Dispense per il corso di

Dicembre 2001—Luglio 2003

Fondamenti dell'Informatica: Linguaggi Formali e Calcolabilità

Agostino Dovier
Dipartimento di Matematica ed Informatica
Università degli Studi di Udine
Via delle Scienze, 206, Loc. Rizzi
33100 Udine, Italy
dovier@dimi.uniud.it

Roberto Giacobazzi
Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15
37134 Verona, Italy
giaco@sci.univr.it

Contents

CHAPTER 1

Introduzione

L'informatica come scienza si fonda sulla teoria della calcolabilità effettiva sviluppata nell'ambito dello studio dei fondamenti della matematica, intorno agli anni '30. Tale avventura intellettuale fornisce non solo le basi concettuali e teoriche dell'informatica, ma anche rappresenta un affascinante viaggio verso i limiti dell'informatica stessa. Ogni disciplina scientifica si definisce pienamente nel momento in cui essa viene delimitata da una teoria in grado di evidenziarne i limiti e le potenzialità. È così per la fisica classica e quantistica, per la psicoanalisi, per la chimica etc. Anche l'informatica ha, come ogni scienza, una teoria che ne definisce in modo universale i limiti e le potenzialità. Tale disciplina, la cui nascita si può far risalire agli anni '30 in un effervescente panorama culturale e scientifico di inizio secolo—sono di quel periodo gli studi sulla materia, sulla meccanica quantistica e sui fondamenti della matematica—si sviluppa pienamente indipendentemente dalla realizzazione, avvenuta solo successivamente negli anni '40, del primo calcolatore elettronico. Possiamo quindi affermare, in modo forse provocatorio, che il successo attuale dell'informatica realizzata mediante dispositivi elettronici piuttosto che biomolecolari, quantistici o altro, sia stato più un caso che ha voluto la maturazione contemporanea dell'informatica come scienza e dell'elettronica intorno agli anni '40 e '50, piuttosto che una necessità intrinseca del processo di calcolo. L'indipendenza da una particolare macchina fisica è uno dei punti di forza dell'informatica come scienza. L'errore comunemente compiuto di identificare l'informatica con il computer moderno, o almeno con l'architettura che conosciamo oggi, ancora basata sulle idee di von Neumann-Turing, limita questa disciplina ad una mera programmazione di particolari macchine. Al contrario, l'informatica prescinde dal particolare strumento di calcolo, sia esso il computer moderno, un insieme di molecole (DNA-computing [**24**]) o particelle (Quantum-computing [**12**]), o sia esso definito da un mero calcolo simbolico (λ-calcolo [**3, 11**]). In questo senso l'informatica può definirsi a pieno titolo come una scienza universale dell'informazione, ovvero di come l'informazione possa essere codificata, manipolata, valutata, analizzata e misurata.

1. Il quadro storico

Questa affascinante avventura intellettuale inizia negli anni '30 grazie al contributo di eminenti studiosi quali Church, Gödel, Kleene, Post e Turing. Il quadro storico-scientifico dell'epoca vede il progressivo sgretolarsi delle teorie classiche del '700 e '800, sia in ambito fisico che matematico che medico. Non è un caso che la scoperta di leggi fisiche nuove che superano le leggi classiche di Newton nello studio di fenomeni atomici, la relatività di Einstein, la scoperta della psicoanalisi di Freud come strumento per investigare l'inconscio, e gli studi sui fondamenti della matematica compiuti da Hilbert, Russell, Weyl, e Gödel, siano tutti riconducibili ad un medesimo quadro storico: gli inizi del 1900. In questo contesto, si sviluppa in particolar modo la teoria della calcolabilità effettiva. Essa si può ragionevolmente collocare nell'ambito degli studi sui fondamenti della matematica, in particolar modo dell'aritmetica. La necessità di capire profondamente la natura di ciò che è *effettivamente costruibile* mediante una sequenza di *passi* elementari di calcolo nasce dalla volontà di costruire effettivamente teorie complesse, quali l'aritmetica. Questo ambizioso programma inizia già nel 1879 con i lavori di Frege [**8**] aprendo l'era della ricostruzione logica della matematica culminata con i *Principia Mathematica* di Russell. Le prime critiche a questo programma emergono già con la scoperta dei paradossi. Il paradosso di Russell (1902-3) è destinato a minare alle fondamenta l'impianto teorico su cui si fonda la teoria degli insiemi di Cantor. Esso riguarda *l'insieme di tutti gli insiemi che non sono membri di sé stessi*. Se chiamiamo T questo insieme, si ha che se $\mathsf{T} \in \mathsf{T}$ allora $\mathsf{T} \notin \mathsf{T}$ e analogamente se $\mathsf{T} \notin \mathsf{T}$ allora $\mathsf{T} \in \mathsf{T}$. Più volgarmente: *consideriamo il barbiere di un villaggio che rade solo coloro del villaggio che non si radono da soli. Il barbiere del villaggio si rade?* Paradossi simili si riscontrano già nell'antica grecia, ad esempio il paradosso del mentitore o di Epimenide si fonda su un simile ragionamento "circolare": *L'affermazione: I cretesi sono bugiardi, è attribuita ad Epimenide di Creta VI B.C. È vera questa affermazione?* [**17**] A seguito dell'emergere di paradossi che minano apparentemente alla base il tentativo di ricostruire in chiave puramente logica l'intera matematica, è emersa la necessità di definire una assiomatizzazione completa per la matematica, in particolar modo la teoria degli insiemi su cui essa si fonda. Assiomatizzare una teoria significa definire un insieme di assiomi universalmente validi ed esenti da contraddizioni per quella teoria ed essere in grado di derivare i teoremi che ne conseguono a partire da quegli assiomi. Il contributo di Hilbert in questo quadro è fondamentale. Secondo Hilbert, la formalizzazione di una teoria, mediante la definizione di assiomi e regole di derivazione, comporta una astrazione dal significato degli oggetti manipolati e la definizioni di principi e metodi per studiare il sistema formale risultante. Nel *programma di Hilbert* è possibile formalizzare in modo completo la matematica, riducendo la matematica stessa ad un mero calcolo simbolico a partire da una sua formalizzazione, ovvero da un insieme di assiomi e regole di inferenza. Il risultato fondamentale che apre definitivamente la strada alla nascita dell'informatica è dovuto a Gödel (1931) [**10**]. In questo fondamentale lavoro, Gödel dimostra che esistono teoremi dell'aritmetica che non sono *decidibili* nell'aritmetica stessa. In particolare la verità di una formula non risulta dimostrabile in modo effettivo (ovvero decisa finitamente) nell'aritmetica. L'importanza di questo risultato sta nel fatto che esso è del tutto indipendente dal particolare sistema formale scelto, purché sufficientemente espressivo da rappresentare l'aritmetica stessa. In questo senso i risultati di incompletezza di Gödel risultano validi per tutti i sistemi formali derivanti da una assiomatizzazione degli insiemi, aggiungendo un qualsiasi numero finito di assiomi, purché questi non generino inconsistenze. L'impatto di questo risultato è enorme e di portata universale: Gödel dimostra l'esistenza di questioni riguardanti i numeri che non sono decidibili nella teoria stessa ed in ogni sua finita estensione consistente. In questo senso, Gödel stabilisce i limiti stessi della formalizzazione di teorie quali l'aritmetica, e pone le basi per il successivo ragionamento su ciò che è effettivamente costruibile. In questo contesto nasce quindi la necessità di approfondire la natura stessa del calcolo logico-formale e di come da questo sia possibile derivare in modo sistematico enunciati la cui verità sia decidibile. Il fallimento quindi dell'idea di ridurre l'intera matematica ad un mero calcolo *automatico* a partire da assiomi e regole di inferenza, costituisce il terreno ideale per lo studio di ciò che effettivamente è calcolabile in questo senso, ovvero stabilisce i limiti di quella disciplina che oggi si chiama informatica. La necessità di formalizzare il processo di calcolo, sia mediante la definizione di una *macchina calcolatrice* che attraverso sistemi formali di calcolo quali il λ-calcolo, ha portato all'analisi della calcolabilità di Turing e Church nel 1936 [**4, 30**]. Dai loro lavori scaturisce di fatto la teoria presentata in modo didattico nel presente volume e si definiscono le basi per i successivi sviluppi pratici e teorici del calcolo mediante calcolatore. Ad esempio è possibile garantire in modo formale l'esistenza di una macchina universale (calcolatore programmabile) o caratterizzare rigorosamente la nozione di programma come sequenza di istruzioni, a prescindere dall'esistenza di una macchina fisica, insieme di circuiti e apparecchiature, in grado di eseguire tali programmi. Si dimostra l'esistenza di funzioni non calcolabili e, di conseguenza, l'esistenza di problemi non risolvibili in modo completo e automatico mediante calcolatore.

2. Una prima classificazione

Una prima classificazione dei problemi tra quelli che possono essere risolti mediante algoritmi e quelli che non ammettono tale soluzione, è ottenibile applicando il Teorema di Cantor. Vedremo nel seguito, nel capitolo 2, questo teorema applicato alla determinazione della cardinalità di insiemi di funzioni. Tuttavia, è possibile darne una dimostrazione del tutto informale, ricorrendo a semplici nozioni intuitive, senza ricorrere a notazioni più complesse sui numeri. Questa dimostrazione permette di dare una prima classificazione, piuttosto grossolana, degli insiemi e distinguere, se non altro per quanto riguarda la loro "dimensione", tra i problemi che possono essere definiti matematicamente e quelli che invece ammettono una soluzione effettiva o algoritmica.

Supponiamo di avere a disposizione un linguaggio per programmare una macchina. Tale linguaggio è caratterizzato da un insieme finito di istruzioni in grado di far eseguire operazioni più o meno complesse alla macchina. I dati sono semplici numeri naturali. Un algoritmo è quindi una sequenza finita di istruzioni nel linguaggio della macchina. Secondo queste ipotesi, è quindi possibile mettere in sequenza tutti i possibili algoritmi, ovvero *enumerare* gli algoritmi così come è possibile enumerare i numeri naturali:

$$P_0, P_1, P_2, \ldots, P_n, \ldots$$

Se un algoritmo è destinato a manipolare dati in *input*, producendo dati in *output*, allora un problema è rappresentabile come una *funzione* ovvero una associazione tra dati in input e dati in output. Nell'esempio, un problema è quindi una funzione sui numeri naturali: $f : \mathbb{N} \longrightarrow \mathbb{N}$. Un problema ammette soluzione algoritmica se esso è programmabile da un algoritmo, ovvero se esiste un n tale che P_n programma la funzione del problema: per ogni $x \in \mathbb{N}$, $P_n(x) = f(x)$. Costruiamo allora il seguente problema $h : \mathbb{N} \longrightarrow \mathbb{N}$:

input: n
determina il programma P_n
calcola $P_n(n)$
output: $P_n(n) + 1$

Se il linguaggio è sufficientemente potente per poter programmare il precedente problema, allora esso corrisponderà ad una sequenza di istruzioni del linguaggio, ovvero esisterà n_0 tale che $P_{n_0} = h$. Calcoliamo ora $h(n_0)$:

$$h(n_0) = P_{n_0}(n_0) + 1 = h(n_0) + 1$$

Questo è un assurdo, poiché *nessun* numero è uguale al suo successore. Da questo assurdo segue il fatto che esistono problemi, ovvero funzioni input/output, non programmabili, ovvero per le quali non è possibile costruire un algoritmo che le risolva. Le ipotesi molto deboli del precedente ragionamento lo rendono applicabile a *tutti* i linguaggi di programmazione noti.

Questa prima osservazione stabilisce già una ripartizione tra i problemi definibili matematicamente come funzioni e quelli effettivamente risolvibili mediante una macchina calcolatrice. Dimostrazioni come quella appena vista saranno ricorrenti nel presente volume.

3. Il presente volume

In questo volume presenteremo le principali nozioni e risultati che riguardano la teoria della calcolabilità effettiva e della ricorsione. La conoscenza di queste nozioni è indispensabile per intraprendere l'avventura intellettuale che ha portato alla nascita dell'informatica, per conoscerne i limiti e le potenzialità e per poterne tracciare i confini nel panorama delle scienze. Studiare la calcolabilità significa capire se un dato problema è risolvibile mediante calcolatore. Al contrario dello studio della complessità (cf. [**9, 23**]), presentato nella parte finale del testo, che studia le condizioni per cui un dato problema è risolvibile o meno avendo a disposizione una quantità limitata di risorse, siano esse il tempo o la memoria (spazio), la calcolabilità non pone restrizioni alle risorse disponibili. Capire se un problema è calcolabile significa capire se esso ammette una soluzione algoritmica indipendentemente da quanto tempo/spazio è necessario per risolverlo. Gli strumenti utilizzati per studiare la calcolabilità sono per molti aspetti simili a quelli utilizzati per studiare la complessità dei problemi (ad esempio il concetto di riduzione funzionale è presente nella definizione di completezza sia per classi di ricorsività che per classi di complessità). In questo senso, la teoria della calcolabilità è di fatto propedeutica ad ogni altra teoria che studi modelli per il calcolo automatico, sia essa la complessità, la semantica etc... Scopo di questo testo è quello di fornire i risultati principali che permettono di analizzare un problema, classificandolo in termini della sua risolvibilità algoritmica. L'approccio seguito in questo testo non corrisponde allo sviluppo storico dei concetti presentati. Partendo da semplici macchine a stati finiti, raggiungeremo i confini di ciò che è calcolabile arricchendo via via le nostre macchine con strutture dati opportune per memorizzare informazioni. Scopriremo poi che l'architettura del computer moderno corrisponde in tutto e per tutto all'architettura ideale della Macchina di Turing (1937) [**30**], ottenibile arricchendo semplici automi a stati finiti con opportune strutture dati. In questo modo arriveremo allo studio astratto (matematico) della calcolabilità effettiva attraversando la teoria dei linguaggi formali e delle macchine preposte al loro riconoscimento. Lo studio formale dei linguaggi generabili a partire da un dato alfabeto è alla base dello sviluppo delle tecniche di traduzione dei moderni linguaggi di programmazione, quali interpreti e compilatori. Lo studio della complessità fornisce metodologie che permettono al programmatore di stabilire a priori il grado di difficoltà in termini di tempo di esecuzione e/o consumo di memoria del problema da risolvere, che rappresenta il limite da raggiungere dall'algoritmo sviluppato per risolverlo. Tali metodologie e risultati devono costituire, per un informatico, ma più in generale per chi utilizza professionalmente un calcolatore, una base di conoscenze indispensabile come le nozioni fondamentali di algebra e di analisi lo sono per il matematico.

In letteratura vi sono molti ottimi testi sull'argomento (si consulti la Bibliografia). La riorganizzazione a "moduli" di 4–6 crediti (30–50 ore di didattica frontale a seconda delle scelte della sede) del sistema universitario italiano rende tuttavia molti di questi testi sovrabbondanti oppure troppo avanzati oppure orientati ad un pubblico con solide basi matematiche (per esempio i testi di Hopcroft-Ullman [**13**], di Rogers [**27**] o di Cutland [**6**]). Si è pertanto cercato di fornire una presentazione il più possibile adatta a studenti del Corso di Laurea in Informatica orientata a uno o due possibili corsi di 6 crediti del II–IV anno. Gli unici prerequisiti sono una conoscenza di base della teoria elementare degli insiemi e di logica che vengono comunque richiamati nel Capitolo 2. Riteniamo pertanto che il testo possa essere utilizzato in un corso di Fondamenti teorici dell'Informatica nei Corsi di laurea in Informatica, Matematica, Fisica, Statistica e Ingegneria.

Il testo è diviso in 4 parti, per ciascuna delle quali sono necessarie poco meno di 30 ore. Complessivamente, pertanto, il testo è complessivamente strutturato per due corsi di 6 crediti l'uno.

Per quanto gli autori comprendano ma non condividano la scelta di un unico corso di Fondamenti da 6 crediti, ritengono che il testo possa comunque venire utilizzato anche in tale contesto. Il contenuto di diversi capitoli della Parte 2, ed i principi generali della parte 4, infatti, vengono talvolta anticipati in altri corsi (per esempio Programmazione, Linguaggi di Programmazione o Algoritmi e Strutture Dati). Questo consente di ridurre a meno di 20 ore la parte 2 e, eventualmente, di omettere la parte 4 o di riassumerne i principali risultati in circa 4 ore. Omettendo inoltre parte delle dimostrazioni dei paragrafi 6.5, 6.6 e 6.7, la Parte 1 può a sua volta essere svolta in circa 20 ore. Le ore rimanenti del corso possono essere impiegate per i principali risultati della parte 3, in particolare per il Capitolo 17. I Teoremi di ricorsione e la riducibilità funzionale possono eventualmente essere omessi in un unico corso di Fondamenti di 6 crediti.

Il presente volume è corredato da un relativo eserciziario, con esercizi risolti, realizzato dalla Dott.ssa Isabella Mastroeni.

Si intende ringraziare gli studenti del corso di laurea in Informatica dell'Università di Verona per i loro commenti che hanno aiutato a chiarire e correggere alcune parti di questo testo. Si ringraziano Isabella Mastroeni, Mila Dalla Preda, Carla Piazza, Andrea Fusiello e Nicola Vitacolonna per le discussioni ed osservazioni relative alla redazione del presente volume, e tutti gli studenti del III e IV anno del Corso di Laurea in Informatica dell'Università di Verona, A.A. 1998–99, per il loro attivo contributo alla realizzazione di questo testo. Un grazie particolare ad Antonella Meneghetti per l'attenta rilettura delle bozze.

Gli autori saranno grati a tutti coloro che avessero la gentilezza di segnalare sviste ed errori o volessero comunque far pervenire i loro commenti sul contenuto e la forma del testo.

Dicembre 2001—Settembre 2003

AGOSTINO DOVIER

ROBERTO GIACOBAZZI

CHAPTER 2

Notazione e concetti di base

In questo breve capitolo viene stabilita la notazione impiegata nel testo relativamente ai concetti base di matematica e logica utilizzati. Con ciò non si intende fornire una presentazione auto-contenuta di tali concetti, per la quale si rimanda ai testi classici, quali ad esempio [**7, 20, 16**].

1. Insiemi

Con lettere latine maiuscole denoteremo in genere *insiemi* di oggetti. Alcuni insiemi hanno un nome particolare che li identifica univocamente: ad esempio $\mathbb{N}$ rappresenta l'insieme dei numeri naturali ($\mathbb{N} = \{0, 1, 2, 3, \ldots\}$).

$x \in A$ significa che x è un *elemento* dell'insieme A. Mediante la rappresentazione *intensionale* di insiemi $\{x \mid E(x)\}$ si identifica l'insieme costituito dagli x che soddisfano (rendono vera) una data espressione E, la quale dipende da x ed ha valori booleani. Se gli elementi di un insieme hanno un indice e sono tutti rappresentabili come una sequenza (anche infinita) di elementi x_i al variare di $i \in I$, allora l'insieme costituito da questi oggetti è rappresentato con $\{x_i\}_{i \in I}$.

$A \subseteq B$ indica che A è un *sottoinsieme* di B, ovvero che ogni elemento di A è anche elemento di B. Con la notazione $A \subset B$ si denoterà l'inclusione *stretta* ovvero vale che $A \subseteq B$ e $A \neq B$.

Con $\wp(A)$ si denota l'*insieme delle parti* dell'insieme A, ovvero l'insieme costituito da tutti i suoi sottoinsiemi: $\wp(A) = \{X \mid X \subseteq A\}$.

Unione ed intersezione di insiemi sono rispettivamente denotati con i simboli $\cup$ e $\cap$. Il simbolo $\setminus$ rappresenta la *differenza insiemistica*: $A \setminus B = \{x \mid x \in A \land x \notin B\}$.

$\bar{A}$ denota il *complemento* di A, ovvero $x \in \bar{A}$ se e solo se $x \notin A$. Se assumiamo (come faremo spesso nel seguito) che gli insiemi sono insiemi di numeri naturali, allora considerando $\mathbb{N}$ come insieme *universo*, si ha che $\bar{A} = \mathbb{N} \setminus A$.

Dati due o più insiemi, è possibile costruire coppie, triple, etc. di oggetti. In generale una n-*upla* di oggetti $x_1, \ldots, x_n$ di un insieme A è rappresentata con $\langle x_1, \ldots, x_n \rangle$. L'insieme delle n-uple di elementi di A è rappresentato con A^n e corrisponde al *prodotto cartesiano* di A n-volte. Insiemi diversi possono essere messi in prodotto cartesiano:

$$A_1 \times A_2 \times \cdots \times A_n = \{\langle x_1, \ldots, x_n \rangle \mid x_1 \in A_1, \ldots, x_n \in A_n\}$$

indica l'insieme delle n-uple di oggetti appartenenti rispettivamente agli insiemi $A_1, \ldots, A_n$.

2. Relazioni e induzione ben fondata

Una relazione (binaria) è un sottoinsieme del prodotto cartesiano di (due) insiemi; dati A e B, $\mathcal{R} \subseteq \mathsf{A} \times \mathsf{B}$ è una relazione su A e B. Ad esempio, la relazione di ordinamento sui numeri naturali "$\leq$" $\subseteq \mathbb{N} \times \mathbb{N}$ è definita nel modo seguente: $\leq = \{\langle x, y\rangle \mid x \in \mathbb{N}, y \in \mathbb{N}, x \leq y\}$. Il fatto che $\langle x, y\rangle \in \leq$ viene solitamente indicato con $x \leq y$. Una relazione binaria su un insieme, $\mathbf{R} \subseteq \mathsf{S} \times \mathsf{S}$, stabilisce una precedenza tra gli oggetti di S: se $\langle a, b\rangle \in \mathcal{R}$ diremo che a *precede* b. $\mathbf{R}$ verrà spesso chiamata *relazione di precedenza*. $\mathbf{R}$ è detta:

***riflessiva*:** se per ogni $a \in \mathsf{S}$ si ha che $a\ \mathsf{R}\ a$,
***simmetrica*:** se per ogni $a, b \in \mathsf{S}$ si ha che se $a\ \mathsf{R}\ b$ allora $b\ \mathsf{R}\ a$, e
***transitiva*:** se per ogni $a, b, c \in \mathsf{S}$ si ha che se $a\ \mathsf{R}\ b$ e $b\ \mathsf{R}\ c$ allora $a\ \mathsf{R}\ c$.

Una relazione $\mathbf{R}$ di *equivalenza* è una relazione binaria riflessiva, simmetrica e transitiva. Per ogni relazione $\mathbf{R} \subseteq \mathsf{S} \times \mathsf{S}$, la chiusura transitiva di $\mathbf{R}$ è il più piccolo insieme $\mathbf{R}^*$ tale che: $\langle a, b\rangle \in \mathbf{R}^* \ \wedge\ \langle b, c\rangle \in \mathbf{R} \ \rightarrow \langle a, c\rangle \in \mathbf{R}^*$. Trattandosi di una definizione induttiva, la chiusura transitiva di una relazione $\mathbf{R}$ è ottenibile come unione delle seguenti relazioni $\mathbf{R}^i$ per $i \in \mathbb{N}$:

$$\begin{cases} \mathbf{R}^1 & = \quad \mathbf{R} \\ \mathbf{R}^{i+1} & = \quad \big\{\langle a, c\rangle : \langle a, b\rangle \in \mathbf{R}^i \ \wedge\ \langle b, c\rangle \in \mathbf{R}\big\} \end{cases}$$

ovvero $\mathbf{R}^* = \bigcup_{i \in \mathbb{N}} \mathbf{R}^i$. Una relazione binaria $\mathbf{R} \subseteq \mathsf{S} \times \mathsf{S}$ è un *preordine* se essa è riflessiva e transitiva. $\mathbf{R}$ è un *ordine parziale* se è un preordine ed è *antisimmetrica*, ovvero

$$a\,\mathbf{R}\,b \ \wedge\ b\,\mathbf{R}\,a \ \rightarrow\ a = b$$

Useremo nel seguito i simboli $\sqsubseteq, \preceq, \leq, \ldots$ per rappresentare relazioni d'ordine parziale. Un insieme S con una relazione di precedenza (non necessariamente una relazione d'ordine) $\prec \subseteq \mathsf{S} \times \mathsf{S}$ è rappresentato nel modo seguente: $(\mathsf{S}, \prec)$. Se $\prec$ è una relazione d'ordine parziale, $(\mathsf{S}, \prec)$ è un *insieme parzialmente ordinato*. $(\mathsf{S}, \preceq)$ è un ordinamento totale se, per ogni $x, y \in \mathsf{S}$, o $x \preceq y$ oppure $y \preceq x$. Sia $(\mathsf{S}, \prec)$ un insieme con relazione di precedenza $\prec$. L'elemento $x \in \mathsf{S}$ è *minimale* rispetto a $\prec$ se $\forall y \in \mathsf{S}.\ y \not\prec x$. Gli elementi *massimali* di un insieme con relazione di precedenza, quando esistono, sono definiti invertendo la relazione $\prec$. Ovvero $x \in \mathsf{S}$ è massimale rispetto a $\prec$ se $\forall y \in \mathsf{S}.\ x \not\prec y$.

Esempio 2.1. Sia $\mathbb{N}$ l'insieme dei numeri naturali, e $\prec$ la relazione definita da $x \prec y \stackrel{\text{def}}{=} y = x + 1$. In altri termini, $x \prec y$ *se e solamente se* y è il successore di x. L'unico elemento minimale rispetto a $\prec$ è 0. Analogamente, possiamo estendere $\prec$ all'insieme $\mathbb{Z}$ dei numeri interi stipulando che $(x \prec y \stackrel{\text{def}}{=} y = x + 1)$ anche per i numeri negativi. Non esistono, però, in $\mathbb{Z}$ elementi minimali rispetto a $\prec$. Un esempio di relazione di precedenza non banale su $\mathbb{N}$ è dato invece dalla seguente definizione di $\sqsubset$:

$$x \sqsubset y \stackrel{\text{def}}{=} x \neq y \wedge \mathit{divide}(x, y)$$

dove *divide*(x, y) esprime il fatto che x è un divisore di y. L'unico elemento minimale di $(\mathbb{N}, \sqsubset)$ è 1. Cos'è la chiusura transitiva di $\prec$?

Una relazione binaria $\mathbf{R} \subseteq S \times S$ è *ben-fondata* se per ogni insieme non vuoto $Y \subseteq S$ esiste $z \in Y$ tale che $\langle y, z \rangle \notin \mathbf{R}$ per ogni $y \in Y \setminus \{z\}$. Se la relazione in questione è una relazione di ordine parziale $\preceq$, allora $\preceq$ è ben-fondata se ogni insieme non vuoto $Y \subseteq S$ ha un elemento minimale. Se $\mathbf{R} \subseteq S \times S$ è ben-fondata, allora si dirà pure che S (o, con abuso di notazione, $(S, \mathbf{R})$) è un insieme ben-fondato.

Veniamo all'enunciato del principio di induzione ben fondata, che generalizza i principi di induzione sui naturali. Sia S un insieme. Il seguente principio di induzione ben fondata permette di dimostrare proprietà di insiemi generici, ovvero non solo di numeri naturali, equipaggiati con una relazione ben fondata.

TEOREMA 2.2 (Induzione Ben Fondata). *Sia* $(S, \prec)$ *un insieme ben fondato e sia* $\phi(x)$ *una proprietà su* S*. Allora vale il seguente asserto:*

$$\textbf{\textit{se}} \qquad \forall x \in S.\ \forall y, y \prec x.\ \phi(y) \rightarrow \phi(x)$$

$$\textbf{\textit{allora}} \qquad \forall x \in S.\ \phi(x).$$

Una lettura informale del principio di induzione ben fondata può essere data come segue: per dimostrare che un asserto $\phi(x)$ vale per ogni elemento x di un insieme ben fondato $(S, \prec)$ è sufficiente dimostrare che ϕ vale su un generico elemento x di S, nell'ipotesi che ϕ valga su ogni elemento y di S che precede x secondo $\prec$. Fissato un generico x in S, la formula

$$\forall y, y \prec x.\ \phi(y)$$

viene detta ipotesi induttiva rispetto a x.

Si noti che, in pratica, la dimostrazione di un asserto ϕ basata sul principio di induzione ben fondata avviene in due passi. Analogamente al caso dell'induzione sui naturali, il primo passo della dimostrazione viene comunemente detto caso base, mentre il secondo viene detto caso induttivo.

Caso Base: si dimostra che ϕ vale su tutti gli elementi di S minimali rispetto a $\prec$ (ricordiamo che dato $(S, \prec)$, un elemento $x \in S$ si dice minimale se non esiste nessun elemento $y \in S$ che preceda x). Infatti, se x è minimale, l'insieme degli $y \in S$ tali che $y \prec x$ è vuoto, e dunque l'ipotesi induttiva rispetto a x si riduce a *True*. Di conseguenza la formula $(\forall y, y \prec x.\ \phi(y)) \rightarrow \phi(x)$ si riduce semplicemente a $\phi(x)$.

Caso induttivo: detto x un generico elemento di S, non minimale rispetto a $\prec$, si dimostra la validità di $\phi(x)$ utilizzando, laddove si renda necessaria, l'ipotesi induttiva rispetto a x, ovvero l'ipotesi che ϕ stessa vale su tutti gli elementi di S che precedono x.

Si osservi come il principio di induzione ben fondata enunciato sopra, generalizzi entrambi i principi di induzione matematica sui naturali, ed induzione completa. Il principio di induzione matematica sui naturali coincide con il principio di induzione ben fondata rispetto all'insieme con precedenza ben fondato $(\mathbb{N} \setminus \{0, \ldots, n_0 - 1\}, \prec)$, dove $\prec$ è definita come segue:

$$x \prec y \stackrel{\text{def}}{=} y = x + 1.$$

Il principio di induzione completa corrisponde invece al principio di induzione ben fondata rispetto alla chiusura riflessiva e transitiva di $\prec$, ovvero rispetto all'insieme ben fondato $(\mathbb{N} \setminus \{0, \ldots, n_0 - 1\}, <)$ (vedi esempio 2.1).

Prima di vedere alcuni esempi di uso del principio di induzione ben fondata, ne diamo una dimostrazione informale.

***Dimostrazione*:** Sia $(S, \prec)$ un insieme ben fondato e sia ϕ una proprietà su S per cui l'ipotesi del teorema di induzione ben fondata è verificata. Consideriamo l'insieme $X = \{x \mid \neg\phi(x)\}$. Quindi, $X \neq \varnothing$ se e solo se esiste un elemento $x_0 \in S$ per cui non vale ϕ. Supponiamo per assurdo che esista $x_0 \in X$. Se x_0 è minimale abbiamo immediatamente una contraddizione con la validità del principio di induzione ben fondata, visto che, come abbiamo appena osservato, tale principio garantisce la validità di ϕ su tutti gli elementi minimali di S (Caso Base). Dunque x_0 non è un elemento minimale: deve allora esistere un elemento $x_1 \in S$ tale che $x_1 \prec x_0$ e $x_1 \neq x_0$, per il quale non vale ϕ. Se così non fosse, infatti, avremmo di nuovo una contraddizione con la validità del principio di induzione ben fondata, che garantisce la validità di ϕ su ogni elemento non minimale x, nell'ipotesi che ϕ valga su tutti gli elementi che precedono x. Possiamo ora ripetere questo ragionamento su x_1. Di nuovo quest'ultimo non può essere un elemento minimale e deve esistere un elemento di S, sia esso x_2 che precede x_1 ($x_2 \prec x_1$ e $x_2 \neq x_1$), e su cui non vale ϕ, etc. In questo modo possiamo costruire una sequenza infinita di elementi, tutti distinti tra loro, tali che

$$\cdots \prec x_k \prec x_{k-1} \prec \cdots \prec x_2 \prec x_1 \prec x_0$$

e tali che, per ogni x_i nella sequenza, $\phi(x_i)$ non vale. Ma, osservando che la sequenza così costruita è una catena decrescente infinita, contraddiciamo l'ipotesi che $(S, \prec)$ sia un insieme ben fondato, ovvero che non contenga catene decrescenti infinite.

3. Funzioni

Consideriamo una relazione n-aria, ovvero $\mathcal{R} \subseteq A_1 \times \cdots \times A_n$. Sia $k < n$. Una relazione $\mathcal{R}$ n-aria è una *funzione* di k argomenti se, dati $x_1 \in A_1, \ldots, x_k \in A_k$, esiste uno ed un solo $z_{k+1} \in A_{k+1}, \ldots, z_n \in A_n$ tale che $\langle x_1, \ldots, x_k, z_{k+1}, \ldots, z_n \rangle \in$

$\mathcal{R}$. In questo caso, la funzione è definita in $A_1 \times \cdots \times A_k$ ed ha valori in $A_{k+1} \times \cdots \times A_n$, ovvero è del tipo:

$$\mathcal{R} : A_1 \times \cdots \times A_k \longrightarrow A_{k+1} \times \cdots \times A_n$$

Nel caso $n = 2$ e $k = 1$, otteniamo la definizione usuale di funzione di un argomento: $\mathcal{R} : A_1 \longrightarrow A_2$ come relazione binaria $\mathcal{R} \subseteq A_1 \times A_2$. A_1 è detto *dominio* di $\mathcal{R}$ mentre A_2 è il suo *co-dominio*. Mentre una funzione è anche a sua volta una relazione, il viceversa non vale sempre. È sempre possibile invece associare una funzione ad una relazione, modificandone il codominio: Sia $\mathbf{R} \subseteq A \times B$ una relazione, definiamo $f_{\mathbf{R}} : A \longrightarrow \wp(B)$ come segue:

$$f_{\mathbf{R}}(x) = \{y \,:\, x \,\mathbf{R}\, y\}$$

Denoteremo come $\vec{x}$ una generica n-upla (vettore) di oggetti. La lunghezza di tale n-upla si evincerà dal contesto. Una funzione è *iniettiva* se $f(\vec{x}_1) = f(\vec{x}_2)$ implica che $\vec{x}_1 = \vec{x}_2$ per ogni $\vec{x}_1, \vec{x}_2$; è *suriettiva* se per ogni elemento $\vec{y}$ del *codominio* (nel caso sopra $A_{k+1} \times \cdots \times A_n$) esiste $\vec{x}$ nel *dominio* (nel caso sopra $A_1 \times \cdots \times A_k$) tale che $f(\vec{x}) = \vec{y}$; è *biiettiva* se è iniettiva e suriettiva. Sia $f : A \longrightarrow B$ una funzione. Con $f(D)$ indichiamo l'*immagine* di D secondo f. L'immagine $f(D)$ è definita per ogni $D \subseteq A$ come l'insieme:

$$f(D) = \{y \in B \,:\, y = f(x) \;\wedge\; x \in D\}$$

Con f^{-1} indichiamo l'*immagine inversa* di f. L'immagine inversa di una funzione è un insieme così definito per ogni $D \subseteq B$:

$$f^{-1}(D) = \{x \in A \,:\, f(x) \in D\}$$

Una funzione può non essere definita su alcuni dei suoi argomenti, ovvero se la funzione di k argomenti è data dalla relazione $\mathcal{R} \subseteq A_1 \times \cdots \times A_n$, con $k < n$, allora possono esistere degli $x \in A_i$ argomenti di $\mathcal{R}$ (per $i \leq k$) tali che nessuna n-upla $\langle \ldots, x, \ldots \rangle$ avente x nella i-esima posizione appartiene a $\mathcal{R}$. In questo caso diremo che la funzione $\mathcal{R}$ è *parziale*. Al contrario, una funzione sempre definita su ogni argomento è detta *totale*. Con la notazione $A \longrightarrow B$ si intende l'insieme delle funzioni totali da A a B.

NOTA 2.3. Normalmente si usa parlare di funzioni $f : A \longrightarrow B$ e $f(x)$ è definita per ogni elemento x dell'insieme A (dominio). Ad esempio, la funzione $f(x) = 1000 \texttt{ div } x$, ove `div` ritorna il quoziente della divisione intera, è definita da $A = \mathbb{N} \setminus \{0\}$ a $B = \mathbb{N}$. In questo testo, per mantenere un'analogia con le funzioni calcolabili mediante un calcolatore (che potrebbe non fornire risultato per certi valori di input), si preferisce parlare di $f : \mathbb{N} \longrightarrow \mathbb{N}$ e parlare di funzioni parziali.

Nel seguito faremo largo uso di funzioni parziali e totali. Al fine di evitare confusione, saremo soliti rappresentare le funzioni secondo la notazione usuale:

$$\mathcal{R} : A_1 \times \cdots \times A_k \longrightarrow A_{k+1} \times \cdots \times A_n$$

identificando in questo modo gli insiemi su cui vengono considerati gli argomenti e gli insiemi su cui la funzione restituisce risultati. In particolare, rappresenteremo con lettere minuscole latine le funzioni totali; ad esempio:

$$f : A_1 \times \cdots \times A_k \longrightarrow A_{k+1} \times \cdots \times A_n$$

rappresenta una funzione totale di k argomenti, mentre rappresenteremo con lettere minuscole greche le funzioni parziali:

$$\varphi : A_1 \times \cdots \times A_k \longrightarrow A_{k+1} \times \cdots \times A_n$$

rappresenta una funzione parziale di k argomenti. Poiché nell'insieme delle funzioni parziali vi sono anche le funzioni totali, nel caso non vi siano ipotesi sulla totalità della funzione considerata, rappresenteremo con lettere greche minuscole una generica funzione.

Per indicare che una funzione φ è definita in x, scriveremo $\varphi(x)\downarrow$. Analogamente, per indicare che una funzione φ non è definita in x, scriveremo $\varphi(x)\uparrow$, o equivalentemente $\varphi(x) = \uparrow$. Il simbolo $\uparrow$ rappresenta la non definizione della funzione φ. Supponiamo quindi di ammettere come possibile valore della funzione anche il valore indefinito $\uparrow$. Questo ci permette di vedere le funzioni parziali come funzioni totali del seguente tipo:

$$\varphi : A_1 \times \cdots \times A_k \longrightarrow (A_{k+1} \cup \{\uparrow\}) \times \cdots \times (A_n \cup \{\uparrow\})$$

Data dunque una generica funzione (parziale o totale) φ chiameremo *dominio* della funzione φ l'insieme su cui φ è definita, ovvero l'insieme:

$$\mathtt{dom}(\varphi) = \{x \mid \varphi(x)\downarrow\}$$

Analogamente chiameremo *immagine* o *range* di φ l'insieme dei risultati definiti della funzione, ovvero:

$$\mathtt{range}(\varphi) = \{x \mid \exists z.\ \varphi(z) = x \neq \uparrow\}$$

Un valore della funzione φ è un elemento di $\mathtt{range}(\varphi)$.

In genere la definizione di una funzione $f(X, Y) = X + Y$ crea un'associazione (relazione) tra il nome f della funzione ed il corpo della funzione stessa $X + Y$. Eseguire f non significa altro che eseguire il suo corpo. In alcuni contesti è utile potersi riferire ad una funzione senza essere costretti ad associare ad essa un nome; il concetto di funzione ha infatti un suo significato a prescindere dal nome che gli viene assegnato con la definizione. A questo scopo viene introdotta la cosiddetta λ-notazione. Per esempio la funzione che riceve due argomenti e ne restituisce la somma può essere definita con: $\lambda X, Y \,.\, X + Y$ dove λ è un simbolo speciale, generalmente seguito da una lista di variabili separate da una virgola (i *parametri* della funzione). Il corpo della funzione segue il punto: si tratta di un'espressione in cui possono apparire le variabili introdotte dopo il simbolo λ. Queste variabili risultano *legate* all'interno dell'espressione (bound variables), mentre si definiscono *libere* quelle variabili che appaiono nel corpo e non appaiono dopo il simbolo lambda.

L'applicazione di una funzione definita con la λ-notazione ha la seguente forma: alla definizione della funzione racchiusa fra parentesi seguono tante espressioni quanti sono i parametri. Ad esempio: $(\lambda X, Y . X + Y)7\ 4$ associa 7 a X e 4 a Y e restituisce 11.

4. Cenni di Logica Matematica

Un *linguaggio del prim'ordine* $\mathcal{L}$ è costituito da

- un insieme di *simboli relazionali* o *predicativi*,
- un insieme di *simboli funzionali*,
- un insieme di *simboli di costante* e da
- un insieme di *simboli logici*.

I simboli logici presenti in ogni linguaggio sono:

- le parentesi ")", "(" e la virgola ","
- un insieme numerabile di variabili $v_0, v_1, \ldots$
- un insieme di connettivi: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- i quantificatori $\forall$ (per ogni) ed $\exists$ (esiste).

Si ricorda che l'implicazione e la doppia implicazione sono superflue in quanto $\varphi \rightarrow \psi$ è equivalente a $(\neg\varphi)\vee\psi$ mentre $\varphi \leftrightarrow \psi$ è equivalente a $(\varphi \rightarrow \psi)\wedge(\psi \rightarrow \varphi)$. Anche uno tra $\vee$ e $\wedge$ è superfluo (si vedano le leggi di De Morgan più avanti), mentre uno solo dei due quantificatori è sufficiente. Tuttavia tali operatori, di uso comune, sono usualmente accettati per fornire maggiore chiarezza alle formule.[1]

Un *termine* di un linguaggio del prim'ordine è definito ricorsivamente nel seguente modo:

(1) una variabile v è un termine;
(2) un simbolo costante c è un termine;
(3) se $t_1, \ldots, t_m$ sono termini e f è un simbolo funzionale m-ario, allora $f(t_1, \ldots, t_m)$ è un termine.

Se p è un simbolo relazionale n-ario di $\mathcal{L}$ e se $t_1, \ldots, t_n$ sono termini allora $p(t_1, \ldots, t_n)$ è una *formula atomica* (o *atomo*). Una *formula* di un linguaggio del prim'ordine è definita ricorsivamente nel seguente modo:

(1) una formula atomica è una formula;
(2) se φ è una formula allora $(\neg\varphi)$ è una formula;
(3) se φ e ψ sono formule, anche $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$ e $(\varphi \rightarrow \psi)$ sono formule;
(4) se φ è una formula e v è una variabile allora $\forall v(\varphi)$ e $\exists v(\varphi)$ sono formule.

Per semplificare la notazione, si assumono le comuni convenzioni per eliminare, qualora ciò non generasse ambiguità, alcune coppie di parentesi dalle formule.

Il significato (la semantica) di una formula del prim'ordine è stabilito qualora venga fornita una *interpretazione*, associata ad un insieme non vuoto detto *dominio* (ad esempio, l'insieme dei numeri naturali $\mathbb{N}$)

[1] Un solo operatore, ad esempio il nor definito come $\overline{\vee}(\varphi, \psi) = \neg(\varphi \vee \psi)$ è in realtà sufficiente, in quanto $\neg\varphi = \overline{\vee}(\varphi, \varphi)$ e $\varphi \vee \psi = \overline{\vee}(\overline{\vee}(\varphi, \psi), \overline{\vee}(\varphi, \psi))$.

- per i simboli di costante (ad esempio, la costante c è il numero 5, d il numero 10),
- per i simboli di funzione (ad esempio, il simbolo funzionale binario f sta a significare il '+' tra numeri interi) e
- venga assegnata un'interpretazione ai predicati sul dominio, ovvero, in parole povere, un valore di verità (*vero* oppure *falso*) agli atomi costruiti con i simboli predicativi e gli oggetti del dominio (ad esempio, $f(5,5) = 10$ è vero).

Fissata una interpretazione per un linguaggio del prim'ordine, usando la semantica degli operatori, è possibile estendere l'interpretazione ad ogni formula chiusa (in cui ogni variabile che vi occorre è quantificata) del linguaggio. Ad esempio, nell'interpretazione accennata sopra, la formula $\exists v(v = f(c,c))$ è vera in quanto per $v = 10$ è soddisfatta, mentre la formula $\forall v(v = f(c,c))$ è falsa (per esempio, si prenda $v = 11$).

Per quanto concerne l'utilizzo della logica del prim'ordine nel prosieguo, sarà fondamentale saper negare una formula logica, ovvero data una formula φ, scrivere in modo equivalente ma più leggibile la formula $\neg\varphi$.

Relativamente alle formule che non hanno quantificatori come simboli più "esterni", le regole da utilizzare sono le seguenti:

Doppia negazione: $\neg(\neg\varphi) = \varphi$
De Morgan 1: $\neg(\varphi \wedge \psi) = (\neg\varphi) \vee (\neg\psi)$
De Morgan 2: $\neg(\varphi \vee \psi) = (\neg\varphi) \wedge (\neg\psi)$

Si osservi in particolare che$\neg(\varphi \to \psi)$ è equivalente a $\varphi \wedge \neg\psi$. Per quanto riguarda la semplificazione della **negazione di quantificatori** valgono:

- $\neg\exists\varphi = \forall\neg\varphi$
- $\neg\forall\varphi = \exists\neg\varphi$

Pertanto, ad esempio,

$$\neg\forall n\exists y(r(y) \wedge y > n \wedge \exists u\exists v(y = u + v \wedge \forall i(r(u + v^i))))$$

ove r è un simbolo unario di predicato, risulta essere:

$$\exists n\forall y(\neg r(y) \vee y \leq n \vee \forall u\forall v(y \neq u + v \vee \exists i(\neg r(u + v^i))))$$

Esplicitando le implicazioni, quest'ultima diventa:

$$\exists n\forall y((r(y) \wedge y > n) \to \forall u\forall v(y = u + v \to \exists i(\neg r(u + v^i))))$$

5. Cardinalità di insiemi

Nel seguito, se S è un insieme, rappresentiamo la sua *cardinalità* con il simbolo $|S|$. In questa sezione richiamiamo i concetti di base in relazione alla cardinalità di insiemi. Diciamo che due insiemi A e B sono *equipotenti* se esiste una funzione biiettiva $f : A \longrightarrow B$. Se A e B sono equipotenti, scriveremo $A \approx B$. $|A| \leq |B|$ se esiste una funzione iniettiva $f : A \longrightarrow B$. Si osservi che la funzione f stabilisce una corrispondenza tra gli elementi dei due insiemi. L'iniettività assicura che la corrispondenza è stabilita elemento per elemento, mentre la suriettività assicura che la quantità degli oggetti nei due insiemi coincide. Per insiemi finiti, la cardinalità sarà quindi un numero naturale, corrispondente al numero di oggetti contenuti nell'insieme. Per insiemi infiniti invece, dobbiamo ricorrere ad una generalizzazione. Più formalmente quindi $|A|$ rappresenta la collezione degli insiemi Y tale che $Y \approx A$. Tale collezione è detta appunto *cardinalità di* A. È evidente che se $A \subseteq B$ allora $|A| \leq |B|$.

Nel seguito saremo particolarmente interessati ad insiemi la cui cardinalità è infinita e numerabile, ovvero che possono essere enumerati, stabilendo un primo, secondo, terzo, etc. elemento. Questo interesse è giustificato dal fatto che i dati manipolati in informatica sono enumerabili, ovvero è possibile metterli in corrispondenza biunivoca con i numeri naturali, essendo essi rappresentati da numeri. Un insieme A è detto *enumerabile* se esso è equipotente all'insieme dei numeri naturali $\mathbb{N}$: $A \approx \mathbb{N}$. La cardinalità degli insiemi infiniti enumerabili è comunemente denotata $\aleph_0$. Un insieme A è finito se $|A| < \aleph_0$. Un insieme A è numerabile se $|A| = \aleph_0$.

Il seguente teorema ha come conseguenza che esistono insiemi non enumerabili.

TEOREMA 2.4 (Cantor). *Sia* S *un insieme.* $|S| < |\wp(S)|$.

PROOF. Per assurdo assumiamo esista una funzione $f : S \longrightarrow \wp(S)$ biiettiva. Sia $A = \{x \in S : x \notin f(x)\}$. Poiché $A \in \wp(S)$ e f è per ipotesi suriettiva, deve esistere $a \in S$ tale che $f(a) = A$. Ora, chiediamoci se a appartenga o meno ad A:

- se $a \in A$ allora, per definizione di A, $a \notin f(a) = A$;
- se $a \notin A = f(a)$ allora, per definizione di A, $a \in A$.

In entrambi i casi si giunge ad una contraddizione: l'unica ipotesi che abbiamo fatto è quella dell'esistenza di una f biiettiva, che pertanto non può essere vera. □

Come conseguenza immediata si ha che: $|\mathbb{N}| < |\wp(\mathbb{N})|$ e che per ogni insieme S, $|S| < |S \longrightarrow \{0,1\}|$. Questo segue dal fatto che è possibile definire un isomorfismo $c : \wp(S) \longrightarrow (S \longrightarrow \{0,1\})$ tale che

$$c(X)(a) = \begin{cases} 1 & \text{se } a \in X \\ 0 & \text{se } a \notin X \end{cases}$$

Inoltre, dal teorema di Cantor segue che è possibile definire una successione di insiemi di cardinalità sempre maggiore:

$$|\mathbb{N}| < |\wp(\mathbb{N})| < |\wp(\wp(\mathbb{N}))| < |\wp(\wp(\wp(\mathbb{N})))| < \cdots$$

Il seguente teorema stabilisce invece una relazione tra la cardinalità dell'insieme dei numeri reali $\mathbb{R}$ e la cardinalità delle funzioni sui naturali.

TEOREMA 2.5. $|\mathbb{R}| = |\wp(\mathbb{N})|$.

Traccia della Dimostrazione. Si dimostra prima l'equicardinalità tra $|\wp(\mathbb{N})|$ e l'intervallo $[0,1)$ di $\mathbb{R}$. La rappresentazione in aritmetica binaria di ogni elemento dell'intervallo $[0,1)$ è costituita da una sequenza infinita di bits della forma:

$$0.b_0b_1b_2b_3b_4b_5\dots$$

con $b_i \in \{0,1\}$. Ad esempio "$0.0000\dots$" rappresenta lo 0, "$0.1000\dots$" rappresenta $\frac{1}{2}$, e così via. Ogni sequenza $b_0b_1b_2\dots$ è inoltre il grafico di una funzione $c(X)$ di quelle descritte sopra che identifica un insieme $X \subseteq \mathbb{N}$ e ogni funzione di quel tipo è rappresentabile in questo modo. Esiste pertanto una biiezione tra $|\wp(\mathbb{N})|$ e l'intervallo $[0,1)$.

Per completare la dimostrazione bisogna mostrare che l'intervallo $[0,1)$ è equipotente a tutto $\mathbb{R}$. Questa parte è lasciata per esercizio (si suggerisce di leggere prima l'intera pagina seguente). $\square$

Le cardinalità di insiemi si possono sommare e moltiplicare tra loro. Intuitivamente, la somma della cardinalità di due insiemi A e B è data dalla cardinalità della loro unione $A \cup B$. Analogamente, il prodotto della cardinalità di due insiemi A e B è data dalla cardinalità dell'insieme prodotto $A \times B$:

$$|A| + |B| = |A \cup B|$$
$$|A| \cdot |B| = |A \times B|$$

È importante osservare che coppie, triple, etc. di numeri sono anch'esse numerabili, ovvero vale il seguente risultato:

Teorema 2.6.
(1) $\aleph_0 + \aleph_0 = \aleph_0$
(2) $\aleph_0 \cdot \aleph_0 = \aleph_0$
(3) $\forall n \in \mathbb{N}.\ \aleph_0^n = \aleph_0$.

Proof. Dimostrare il punto (1) per esercizio. Per quanto riguarda il punto (2), consideriamo l'insieme $\mathbb{N} \times \mathbb{N}$. Per questo insieme vogliamo trovare una enumerazione, ovvero un insieme $S = \{\langle n_i, m_i\rangle : n_i, m_i, i \in \mathbb{N}\}$ tale che $S \approx \mathbb{N}$. Ordiniamo $\mathbb{N} \times \mathbb{N}$ nel modo seguente: Il primo elemento è la coppia $\langle n_0, m_0\rangle = \langle 0, 0\rangle$. Quindi ordiniamo le coppie $\langle n, m\rangle$ tali che $n_i + m_j = 1$ in base alla prima componente: $\langle n_1, m_1\rangle = \langle 0, 1\rangle$, $\langle n_2, m_2\rangle = \langle 1, 0\rangle$. Quindi in modo analogo ordiniamo le coppie $\langle n, m\rangle$ tali che $n_i + m_j = 2$: $\langle n_3, m_3\rangle = \langle 0, 2\rangle$, $\langle n_4, m_4\rangle = \langle 1, 1\rangle$, etc. È facile vedere che la funzione $\iota : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$ definita nel modo seguente: $\iota(i) = \langle n_i, m_i\rangle$ è un isomorfismo tra $\mathbb{N}$ e $\mathbb{N} \times \mathbb{N}$. La seguente funzione è l'inversa di ι [dimostrare per esercizio] ;

$$\delta(n, m) = n + \sum_{\ell=0}^{n+m+1} (\ell + 1) = n + \frac{(n+m)(n+m+1)}{2}$$

Il punto (3) segue per induzione dal punto (2). $\square$

Abbiamo visto che $\aleph_0$ corrisponde alla cardinalità degli insiemi enumerabili infiniti, ovvero dei numeri naturali. Inoltre, $\forall n \in \mathbb{N}.\ \aleph_0^n = \aleph_0$ e, per il Teorema di Cantor: $\aleph_0 = |\mathbb{N}| < |\wp(\mathbb{N})| = |\mathbb{R}|$. Ci chiediamo quindi se $|\wp(\mathbb{N})|$ è la *più piccola cardinalità non enumerabile*. Questa congettura è nota con il nome di *ipotesi del continuo*:

> *Ipotesi del continuo (CH)*: Ogni sottoinsieme di $\mathbb{R}$ è o numerabile o di cardinalità $|\wp(\mathbb{N})|$.

Assumendo valida l'ipotesi del continuo[2], è quindi possibile affermare che un insieme A è non-numerabile se $|A| \geq |\wp(\mathbb{N})| > \aleph_0$.

6. Ordinali

In questo paragrafo parleremo del concetto di (insieme) ordinale. Mentre i cardinali sono introdotti per misurare in qualche modo il numero di elementi di un dato insieme, gli ordinali sono introdotti per confrontare tra loro insiemi bene ordinati. Gli ordinali sono introdotti anche per poter definire un'aritmetica all'interno della teoria degli insiemi.

Un insieme x si dice *transitivo* se e solo se ogni elemento di x è sottoinsieme di x. In breve, se $\forall y \in x. \forall z \in y. z \in x$. Per esempio:

- $\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}$ sono transitivi.
- $\{\{\varnothing\}\}$ non è transitivo.
- Se Ω è un insieme tale che $\Omega = \{\Omega\}$, allora Ω è transitivo.

Si noti però che in quest'ultimo insieme l'appartenenza non è una relazione ben fondata su di esso (vale $\Omega \in \Omega$ che permette di generare catene discendenti infinite di appartenenza). Questo insieme è un insieme non ben fondato [**1**].

Un insieme x è un *ordinale* se x è transitivo e l'appartenenza è ben fondata su x (ovvero, $(x, \in)$ è ben fondato). Per esempio:

- $\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}$ sono ordinali.
- $\{\{\varnothing\}\}$ e Ω non sono ordinali.

TEOREMA 2.7 (Paradosso Burali-Forti). *Non esiste l'insieme* $\mathsf{ON} = \big\{x : x \text{ è un ordinale}\big\}$.

La dimostrazione di questo teorema è lasciata per esercizio. La parte tecnica è quella di dimostrare che se ON esistesse, allora sarebbe un ordinale. Ottenuto ciò il paradosso emerge immediatamente.

Dati due insiemi con una relazione (A, R) e (B, S), un *isomorfismo* tra (A, R) e (B, S) è una funzione biiettiva $f : A \longrightarrow B$ tale che $\forall x \in A. \forall y \in A. x \mathrel{R} y \leftrightarrow f(x) \mathrel{S} f(y)$. Se esiste un isomorfismo tra (A, R) e (B, S), diremo che sono isomorfi e lo denoteremo con $(A, R) \cong (B, S)$. Quando X è un ordinale, scriveremo semplicemente X in luogo di $(X, \in)$.

[2] L'indipendenza dell'ipotesi del continuo dalla formulazione classica della teoria degli insiemi ZF, è dovuta a Gödel e Cohen. Rimandiamo a [**16**] per maggiori dettagli su questa affascinante branca della moderna teoria degli insiemi.

Vale il seguente:

TEOREMA 2.8. *Se* (A, R) *è un buon ordine, allora esiste un unico ordinale* C *tale che* $(A, R) \cong C$.

Tale unico ordinale viene indicato come $\mathsf{type}(A, R)$ e vuole rappresentare una *misura* di quel buon ordine. Gli ordinali sono tutti basati su insiemi *puri* (ovvero $\varnothing, \{\varnothing, \{\varnothing\}\}$ etc.) e sulla relazione di appartenenza e sono confrontabili tra loro, come vedremo nel seguito. Il poter assegnare univocamente un ordinale ad ogni buon ordine permette pertanto di confrontare buoni ordini basati su insiemi qualunque (per esempio $\{\mathsf{mario}, \mathsf{luigi}, \dots\}$) e su relazioni d'ordine ben fondate qualunque (per esempio l'ordine lessicografico tra i codici fiscali).

Più precisamente, dati due ordinali α e β, diciamo che $\alpha < \beta$ se $\alpha \in \beta$; diciamo che $\alpha \leq \beta$ se $\alpha < \beta$ oppure $\alpha = \beta$.

Se X è un insieme di ordinali, definiamo come $\sup(X)$ l'unione unaria di X ovvero l'insieme $\{z : \exists y \in X. z \in y\}$. Si può dimostrare che $\sup(X)$ è il minimo ordinale maggiore o uguale a tutti gli elementi di X.

Se α è un ordinale, allora l'*ordinale successore di* α è definito come $S(\alpha) = \alpha \cup \{\alpha\}$. Per ogni ordinale α si ha che $S(\alpha)$ è un ordinale. β è detto un *ordinale successore* se esiste un ordinale α tale che $\beta = S(\alpha)$. α è invece un *ordinale limite* se è un ordinale diverso da $\varnothing$ e non è un *ordinale successore*.

A partire da queste nozioni siamo in grado di definire alcuni ordinali familiari:

$$
\begin{array}{rcl}
\underline{0} & \stackrel{\text{def}}{=} & \varnothing \\
\underline{1} & \stackrel{\text{def}}{=} & S(\underline{0}) = \varnothing \cup \{\varnothing\} = \{\varnothing\} = \{\underline{1}\} \\
\underline{2} & \stackrel{\text{def}}{=} & S(\underline{1}) = \{\varnothing\} \cup \{\{\varnothing\}\} = \{\varnothing, \{\varnothing\}\} = \{\underline{0}, \underline{1}\} \\
\underline{3} & \stackrel{\text{def}}{=} & S(\underline{2}) = \{\varnothing, \{\varnothing\}, \{\{\varnothing, \{\varnothing\}\}\} = \{\underline{0}, \underline{1}, \underline{2}\} \\
\vdots & \vdots & \vdots \\
\underline{n+1} & = & S(\underline{n}) = \underline{n} \cup \{\underline{n}\} = \{\underline{0}, \underline{1}, \dots, \underline{n}\}
\end{array}
$$

Questa successione corrisponde alla successione dei numeri naturali. Si osservi che la definizione di $<$ data sopra per gli ordinali è consistente con quella usuale sui numeri naturali. In teoria degli insiemi (p. es., [**18**]) si definiscono i *numeri naturali* esattamente in questo modo: un ordinale α è un numero naturale se $\forall \beta \leq \alpha. (\beta = \varnothing \vee \beta$ è un ordinale successore).

Per quanto riguarda gli ordinali limite, il primo ordinale limite è ω definito come $\omega = \sup\{\alpha : \alpha$ è un naturale$\}$ ovvero

$$\omega = \underline{0} \cup \underline{1} \cup \underline{2} \cup \cdots = \{\underline{0}, \underline{1}, \underline{2}, \dots\}$$

Abbiamo visto come gli insiemi ordinali permettano di definire in qualche modo i numeri naturali e la relazione d'ordine $<$. Vedremo ora alcune nozioni di aritmetica degli ordinali.

Dati due ordinali α e β l'ordinale *somma* $\alpha+\beta$ è l'ordinale

$$\mathsf{type}(\alpha \times \{\underline{0}\} \cup \beta \times \{\underline{1}\}, \mathbf{R})$$

dove $\mathbf{R}$ è la relazione:

$$\begin{aligned}\mathbf{R} \quad = \quad & \big\{\langle\langle x,\underline{0}\rangle,\langle y,\underline{0}\rangle\rangle \,:\, x<y<\alpha\big\}\cup\\ & \big\{\langle\langle x,\underline{1}\rangle,\langle y,\underline{1}\rangle\rangle \,:\, x<y<\beta\big\}\cup\\ & ((\alpha\times\{\underline{0}\})\times(\beta\times\{\underline{1}\}))\end{aligned}$$

A parole, gli elementi di α e β vengono messi in sequenza. Prima tutti quelli di α, poi quelli di β. Ad esempio, $\underline{2}+\underline{3}$ risulta il type dell'insieme costituito dalle coppie (ordinate da sinistra a destra):

$$\{\langle\underline{0},\underline{0}\rangle,\langle\underline{1},\underline{0}\rangle,\langle\underline{0},\underline{1}\rangle,\langle\underline{1},\underline{1}\rangle,\langle\underline{2},\underline{1}\rangle\}$$

ovvero all'ordinale $\underline{5}=\{\underline{0},\underline{1},\underline{2},\underline{3},\underline{4}\}$ ad esso isomorfo. Dunque, 2 + 3 fa proprio 5 come si sperava avvenisse.

Per l'aritmetica ordinale valgono le seguenti proprietà:

Teorema 2.9.
(1) $\alpha+(\beta+\gamma)=(\alpha+\beta)+\gamma$
(2) $\alpha+\underline{0}=\alpha$
(3) $\alpha+\underline{1}=\mathsf{S}(\alpha)$
(4) $\alpha+\mathsf{S}(\beta)=\mathsf{S}(\alpha+\beta)$
(5) *Se* β *è ordinale limite, allora* $\alpha+\beta=\sup\big\{\alpha+\xi \,:\, \xi<\beta\big\}$.

Si osservi però che il + non è in generale commutativo (lo è, per fortuna, sui numeri naturali). Studiamo infatti chi sono $1+\omega$ e $\omega+1$.

- $1+\omega$ è il type dell'insieme: $\{\underline{0}\}\times\{\underline{0}\}\cup\omega\times\{\underline{1}\}$ ovvero:

  $$\{\langle\underline{0},\underline{0}\rangle,\langle\underline{0},\underline{1}\rangle,\langle\underline{1},\underline{1}\rangle,\langle\underline{2},\underline{1}\rangle,\langle\underline{3},\underline{1}\rangle,\dots\}$$

  che altri non è se non ω stesso. Dunque $1+\omega=\omega$.
- $\omega+1$ è il type dell'insieme: $\omega\times\{\underline{0}\}\cup\{\underline{0}\}\times\{\underline{1}\}$ ovvero:

  $$\{\langle\underline{0},\underline{0}\rangle,\langle\underline{1},\underline{0}\rangle,\langle\underline{2},\underline{0}\rangle,\langle\underline{3},\underline{0}\rangle,\langle\underline{4},\underline{0}\rangle,\dots,\langle\underline{0},\underline{1}\rangle\}$$

  Si tratta di un ordine diverso da ω. Vi è prima una quantità numerabile di elementi ordinati come ω. Poi c'è un unico elemento che sta sopra tutti loro. L'ordinale corrispondente è $\mathsf{S}(\omega)=\omega\cup\{\omega\}$.

Studiamo ora la moltiplicazione tra ordinali. Dati due ordinali α e β l'ordinale *prodotto* $\alpha\cdot\beta$ è l'ordinale

$$\mathsf{type}(\beta\times\alpha,\mathbf{R})$$

dove $\mathbf{R}$ è la relazione d'ordine lessicografico su $\beta\times\alpha$, ovvero:

$$\langle x,y\rangle\,\mathbf{R}\,\langle x',y'\rangle \quad \mathit{sse} \quad x<x'\vee(x=x'\wedge y<y')$$

In parole povere, viene ricopiato α tante volte quante β. Ad esempio, $\underline{2} \cdot \underline{3}$ risulta il type dell'insieme costituito dalle coppie (ordinate da sinistra a destra):

$$\{\langle \underline{0}, \underline{0}\rangle, \langle \underline{0}, \underline{1}\rangle, \langle \underline{1}, \underline{0}\rangle, \langle \underline{1}, \underline{1}\rangle, \langle \underline{2}, \underline{0}\rangle, \langle \underline{2}, \underline{1}\rangle\}$$

ovvero all'ordinale $\underline{6} = \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}\}$ ad esso isomorfo. Dunque, $2 \cdot 3$ fa proprio 6 come si sperava avvenisse.

Per la moltiplicazione valgono le seguenti proprietà:

TEOREMA 2.10. (1) $\alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$
(2) $\alpha \cdot \underline{0} = \underline{0}$
(3) $\alpha \cdot \underline{1} = \alpha$
(4) $\alpha \cdot S(\beta) = \alpha \cdot \beta + \alpha$
(5) $\alpha \cdot (\beta + \gamma) = \alpha \cdot \beta + \alpha \cdot \gamma$
(6) *Se* β *è ordinale limite, allora* $\alpha \cdot \beta = \sup\{\alpha \cdot \xi \,:\, \xi < \beta\}$.

Anche in questo caso la commutatività vale solo per i naturali. Studiamo $2 \cdot \omega$ e $\omega \cdot 2$.

- $2 \cdot \omega$ è il type dell'insieme: $\omega \times \{\underline{0}, \underline{1}\}$ ordinato lessicograficamente, ovvero:
$$\{\langle \underline{0}, \underline{0}\rangle, \langle \underline{0}, \underline{1}\rangle, \langle \underline{1}, \underline{0}\rangle, \langle \underline{1}, \underline{1}\rangle, \langle \underline{2}, \underline{0}\rangle, \langle \underline{2}, \underline{1}\rangle, \dots\}$$
che altri non è se non ω stesso. Dunque $2 \cdot \omega = \omega$.
- $\omega \cdot 2$ è invece il type dell'insieme: $\{\underline{0}, \underline{1}\} \times \omega$ ovvero:
$$\{\langle \underline{0}, \underline{0}\rangle, \langle \underline{0}, \underline{1}\rangle, \langle \underline{0}, \underline{2}\rangle, \langle \underline{0}, \underline{3}\rangle, \langle \underline{0}, \underline{4}\rangle, \dots, \langle \underline{1}, \underline{0}\rangle, \langle \underline{1}, \underline{1}\rangle, \langle \underline{1}, \underline{2}\rangle, \langle \underline{1}, \underline{3}\rangle, \langle \underline{1}, \underline{4}\rangle, \dots,\}$$
Si tratta di un ordine diverso da ω. Vi è prima una quantità numerabile di elementi ordinati come ω. Poi c'è un un elemento che sta sopra tutti loro ($\langle \underline{1}, \underline{0}\rangle$) ed infine una quantità numerabile di elementi sopra ad esso. L'ordinale corrispondente è $\omega \cdot 2 = \sup\{\omega + \underline{n} \,:\, n < \omega\}$.

Non è da confondere la nozione di ordinale con quella di cardinalità. Ad esempio, è facile osservare in base al Teorema 2.6, che $|\omega \cdot 2| = |\omega + 1| = |\omega| = \aleph_0$. Tuttavia, vista la corrispondenza tra il primo ordinale limite infinito ω e l'insieme dei numeri naturali $\mathbb{N}$, faremo abuso di notazione identificando ω con $\aleph_0$. Pertanto, scriveremo $|A| = \omega$ per affermare che A è un insieme infinito enumerabile.

7. Il problema dell'informatica

Abbiamo visto come la cardinalità dell'insieme delle funzioni sui naturali sia strettamente superiore alla cardinalità dei numeri naturali stessi (Teorema 2.4). Già questo risultato permette di stabilire una prima distinzione tra ciò che possiamo formulare come funzione (problema) e ciò che possiamo dare come algoritmo (soluzione algoritmica del problema).

Se pensiamo che i programmi di un calcolatore siano tutti rappresentabili come sequenze finite di istruzioni date in un determinato linguaggio Σ, allora l'insieme dei programmi che possiamo scrivere è equipotente all'insieme dei numeri naturali.

Infatti, se Σ è l'alfabeto con cui possiamo scrivere i nostri programmi, allora un programma non è altro che una sequenza finita di simboli di Σ:

$$s_1 s_2 s_3 \dots s_n$$

con $n \in \mathbb{N}$. Gli algoritmi o programmi che possiamo descrivere sono quindi tanti quante le sequenze finite di simboli di Σ. Chiamiamo Alg l'insieme degli algoritmi che possiamo descrivere in un dato linguaggio. La seguente funzione stabilisce una biiezione tra algoritmi e numeri:

$$\pi : \mathsf{Alg} \longrightarrow \mathbb{N}$$

dove, se $P \in \mathsf{Alg}$ e $P = s_1 s_2 s_3 \dots s_n$, allora $\pi(P) = \sum_{i=1}^{n} p(s_i) \cdot 2^{n-i}$ dove $p(s_i)$ è il numero primo associato in modo univoco al simbolo s_i.

Se supponiamo inoltre che i dati manipolabili dai programmi siano anch'essi numerabili, ovvero siano numeri, o sequenze di simboli in un alfabeto Σ' (ad esempio immagini come sequenze di bytes che rappresentano il colore, la luminosità ed il contrasto per ogni pixel dell'immagine) allora un problema di tipo informatico è descrivibile come una funzione sui dati $\mathcal{D}$:

$$f : \mathcal{D} \longrightarrow \mathcal{D}$$

che associa ad ogni dato in input un dato in output. Come nel ragionamento sopra si ha che: $|\mathcal{D}| = |\mathbb{N}|$. Ne consegue dal Teorema di Cantor che l'insieme dei problemi è equipotente all'insieme delle funzioni $\mathcal{D} \longrightarrow \mathcal{D}$ che in cardinalità è strettamente maggiore dell'insieme di tutti i possibili programmi che possiamo scrivere in un dato alfabeto:

$$|\mathcal{D} \longrightarrow \mathcal{D}| > |\mathsf{Alg}|$$

In particolare, ed in modo un po' provocatorio [**22**], possiamo affermare che, come l'analisi, lo studio della ricorsività coincide con lo studio di $|\mathcal{D} \longrightarrow \mathcal{D}|$, ovvero di un insieme equipotente ai numeri reali. I successivi capitoli permetteranno quindi di appropriarci degli strumenti per analizzare tale insieme ed in particolare la relazione tra problemi e loro soluzioni algoritmiche (programmi).

Part 1

Linguaggi formali

CHAPTER 3

Automi a stati finiti

In questo capitolo sarà presentato il concetto di automa a stati finiti. Si partirà dal formalismo più semplice (automi a stati finiti deterministici) per poi presentare diverse tipologie di automa, che si dimostreranno essere tutte equivalenti dal punto di vista delle potenzialità del loro utilizzo, ovvero del riconoscimento di un dato linguaggio.

1. Alfabeti e Linguaggi

Un *simbolo* è un'entità primitiva astratta che non sarà definita formalmente (come punto, linea, etc.). Lettere e caratteri numerici sono esempi di simboli.

Una *stringa* (o *parola*) è una sequenza finita di simboli giustapposti (uno dietro l'altro). Ad esempio, se a, b, c sono simboli, $abcba$ è una stringa.

La *lunghezza* di una stringa w, denotata come $|w|$, è il numero di *occorrenze* di simboli che compongono una stringa. Ad esempio, $|abcba| = 5$. La stringa vuota, denotata con ε è la stringa costituita da zero simboli: $|\varepsilon| = 0$.

Sia $w = a_1 \cdots a_n$ una stringa. Ogni stringa della forma:

- $a_1 \cdots a_j$, con $j \in \{1, \dots, n\}$ è detta un *prefisso* di w;
- $a_i \cdots a_n$, con $i \in \{1, \dots, n\}$ è detta un *suffisso* di w;
- $a_i \cdots a_j$, con $i, j \in \{1, \dots, n\}$, $i \leq j$, è detta una *sottostringa* di w;
- ε è sia prefisso che suffisso che sottostringa di w.

Si osservi che se $n = 0$, allora $w = \varepsilon$. Dunque ε è sia prefisso che suffisso di ε. Ad esempio, i prefissi di abc sono ε, a, ab, e abc. I suffissi sono ε, c, bc, e abc. Le sottostringhe sono:

$$\begin{array}{cccc} \varepsilon & & & \\ & a & ab & abc \\ & & b & bc \\ & & & c \end{array}$$

Un prefisso, un suffisso o una sottostringa di una stringa, quando non sono la stringa stessa, sono detti *propri*.

Esercizio 3.1. Sia data una stringa w di lunghezza $|w| = n$. Quanti sono i suoi prefissi, i suoi suffissi e le sue sottostringhe?

La *concatenazione* di due stringhe v e w è la stringa vw che si ottiene facendo seguire alla prima la seconda. La concatenazione è una operazione (associativa) che ammette come identità la stringa vuota ε.

Un *alfabeto* Σ è un insieme finito di simboli. Un *linguaggio formale* (in breve linguaggio) è un insieme di stringhe di simboli da un alfabeto Σ. L'insieme vuoto $\emptyset$ e l'insieme $\{\varepsilon\}$ sono due linguaggi formali di qualunque alfabeto. Con Σ^* verrà denotato il linguaggio costituito da tutte le stringhe su un fissato alfabeto Σ. Dunque

$$\Sigma^* = \{a_1 \cdots a_n : n \geq 0, a_i \in \Sigma\}$$

Ad esempio, se $\Sigma = \{0\}$, allora $\Sigma^* = \{\varepsilon, 0, 00, 000, \dots\}$; se $\Sigma = \{0, 1\}$, allora $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$.

Esercizio 3.2. Si provi che se $\Sigma \neq \emptyset$, allora Σ^* è numerabile. Definendo inoltre:

$$\begin{cases} \Sigma^0 & = & \{\varepsilon\} \\ \Sigma^{n+1} & = & \{ax : a \in \Sigma, x \in \Sigma^n\} \end{cases}$$

si osservi che $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i$. Qual è la cardinalità di Σ^i ?

2. Automi

Un automa a stati finiti è un modello matematico di un sistema avente input, ed eventualmente output, a valori discreti. Il sistema può essere in uno stato tra un insieme finito di stati possibili. L'essere in uno stato gli permette di tener traccia della storia precedente. Un buon esempio di automa a stati finiti è costituito dall'ascensore: l'input è la sequenza di tasti premuti, mentre lo stato è il piano in cui si trova.

Un automa a stati finiti si può rappresentare mediante una testina che legge, spostandosi sempre nella stessa direzione, un nastro di lunghezza illimitata contenente dei simboli. La testina si può trovare in un certo *stato*; a seconda dello stato q e del simbolo s_i letto, la testina si porta in un altro stato (o rimane nello stesso) e si sposta a destra per apprestarsi a leggere il simbolo sucessivo:

$$\begin{array}{|c|c|c|c} \hline s_1 & s_2 & s_3 & \cdots \\ \hline \end{array}$$

$$\Uparrow$$

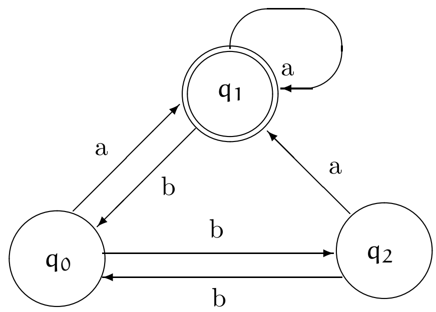
$$\boxed{q}$$

Quando la lettura dei simboli termina, a seconda dello stato raggiunto dalla testina, l'automa fornisce un risultato di accettazione o di refutazione della stringa (parola) letta.

Il comportamento dell'automa si definisce in maniera univoca mediante una tabella, detta *matrice di transizione*, come ad esempio:

	a	b
q_0	q_1	q_2
q_1	q_1	q_0
q_2	q_1	q_0

Un'automa siffatto si rappresenta bene anche con un grafo della forma seguente:



Nel prossimo paragrafo si fornirà una definizione formale del concetto ora esposto, necessaria per una trattazione precisa dell'argomento.

NOTA 3.3. Il fatto che, a differenza delle MdT (Capitolo 11.1), la testina non possa produrre degli output (eventualmente sul nastro) non è essenziale (si vedano, ad esempio, le macchine di Moore e di Mealy (Capitolo 8 e [**13**]). Intuitivamente, anche se scrivesse qualcosa sul nastro, non lo potrebbe più riutilizzare.
Neppure il permettere la bidirezionalità aumenterebbe di fatto la potenzialità del dispositivo (two-way automata [**13**]). Intuitivamente, la testina andrebbe su e giù ma sempre sugli stessi dati e con un controllo finito.
E' la somma delle due caratteristiche che permette di passare da questo formalismo al più potente formalismo di calcolo costituito dalla MdT.

3. Automi deterministici

Un *automa a stati finiti deterministico* (DFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F\rangle$ dove:

- Q è un insieme finito di *stati*;
- Σ è un alfabeto (alfabeto di input);
- $\delta : Q \times \Sigma \longrightarrow Q$ è la *funzione di transizione*;
- q_0 è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli *stati finali.*

NOTAZIONE 3.4. Useremo p, q, r con o senza pedici per denotare stati, P, Q, R, S per insiemi di stati, a, b con o senza pedici per denotare simboli di Σ, x, y, z, u, v, w sempre con o senza pedici per denotare stringhe.

Dalla funzione δ si ottiene in modo univoco la funzione $\hat{\delta} : Q \times \Sigma^* \longrightarrow Q$ nel modo seguente:

$$\begin{cases} \hat{\delta}(q, \varepsilon) & = & q \\ \hat{\delta}(q, wa) & = & \delta(\hat{\delta}(q, w), a) \end{cases}$$

Una stringa x è detta essere *accettata* da un DFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ se $\hat{\delta}(q_0, x) \in F$. Il *linguaggio accettato da* M, denotato come $L(M)$ è l'insieme delle stringhe accettate, ovvero:

$$L(M) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \in F\}.$$

Un linguaggio L è detto *regolare* se è accettato da qualche DFA, ovvero se esiste M tale che $L = L(M)$.

ESEMPIO 3.5. $\varnothing$ e Σ^* sono linguaggi regolari. Sia $\Sigma = \{s_1, \dots, s_n\}$: un automa M_0 che riconosce il linguaggio $\varnothing$ (ovvero: nessuna stringa è accettata) è il seguente:

	s_1	$\dots$	s_n
q_0	q_0	$\dots$	q_0

ove $F = \varnothing$. Infatti, poiché $\forall x\, x \notin \varnothing$, si ha che:

$$(\forall x \in \Sigma^*)(\hat{\delta}(q_0, x) \notin F).$$

Un automa per Σ^*, è invece l'automa M_1:

	s_1	$\dots$	s_n
q_0	q_0	$\dots$	q_0

ove $F = \{q_0\}$. Si dimostra facilmente infatti, per induzione su $|x|$ che

$$(\forall x \in \Sigma^*)(\hat{\delta}(q_0, x) = q_0).$$

ESERCIZIO 3.6. Si determini il linguaggio accettato dai seguenti automi rappresentati mediante la matrice di transizione (in tutti $F = \{q_1\}$).

(1)

	0	1
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_1	q_0

(2)

	0	1
q_0	q_2	q_0
q_1	q_2	q_0
q_2	q_1	q_2

Esercizio 3.7. Si verifichi che i seguenti linguaggi, con $\Sigma = \{0, 1\}$, sono regolari:

(1) l'insieme di tutte le stringhe aventi tre 0 consecutivi;
(2) l'insieme di tutte le stringhe tali che il penultimo simbolo è 0;
(3) l'insieme di tutte le stringhe tali che il terzultimo simbolo è 0;
(4) l'insieme di tutte le stringhe tali che, se interpretate come numero intero (binario), sono divisibili per 2;
(5) l'insieme di tutte le stringhe tali che, se interpretate come numero intero (binario), sono divisibili per 4;
(6) l'insieme di tutte le stringhe tali che, se interpretate come numero intero (binario), sono divisibili per 5.

Esercizio 3.8. Si dimostrino le seguenti proprietà

(1) Siano vx e wx due stringhe e $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un DFA. Allora, $\hat{\delta}(q_0, v) = \hat{\delta}(q_0, w)$ implica $\hat{\delta}(q_0, vx) = \hat{\delta}(q_0, wx)$.
(2) Siano u e v due stringhe e q uno stato. Allora $\hat{\delta}(q, uv) = \hat{\delta}(\hat{\delta}(q, u), v)$.
(3) Siano u e v due stringhe e q uno stato. Se $\hat{\delta}(q, u) = q_1$ e $\hat{\delta}(q, uv) = q_2$ allora $\hat{\delta}(q1, v) = q_2$.
(4) Sia $L \subseteq \Sigma^*$ un linguaggio accettato da un DFA M. Allora esiste $M' = \langle Q', \Sigma', \delta', q_0', F' \rangle$ tale che $\Sigma' = \Sigma$ e $L(M') = L(M)$.

4. Automi non-deterministici

Un *automa a stati finiti non-deterministico* (NFA) è una quintupla $\langle Q, \Sigma, \delta, q_0, F \rangle$ dove Q, Σ, q_0 e $F \subseteq Q$ mantengono il significato visto per gli automi deterministici, mentre la *funzione di transizione* δ è ora definita

$$\delta : Q \times \Sigma \longrightarrow \wp(Q)\,.$$

Si osservi che è ora ammesso: $\delta(q, a) = \varnothing$ per qualche $q \in Q$ ed $a \in \Sigma$. Anche per gli NFA dalla funzione δ si ottiene in modo univoco la funzione $\hat{\delta} : Q \times \Sigma^* \longrightarrow \wp(Q)$ nel modo seguente:

$$\begin{cases} \hat{\delta}(q, \varepsilon) & = & \{q\} \\ \hat{\delta}(q, wa) & = & \bigcup_{p \in \hat{\delta}(q, w)} \delta(p, a) \end{cases}$$

Una stringa x è *accettata* da un NFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ se $\hat{\delta}(q_0, x) \cap F \neq \varnothing$. Il *linguaggio accettato da* M è l'insieme delle stringhe accettate, ovvero:

$$L(M) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \cap F \neq \varnothing\}.$$

Mentre la rappresentazione mediante tabella degli automi non deterministici è profondamente diversa da quella per i deterministici (in ogni casella si deve inserire ora un insieme di stati), la rappresentazione a grafo rimane pressoché immutata. L'unica differenza è che da un nodo possono uscire più archi (o nessuno) etichettati dallo stesso simbolo. Dal punto di vista invece del modello con testina e nastro, il non-determinismo va immaginato come la possibilità *contemporanea* di procedere la computazione in ognuno degli stati raggiunti. Si apre dunque un gran numero di computazioni virtuali parallele.

5. Equivalenza tra DFA e NFA

In questo paragrafo si mostrerà che i linguaggi accettati dai DFA e dagli NFA coincidono. Poiché un DFA si può vedere come un NFA in cui $\delta(q, a)$ restituisce sempre insiemi costituiti da un solo stato (detti anche *singoletti*), si ha che ogni linguaggio regolare è un linguaggio accettato da un qualche NFA. Per l'implicazione inversa ci serve il seguente teorema:

TEOREMA 3.9 (Rabin-Scott, 1959). *Sia* $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ *un NFA. Allora esiste un DFA* M' *tale che* $L(M) = L(M')$.

PROOF. Definisco $M' = \langle Q', \Sigma', \delta', q_0', F'\rangle$ come segue:

- $\Sigma' = \Sigma$;
- $Q' = \wp(Q)$ (sarebbe più preciso definire $Q' = \{q_1, \dots, q_{2^{|Q|}}\}$ e poi stabilire una corrispondenza biunivoca fra tali stati e gli elementi di $\wp(Q)$. Tuttavia, con tale abuso sintattico la dimostrazione diventa molto più snella);
- $q_0' = \{q_0\}$;
- $F' = \{P \subseteq Q : P \cap F \neq \varnothing\}$;
- $\delta'(P, a) = \bigcup_{p \in P} \delta(p, a)$, per $P \in \wp(Q)$.

Mostriamo per induzione sulla lunghezza della stringa di input x che

$$\hat{\delta}(q_0, x) = \hat{\delta}'(q_0', x)$$

Base: Per $|x| = 0$ il risultato è banale, poiché $q_0' = \{q_0\}$ e $x = \varepsilon$.
Passo: Supponiamo che l'ipotesi induttiva valga per tutte le stringhe x tali che $|x| \leq m$. Sia xa una stringa di lunghezza $m + 1$. Allora:

$$\begin{array}{rcll} \hat{\delta}'(q_0', xa) & = & \delta'(\hat{\delta}'(q_0', x), a) & \text{Def. di } \hat{\cdot} \text{ nei DFA} \\ & = & \delta'(\hat{\delta}(q_0, x), a) & \text{Ip. ind.} \\ & = & \bigcup_{p \in \hat{\delta}(q_0, x)} \delta(p, a) & \text{Def. di } \delta' \\ & = & \hat{\delta}(q_0, xa) & \text{Def. di } \hat{\cdot} \text{ nei NFA} \end{array}$$

Il teorema segue dal fatto che:

$$\begin{array}{llll} x \in L(M) & \text{sse} & \hat{\delta}(q_0, x) \cap F \neq \varnothing & \text{def. di linguaggio NFA} \\ & \text{sse} & \hat{\delta}'(q_0', x) \cap F \neq \varnothing & \text{proprietà sopra} \\ & \text{sse} & \hat{\delta}'(q_0', x) \in F' & \text{def. di } F' \\ & \text{sse} & x \in L(M') & \text{def. di linguaggio DFA} \end{array}$$

□

Esercizio 3.10. Si determini il DFA equivalente al NFA:

	0	1
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_1\}$	$\{q_0, q_2\}$
q_2	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$

ove $F = \{q_2\}$. Qual è il linguaggio accettato?

Esercizio 3.11. Si descriva un NFA a 4 stati che riconosce il linguaggio delle stringhe di 0 e 1 con terzultimo elemento a 0. Si passi poi al DFA equivalente e lo si confronti con quello ottenuto nell'esercizio 3.7(3).

6. Automi con ε-transizioni

In questo paragrafo sarà presentato un terzo tipo di automa che estende il modello non-deterministico ma che, come sarà mostrato nel Teorema 3.12, ne è equivalente dal punto di vista dei linguaggi accettati.

Un *NFA con ε-transizioni* è una quintupla $\langle Q, \Sigma, \delta, q_0, F\rangle$ dove: Q, Σ, q_0 e $F \subseteq Q$ sono come per gli automi non deterministici, mentre la *funzione di transizione* δ è ora definita

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \wp(Q)\,.$$

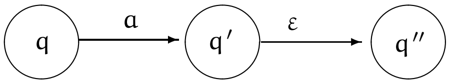
L'idea è che da uno stato è permesso passare ad un altro stato anche senza "leggere" caratteri di input.
La costruzione della funzione $\hat{\delta} : Q \times \Sigma^* \longrightarrow \wp(Q)$ nel caso dei ε-NFA risulta leggermente più complessa che nei casi precedenti. Per far ciò si introduce la funzione *ε-closure* che, applicata ad uno stato, restituisce l'insieme degli stati raggiungibili da esso (compreso sè stesso) mediante ε-transizioni. La costruzione di tale funzione è equivalente a quella che permette di conoscere i nodi raggiungibili da un nodo in un grafo e può facilmente essere calcolata a partire dalla funzione δ (un arco $p \to q$ si ha quando $q \in \delta(p, \varepsilon)$). Il concetto di ε-closure si estende in modo intuitivo ad insiemi di stati:

$$\varepsilon\text{-closure}(P) = \bigcup_{p \in P} \varepsilon\text{-closure}(p)$$

$\hat{\delta}$ si può ora definire nel modo seguente:

$$\begin{cases} \hat{\delta}(q, \varepsilon) & = & \varepsilon\text{-closure}(q) \\ \hat{\delta}(q, wa) & = & \bigcup_{p \in \hat{\delta}(q,w)} \varepsilon\text{-closure}(\delta(p, a)) \end{cases}$$

Si noti che in questo caso $\hat{\delta}(q, a)$ può essere diverso da $\delta(q, a)$. Ad esempio, nell'automa:



Si ha che $\delta(q, a) = \{q'\}$, mentre $\hat{\delta}(q, a) = \bigcup_{p \in \hat{\delta}(q,\varepsilon)} \varepsilon\text{-closure}(\delta(p, a)) = \{q', q''\}$.

Definiamo dunque il *linguaggio accettato* dall'automa $L(M) = \{x \in \Sigma^* : \hat{\delta}(q_0, x) \cap F \neq \varnothing\}$. Si osservi come per questa classe di linguaggi si potrebbe assumere che l'insieme F abbia esattamente un elemento.

7. Equivalenza di ε-NFA e NFA

Ogni NFA è, per definizione, un caso particolare di un ε-NFA. Con il seguente teorema mostreremo che la classe dei linguaggi riconosciuti dagli automi ε-NFA non estende propriamente quella dei linguaggi riconosciuti da automi NFA:

TEOREMA 3.12. *Sia* $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ *un ε-NFA. Allora esiste un NFA* M' *tale che* $L(M) = L(M')$.

PROOF. Definisco $M' = \langle Q', \Sigma', \delta', q'_0, F'\rangle$ come segue:

- $Q' = Q$,
- $\Sigma' = \Sigma$,
- $q'_0 = q_0$,
- $F' = \begin{cases} F \cup \{q_0\} & \text{Se } \varepsilon\text{-closure}(q_0) \cap F \neq \varnothing \\ F & \text{altrimenti} \end{cases}$
- $\delta'(q, a) = \hat{\delta}(q, a)$.

Dobbiamo mostrare che

$$\hat{\delta}(q_0, x) \cap F \neq \varnothing \quad \text{sse} \quad \hat{\delta}'(q'_0, x) \cap F' \neq \varnothing \quad (1)$$

Se $x = \varepsilon$, si ha che: $\hat{\delta}(q_0, \varepsilon) = \varepsilon\text{-closure}(q_0)$ per definizione.
D'altro canto: $\hat{\delta}'(q_0, \varepsilon) = \{q_0\}$.
Se $\varepsilon\text{-closure}(q_0) \cap F \neq \varnothing$ allora, per def. di F' vale che $q_0 \in F'$; dunque $\{q_0\} \cap F' \neq \varnothing$.
Sia ora $\{q_0\} \cap F' \neq \varnothing$. Allora $q_0 \in F'$. Due casi sono possibili: se $q_0 \in F$ allora $\varepsilon\text{-closure}(q_0) \cap F \neq \varnothing$ (in quanto $q_0 \in \varepsilon\text{-closure}(q_0)$). Altrimenti, se $q_0 \in F' \setminus F$ per definizione di F' si ha che $\varepsilon\text{-closure}(q_0) \cap F \neq \varnothing$.

Mostreremo per induzione su $|x| \geq 1$ che $\hat{\delta}'(q'_0, x) = \hat{\delta}(q_0, x)$.

Base: $|x| = 1$. Allora $x = a$ per qualche simbolo $a \in \Sigma$. Ma allora $\hat{\delta}'(q'_0, a) = \delta'(q'_0, a) = \hat{\delta}(q_0, a)$ per definizione.
Passo: Assumiamo che la tesi valga per tutte le stringhe x tali che $1 \leq |x| \leq m$. Sia xa una stringa di lunghezza $m + 1$. Allora:

$$\begin{array}{rcll} \hat{\delta}'(q'_0, xa) & = & \bigcup_{p \in \hat{\delta}'(q'_0, x)} \delta'(p, a) & \text{def. di } \hat{\delta}' \\ & = & \bigcup_{p \in \hat{\delta}'(q'_0, x)} \hat{\delta}(p, a) & \text{def. di } \delta' \\ & = & \bigcup_{p \in \hat{\delta}(q_0, x)} \hat{\delta}(p, a) & \text{ip. ind.} \\ & = & \bigcup_{p \in \hat{\delta}(q_0, x)} \varepsilon\text{-closure}(\delta(p, a)) & \text{def. di } \hat{\delta} \text{ e } (*) \\ & = & \hat{\delta}(q_0, xa) & \text{def. di } \hat{\delta} \end{array}$$

$(*)$ si osservi che per definizione di $\hat{\delta}$, la ε-closure(p) è contenuta in $\hat{\delta}(q_0, x)$.

Per concludere la dimostrazione, si deve mostrare la proprietà (1), ovvero che $\hat{\delta}'(q'_0, x)$ contiene uno stato di F' sse $\hat{\delta}(q_0, x)$ contiene uno stato di F.
Poiché $F \subseteq F'$ e $\hat{\delta}'(q'_0, x) = \hat{\delta}(q_0, x)$ l'implicazione $(\rightarrow)$ deriva immediatamente.
Per l'altra implicazione, l'unico problema si avrebbe nel caso:

(1) $q_0 \in \hat{\delta}'(q'_0, x)$,
(2) $q'_0 \in F'$,
(3) $q_0 \notin F$.

Poiché $\hat{\delta}'(q'_0, x) = \hat{\delta}(q_0, x)$, si ha che $q_0 \in \hat{\delta}(q_0, x)$. Ma, per definizione di $\hat{\delta}$, anche ogni elemento della sua ε-closure appartiene a $\hat{\delta}(q_0, x)$. Tale ε-closure, poiché $q_0 \in F'$, interseca F. □

Corollario 3.13. Le classi di linguaggi riconosciute da DFA, NFA e ε-NFA coincidono (linguaggi regolari).

Proof. Immediato dai Teoremi 3.9 e 3.12. □

8. Automi con output

Gli automi visti fin qui, tutti equivalenti dal punto di vista della classe di linguaggi accettati (riconosciuti) non sono in grado di fornire alcun tipo di output se non il messaggio (booleano) del raggiungimento di uno stato finale o meno. Esistono in letteratura due tipi di automi provvisti di output: le macchine di *Moore* e quelle di *Mealy*. Tali *macchine* sono automi a stati finiti deterministici (e dunque ereditano la classe di linguaggi accettati) che possiedono un alfabeto di output:

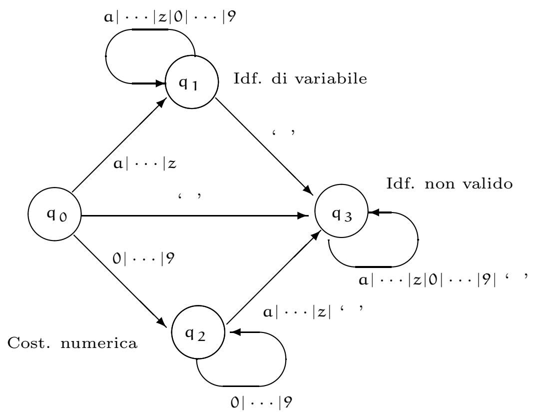
- le macchine di Moore forniscono in output un simbolo in funzione di ogni stato raggiunto. Poiché per una stringa di n elementi si raggiungono al più $n + 1$ stati (compreso quello iniziale), si può ottenere un output di (al più) $n + 1$ caratteri.

- le macchine di Mealy forniscono in output un simbolo ogni volta che avviene una transizione $\delta(q, a)$. In questo caso al più n caratteri saranno forniti in output.

Gli output possono essere utili per differenziare stati finali. Così, ad esempio, una stringa accettata può anche avere una più precisa classificazione (si veda l'Esempio 3.14).

Le macchine di Moore e Mealy vengono impiegate per la sintesi di reti sequenziali e sono formalismi tra loro equivalenti. Per maggiori dettagli, si veda [**13**].

Esempio 3.14. Il seguente automa è una macchina di Moore che, oltre a riconoscere se una stringa sia o meno un elemento di una espressione numerica, segnala se si tratta di un numero intero o di un identificatore di variabile. L'output è rappresentato dalle scritte (Idf. di variabile, etc.) posizionate accanto ai vari stati.



Si osservi come esso racchiuda in sè due DFA privi di output.

CHAPTER 4

Espressioni regolari

In questo capitolo verrà illustrato un modo alternativo per descrivere i linguaggi accettati da automi finiti (linguaggi regolari). definiremo un algebra di operazioni per manipolare linguaggi regolari.

1. Operazioni sui linguaggi

Sia Σ un alfabeto e L, L_1, L_2 insiemi di stringhe di Σ^*. La *concatenazione di* L_1 *e* L_2, denotata come L_1L_2 è l'insieme:

$$L_1L_2 = \{xy \in \Sigma^* : x \in L_1, y \in L_2\}.$$

Definiamo ora

$$\begin{cases} L^0 & = & \{\varepsilon\} \\ L^{i+1} & = & LL^i \end{cases}$$

Si osservi che $L^0 = \{\varepsilon\} \neq \varnothing$. La *chiusura (di Kleene) di* L, denotata come L^* è l'insieme:

$$L^* = \bigcup_{i \geq 0} L^i$$

mentre la *chiusura positiva di* L, denotata come L^+ è l'insieme: $L^+ = \bigcup_{i \geq 1} L^i$ (si verifica immediatamente che $L^+ = LL^*$, dunque tale operatore, seppur comodo, non è essenziale). Si osservi come la definizione sia consistente con la nozione di Σ^* fin qui adoperata.

2. Definizione formale

Sia Σ un alfabeto. Le *espressioni regolari su* Σ e gli *insiemi* che esse denotano sono definiti ricorsivamente nel modo seguente:

(1) $\varnothing$ è una espressione regolare che denota l'insieme vuoto.
(2) ε è una espressione regolare che denota l'insieme $\{\varepsilon\}$.
(3) Per ogni simbolo $a \in \Sigma$, a è una espressione regolare che denota l'insieme $\{a\}$.
(4) Se r e s sono espressioni regolari denotanti rispettivamente gli insiemi R ed S, allora $(r + s)$, (rs), e (r^*) sono espressioni regolari che denotano gli insiemi $R \cup S$, RS, e R^* rispettivamente.

Se r è una espressione regolare, indicheremo con $\mathsf{L}(\mathsf{r})$ il linguaggio denotato da r. Tra espressioni regolari valgono delle uguaglianze che permettono la loro manipolazione algebrica. Varrà che $\mathsf{r} = \mathsf{s}$ se e solo se $\mathsf{L}(\mathsf{r}) = \mathsf{L}(\mathsf{s})$.

NOTAZIONE 4.1. Qualora non si crei ambiguità, nello scrivere espressioni regolari saranno omesse le parentesi. Le precedenze degli operatori sono le stesse dell'algebra, interpretando * come esponente e la concatenazione come prodotto (si veda Esercizio 4.2). E' ammesso usare l'abbreviazione r^+ per l'espressione regolare $\mathsf{r}\mathsf{r}^*$.

ESERCIZIO 4.2. Si provino o refutino le seguenti identità tra espressioni regolari:

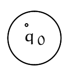
(1) $\mathsf{r} + \mathsf{s} = \mathsf{s} + \mathsf{r}$;
(2) $(\mathsf{r} + \mathsf{s}) + \mathsf{t} = \mathsf{r} + (\mathsf{s} + \mathsf{t})$;
(3) $\mathsf{r}(\mathsf{s}\mathsf{t}) = (\mathsf{r}\mathsf{s})\mathsf{t}$;
(4) $\mathsf{r}(\mathsf{s} + \mathsf{t}) = \mathsf{r}\mathsf{s} + \mathsf{r}\mathsf{t}$;
(5) $(\mathsf{r} + \mathsf{s})\mathsf{t} = \mathsf{r}\mathsf{t} + \mathsf{s}\mathsf{t}$;
(6) $\varnothing^* = \varepsilon$;
(7) $(\mathsf{r}^*)^* = \mathsf{r}^*$;
(8) $(\varepsilon + \mathsf{r}^*\mathsf{r}) = \mathsf{r}^*$;
(9) $(\varepsilon + \mathsf{r})^* = \mathsf{r}^*$;
(10) $(\mathsf{r}^*\mathsf{s}^*)^* = (\mathsf{r} + \mathsf{s})^*$;
(11) $(\mathsf{r}\mathsf{s} + \mathsf{r})^*\mathsf{r} = \mathsf{r}(\mathsf{s}\mathsf{r} + \mathsf{r})^*$;
(12) $\mathsf{s}(\mathsf{r}\mathsf{s} + \mathsf{s})^*\mathsf{r} = \mathsf{r}\mathsf{r}^*\mathsf{s}(\mathsf{r}\mathsf{r}^*\mathsf{s})^*$;
(13) $(\mathsf{r} + \mathsf{s})^* = \mathsf{r}^* + \mathsf{s}^*$;
(14) $(\mathsf{r}^* + \mathsf{s}^*)^* = (\mathsf{r}\mathsf{s})^*$.

3. Equivalenza tra DFA e ER

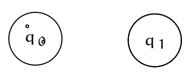
TEOREMA 4.3 (McNaughton & Yamada, 1960). *Sia* r *una espressione regolare. Allora esiste un* ε*-NFA* M *tale che* $\mathsf{L}(\mathsf{M}) = \mathsf{L}(\mathsf{r})$.

PROOF. Costruiremo un ε-NFA siffatto, con un unico stato finale, per induzione sulla complessità strutturale dell'espressione regolare r.
Base: Ci sono tre casi base:

- l'automa:

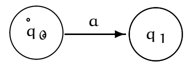


  riconosce il linguaggio $\{\varepsilon\}$;
- l'automa



  riconosce il linguaggio $\varnothing$;

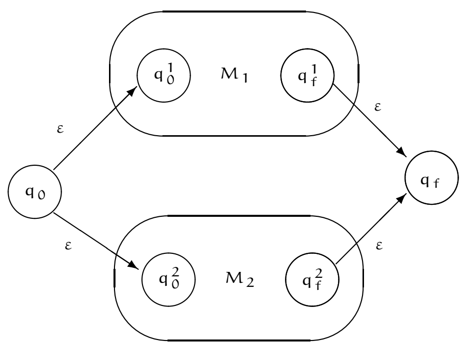
- l'automa



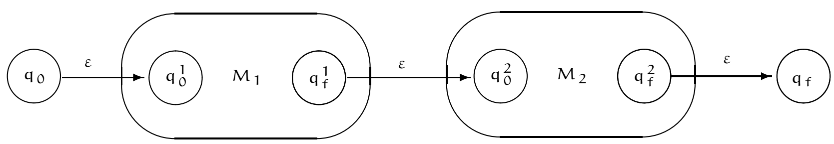
riconosce il linguaggio $\{a\}$.

Passo: Anche qui abbiamo tre casi da analizzare:

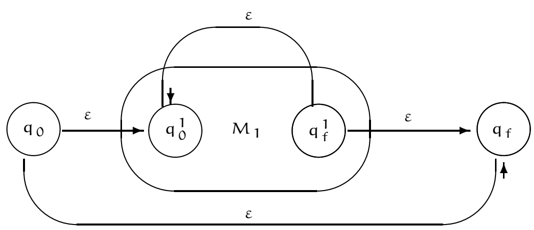
- $r = r_1 + r_2$. Per $i = 1, 2$, sia M_i, con stato iniziale q_0^i e stato finale q_f^i l'automa che riconosce $L(r_i)$. L'esistenza di tali automi è assicurata dall'ipotesi induttiva. Il seguente automa, con stato iniziale q_0 e stato finale q_f riconosce il linguaggio $L(r) = L(r_1) \cup L(r_2)$:



- $r = r_1 r_2$. Per $i = 1, 2$, sia M_i, con stato iniziale q_0^i e stato finale q_f^i l'automa che riconosce $L(r_i)$. L'esistenza di tali automi è assicurata dall'ipotesi induttiva. Il seguente automa, con stato iniziale q_0 e stato finale q_f riconosce il linguaggio $L(r_1)L(r_2)$:



- $r = r_1^*$. Sia M_1, con stato iniziale q_0^1 e stato finale q_f^1 l'automa che riconosce $L(r_1)$. L'esistenza di tale automa è assicurata dall'ipotesi induttiva. Il seguente automa, con stato iniziale q_0 e stato finale q_f riconosce $L(r) = (L(r_1))^*$:



Le dimostrazioni che tali automi riconoscono esattamente i linguaggi a loro assegnati è lasciata per esercizio. □

ESERCIZIO 4.4. Si costruisca l'ε-NFA per l'espressione regolare: $(0+1)^*0(0+1)(0+1)$. Si passi poi automaticamente al NFA e quindi al DFA equivalente. Lo si confronti dunque con l'automa dell'esercizio 3.11.

Si vuole ore mostrare il contrario, ovvero che preso a caso un automa M (senza perdita di generalità è sufficiente considerare un DFA), il linguaggio accettato da M è denotato da una espressione regolare. Si darà una dimostrazione costruttiva di questo fatto. Per una dimostrazione alternativa (e più rigorosa), si veda [**13**].

TEOREMA 4.5. *Sia* M *un DFA. Allora esiste una espressione regolare* r *tale che* $L(M) = L(r)$.

PROOF. (Traccia) Si costruirà, a partire da $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ un sistema di equazioni in cui ad ogni stato q_i viene associata una variabile Q_i. Si fornirà una metodologia per calcolare una soluzione a tale sistema. La soluzione calcolata assegnerà una espressione regolare r_i ad ogni variabile Q_i, che ne denota il linguaggio. Se $F = \{q_{j_1}, \ldots, q_{j_p}\}$, avremo che:

$$r = r_{j_1} + \cdots + r_{j_p}$$

e $L(r)$ è il linguaggio accettato dall'automa.
Per ogni $q_i \in Q$, siano:

- $q_{i_1}, \ldots, q_{i_h}$ gli stati diversi da q_i tali che esiste $a_{i_j} \in \Sigma$ t.c. $\delta(q_{i_j}, a_{i_j}) = q_i$;
- $b_{i_1}, \ldots, b_{i_k}$ i simboli tali che $\delta(q_i, b_{i_j}) = q_i$.

Assegnamo a q_i l'equazione:

$$Q_i \quad = \quad (Q_{i_1} a_{i_1} + \cdots + Q_{i_h} a_{i_h})(b_{i_1} + \cdots + b_{i_k})^*$$

Nel caso che q_i fosse q_0 bisogna aggiungere $+\varepsilon$ alla parte destra.
Un tale sistema si può risolvere mediante manipolazioni algebriche sulle espressioni regolari presenti ed usando il metodo di *sostituzione* (standard) qualora (come accade all'inizio) Q_i non compaia nella parte destra dell'equazione che lo definisce. Se non fosse così, poiché le variabili Q_j compaiono sempre all'inizio delle espressioni presenti nelle parti destre delle equazioni, ci si riesce a ricondurre, mediante

manipolazioni algebriche, ad una equazione della forma:

$$Q_i = Q_i s_1 + s_2$$

per opportuni s_1 ed s_2 in cui Q_i non compare. La soluzione di questa equazione è $Q_i = s_2 s_1^*$; infatti:

$$(s_2 s_1^*) s_1 + s_2 = s_2 s_1^+ + s_2 = s_2(s_1^+ + \varepsilon) = s_2 s_1^* .$$

Si applichi dunque la sostituzione $Q_i = s_2 s_1^*$ al resto del sistema. □

ESERCIZIO 4.6. Mediante la costruzione del Teorema 4.3 si identifichi un ε-NFA che riconosca il linguaggio denotato dall'espressione regolare $((0+1)^*0011(0+1)^*)^*$. Si passi poi, mediante le trasformazioni suggerite nei Teoremi 3.12 e 3.9 al corrispondente DFA. Di quanti stati si ha bisogno? Si cerchi di costruire direttamente un DFA che riconosca lo stesso linguaggio.

ESERCIZIO 4.7. Si determinino le espressioni regolari associate ai DFA finora presentati nel testo o descritti per risolvere gli esercizi.

CHAPTER 5

Proprietà dei linguaggi regolari

Dai Capitoli 2 e 4 si è imparato che se un linguaggio è riconosciuto da un automa a stati finiti (equivalentemente, DFA, NFA, ε-NFA) allora è un linguaggio regolare. Argomenti di questo capitolo saranno i principali risultati e metodi utili per stabilire se un dato linguaggio sia o meno regolare.

1. Il "Pumping Lemma"

Il primo metodo che proponiamo e che viene spesso usato per mostrare che un linguaggio non è regolare deriva dal seguente risultato noto come *Pumping Lemma*:

LEMMA 5.1 (Bar-Hillel, Perles, Shamir, 1961). *Sia* L *un linguaggio regolare. Allora esiste una costante* $n \in \mathbb{N}$ *tale che per ogni* $z \in L$ *tale che* $|z| \geq n$ *esistono tre stringhe* u, v, w *tali che:*

(1) $z = uvw$,
(2) $|uv| \leq n$,
(3) $|v| > 0$, *e*
(4) *per ogni* $i \geq 0$ *vale che* $uv^iw \in L$.

PROOF. Sia $M = \langle \{q_0, \ldots, q_{n-1}\}, \Sigma, \delta, q_0, F \rangle$ DFA tale che $L = L(M)$ (esiste poiché L è regolare). Sia $z = a_1 \cdots a_m$, $m \geq n$, $z \in L$ (si noti l'arbitrarietà della scelta di z; se una tale z non esistesse, il lemma varrebbe banalmente). Per $i = 1, \ldots, n$ ($n \leq m$) si consideri l'evoluzione degli stati $\hat{\delta}(q_0, a_1 \cdots a_i)$. In tal modo, considerando anche lo stato iniziale q_0, si raggiungono in tutto $n + 1$ stati. Poiché gli stati dell'automa sono n per ipotesi, esiste (almeno) uno stato $\bar{q}$ raggiunto (almeno) due volte.

Dunque avremo una situazione del tipo

$$\begin{aligned} \hat{\delta}(q_0, a_1 \cdots a_{i_1}) &= \bar{q} \\ &= \hat{\delta}(q_0, a_1 \cdots a_{i_1} \cdots a_{i_2}) \quad i_2 > i_1 \end{aligned}$$

A questo punto, si prenda $u = a_1 \cdots a_{i_1}$, $v = a_{i_1+1} \cdots a_{i_2}$, $w = a_{i_2+1} \cdots a_m$.

Sappiamo dunque che $\hat{\delta}(q_0, u) = \bar{q}$ e $\hat{\delta}(q_0, uv) = \bar{q}$. Usando l'esercizio 3.8 si ha che $\hat{\delta}(\bar{q}, v) = \bar{q}$.

Inoltre, sappiamo per ipotesi che $\hat{\delta}(q_0, uvw) = q'$ per qualche $q' \in F$; assieme al fatto che $\hat{\delta}(q_0, uv) = \bar{q}$, usando l'esercizio 3.8 si ha che $\hat{\delta}(\bar{q}, w) = q'$.

Per induzione su $n \geq 0$ si mostra che $\hat{\delta}(q_0, uv^i) = \bar{q}$. Con l'ultima proprietà vista, si ha dunque che $\hat{\delta}(q_0, uvw^i) = q' \in F$.

$|uv| \leq n$ per costruzione. □

COROLLARIO 5.2. n del Lemma 5.1 può essere preso come il numero di stati del più piccolo automa riconoscente L.

PROOF. Immediato dalla dim. del Lemma. □

Si osservi come il lemma asserisca la veridicità di una formula logica quantificata non banale, ovvero, per ogni linguaggio L, se L è regolare, allora vale:

$$\exists n \in \mathbb{N}\ \forall z \left(\begin{array}{l} (z \in L \land |z| \geq n) \to \exists u, v, w \\ \left(\begin{array}{l} z = uvw \land |uv| \leq n \land |v| > 0 \land \\ \forall i\, (i \in \mathbb{N} \to uv^i w \in L) \end{array} \right) \end{array} \right) \tag{1.1}$$

Il Pumping lemma viene largamente usato per mostrare che un dato linguaggio non è regolare (qualora non lo sia!). Per mostrare ciò, si assume che L non sia regolare e si deve mostrare che vale la negazione della formula (1.1), ovvero (si veda il Capitolo 4 per le tecniche di complementazione di una formula logica):

$$\forall n \in \mathbb{N}\ \exists z \left(\begin{array}{l} z \in L \land |z| \geq n \land \\ \forall u, v, w\, (z = uvw \land |uv| \leq n \land |v| > 0 \\ \qquad \to \exists i\, (i \in \mathbb{N} \land uv^i w \notin L) \end{array} \right) \tag{1.2}$$

Nel prossimo esempio si illustrerà un corretto uso di questa tecnica, che dato l'annidamento di quantificazioni, può indurre ad errore.

ESEMPIO 5.3. Il linguaggio $L = \{0^i 1^i : i \geq 0\}$ non è regolare. Supponiamo, per assurdo, sia regolare e cerchiamo di verificare la formula (1.2). Si prenda un numero naturale n arbitrario e si scelga (questo è il punto in cui bisogna avere la "fortuna" di scegliere una stringa "giusta" tra tutte quelle di lunghezza maggiore o uguale a n) la stringa $z = 0^n 1^n$. Per tale stringa vale che $z \in L$ e $|z| \geq n$. Rimane da dimostrare che comunque si suddivida la stringa z in sottostringhe uvw (in modo tale cioè che $z = uvw$) tali da soddisfare i vincoli $|uv| \leq n$ e $|v| > 0$ esiste almeno un $i \geq 0$ tale per cui $uv^i w \notin L$.

I modi per partizionare z sono molti (per esercizio, si provi a pensare quanti) però, il fatto che $|uv| \leq n$ permette di ricondursi al seguente unico "schema di suddivisione":

- $u = 0^a$,
- $v = 0^b$,
- $w = 0^c 1^n$,

con $a + b + c = n$, $b > 0$. "Pompando" v si ottengono stringhe al di fuori del linguaggio, da cui l'assurdo. Ad esempio, con $i = 0$ vale che $uv^0 w = 0^a 0^c 1^n \notin L$, poiché $a + c < n$.

Esercizio 5.4. Dimostrare che i seguenti linguaggi non sono regolari:
(1) $\{0^{2^n} : n \geq 1\}$;
(2) $\{0^n : n$ è primo$\}$;
(3) $\{0^m 1^{m+1} 0^{m+2} : m \geq 0\}$;
(4) $\{0^m 1^n 0^{m+n} : m \geq 1, n \geq 1\}$;
(5) $\{0^m 1^n : m, n \geq 0, m < n\}$.
(6) $\{x \in \{0,1\}^* : \#$ di 0 in $x = \#$ di 1 in $x\}$

2. Proprietà di chiusura

La definizione di linguaggio è una definizione di tipo insiemistico ($L \subseteq \Sigma^*$). Pertanto è lecito parlare di unione, intersezione, complementazione (etc.) di linguaggi. In questo paragrafo mostreremo che la proprietà di essere un linguaggio regolare si conserva sotto queste usuali operazioni.

Teorema 5.5. *I linguaggi regolari sono chiusi rispetto alle operazioni di unione, concatenazione e chiusura di Kleene.*

Proof. Immediato dalla definizione di espressione regolare e dai Teoremi 4.3 e 4.5. □

Teorema 5.6. *I linguaggi regolari sono chiusi rispetto alla operazione di complementazione. Ovvero, se $L \subseteq \Sigma^*$ è regolare, anche $\bar{L} = \Sigma^* \setminus L$ è regolare.*

Proof. Sia $M = \langle Q, \Sigma', \delta, q_0, F\rangle$ il DFA che riconosce L. Per l'esercizio 3.8.4, possiamo assumere $\Sigma' = \Sigma$. Allora, banalmente, $M' = \langle Q, \Sigma, \delta, q_0, Q \setminus F\rangle$ riconosce $\bar{L}$. □

Corollario 5.7. I linguaggi regolari sono chiusi rispetto all'intersezione.

Proof. Immediato dal fatto che $L_1 \cap L_2 = \overline{(\bar{L}_1 \cup \bar{L}_2)}$. □

Esercizio 5.8. Dati due DFA M_1 e M_2, si costruisca l'automa che riconosce $M_1 \cap M_2$ in maniera diretta. Di quanti stati avete bisogno?

Esercizio 5.9. Si dimostri che l'insieme delle stringhe di 0 e 1 tali che:

- non vi sono mai 3 "0" consecutivi e non vi sono mai tre "1" consecutivi, oppure
- vi sono almeno 2 "0" seguiti da 2 "1"

è un linguaggio regolare (Suggerimento: si suddivida il problema in problemi elementari e si usino i risultati di questo paragrafo per combinarli).

Esercizio 5.10. Si dimostri che il linguaggio composto da stringhe di 0 e 1 tale che:

- contengono almeno 5 "0" consecuitivi, e
- se vi sono 3 "1" consecutivi, allora essi sono seguiti da almeno due "0"

è regolare.

3. Risultati di decidibilità

In questo paragrafo saranno mostrati alcuni risultati relativi al problema di effettuare delle decisioni riguardanti linguaggi regolari. Un primo risultato scontato in questa classe di linguaggi (meno in altre!) è la decidibilità del *problema dell'appartenenza*, ovvero: data una descrizione del linguaggio L (mediante e.r. o automa) e una stringa x, decidere se $x \in L$ o meno. Se è data l'e.r., possiamo prima calcolare l'automa. Dato l'automa il problema diventa banale: tale problema è evidentemente decidibile. Vediamo ora altri problemi.

TEOREMA 5.11 (Vuoto-infinito). *L'insieme delle stringhe accettate da un DFA M con n stati è:*

(1) *non vuoto se e solo se accetta una stringa di lunghezza inferiore a n;*
(2) *infinito se e solo se l'automa accetta una stringa di lunghezza ℓ, $n \leq \ell < 2n$.*

PROOF. (1) ($\leftarrow$) Se accetta una stringa (di lunghezza di inferiore a n) allora è banalmente non vuoto.
($\rightarrow$) Supponiamo sia non vuoto. Allora, per definizione M accetta almeno una stringa z, $|z| = m$.
Se $m < n$, la tesi è provata. Altrimenti come conseguenza del il pumping lemma, $z = uvw$ con $|v| \geq 1$, e $uv^0w = uw$ è accettata da M. Se $|uw| < n$ la tesi è provata, altrimenti si proceda iterativamente ripartendo con $z = uw$ (che ha lunghezza strettamente minore a m). Dopo al più $m - n$ iterazioni si otterrà una stringa di lunghezza inferiore a n;
(2) ($\leftarrow$) Supponiamo M accetti una stringa z di lunghezza ℓ, $n \leq \ell < 2n$. Allora, per il pumping Lemma, $z = uvw$, $|v| \geq 1$ e $\{uv^iw : i \in \mathbb{N}\} \subseteq L(M)$. Ma $\{uv^iw : i \in \mathbb{N}\}$ è un insieme infinito.
($\rightarrow$) Sia $L(M)$ infinito. Allora esiste $z \in L(M)$ tale che $|z| = m \geq 2n$. Per il pumping lemma $z = uvw$ con $|uv| \leq n$ e $|v| \geq 1$ (e dunque $|uw| \geq n$). Per il pumping lemma, $z' = uw \in L(M)$. Se $|z'| < 2n$, allora la tesi è dimostrata. Altrimenti, si reiteri il procedimento partendo dalla stringa (più corta) $z' = uw$. In un numero finito di passi si trova la stringa cercata.

□

COROLLARIO 5.12. Sia M DFA. Allora i problemi '$L(M) = \varnothing$' e '$L(M)$ è infinito' sono entrambi decidibili (cioè esiste una procedura effettiva per determinare la loro veridicità o falsità).

TEOREMA 5.13 (Equivalenza). *Dati due DFA M_1 e M_2, il problema di stabilire se $L(M_1) = L(M_2)$ è decidibile.*

PROOF. $L(M_1) = L(M_2)$ è insiemisticamente equivalente a

$$(L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2)) = \varnothing\,.$$

Dai Teoremi 5.5 e 5.6 sappiamo esistere un automa M_3 tale che

$$L(M_3) = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2)) .$$

Il risultato segue dalla decidibilità del problema del vuoto (Corollario 5.12). $\square$

4. Il teorema di Myhill-Nerode

Iniziamo il paragrafo richiamando alcune definizioni. Dato un insieme S, una relazione di equivalenza $R \subseteq S \times S$ (si veda il Capitolo 2) induce (univocamente, a meno di ridenominazione degli indici) una *partizione* di $S = S_1 \cup S_2 \cup \dots$ ove per ogni i $S_i \neq \varnothing$ e per ogni i, j, $i \neq j$, si ha che:

(1) $S_i \cap S_j = \varnothing$;
(2) $(\forall a, b \in S_i) a \mathrel{R} b$;
(3) $(\forall a \in S_i)(\forall b \in S_j) \neg(a \mathrel{R} b)$.

Le varie S_i sono dette *classi di equivalenza*. Se $a \in S_i$ allora con la notazione $[a]_R$ (o semplicemente $[a]$ quando il contesto è chiaro) si denota la classe S_i. Se S è partizionato in un numero finito di classi $S_1 \cup S_2 \cup \cdots \cup S_k$ allora R si dice di *indice finito* (k) su S. Date due relazioni di equivalenza R_1 e R_2 sullo stesso insieme S, R_1 è un *raffinamento* di R_2 se ogni classe di equivalenza della partizione indotta da R_1 è sottoinsieme di qualche classe di equivalenza della partizione indotta da R_2.

Ad esempio, se $S = \{2, 3, 4, 5\}$, $P_1 = \{\{2, 3, 5\}, \{4\}\}$ e $P_2 = \{\{2\}, \{3, 5\}, \{4\}\}$ denotano le partizioni di S ottenute a partire dalle relazioni

(R_1) xR_1y sse x e y sono entrambi primi o uguali tra loro

(R_2) xR_2y sse x e y sono entrambi primi e dispari, o uguali tra loro

allora R_2 è un raffinamento di R_1.

Esercizio 5.14. Dato un insieme S finito, quanti sono i modi distinti per partizionarlo?

Dato un linguaggio $L \subseteq \Sigma^*$, gli associamo la relazione $R_L \subseteq \Sigma^* \times \Sigma^*$ definita come segue:

$$x \mathrel{R_L} y \quad \text{sse} \quad (\forall z \in \Sigma^*)(xz \in L \leftrightarrow yz \in L) .$$

Similmente, se $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ è un DFA, gli associamo la relazione $R_M \subseteq \Sigma^* \times \Sigma^*$ definita come:

$$x \mathrel{R_M} y \quad \text{sse} \quad \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) .$$

Definiamo come $L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}$ il linguaggio associato allo stato q dell'automa. Le classi di equivalenza di R_M sono esattamente i linguaggi associati ad ogni stato dell'automa M.

Lemma 5.15. *R_L e R_M sono relazioni di equivalenza.*

PROOF. (Esercizio) □

Una relazione $R \subseteq \Sigma^* \times \Sigma^*$ che gode della proprietà:

$$x \; R \; y \quad \text{implica} \quad (\forall z \in \Sigma^*)(xz \; R \; yz)$$

si dice *invariante a destra* (rispetto alla concatenazione). Dall'esercizio 3.8.1, si evince che R_M è invariante a destra. Anche R_L è invariante a destra: siano $x, y \in \Sigma^*$ tali che $x R_L y$. Sia $z \in \Sigma^*$: dobbiamo mostrare che xzR_Lyz. Se così non fosse, allora esisterebbe w tale che $xzw \in L$ e $yzw \notin L$ (o viceversa). Ma in tal caso, con $z' = zw$, mostreremmo che $x \not R_L \; y$: assurdo.

TEOREMA 5.16 (Myhill-Nerode, 1957–58). *I seguenti enunciati sono equivalenti:*

(1) $L \subseteq \Sigma^*$ *è accettato da un qualche DFA;*
(2) L *è l'unione di classi di equivalenza di* Σ^* *indotte da una relazione invariante a destra e di indice finito;*
(3) R_L *è di indice finito.*

PROOF. $(1) \to (2)$ Sia L accettato da un DFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$. Mostreremo che R_M è la relazione che soddisfa il punto (2). Per definizione di linguaggio riconosciuto da un automa:

$$L = \bigcup_{q \in F} \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}$$

Per l'esercizio 3.8.1, R_M è invariante a destra, dunque i vari insiemi:

$$L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}$$

costituiscono le classi di equivalenza della partizione indotta da R_M.

$(2) \to (3)$ Mostreremo che ogni relazione di equivalenza R che soddisfa (2) è un raffinamento di R_L. Sia $x \in \Sigma^*$, vogliamo mostrare che $[x]_R \subseteq [x]_{R_L}$. Sia $y \in [x]_R$ (dunque $x \; R \; y$). Poiché R è invariante a destra per ipotesi, allora $xz \; R \; yz$ per ogni $z \in \Sigma^*$. Poiché L è unione di classi di equivalenza di R, ciò implica che ogni qualvolta vRw si ha che $v \in L$ sse $w \in L$. Pertanto per ogni $z \in \Sigma^*$, $xz \in L$ sse $yz \in L$. Ma allora $x \; R_L \; y$ per definizione, dunque $y \in [x]_{R_L}$. L'indice di R_L è minore di quello di R, che per ipotesi è finito.

$(3) \to (1)$ Si costruisce un DFA $M' = \langle Q', \Sigma', \delta', q'_0, F'\rangle$ che riconosce L. Sia

- Q' l'insieme (finito per ipotesi) di classi di equivalenza di R_L,
- Σ' lo stesso di L,
- $\delta'([x], a) = [xa]$ (la definizione ha senso indipendentemente dalla scelta di x in quanto R_L è invariante a destra),
- $q'_0 = [\varepsilon]$,
- $F' = \{[x] : x \in L\}$.

Si tratta di mostrare che $L(M') = L$. Si verifica per induzione su $|y| \geq 0$ che $\hat{\delta}'([x], y) = [xy]$. Pertanto si ha che:

$$\hat{\delta}'(q'_0, x) = \hat{\delta}'([\varepsilon], x) = [\varepsilon x] = [x]$$

e dunque

$$x \in L(M') \quad \text{sse} \quad \hat{\delta}'(q'_0, x) \in F' \quad \text{sse} \quad [x] \in F \quad \text{sse} \quad x \in L\,.$$

□

5. Minimizzazione di DFA

Il Teorema 5.16 ha, tra le sue varie conseguenze, quella di stabilire che se un linguaggio L è riconosciuto da un DFA M allora ne esiste uno minimo M' (nel senso del numero degli stati) equivalente a M.

Teorema 5.17. *Per ogni linguaggio regolare* L *esiste un automa* M *con minimo numero di stati tale che* $L = L(M)$*, unico a meno di isomorfismo (ovvero ridenominazione di stati).*

Proof. Sia L accettato da un DFA M. Σ^* è partizionato dalla relazione di equivalenza R_M negli insiemi di linguaggi accettati dai singoli stati di M.

Dal Teorema 5.16 ($1 \to 2$) sappiamo che R_M raffina R_L e pertanto il numero di stati di M è maggiore o uguale a quello di M' costruito come nella dimostrazione ($3 \to 1$) del Teorema 5.16. Dunque l'automa M' ha il minor numero di stati possibile. Supponiamo che tale numero coincida con il numero di stati di M.

Definiamo la funzione f tra gli insiemi di stati dei due automi nel modo seguente:

$$f(q) = [x] \quad \text{sse} \quad \hat{\delta}(q_0, x) = q\,.$$

Mostriamo che questa è una buona definizione. Vogliamo essere sicuri che $f(q)$ è univoca. Supponiamo, per assurdo, $f(q) = [y_1]$ e $f(q) = [y_2]$. Allora $\hat{\delta}(q_0, y_1) = q$ e $\hat{\delta}(q_0, y_2) = q$. Ma allora $y_1 \; R_M \; y_2$. Poiché R_M è raffinamento di R_L si ha anche $[y_1] = [y_2]$.
La funzione è suriettiva, in quanto per ogni $x \in \Sigma^*$ si ha che esiste $q = \hat{\delta}(q_0, x)$ tale che $f(q) = [x]$. Poiché $|Q| = |Q'|$ ciò implica che f è biiettiva, dunque vi è una funzione di rinomina dei nodi.

Per mostrare che i due automi sono isomorfi rimane da mostrare che il seguente diagramma commuta, per ogni $q \in Q$ e per ogni $a \in \Sigma$:

$$\begin{array}{ccc} q & \xrightarrow{f} & [x] \\ \downarrow a & & \downarrow a \\ q' & \xrightarrow{f} & [xa] \end{array}$$

In altri termini, dati $f(q) = [x]$, $\delta(q, a) = q'$, $\delta'([x], a) = [xa]$, dobbiamo mostrare che $f(q') = [xa]$:

$$
\begin{array}{llll}
f(q') = [xa] & \text{sse} & q' = \hat{\delta}(q_0, xa) & \text{def. di } f \\
 & \text{sse} & q' = \delta(\hat{\delta}(q_0, x), a) & \text{def. di } \hat{\delta} \\
 & \text{sse} & q' = \delta(q, a) & \text{vero per ipotesi.}
\end{array}
$$

□

In Fig. 1 è presentato un algoritmo che permette di ottenere automaticamente l'automa minimo corrispondente ad uno dato.

Esempio 5.18. *La minimizzazione dell'automa:*

	0	1	
q_0	q_1	q_3	
q_1	q_2	q_3	
q_2	q_0	q_3	
q_3	q_3	q_3	F

conduce all'automa:

	0	1	
q_0	q_0	q_3	
q_3	q_3	q_3	F

in un solo passo. Invece la minimizzazione dell'automa:

	0	1	
q_0	q_0	q_1	
q_1	q_1	q_2	
q_2	q_2	q_3	
q_3	q_3	q_3	F

restituisce l'automa stesso. Sono però necessarie 3 ($= |Q| - 1$*) iterazioni.*

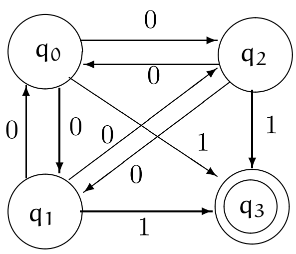
Esercizio 5.19 (Facoltativo). Si dimostri che l'algoritmo di Fig. 1 permette effettivamente di calcolare un automa equivalente e minimo. (Suggerimento: dato $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ e $p, q \in Q$, diciamo che p è distinguibile da q se esiste una stringa x tale che $\hat{\delta}(p, x) \in F \leftrightarrow \hat{\delta}(q, x) \notin F$. Per il pumping lemma, sappiamo che per verificare la distinguibilità di due stati è sufficiente provare tutte le stringhe x di lunghezza minore a $|Q|$. Si tratta di mostrare che due stati sono non distinguibili se e solo se sono equivalenti per $R_{L(M)}$ e che con al più $|Q|$ iterazioni dell'algoritmo tutti gli stati distinguibili sono scovati.)

procedure minimizza;
input: $M = \langle Q, \Sigma, \delta, q_0, F\rangle$;
output: $M' = \langle Q', \Sigma, \delta', q_0', F'\rangle$ t.c. $L(M) = L(M')$ e
$(\forall M'' = \langle Q'', \Sigma, \delta'', q_0'', F''\rangle)(L(M) = L(M'') \to |Q'| \leq |Q''|)$;
begin
$\Pi_{new} := \{Q - F, F\}$;
repeat
$\Pi := \Pi_{new}$;
$\Pi_{new} := \varnothing$;
for S in Π do
begin
partiziona S in $S_1, \ldots, S_m$
usando il valore di m più piccolo possibile
t.c. $p, q \in S_i$ sse
$(\forall a \in \Sigma)(\exists S' \in \Pi)(\delta(p, a) \in S' \leftrightarrow \delta(q, a) \in S')$
$\Pi_{new} := \Pi_{new} \cup \{S_1, \ldots, S_m\}$
end
until $\Pi_{new} = \Pi$;
for S in Π let $\bar{S} = \min_i\{q_i \in S\}$;
$Q' := \{\bar{S} : S \in \Pi\}$;
$\delta'(\bar{S}, a) := \min_i\{q_i : q_j \in S, \delta(q_j, a) = q_i\}$;
$F' := \{[S] : S \in \Pi, (\exists q \in S)(q \in F)\}$;
$q_0' := q_0$;
si eliminino ricorsivamente da Q' e F' gli stati diversi da q_0' privi di archi entranti e si eliminino da δ gli archi uscenti da essi.
end.

FIGURE 1. Algoritmo di minimizzazione

Esercizio 5.20. Si concretizzi in un linguaggio di programmazione ad alto livello l'algoritmo di Fig. 1 in modo tale che il problema sia risolto con complessità $O(|Q|^2|\Sigma|)$. Si mediti su come arrivare alla complessità $O(|Q|\log|Q||\Sigma|)$.

Esercizio 5.21. Si determini, usando le tecniche di trasformazione viste finora, il DFA minimo equivalente all'automa:



Si determini inoltre, dimostrando in modo formale la propria affermazione, il linguaggio riconosciuto dal DFA calcolato.

Esercizio 5.22. Si determini l'automa minimo per il linguaggio denotato dall'espressione regolare:

$$(0^* + 1^* + (01)^*)$$

CHAPTER 6

Grammatiche libere dal contesto

Una grammatica è, intuitivamente, un insieme di regole che permettono di generare un linguaggio. Un ruolo fondamentale tra le grammatiche è costituito dalle grammatiche libere dal contesto mediante le quali vengono solitamente descritti i linguaggi di programmazione.

1. Definizione formale

Una *grammatica libera dal contesto* (CF) è una quadrupla $G = \langle V, T, P, S \rangle$, ove:

- V è un insieme finito di variabili (dette anche simboli *non terminali*),
- T è un insieme finito di simboli *terminali* ($V \cap T = \varnothing$),
- P è un insieme finito di produzioni; ogni *produzione* è della forma $A \rightarrow \alpha$,[1] ove:
  - $A \in V$ è una variabile, e
  - $\alpha \in (V \cup T)^*$.
- $S \in V$ è una variabile speciale, detta *simbolo iniziale.*

Esempio 6.1. Sia G la grammatica seguente:

$$G = \{\{E\}, \{\mathtt{or}, \mathtt{and}, \mathtt{not}, (,), 0, 1\}, P, E\}$$

ove P è costituito da:

$$\begin{array}{lcl} E & \mapsto & 0 \\ E & \mapsto & 1 \\ E & \mapsto & (E \text{ or } E) \\ E & \mapsto & (E \text{ and } E) \\ E & \mapsto & (\text{not } E) \end{array}$$

Come vedremo, G genererà le possibili espressioni booleane.

Notazione 6.2. Per le grammatiche saranno utilizzate le seguenti notazioni: lettere maiuscole A, B, C, D, E, S denotano variabili, S—a meno che non sia esplicitamente detto il contrario—il simbolo iniziale. $a, b, c, d, e, 0, 1$ denotano simboli terminali; X, Y, Z denotano simboli che possono essere sia terminali che variabili;

[1]Oppure, equivalentemente, si può vedere come una coppia $\langle A, \alpha \rangle \in V \times (V \cup T)^*$.

u, v, w, x, y, z denotano stringhe di terminali, $\alpha, \beta, \gamma, \delta$ stringhe generiche di simboli (sia terminali che non). Se $A \to \alpha_1, \ldots, A \to \alpha_n \in P$, esprimeremo questo fatto scrivendo: $A \to \alpha_1 | \cdots | \alpha_n$. Con A-*produzione* denoteremo una generica produzione con la variabile A a sinistra.

La grammatica dell'esempio 6.1 può essere dunque schematicamente rappresentata da:

$$E \mapsto 0|1|(E \text{ or } E)|(E \text{ and } E)|(\text{not } E)\,.$$

2. Linguaggio generato

Le grammatiche servono a generare un linguaggio. Daremo ora le definizioni necessarie a definire il linguaggio generato da una grammatica $G = \langle V, T, P, S\rangle$. Definiremo le relazioni $\overset{G}{\Rightarrow}, \overset{G}{\Rightarrow}_i, \overset{G}{\Rightarrow}_* \subseteq (V \cup T)^* \times (V \cup T)^*$ nel seguente modo:

- se $A \to \beta \in P$ e $\alpha, \gamma \in (V \cup T)^*$, allora $\alpha A \gamma \overset{G}{\Rightarrow} \alpha\beta\gamma$. Diremo in questo caso che da $\alpha A \gamma$ *deriva immediatamente* $\alpha\beta\gamma$;
- se $\alpha_1, \ldots, \alpha_i \in (V \cup T)^*$, $i \geq 1$, e
$$\bigwedge_{j=1}^{i-1} \alpha_j \overset{G}{\Rightarrow} \alpha_{j+1}\ ,$$
allora $\alpha_1 \overset{G}{\Rightarrow}_i \alpha_m$. In questo caso diremo che da α_1 *deriva* α_m in i passi. Per ogni α, diremo anche che $\alpha \overset{G}{\Rightarrow}_0 \alpha$;
- se esiste i tale per cui $\alpha \overset{G}{\Rightarrow}_i \beta$, allora $\alpha \overset{G}{\Rightarrow}_* \beta$. Diremo in tal caso che da α deriva β.

Si noti che $\overset{G}{\Rightarrow}_*$ è la chiusura transitiva e riflessiva della relazione $\overset{G}{\Rightarrow}$. In particolare vale che $\alpha \overset{G}{\Rightarrow}_* \alpha$ per ogni $\alpha \in (V \cup T)^*$.

Il *linguaggio generato da* G è:

$$L(G) = \{w \in T^* : S \overset{G}{\Rightarrow}_* w\}\,.$$

L è un liguaggio *libero dal contesto* (CF) se esiste una grammatica CF G tale che $L = L(G)$. Due grammatiche G_1 e G_2 sono equivalenti se $L(G_1) = L(G_2)$.

Esempio 6.3. Σ^* è un linguaggio CF. Infatti, sia $\Sigma = \{s_1, \ldots, s_n\}$, allora Σ^* è generato da:

$$S \to \varepsilon | s_1 S | \cdots | s_n S\,.$$

Esempio 6.4. La grammatica schematicamente definita da:

$$S \to ASB \mid \varepsilon, A \to 0, B \to 1$$

genera il linguaggio

$$\{0^n 1^n : n \geq 0\}$$

(dimostrare ciò per induzione su n come esercizio).

Per quanto dimostrato nell'esempio 5.3 usando il pumping lemma, il linguaggio $\{0^n1^n : n \geq 0\}$ non è un linguaggio regolare. Pertanto l'insieme dei linguaggi regolari e quello dei linguaggi CF non sono lo stesso insieme. Vedremo, comunque, che il primo insieme è incluso nel secondo.

Esempio 6.5. La grammatica dell'Esempio 6.1 genera tutte le espressioni booleane (si dimostri questo fatto per per induzione sulla struttura dell'espressione).

3. Alberi di derivazione

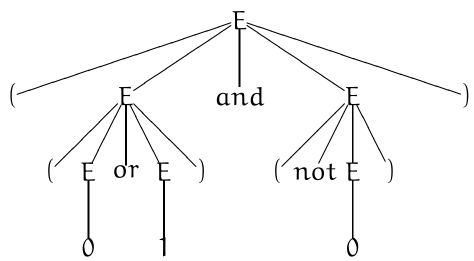
Le derivazioni generate da una grammatica vengono ben visualizzate mediante l'ausilio di strutture dati ad albero.

Sia $G = \langle V, T, P, S \rangle$ una grammatica CF. Un albero è un *albero di derivazione* (parse tree) per G se:

(1) ogni vertice ha una *etichetta*, presa tra $V \cup T \cup \{\varepsilon\}$;
(2) l'etichetta della radice appartiene a V;
(3) ogni vertice interno (ovvero, non una foglia) ha etichetta appartenente a V;
(4) se un vertice n è etichettato con A e $n_1, \ldots, n_k$ sono (ordinatamente, da sinistra a destra) i vertici *figli* di A etichettati con $X_1, \ldots, X_k$, allora
$$A \to X_1 \cdots X_k \in P\,;$$
(5) se un vertice n ha etichetta ε, allora n è una foglia ed è l'unico figlio di suo padre.

Si noti che la definizione di albero non impone che tutte le foglie siano etichettate con simboli terminali. L'albero avente un solo nodo etichettato da S è un caso particolare di albero di derivazione. Gli alberi verrano utilizzati per dare una controparte grafica delle derivazioni. Se vi sono foglie etichettate con simboli non terminali, allora l'albero rappresenta una derivazione *parziale*. Diremo che un albero *descrive* una stringa $\alpha \in (V \cup T)^*$ se α è proprio la stringa che possiamo leggere dalle etichette delle foglie da sinistra a destra.

Esempio 6.6. La grammatica dell'esempio 6.1 genera, in particolare, la stringa $w = ((0 \texttt{ or } 1) \texttt{ and } (\texttt{not } 0))$. La derivazione per w è rappresentata dall'albero seguente (che descrive w):



Dato un albero ed un suo nodo n, il *sottoalbero* identificato da quel nodo è costituito da quel nodo più tutti i suoi discendenti (figli, nipoti, etc.), nonchè le varie etichette associate ad essi.

TEOREMA 6.7. *Sia* $G = \langle V, T, P, S \rangle$ *una grammatica CF. Allora* $S \overset{G}{\Rightarrow}_* \alpha$ *se e solo se esiste un albero di derivazione con radice etichettata* S *per* G *che descrive* α*.*

PROOF. Si proverà un enunciato più forte (e più comodo per la dimostrazione induttiva), ovvero: dato un qualunque simbolo non terminale $A \in V$, $A \overset{G}{\Rightarrow}_* \alpha$ se e solo se esiste un albero di derivazione per G che descrive α la cui radice è etichettata A.
($\rightarrow$) Proviamo l'enunciato per induzione sul numero i di passi di derivazione per $A \overset{G}{\Rightarrow}_i \alpha$.
Base: $i = 0$: $A \overset{G}{\Rightarrow}_0 \alpha$ implica, per definizione, che $\alpha = A$. Ma l'albero con unico nodo (radice) etichettato con A è un albero di derivazione (parziale) che descrive A stesso.
Passo: Supponiamo $A \overset{G}{\Rightarrow}_i \beta_1 B \beta_3 \overset{G}{\Rightarrow}_1 \underbrace{\beta_1 \beta_2 \beta_3}_{\alpha}$. Per ipotesi induttiva esiste un albero di derivazione T con radice etichettata A che descrive $\beta_1 B \beta_3$. Ma per definizione, $\beta_1 B \beta_3 \overset{G}{\Rightarrow}_1 \beta_1 \beta_2 \beta_3$ se e solo se esiste la produzione $B \rightarrow \beta_2$ in P. Ma allora, applicando la regola (4) di definizione di albero, dall'albero T si ottiene l'albero che soddisa i requisiti.
($\leftarrow$) Proviamo l'enunciato per induzione sul numero di nodi interni dell'albero di derivazione etichettato in A che descrive α. [Completare per esercizio] $\square$

4. Ambiguità delle derivazioni

Se ad ogni passo di una derivazione la produzione è applicata al simbolo non terminale più a sinistra, allora la derivazione è detta *sinistra* (leftmost). Similmente, se viene applicata sempre a quello a destra, allora è detta *destra* (rightmost). Se $w \in L(G)$, dal Teorema 6.7 sappiamo che per essa esiste almeno un albero di derivazione con radice etichettata S. Si può dimostrare che, fissato un albero di derivazione con radice etichettata S, esattamente una derivazione sinistra (suggerimento: si visiti l'albero in preordine) ed una derivazione destra (simmetricamente) possono essere desunte.

Un albero di derivazione rappresenta dunque un certo insieme di possibili derivazioni distinte di una stessa stringa. Ci possono tuttavia essere più alberi di derivazione per la stessa parola:

ESEMPIO 6.8. Si consideri la grammatica

$$E \rightarrow E + E | E * E | 0 | 1 | 2 \, .$$

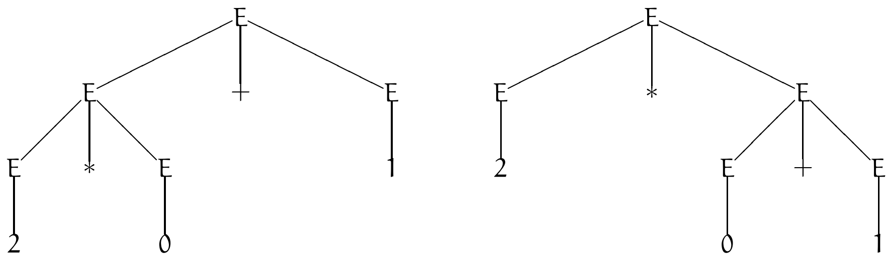
Una derivazione sinistra è:

$$E \overset{G}{\Rightarrow} E + E \overset{G}{\Rightarrow} E * E + E \overset{G}{\Rightarrow}_3 2 * 0 + 1 \, .$$

Un'altra derivazione sinistra per la stessa stringa è:

$$\mathsf{E} \overset{G}{\Rightarrow} \mathsf{E} * \mathsf{E} \overset{G}{\Rightarrow} 2 * \mathsf{E} \overset{G}{\Rightarrow} 2 * \mathsf{E} + \mathsf{E} \overset{G}{\Rightarrow}_2 2 * 0 + 1\,.$$

Tuttavia gli alberi associati alle due derivazioni (che potrebbero essere visti come alberi per la computazione dell'espressione associata alla stringa generata) sono diversi:



Il primo è intuitivamente associato a $(2*0)+1 = 1$ il secondo, invece, $2*(0+1) = 2$.

Una grammatica CF tale per cui esiste una parola con più di un albero di derivazione con radice etichettata S è detta *ambigua*. Un linguaggio CF per cui ogni grammatica che lo genera è ambigua è detto essere *inerentemente ambiguo*. Un esempio di tali linguaggi è:

$$\mathsf{L} = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}\,.$$

La dimostrazione di questa proprietà (3 pagine!) si può trovare in [**13**].

5. Semplificazione

Vi sono molti modi per restringere la forma delle produzioni permesse alle grammatiche CF senza alterare la classe dei linguaggi riconosciuti dall'insieme delle possibili grammatiche. In questo paragrafo si presenteranno due di queste trasformazioni, che conducono alla forma normale di Chomsky e a quella di Greibach.

Mostreremo che ogni linguaggio CF può essere generato da una grammatica G dalle seguenti due proprietà:

(1) ogni variabile e ogni simbolo terminale di G compaiono in almeno una derivazione di una qualche parola di L;
(2) non ci sono produzioni della forma $\mathsf{A} \to \mathsf{B}$ con A e B variabili.

Se, inoltre, $\varepsilon \notin \mathsf{L}$, non avremo bisogno di produzioni $\mathsf{A} \to \varepsilon$; sempre sotto tale ipotesi potremo richiedere che ogni produzione sia della forma $\mathsf{A} \to \mathsf{BC}$ o $\mathsf{A} \to a$ (Chomsky) oppure, alternativamente, che ogni produzione sia della forma $\mathsf{A} \to a\alpha$ ove α è una stringa di variabili (Greibach).
Nel prosieguo del capitolo alcune dimostrazioni tecniche ma dimostrabili immediatamente per induzione saranno lasciate per esercizio.

5.1. Eliminazione di simboli inutili. Sia $G = \langle V, T, P, S\rangle$ una grammatica CF. Un simbolo $X \in V \cup T$ è *utile* se esiste una derivazione

$$S \overset{G}{\Rightarrow}_* \alpha X \beta \overset{G}{\Rightarrow}_* w\,;$$

altrimenti è detto inutile. Mostreremo come simboli inutili si possano eliminare in due passi.
Innanzitutto si eliminano le variabili che non conducono in nessun modo ad una stringa di terminali.

LEMMA 6.9. *Data una grammatica CF* $G = \langle V, T, P, S\rangle$ *tale che* $L(G) \neq \varnothing$*, si può calcolare in modo effettivo una grammatica equivalente* $G' = \langle V', T, P', S\rangle$ *tale che per ogni* $A \in V'$ *esiste* $w \in T^*$ *tale che* $A \overset{G}{\Rightarrow}_* w$*.*

PROOF. Per calcolare V' usiamo il seguente operatore $\Gamma : \wp(V) \longrightarrow \wp(V)$:

$$\Gamma(W) = \{A \in V : (\exists \alpha \in (T \cup W)^*)(A \to \alpha \in P)\}$$

Γ è monotono—ovvero $X \subseteq Y \to \Gamma(X) \subseteq \Gamma(Y)$—per definizione. Definiamo inoltre l'applicazione iterata di un operatore:

$$\begin{cases} \Gamma^0(W) & = & W \\ \Gamma^{i+1}(W) & = & \Gamma(\Gamma^i(W)) \end{cases}$$

Per induzione su $i \geq 1$, si mostra simultaneamente su tutti gli $A \in T$ che:

$$\left\{ \begin{array}{l} \text{esiste un albero di derivazione che descrive } w \in T^* \\ \text{con radice etichettata con } A \text{ e di altezza } \leq i \end{array} \right\} \quad \text{sse} \quad A \in \Gamma^i(\varnothing)$$

(completare per esercizio).

Dunque, per il Teorema 6.7, si ha che $A \in \Gamma^i(\varnothing)$ per qualche i sse $A \overset{G}{\Rightarrow}_* w$.

Poiché V è finito e Γ monotono, si ha che esiste $i \leq |V|$ tale per cui

$$\Gamma^i(\varnothing) = \Gamma(\Gamma^i(\varnothing)) = \Gamma^{i+1}(\varnothing)(= \Gamma^{i+2}(\varnothing) = \cdots)$$

Pertanto si riesce a calcolare finitamente l'insieme $V' = \Gamma^{|V|}(\varnothing)$, e $P' = \{A \to \alpha \in P : A \in V' \wedge \alpha \in (V' \cup T)^*\}$. $\square$

L'ipotesi che il linguaggio sia non vuoto è indispensabile. Se infatti fosse $L = \varnothing$, allora l'applicazione del Lemma 6.9 fornirebbe una grammatica con $S \notin V'$ e dunque priva di significato in quanto S sarebbe il simbolo iniziale ma non apparterrebbe alla grammatica. In ogni caso, per definizione, ogni simbolo di una grammatica che genera un linguaggio vuoto è inutile. Si osservi che l'applicazione della procedura del lemma permette anche di determinare se il linguaggio riconosciuto da un automa sia vuoto o meno.

Fatto ciò eliminiamo le variabili che non sono mai raggiunte da S:

LEMMA 6.10. *Data una grammatica CF* $G = \langle V, T, P, S\rangle$, *si può calcolare in modo effettivo una grammatica equivalente* $G' = \langle V', T', P', S'\rangle$ *tale che per ogni* $X \in (V' \cup T')$ *esistono* α *e* β *in* $(V' \cup T')^*$ *per cui* $S \overset{G}{\Rightarrow}_* \alpha X \beta$.

PROOF. Definiamo un operatore $\Gamma : \wp(V \cup T) \longrightarrow \wp(V \cup T)$:

$$\Gamma(W) \quad = \quad \{X \in V \cup T \,:\, (\exists A \in W)(A \to \alpha X \beta \in P)\,\} \cup \{S\}$$

Per induzione su $i \geq 0$, si mostra (esercizio) che:

$$\left\{\begin{array}{l} \text{esiste un albero di derivazione (parziale)} \\ \text{con radice etichettata con } S \text{ di altezza} \leq i \\ \text{in cui } X \text{ compare come foglia} \end{array}\right\} \quad \text{sse} \quad X \in \Gamma^i(\{S\})$$

Pertanto, per il Teorema 6.7 si ha che $X \in \Gamma^i(\{S\})$ per qualche i se e solo se esistono α e β in $(V \cup T)^*$ per cui $S \overset{G}{\Rightarrow}_* \alpha X \beta$.

Γ è monotono, V e T sono finiti, dunque esiste $i \leq |V| + |T|$ tale che

$$\Gamma^i(\{S\}) = \Gamma^{i+1}(\{S\})(= \Gamma^{i+2}(\{S\}) = \cdots)$$

Siano dunque:

- $V' = \Gamma^{|V|+|T|}(\{S\}) \cap V$;
- $T' = \Gamma^{|V|+|T|}(\{S\}) \cap T$;
- $P' = \{A \to \alpha \in P : A \in V' \wedge \alpha \in (V' \cup T')^*\}$.

$\square$

Si osservi come, introducendo anche ε in $\Gamma(W)$ qualora $A \longrightarrow \varepsilon \in P$ e $A \in W$, saremmo anche in grado di capire se $S \overset{G}{\Rightarrow}_* \varepsilon$.

TEOREMA 6.11. *Ogni linguaggio CF non vuoto è generato da una grammatica CF priva di simboli inutili.*

PROOF. Immediato, applicando prima il Lemma 6.9 e poi il Lemma 6.10. $\square$

ESERCIZIO 6.12. Si cerchi una grammatica con linguaggio non vuoto tale per cui applicando prima la procedura del Lemma 6.10 e poi quella del Lemma 6.9 non si giunge all'eliminazione dei simboli inutili.

5.2. Eliminazione di ε-produzioni. Data una grammatica CF che genera un linguaggio L, si desidera eliminare le produzioni della forma $A \to \varepsilon$. Qualora $\varepsilon \in L$, ammetteremo la presenza della produzione $S \to \varepsilon$.

Il metodo è quello di determinare quali variabili $A \in V$ sono tali che $A \overset{G}{\Rightarrow}_* \varepsilon$. Tali variabili sono dette *annullabili.* Ciò che poi si farà sarà quello di eliminare, tra le X_i di ogni produzione $A \to X_1 \cdots X_n$ zero, una, ..., o tutte le variabili annullabili. Se togliendo tutte le variabili annullabili da $X_1 \cdots X_n$ la stringa diventasse vuota, allora anche A sarà annullabile (ma non aggiungeremo la produzione $A \to \varepsilon$, a meno che A non sia S).

TEOREMA 6.13. *Se* $L = L(G)$ *per qualche grammatica CF* $G = \langle V, T, P, S\rangle$, *allora* $L \setminus \{\varepsilon\}$ *è un linguaggio CF generato da una grammatica* $G' = \langle V', T', P', S'\rangle$ *senza simboli inutili e senza* ε*-produzioni.*

PROOF. Definiamo un operatore $\Gamma : \wp(V) \longrightarrow \wp(V)$:

$$\begin{aligned} \Gamma(W) \;\; = \;\; \{A \in V : \;\; & (\exists A \to \alpha \in P) \\ & \alpha \text{ ``privata'' delle variabili di } W \text{ è } \varepsilon\} \end{aligned}$$

Anche qui, Γ è monotono, V è finito, dunque esiste $i \leq |V|$ tale che $\Gamma^i(\emptyset) = \Gamma^{i+1}(\emptyset)$. Siano dunque $N = \Gamma^{|V|}(\emptyset)$ e

$$\begin{aligned} P'' \;\; = \;\; \{A \to \alpha_1 \cdots \alpha_n : \;\; & A \to X_1 \cdots X_n \in P, \\ & \text{se } X_i \notin N, \text{ allora } \alpha_i = X_i, \\ & \text{se } X_i \in N, \text{ allora } \alpha_i = X_i \text{ o } \alpha_i = \varepsilon, \\ & \alpha_1 \cdots \alpha_n \neq \varepsilon\,\} \end{aligned}$$

E' immediato mostrare che $L(G) = L(\langle V, T, P'', S\rangle)$. Applichiamo dunque il Teorema 6.11 alla grammatica $\langle V, T, P'', S\rangle$. $\square$

Ovviamente, se $\varepsilon \in L$, aggiungeremo la produzione $S \to \varepsilon$. Si osservi come questa procedura permetta di capire se $\varepsilon \in L$. E' infatti sufficiente vedere se esiste in P una produzione $S \to A_1 \cdots A_k$ con le $A_i \in N$ per $i = 1, \ldots, k$, oppure direttamente la produzione $S \to \varepsilon$.

5.3. Eliminazione di produzioni unitarie. Una produzione della forma $A \to B$, con A e B variabili, si dice *unitaria*.

TEOREMA 6.14. *Ogni grammatica CF senza* ε *è definita da una grammatica senza simboli inutili, senza* ε*-produzioni e senza produzioni unitarie.*

PROOF. Grazie al Teorema 6.13, possiamo partire da una grammatica senza simboli inutili e senza ε-produzioni. Innanzitutto identifichiamo tutte le coppie di variabili A, B che permettono derivazioni della forma $A \overset{G}{\Rightarrow}_* B$ (immediato, seguendo i cammini del grafo ottenuto prendendo come nodi le variabili e come archi le produzioni unitarie). Sia, per ogni variabile A, $\mathsf{seguenti}(A)$ l'insieme delle variabili B soddisfacenti la richiesta sopra.
A questo punto togliamo da P tutte le produzioni unitarie e, per ogni $A \in V$ e per ogni produzione non unitaria $B \to \beta \in P$ con $B \in \mathsf{seguenti}(A)$, $B \neq A$, aggiungo la produzione $A \to \beta$.
Dimostrare l'equivalenza dei linguaggi è immediato (induzione sul numero di produzioni unitarie presenti in una derivazione). Tuttavia, la procedura potrebbe aver generato simboli inutili (si pensi alla grammatica $S \longrightarrow A, A \longrightarrow a$): applichiamo il Teorema 6.11 (si noti che produzioni unitarie non possono essere introdotte dalla procedura di eliminazione di simboli inutili) e otteniamo il risultato cercato. $\square$

6. Forma normale di Chomsky

TEOREMA 6.15 (Chomsky, 1959). *Ogni linguaggio CF senza* ε *è generato da una grammatica in cui tutte le produzioni sono della forma* $A \to BC$ *e* $A \to a$.

PROOF. Per il Teorema 6.14 possiamo partire da una grammatica $G = \langle V, T, P, S\rangle$ senza ε-produzioni, simboli inutili e produzioni unitarie.
Sia $A \to X_1 \cdots X_m \in P$, $m \geq 1$ (non vi sono ε-produzioni):

- se $m = 1$, allora, poiché non vi sono produzioni unitarie, $X_1 \in T$: è già nella forma cercata;
- se $m = 2$ e $X_1, X_2 \in V$, siamo nella forma cercata;
- se $m \geq 2$ e qualcuno degli $X_i \in T$, allora per ogni $X_i \in T$ introduco in V una nuova variabile, diciamo B_i, aggiungo la produzione $B_i \to X_i$ e rimpiazzo la produzione $A \to X_1 \cdots X_m$ con $A \to Y_1 \cdots Y_m$, ove $Y_i = X_i$ se $X_i \in V$, $Y_i = B_i$ altrimenti;
- se $m > 2$ e tutti gli X_i sono variabili, allora rimpiazzo la produzione $A \to X_1 \cdots X_m$ con le produzioni $A \to BX_3 \cdots X_m, B \to X_1X_2$, ove B è una nuova variabile da aggiungere a V.

E' immediato verificare che il procedimento termina e che l'insieme finale di produzioni (P') è nella forma normale di Chomsky. Si tratta di verificare che le grammatica $G' = \langle V', T, P', S\rangle$ (ove V' si ottiene da V con l'aggiunta delle nuove variabili) ottenuta è equivalente a quella di partenza. Ovvero, per $A \in V$:

$$A \overset{G}{\Rightarrow}_* w \text{ sse } A \overset{G'}{\to}_* w$$

Dimostriamo, per induzione su $i \geq 1$ che $A \overset{G}{\Rightarrow}_i w$ implica che esiste k tale per cui $A \overset{G'}{\to}_{i+k} w$. [Esercizio]
Viceversa, dimostriamo, per induzione su $i \geq 1$ che $A \overset{G'}{\to}_i w$ implica che $A \overset{G}{\Rightarrow}_j w$ per qualche $j \leq i$. [Esercizio] □

7. Forma normale di Greibach

Per raggiungere questa forma normale abbiamo bisogno di due lemmi preliminari.

LEMMA 6.16 (Unfolding). *Sia* $G = \langle V, T, P, S\rangle$ *una grammatica CF, sia* $A \to \alpha B\gamma$ *una produzione di* P *e* $B \to \beta_1|\cdots|\beta_n$ *l'insieme delle* B*-produzioni. Sia* $G' = \langle V, T, P', S\rangle$ *ottenuta da* G *eliminando la produzione* $A \to \alpha B\gamma$ *da* P *e aggiungendo le produzioni* $A \to \alpha\beta_1\gamma|\cdots|\alpha\beta_n\gamma$. *Allora* $L(G) = L(G')$.

PROOF. (Esercizio) □

LEMMA 6.17 (Eliminazione ricorsione sinistra). *Sia* $G = \langle V, T, P, S\rangle$ *una grammatica CF e sia* $A \to A\alpha_1|\cdots|A\alpha_m$ *l'insieme delle* A*-produzioni di* G *per cui* A *è anche il simbolo più a sinistra della stringa di destra delle produzioni. Siano*

$A \to \beta_1|\cdots|\beta_n$ *le rimanenti* A*-produzioni. Sia* $G' = \langle V \cup \{B\}, T, P', S\rangle$ *la grammatica CF ottenuta aggiungendo una nuova variabile* B *a* V *e rimpiazzando tutte le* A*-produzioni con:*

(1) $A \to \beta_i|\beta_i B$ *per* $i \in \{1,\dots,n\}$*;*
(2) $B \to \alpha_i|\alpha_i B$ *per* $i \in \{1,\dots,m\}$.

Allora $L(G') = L(G)$.

PROOF. Mostriamo prima che se $x \in L(G)$ allora $x \in L(G')$. Se $S \overset{G}{\Rightarrow}_* x$ e non vi sono in essa produzioni della forma $A \to A\alpha_i$ non c'è nulla da dimostrare. Altrimenti, sia

$$A \overset{G}{\Rightarrow} A\alpha_{i_1} \overset{G}{\Rightarrow}_* A\gamma_1 \overset{G}{\Rightarrow} A\alpha_{i_2}\gamma_1 \overset{G}{\Rightarrow}_* A\gamma_2 \overset{G}{\Rightarrow} A\alpha_{i_3}\gamma_2 \overset{G}{\Rightarrow}_* \cdots\cdots$$

una sottoderivazione in cui tali produzioni si usano (e vengono esplicitate: nei tratti $\overset{G}{\Rightarrow}_*$ assumiamo che tali derivazioni non ci sono). Poiché alla fine si deve ottenere una stringa di terminali, ad un certo punto la A sempre presente in testa sarà rimpiazzata da una β_j usando la produzione $A \to \beta_j$. La derivazione può essere mimata con G' nel modo seguente:

$$\begin{array}{ccccccccc}
A & \overset{G}{\Rightarrow} & A\alpha_{i_1} & \overset{G}{\Rightarrow}_* & A\gamma_{i_1} & \overset{G}{\Rightarrow}_* & A\alpha_{i_2}\gamma_1 & \overset{G}{\Rightarrow}_* \cdots \\
A\gamma_{k-1} & \overset{G}{\Rightarrow} & A\alpha_{j_k}\gamma_{k-1} & \overset{G}{\Rightarrow}_* & A\gamma_k & \overset{G}{\Rightarrow} & \beta_j\gamma_k & \\
\hline
A & \overset{G}{\Rightarrow} & \beta_j B & \overset{G}{\Rightarrow}_* & \beta_j B\alpha_{i_1} & \overset{G}{\Rightarrow}_* & \beta_j B\gamma_1 & \overset{G}{\Rightarrow}_* \\
\beta_j B\alpha_{j_2}\gamma_1 & \overset{G}{\Rightarrow}_* \cdots & \beta_j B\gamma_{k-1} & \overset{G}{\Rightarrow} & \beta_j\alpha_{i_{k-1}}\gamma_{k-1} & \overset{G}{\Rightarrow}_* & \beta_j\gamma_k &
\end{array}$$

L'implicazione inversa è lasciata per esercizio. □

TEOREMA 6.18 (Greibach, 1965). *Ogni linguaggio CF senza* ε *è generato da una grammatica in cui tutte le produzioni sono della forma* $A \to a\alpha$*, ove* α *è una stringa (eventualmente vuota) di simboli non terminali.*

PROOF. Sia G una grammatica in forma normale di Chomsky. Sia $V = \{A_1,\dots,A_m\}$. Si costruirà in tre passi una grammatica G' in forma normale di Greibach equivalente a G.

(1) Si applichi il seguente algoritmo:

```
for k := 1 to m do
begin
    for j := 1 to k − 1 do
        elimina le produzioni A_k → A_j α mediante unfolding;
    elimina le produzioni A_k → A_k α mediante elim. ricors. sinistra
end;
```

Nuove variabili $B_1, \ldots, B_h$ sono state introdotte dalla fase di eliminazione della ricorsione sinistra. L'equivalenza tra G e la grammatica ottenuta deriva dai lemmi appena dimostrati.

A questo punto tutte le produzioni sono della forma:

$$\begin{array}{lcll} A_i & \to & A_j\alpha & j > i \\ A_i & \to & a\beta & a \in T \\ B_i & \to & \gamma & \end{array}$$

Inoltre, poiché siamo partiti da una grammatica in forma normale di Chomsky, si osserva che α ha un prefisso della forma $A_{i_1} \cdots A_{i_h}$ seguito da un simbolo non terminale, seguito ancora da una sequenza eventualmente vuota di variabili. Per lo stesso motivo, β è una stringa di variabili. Per quanto riguarda le stringhe γ, esse non iniziano mai con un B_i.

(2) Pertanto, ogni produzione $A_m \to \alpha$ ha la parte destra iniziante con un simbolo terminale. Posso effettuare l'unfolding delle A_i-produzioni rimpiazzando A_m con ciascuna delle sue possibili parti destre. Posso iterare questo procedimento all'indietro, ottenendo produzioni della forma:

$$A_i \quad \to \quad a\beta \quad a \in T$$

per ogni $i = 1, \ldots, m$, con β stringa di variabili.

(3) Si tratta ora di semplificare le produzioni $B_i \to \gamma$. Applicando la sostituzione sul primo (eventuale) simbolo A_i della stringa, si giunge al risultato.

$\square$

Esercizio 6.19. Si applichi la procedura descritta per condurre in forma normale di Greibach la grammatica dalle produzioni:

$$\begin{array}{lcl} A_1 & \longrightarrow & A_3A_3 \mid a \\ A_2 & \longrightarrow & A_1A_2 \mid b \\ A_3 & \longrightarrow & A_3A_1 \mid c \end{array}$$

Esercizio 6.20. Si riconduca alle forme normali di Chomsky e di Greibach la grammatica:

$$S \longrightarrow \varepsilon \mid aSb$$

Un importante caso particolare di grammatiche in forma normale di Greibach sono le grammatiche CF in cui le produzioni hanno tutte la forma $A \to a$ oppure $A \to aB$. Tali grammatiche, studiate nel Capitolo 9, sono grammatiche lineari destre ed hanno l'importante caratteristica di generare esattamente i linguaggi regolari.

CHAPTER 7

Automi a pila

In questo capitolo descriviamo una macchina astratta per il riconoscimento dei linguaggi CF. Questa macchina corrisponde ad un arricchimento degli automi già visti per il caso dei linguaggi regolari, mediante una struttura dati opportuna per memorizzare informazione: la pila. Abbiamo visto che, grazie al pumping-lemma per i linguaggi regolari, il linguaggio

$$\left\{a^n b^n : n \geq 0\right\}$$

non è regolare. Intuitivamente l'impossibilità di riconoscere questo linguaggio da parte di un automa a stati finiti è legata all'impossibilità da parte di questi automi di memorizare una quantità arbitrariamente grande (anche se finita) di simboli e ricordarne il numero. Solo in questo modo infatti si può pensare di poter confrontare il numero di occorrenze di "a" e "b" nella stringa da leggere. Una struttura dati adeguata per risolvere questo problema è la pila, ovvero una struttura dati LIFO (Last In First Out). Da un punto di vista architetturale, un automa a pila è una macchina costituita da un *controllo* che è un automa a stati finiti, un nastro di input che, come negli NFA e DFA, è di sola lettura, ed una *pila* sulla quale sono possibili le operazioni di: `push(X)`, `pop(X)`, `empty` (si veda la Figura 1):

- L'istruzione `push(X)` sposta una stringa X in testa alla pila,
- l'istruzione `pop(X)` preleva il simbolo in testa alla pila, e
- l'istruzione `empty` verifica, restituendo valore booleano, se la pila è vuota o meno.

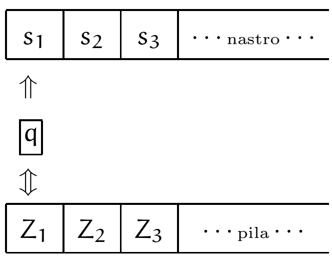


FIGURE 1. Automa a pila

Le azioni possibili di un automa a pila sono:

- Leggere dal nastro e dalla pila (`pop`) contemporaneamente
  - avanzare la testina a destra sul nastro
  - cambiare stato nel controllo
  - inserire una stringa (anche vuota) sulla pila (`push`)
- Leggere solo dalla pila (`pop`)
  - cambiare stato nel controllo
  - inserire una stringa (anche vuota) sulla pila (`push`)

Definizione 7.1. Un automa a pila non-deterministico (APND) M è una 7-upla:

$$M = \langle Q, \Sigma, R, \delta, q_0, Z_0, F\rangle$$

dove:

- Q è un insieme finito di stati
- Σ è un alfabeto finito
- R è l'alfabeto finito della pila
- $q_0 \in Q$ è lo stato iniziale
- $Z_0 \in R$ è il simbolo iniziale sulla pila
- $F \subseteq Q$ è l'insieme degli stati finali
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times R \to \wp_f(Q \times R^*)$ è la funzione di transizione

L'evoluzione di un automa a pila è determinata dall'evoluzione delle sue descrizioni istantanee. Una descrizione istantanea rappresenta la "fotografia" dello stato (stato + nastro + pila) della macchina ad un certo istante.

Definizione 7.2. Una descrizione istantanea per un APND è una tripla

$$(q, x, \gamma)$$

dove $q \in Q$ è lo stato corrente, $x \in \Sigma^*$ è la stringa ancora da leggere sul nastro e $\gamma \in R^*$ è la sequenza di simboli contenuti sulla pila. Se $(p, \gamma) \in \delta(q, a, Z)$ allora è definito un passo di derivazione dell'APND:

$$(q, aw, Z\alpha) \mapsto_M (p, w, \gamma\alpha)$$

Il linguaggio $L(M)$ riconosciuto da un APND M si può definire in due modi diversi: per pila vuota $L_p(M)$ o per stato finale $L_F(M)$:

$$\begin{aligned} L_p(M) &= \left\{x \in \Sigma^* : (q_0, x, Z_0) \mapsto_M^* (q, \varepsilon, \varepsilon), q \in Q\right\} \\ L_F(M) &= \left\{x \in \Sigma^* : (q_0, x, Z_0) \mapsto_M^* (q, \varepsilon, \gamma), \gamma \in R^*, q \in F\right\} \end{aligned}$$

Proposizione 7.3. *Per ogni APND* M*, esiste un APND* M' *tale che* $L_F(M) = L_p(M')$.

Nel seguito, quando l'APND è previsto per riconoscere le stringhe per pila vuota, assumeremo $F = \varnothing$.

Esempio 7.4. Consideriamo l'APND che riconosce, per pila vuota, il linguaggio xcx^r, dove $c \in \{a, b\}^*$ e x^r è la stringa inversa della stringa x (ad esempio, se $x = aab$ allora $x^r = baa$). L'APND è definito nel modo seguente:

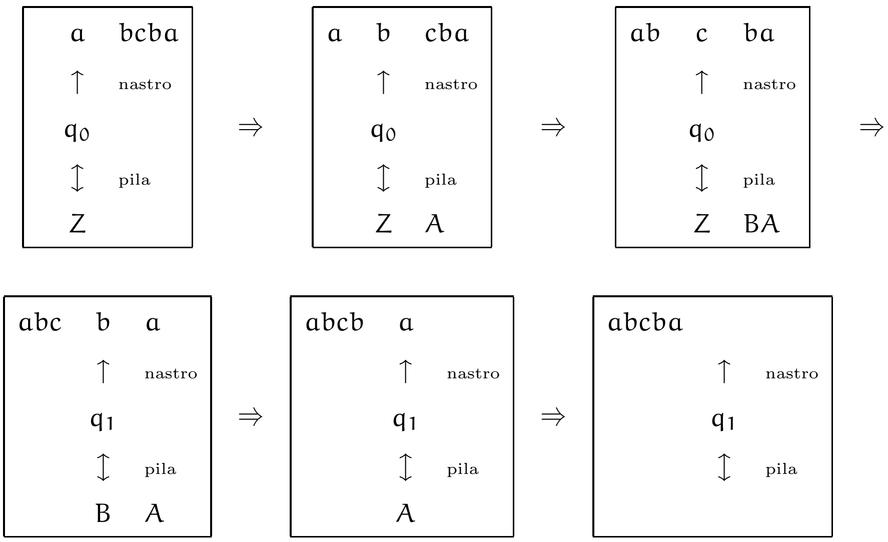
$$\langle \{q_0, q_1\}, \{a, b, c\}, \{Z, A, B\}, \delta, q_0, Z, \varnothing \rangle$$

Dove δ risulta definita dalla seguente matrice (le celle vuote possono essere definite in qualunque modo che conduca sicuramente ad una refutazione):

q_0	ε	a	b	c
Z		q_0, ZA	q_0, ZB	q_1, ε
A				
B				

q_1	ε	a	b	c
Z		q_1, Z	q_1, Z	
A		q_1, ε	q_1, Z	q_1, Z
B		q_1, Z	q_1, ε	q_1, Z

Ad esempio, il riconoscimento per pila vuota della stringa $abcba$ avviene come segue:



Si osserva che il precedente APND è in realtà deterministico. Al contrario di quanto accade per gli automi a stati finiti gli automi a pila non-deterministici sono strettamente più potenti di quelli deterministici. Nel caso del linguaggio dell'esempio precedente, questo è facilmente riconoscibile da un automa a pila deterministico in quanto esso è costituito da tutte e sole le stringhe del tipo xcx^r. La presenza del simbolo ausiliario c a delimitare le due parti simmetiche della stringa, permette di avere sulla stringa stessa, un elemento di demarcazione tra le fasi di caricamento della pila (la lettura della stringa x) ed il suo scaricamento (la lettura della stringa x^r). In questo modo, il riconoscimento della stringa avviene in modo deterministico: non è necessario attivare nessun processo parallelo di verifica alla ricerca del punto mediano della stringa. Al contrario, ciò non è possibile nel riconoscimento del linguaggio CF delle stringe palindrome. In questo caso, per determinare se la lettura del simbolo corrente sul nastro corrisponde alla lettura del simbolo mediano della stringa, da cui inizia lo scaricamento della pila, ci appoggiamo ad una computazione non-deterministica.

È importante notare che gli automi a pila deterministici sono solitamente utilizzati per il riconoscimento sintattico dei linguaggi di programmazione. In particolare, essi costituiscono la base per la realizzazione di compilatori (fase di parsing). A tal proposito, si noti come la sintassi degli usuali linguaggi di programmazione sia strutturata in modo tale da interporre tra categorie sintattiche dei simboli o sequenze di simboli marcatori, ad esempio `if`, `while`, `then`, etc., come nel caso dell'Esempio 7.4. Questo permette di realizzare la fase di parsing del linguaggio attraverso un APD.

Esempio 7.5. Costruiamo un APND che riconosca il linguaggio delle stringhe palindrome sull'alfabeto $\{a, b\}$: $L = \{ww^r : w \in \{a, b\}^*\}$. L'automa è definito come segue:

- $Q = \{q_0, q_1\}$
- $\Sigma = \{a, b\}$
- $R = \{Z, A, B\}$

e relazione di transizione:

q_0	ε	a	b
Z		q_0, AZ	q_0, BZ
A		q_0, AA	q_0, BA
		q_1, ε	
B		q_0, AB	q_0, BB
			q_1, ε

q_1	ε	a	b
Z	q_1, ε		
A		q_1, ε	
B			q_1, ε

Esercizio 7.6. Scrivere le derivazioni necessarie per riconoscere la stringa: $abbba$.

Lemma 7.7. *Se* $L = L_p(M)$ *con* M *APND, allora* $L = L_p(M')$ *con* M' *APND con 1 solo stato.*

Proof. (Idea). Sia $M = \langle Q, \Sigma, R, \delta, q_0, Z_0, F\rangle$. L'idea è quella di aumentare i simboli nella pila per descrivere ogni coppia (q_i, Z_i) con $q_i \in Q$ e $Z_i \in R$. Per ogni coppia di questo tipo, si aggiunge un nuovo simbolo Z_i' in R e si modifica δ in modo tale che se Z' è il simbolo nuovo in corrispondenza della coppia (q, Z), allora: $\delta(q, a, Z) = \delta'(q_0, a, Z')$. Il lemma segue quindi banalmente per costruzione. □

I risultati seguenti dimostrano l'equivalenza tra i linguaggi riconosciuti dagli APND e i linguaggi CF.

TEOREMA 7.8. *Se* $L = L_p(M)$ *con* M *APND con 1 stato, allora* L *è CF.*

PROOF. Sia $M = \langle \{q\}, \Sigma, R, \delta, q, Z_0, \varnothing \rangle$ un APND con 1 solo stato che riconosce il linguaggio $L_p(M)$. Definiamo la grammatica libera da contesto: $G = \langle R, \Sigma, Z_0, P \rangle$, avente R come alfabeto di simboli non terminali, $Z_0 \in R$ come simbolo iniziale e le seguenti produzioni:

$$Z \to aZ_1Z_2\cdots Z_n \text{ se } (q, Z_1Z_2\cdots Z_n) \in t(q, a, Z)$$

È ovvio che se $\alpha \in R^*$, allora:

$$(q, a, Z\alpha) \mapsto_M (q, \varepsilon, Z_1Z_2\cdots Z_n\alpha) \text{ sse } Z\alpha \overset{G}{\Rightarrow} aZ_1Z_2\cdots Z_n\alpha$$

Per induzione si dimostra che per ogni $x \in \Sigma^*$:

$$(q, x, Z_0) \mapsto_M^* (q, \varepsilon, \varepsilon) \text{ sse } Z_0 \overset{G}{\Rightarrow} x$$

da cui segue la tesi. $\square$

Il seguente teorema stabilisce l'equivalenza tra i linguaggi riconosciuti dagli APND ed il linguaggi CF. Prima di vedere questo risultato, osserviamo l'analogia tra la derivazione da una grammatica CF ed i passi di caricamento/svuotamento della pila durante le varie fasi di riconoscimento delle sottostringhe di una data stringa da parte di un APND.

ESEMPIO 7.9. Consideriamo la seguente grammatica G in forma normale di Greibach.

$$\begin{array}{lcl@{\qquad\qquad}lcl} S & \to & aB & A & \to & a \\ S & \to & bA & B & \to & bS \\ A & \to & aS & B & \to & aBB \\ A & \to & bAA & B & \to & b \end{array}$$

Sia M l'APND con 1 solo stato q definito dalla seguente relazione di transizione:

q	ε	a	b
S		q, B	q, A
A		q, ε q, S	q, AA
B		q, BB	q, ε q, S

Si noti come le seguenti derivazioni siano del tutto simmetriche:

TEOREMA 7.10. *Sia* L *CF. Allora esiste un APND* M *tale che* $L = L_p(M)$.

PROOF. Sia $L = L(G)$ con $G = \langle V, T, P, S \rangle$ grammatica CF in forma normale di Greibach. Definiamo:

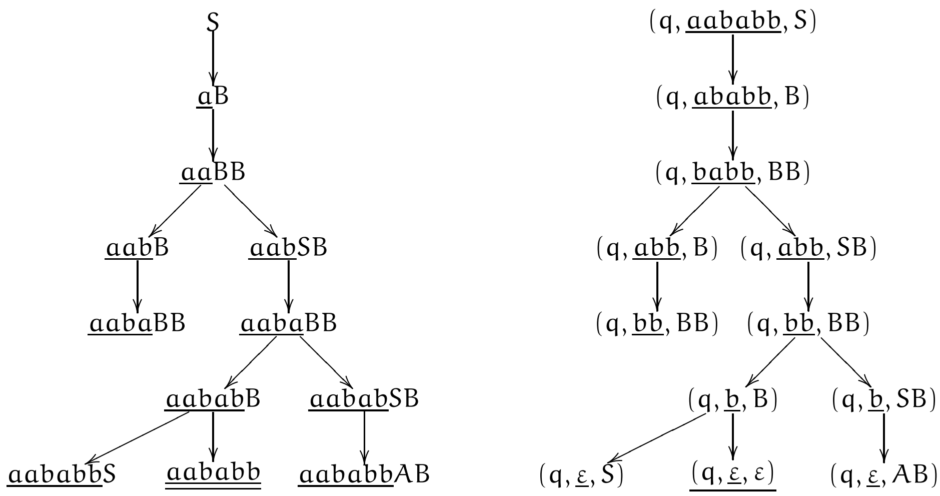


FIGURE 2. Simmetria tra albero di derivazione e APND.

- $Q := \{q\}$
- $\Sigma := T$
- $R := V$
- $Z_0 := S$
- $F := \varnothing$

e dove:

$$(q, \alpha) \in t(q, a, A) \text{ sse } A \to a\alpha \in P$$

Si osserva che:

$$xA\beta \overset{G}{\Rightarrow}^1 xa\alpha\beta \text{ dove } \alpha, \beta \in V^* \text{ e } x \in T^*$$

Ma questo avviene se e solo se l'APND così definito fa una mossa del tipo:

$$(q, a, A\beta) \mapsto^1_M (q, \varepsilon, \alpha\beta)$$

Per induzione [completare per esercizio] sul numero di passi della derivazione $\overset{G}{\Rightarrow}$ si dimostra che:

$$xA\beta \overset{G}{\Rightarrow}^n xy\alpha\beta \text{ sse } (q, y, A\beta) \mapsto^n_M (q, \varepsilon, \alpha\beta)$$

Ne consegue che: $S \overset{G}{\Rightarrow}^* y$ sse $(q, y, S) \mapsto^*_M (q, \varepsilon, \varepsilon)$. □

CHAPTER 8

Proprietà dei linguaggi liberi dal contesto

Per i linguaggi CF valgono delle proprietà simili a quelle studiate per i linguaggi regolari nel Capitolo 5.

1. Il pumping lemma per i linguaggi CF

LEMMA 8.1 (Bar-Hillel, Perles, Shamir, 1961). *Sia* L *un linguaggio CF. Allora c'è una costante* $n \in \mathbb{N}$ *(dipendente dal linguaggio* L*) tale che per ogni* $z \in L, |z| \geq n$*, esistono stringhe* $uvwxy$ *tali che*

(1) $z = uvwxy$,
(2) $|vx| \geq 1$,
(3) $|vwx| \leq n$, *e*
(4) *per ogni* $i \geq 0$ *vale che* $uv^i wx^i y \in L$.

PROOF. Sia $G = \langle V, T, P, S \rangle$ la grammatica che genera $L \setminus \{\varepsilon\}$. Senza perdita di generalità, possiamo assumere che G sia in forma normale di Chomsky (se $\varepsilon \in L$, potremmo costruire una nuova grammatica G' che estende G con un nuovo simbolo iniziale S' e le produzioni $S' \to \varepsilon$ e $S' \to S$ e modificare appena la dimostrazione—si noti che G' non è in forma normale. Poiché il significato del lemma è per stringhe "lunghe", non complicheremo la dimostrazione in questo modo).

Per induzione su $i \geq 1$, si mostra (esercizio) che se un albero di derivazione per una stringa $z \in T^*$ ha tutti i cammini di lunghezza minore o uguale a i, allora $|z| \leq 2^{i-1}$.

Sia $|V| = k$ $(k > 0)$ e sia $n = 2^k$ $(n > 1)$. Se $z \in L$ e $|z| \geq n$, allora ogni albero di derivazione per z deve avere un cammino di lunghezza almeno $k+1$. Ma tale cammino ha almeno $k+2$ nodi, tutti, eccetto l'ultimo, etichettati da variabili. Pertanto vi deve essere una variabile ripetuta nel cammino.

In particolare, dunque, vi devono essere due vertici v_1, v_2 tali che:

(1) entrambi sono etichettati dalla stessa variabile, diciamo A;
(2) v_1 è più vicino alla radice di v_2;
(3) poiché $n \geq 2$, a v_1 è associata una produzione del tipo: $A \to BC$;
(4) da v_1 alla foglia al più la lunghezza è $k+1$.

Si prenda ora il sottoalbero T_1 con radice in v_1; esso rappresenta una derivazione di una parola z_1 di lunghezza al più 2^k. Sia z_2 invece la parola relativa al sottoalbero T_2 di T_1 con radice in v_2. Possiamo scrivere $z_1 = vz_2x$. Inoltre almeno una tra v e x deve essere non vuota, in quanto a v_1 è associata una produzione del tipo: $A \to BC$ ed il sottoalbero T_2 è un sottoalbero di esattamente uno dei due sottoalberi individuati da B e C.

Allora abbiamo che:

$$A \overset{G}{\Rightarrow}_* vAx \quad \text{e} \quad A \overset{G}{\Rightarrow}_* z_2$$

ove $|vz_2x| \leq 2^k = n$. Ma allora ne segue anche che

$$A \overset{G}{\Rightarrow}_* v^i z_2 x^i$$

per ogni $i \geq 0$. Si ponga $w = z_2$ e u e y il prefisso e il suffisso di z necessari affinchè $z = uvwxy$. $\square$

ESEMPIO 8.2. Il linguaggio $\{a^i b^i c^i : i \geq 1\}$ non è CF. Supponiamo per assurdo che lo sia. Come fatto nel caso dei linguaggi regolari (Lemma 5.1), dobbiamo mostrare la veridicità del complemento della formula dimostrata nel lemma, ovvero che per ogni $n \geq 1$ sappiamo trovare una stringa $z \in L$, $|z| \geq n$ che soddisfa una certa condizione. Scegliamo $z = a^n b^n c^n$. Dobbiamo mostrare che per ogni quintupla di stringhe u, v, w, x, y tali che $a^n b^n c^n = uvwxy$, $|vx| \geq 1$ e $|vwx| \leq n$ esiste un $i \geq 0$ tale per cui $uv^i wx^i y \notin L$.

Vi sono molti modi di partizionare z in $uvwxy$ che soddisfano i requisiti. Tuttavia questi si possono raggruppare nelle seguenti casistiche:

(1) $uvwx = a^h$ con $h \leq n$
(2) $u = a^h$, $vwx = a^k b^m$ con $k > 0$, $k + m \leq n$
(3) $u = a^n b^h$, $vwx = b^m$ con $m \leq n$
(4) $u = a^n b^h$, $vwx = b^k c^m$ con $k + m \leq n$
(5) $u = a^n b^n c^h$, $vwx = c^m$ con $m \leq n$

In ciascuna delle casistiche, tuttavia, si ha che $uv^0 wx^0 y = uwy \notin L$, in quanto vengono alterati al massimo due tra i numeri di ripetizioni dei tre tipi di carattere a, b e c.

ESERCIZIO 8.3. Si dimostri che i seguenti linguaggi non sono liberi dal contesto:

(1) $\{0^a 1^b 0^{ab} : a, b \in \mathbb{N}, a, b > 0\}$
(2) $\{0^a 1^b 0^{a^b} : a, b \in \mathbb{N}, a, b > 0\}$
(3) $\{0^a 1^b 0^a 1^b : a, b \in \mathbb{N}, a, b > 0\}$

2. Proprietà di chiusura

TEOREMA 8.4. *I linguaggi CF sono chiusi rispetto all'unione, alla concatenazione, ed alla chiusura di Kleene.*

PROOF. La dimostrazione è costruttiva. Siano $G_1 = \langle V_1, T_1, P_1, S_1 \rangle$ e $G_2 = \langle V_2, T_2, P_2, S_2 \rangle$ le grammatiche generanti i linguaggi L_1 e L_2. Per semplicità si assuma che V_1 e V_2 siano disgiunti (altrimenti si proceda a ridenominare le variabili). Sia S una nuova variabile.

Allora

- $G_\cup = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup P, S\rangle$, ove P consta delle produzioni:
$$S \to S_1 | S_2$$
è la grammatica che genera $L_1 \cup L_2$.
- $G_\circ = \langle V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup P, S\rangle$, ove P consta delle produzioni:
$$S \to S_1 S_2$$
è la grammatica che genera $L_1 L_2$.
- $G_* = \langle V_1 \cup \{S\}, T_1, P_1 \cup P, S\rangle$, ove P consta delle produzioni:
$$S \to \varepsilon | S_1 S$$
è la grammatica che genera L_1^*.

□

TEOREMA 8.5. *I linguaggi CF non sono chiusi rispetto all'intersezione.*

PROOF. Consideriamo $L_1 = \{a^i b^i c^j : i \geq 1, j \geq 1\}$ generato da:
$$S \to RC, R \to ab \mid aRb, C \to c \mid cC$$
e $L_2 = \{a^i b^j c^j : i \geq 1, j \geq 1\}$ generato da:
$$S \to AR, R \to bc \mid bRc, C \to c \mid cC$$
La loro intersezione è $\{a^i b^i c^i : i \geq 1\}$, mostrato essere non CF nell'esempio 8.2. □

COROLLARIO 8.6. I linguaggi CF non sono chiusi rispetto alla complementazione.

PROOF. Se lo fossero, allora lo sarebbero anche rispetto all'intersezione, in quanto $A \cap B = \overline{\overline{A} \cup \overline{B}}$, contraddicendo il Teorema 8.5. □

3. Algoritmi di decisione

TEOREMA 8.7 (Vuoto, Finito, Infinito). *Data una grammatica CF* $G = \langle V, T, P, S\rangle$, *i problemi:*

(1) $L(G) = \varnothing$,
(2) $L(G)$ *è finito,*
(3) $L(G)$ *è infinito,*

sono decidibili.

PROOF. 1) Come osservato nel paragrafo 5.1, se la grammatica ottenuta dall'applicazione dell'algoritmo presente nella dimostrazione del Lemma 6.9 è tale che $S \notin V'$, allora il linguaggio è $\varnothing$. Altrimenti no.

2,3) Se $L(G) = \varnothing$ oppure $L(G) = \{\varepsilon\}$ (entrambe proprietà che si dimostrano durante la normalizzazione) il linguaggio è chiaramente finito. Altrimenti, $G' = \langle V', T, P', S\rangle$ una grammatica t.c. $L(G') = L(G) \setminus \varepsilon$ in forma normale di Chomsky e priva di simboli inutili. Creiamo un grafo $\mathcal{G}$ avente:

- un nodo per ogni variabile $A \in V'$;
- un arco $\langle A, B\rangle$ ed un arco $\langle A, C\rangle$ per ogni produzione $A \to BC \in P'$.

Allora $L(G)$ $(= L(G'))$ è finito se e solo se $\mathcal{G}$ non ha cicli.

Se $\mathcal{G}$ non ha cicli, la finitezza è immediata: solo un numero finito di alberi di derivazione con radice etichettata da S possono essere costruiti.

Viceversa, supponiamo $\mathcal{G}$ abbia (almeno) un ciclo passante per un nodo associato ad una variabile A. Allora riusciamo a costruire un albero con un cammino dalla radice verso le foglie (qui è importante l'ipotesi che la grammatica non abbia simboli inutili) che attraversa due nodi etichettati con A. Ripetendo la dimostrazione del pumping lemma, si riescono a generare infiniti alberi di derivazione diversi. □

TEOREMA 8.8 (Appartenenza). *Data una grammatica CF* $G = \langle V, T, P, S\rangle$, *e una stringa* z, *il problema* $z \in L(G)$ *è decidibile.*

PROOF. Per il caso $z = \varepsilon$, dal Teorema 6.13 si ha che $\varepsilon \in L$ sse esiste $S \to A_1 \cdots A_n$ $(n \geq 0)$ in P tale per cui $A_i \in N$ per $i = 1, \dots, n$.

Sia $z \neq \varepsilon$ e $G' = \langle V', T, P', S\rangle$ la grammatica equivalente a G in forma normale di Greibach. Allora, se z ha una derivazione, ne ha una di esattamente $|z|$ passi. Generiamo esaustivamente tutte le derivazioni di $|z|$ passi e verifichiamo se esiste una di queste che deriva z. □

L'algoritmo impiegato nella dimostrazione del Teorema 8.8 ha complessità esponenziale. Fortunatamente per chi necessita di riconoscere o refutare stringhe (ad esempio, per capire se un programma Pascal o C contiene degli errori sintattici) può avvalersi di metodi più efficienti. In particolare, l'algoritmo di Cocke-Younger-Kasami (1965–67) che è $O(|z|^3)$, oppure l'algoritmo di Valiant (1975) evente complessità $O(|z|^{2.8})$.

Si osservi che nel caso di linguaggi regolari invece tale problema viene deciso in tempo $O(n)$. Infatti, dato il DFA che accetta il linguaggio, la computazione avviene in tempo proporzionale alla lunghezza della stringa.

Per quanto riguarda il problema dell'equivalenza di due linguaggi liberi dal contesto, il problema è mostrato essere indecidibile in [**13**], Teorema 8.12.

Il problema corrispondente nel caso dei linguaggi regolari è invece decidibile (si confronti il Teorema 5.13) ed è dimostrato essere NL-completo in [**14**].

CHAPTER 9

Le grammatiche regolari e la gerarchia di Chomsky

In questo capitolo si studieranno le famiglie di grammatiche CF che generano esattamente i linguaggi regolari. Saranno dunque collocate in un contesto più ampio: riassumeremo brevemente dei risultati (alcuni dei quali presenti in questo testo) che permettono di classificare le grammatiche in base all'espressività dei linguaggi generabili e presenteremo la classificazione insiemistica completa delle grammatiche.

1. Grammatiche Regolari

Una grammatica CF si dice *lineare destra* se ogni produzione è della forma:

$$\begin{array}{lll} A & \to & wB \quad w \in T^+, \text{ oppure} \\ A & \to & w \quad\;\;\, w \in T^+ \end{array}$$

più eventualmente $S \to \varepsilon$. Se invece tutte le produzioni sono della forma:

$$\begin{array}{lll} A & \to & Bw \quad w \in T^+, \text{ oppure} \\ A & \to & w \quad\;\;\, w \in T^+ \end{array}$$

più eventualmente $S \to \varepsilon$, si dice *lineare sinistra*.

LEMMA 9.1. *Data una grammatica lineare destra* G *esiste una grammatica* G' *equivalente (in forma normale di Greibach) tale che tutte le produzioni sono della forma:*

$$\begin{array}{lll} A & \to & aB \quad a \in T, \textit{ oppure} \\ A & \to & a \quad\;\;\, a \in T \end{array}$$

più eventualmente $S \to \varepsilon$.

PROOF. Ogni produzione della forma $A \to a_1 a_2 \cdots a_n B$ viene sostituita dalle produzioni: $A \to a_1 C_1, C_1 \to a_2 C_2, \ldots, C_{n-1} \to a_n B$, ove $C_1, \ldots, C_n$ sono nuovi simboli non terminali. Similmente, $A \to a_1 a_2 \cdots a_n$ viene sostituita dalle produzioni: $A \to a_1 C_1, C_1 \to a_2 C_2, \ldots, C_{n-1} \to a_n$. □

TEOREMA 9.2. *Se* L *è generato da una grammatica lineare destra, allora* L *è un linguaggio regolare.*

PROOF. Grazie al Lemma 9.1 possiamo supporre che $G = \langle V, T, P, S\rangle$ tale che $L(G) = L$ sia nella forma semplificata descritta nell'enunciato di tale lemma. Costruiremo un NFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ che riconosce L.

- $Q = V \cup \{\perp\}$;
- $\Sigma = T$;
- $B \in \delta(A, a)$ sse $A \to aB \in P$, $\perp \in \delta(A, a)$ sse $A \to a \in P$; $\delta(\perp, a)$ è indefinito per ogni simbolo $a \in \Sigma$.
- $q_0 = S$;
-

$$F = \begin{cases} \{\perp, S\} & \text{se } S \to \varepsilon \in P \\ \{\perp\} & \text{altrimenti} \end{cases}$$

Si deve mostrare che $\hat{\delta}(S, w) \cap F \neq \varnothing$ se e solo se $S \overset{G}{\Rightarrow}_* w$, per ogni stringa w.

Iniziamo mostrando, per induzione su $|w| \geq 0$, che per ogni $A \in V$ (dunque $A \neq \perp$)

$$A \in \hat{\delta}(S, w) \quad \text{sse} \quad S \overset{G}{\Rightarrow}_* wA$$

Base: Sia $w = \varepsilon$. Per definizione, $\hat{\delta}(S, \varepsilon) = \{S\}$. Per la forma della grammatica $S \overset{G}{\Rightarrow}_* \varepsilon A$ sse $S = A$.
Passo: Sia $w = va$.

$$\begin{array}{llll} A \in \hat{\delta}(S, va) & \text{sse} & A \in \bigcup_{B \in \hat{\delta}(S,v)} \delta(B, a) & \text{def di } \hat{\delta} \text{ nei NFA} \\ & \text{sse} & A \in \bigcup_{B\,:\,S \overset{G}{\Rightarrow}_* vB} \delta(B, a) & \text{ip. induttiva} \\ & \text{sse} & S \overset{G}{\Rightarrow}_* vaA & \text{def di } \delta \ (B \longrightarrow aA \in P) \end{array}$$

Da questo risultato, per definizione di F, si ha che:

(1) $S \overset{G}{\Rightarrow}_* \varepsilon$ se e solo se $\hat{\delta}(S, \varepsilon) \cap F \neq \varnothing$;
(2) $S \overset{G}{\Rightarrow}_* va$ se e solo se esiste A tale che $S \overset{G}{\Rightarrow}_* vA$ e $A \to a \in P$. Ciò accade se e solo se $A \in \hat{\delta}(S, v)$ e $\perp \in \delta(A, a)$, ovvero $\perp \in \hat{\delta}(S, va)$.

$\square$

TEOREMA 9.3. *Se* L *è un linguaggio regolare, allora* L *è generato da una grammatica lineare destra.*

PROOF. Sia $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ il DFA che riconosce L. Costruiamo una grammatica lineare destra $G = \langle V, T, P, S\rangle$ tale che $L(G) = L$ nel modo seguente:

- $V = Q$;
- $T = \Sigma$;

- $$\begin{cases} A \to aB & \in & P & \text{sse } \delta(A, a) = B \\ A \to \varepsilon & \in & P & \text{sse } A \in F \end{cases}$$
- $S = q_0$.

Si mostra per induzione su $|w|$ che $\hat{\delta}(q_0, w) = p$ se e solo se $q_0 \overset{G}{\Rightarrow}_{|w|} wp$ [Esercizio]. A questo punto si ha immediatamente che $\hat{\delta}(q_0, w) \in F$ se e solo se $q_0 \overset{G}{\Rightarrow}_{|w|} w$.

Per costruzione, la grammatica ottenuta può avere ε-produzioni e dunque non è della forma desiderata. Si applichi dunque l'eliminazione di tali ε-produzioni per ottenere la grammatica nella forma voluta. □

Il Teorema 9.3 implica che ogni linguaggio regolare è anche CF.

Si possono mostrare risultati analoghi ai teoremi 9.2 e 9.3 per le grammatiche lineari sinistre (si veda [**13**]).

2. Grammatiche di tipo 0

Le *grammatiche di tipo 0* (a struttura di frase) sono le grammatiche della forma $G = \langle V, T, P, S \rangle$ in cui P contiene produzioni

$$\alpha \to \beta$$

ove $\alpha \in (V \cup T)^+, \beta \in (V \cup T)^*$.

Teorema 9.4. *$L = L(G)$ per G grammatica a struttura di frase se e solo se L è un insieme ricorsivamente enumerabile.*

Proof. (Si veda [**13**]) □

3. Grammatiche di tipo 1

Le *grammatiche di tipo 1* (dipendenti dal contesto) sono le grammatiche della forma $G = \langle V, T, P, S \rangle$ in cui P contiene produzioni

$$\alpha \to \beta$$

ove $\alpha \in (V \cup T)^+, \beta \in (V \cup T)^+$, e $|\alpha| \leq |\beta|$, con l'eccezione dell'eventuale produzione $S \longrightarrow \varepsilon$ (in tal caso richiediamo che S non occorra mai in β).

Per queste grammatiche esiste una forma normale per le produzioni, che devono essere del tipo:

$$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$$

con $\beta \neq \varepsilon$.

Teorema 9.5. *Ogni linguaggio L generato da una grammatica dipendente dal contesto è ricorsivo.*

Proof. (Si veda [**13**]) □

Tuttavia:

TEOREMA 9.6. *Ci sono insiemi ricorsivi che non sono generati da nessuna grammatica dipendente dal contesto.*

PROOF. (Si veda [**13**]) □

Nell'Esempio 8.2 abbiamo mostrato che $\{a^i b^i c^i : i \geq 1\}$ non è un linguaggio CF. Mostreremo ora che tale linguaggio è un linguaggio dipendente dal contesto.

ESEMPIO 9.7. La grammatica di tipo 1:

$$\begin{array}{rclcrcl} S & \rightarrow & aSBC \mid aBC & \quad & bC & \rightarrow & bc \\ CB & \rightarrow & BC & & cC & \rightarrow & cc \\ bB & \rightarrow & bb & & aB & \rightarrow & ab \end{array}$$

genera esattamente il linguaggio $\{a^i b^i c^i : i \geq 1\}$.

4. Gerarchia

Ricordiamo:

- dagli esempi 5.3 e 6.4 sappiamo che il linguaggio $\{a^i b^i : i \geq 0\}$ è un linguaggio CF ma non regolare. Sappiamo inoltre che le grammatiche (CF) lineari destre e sinistre generano esattamente i linguaggi regolari.
- Dagli esempi 8.2 e 9.7 sappiamo che il linguaggio $\{a^i b^i c^i : i \geq 1\}$ è un linguaggio dipendente dal contesto ma non CF.
- Dai Teoremi 9.6 e 9.4 sappiamo che la classe di linguaggi generati da grammatiche dipendenti dal contesto è inclusa strettamente nella classe di linguaggi generati da grammatiche a struttura di frase.

Pertanto viene indotta una gerarchia di grammatiche, nota come *gerarchia di Chomsky*, fatta di inclusioni proprie, che si può trovare in Figura 1.

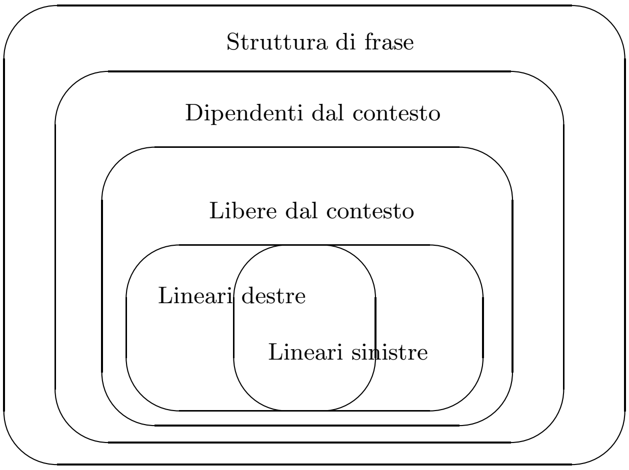


Figure 1. La gerarchia di Chomsky

Part 2

Teoria della calcolabilità

CHAPTER 10

Nozione intuitiva di algoritmo

In questa parte del testo presenteremo alcuni sistemi formali introdotti per studiare in modo rigoroso i concetti di algoritmo, programma e funzione calcolabile. In generale, un *sistema formale* $\mathcal{S}$ è un sistema di formule e regole per combinare formule, descritto mediante un insieme finito o numerabile S di simboli, e tale che per ogni regola $\mathsf{R} \subseteq \mathsf{S}^k$ per un qualche $k \in \mathbb{N}$, sia possibile stabilire in modo finito (decidibile) se una formula f è *conseguenza* di $f_1, \ldots, f_{k-1}$ secondo R, ovvero se $\langle f_1, \ldots, f_{k-1}, f \rangle \in \mathsf{R}$. Il senso dei sistemi formali è quello di definire in modo rigoroso, mediante simboli e regole, l'impalcatura entro cui derivare asserzioni su un dato modello della realtà. Il calcolo proposizionale e la logica del prim'ordine sono esempi noti di sistemi formali.

Studieremo dunque alcuni sistemi formali per descrivere in modo rigoroso la nozione intuitiva di calcolabilità. Ciò permetterà di stabilire al tempo stesso i limiti intrinseci dell'informatica moderna e studiare alcuni formalismi per rappresentare il "calcolo", apparentemente lontani tra loro, ma equivalenti nel potere espressivo riguardante ciò che è calcolabile.

La corrente Parte 2 del testo, che riguarda questi argomenti, è strutturata nel modo seguente: nel par. 1 del presente capitolo sarà introdotta la nozione informale ed intuitiva di algoritmo, o di funzione calcolabile. In seguito, nel Capitolo 11 sarà presentata la nozione di Macchina di Turing (MdT in breve), come primo sistema formale per definire il concetto di funzione calcolabile. L'importanza della MdT è, oltre che culturale, essendo una delle prime formalizzazioni del concetto di calcolabilità, anche tecnica: ad esempio, mediante MdT sono attualmente formalizzate le nozioni più importanti nella teoria moderna della complessità degli algoritmi che studieremo nella parte 4 del testo. Nel Capitolo 12 introdurremo le funzioni parziali ricorsive, come modello matematico per le funzioni calcolabili. L'equivalenza tra questi due modelli di calcolo ci porterà, nel Capitolo 13, all'analisi della *Tesi di Church-Turing* e successivamente, nei Capitoli 14 e 15 ad enunciare alcuni risultati generali sulla calcolabilità ed alcuni problemi algoritmicamente insolubili.

1. Requisiti di un algoritmo

Discuteremo ora le caratteristiche che deve avere un algoritmo partendo da quella che è l'esperienza e l'idea intuitiva che tutti ne abbiamo. Un algoritmo viene descritto in un certo linguaggio, che può anche essere semplicemente l'italiano, così come usando i linguaggi nati appositamente per descrivere algoritmi quali i linguaggi di programmazione e i sistemi formali che vedremo nei prossimi capitoli. Facciamo qui riferimento all'esperienza personale e immaginiamo di guardare un algoritmo. Possiamo concordare che, indipendentemente dal problema che intende risolvere, ha delle caratteristiche comuni.

- **a:** Un algoritmo è di lunghezza finita.
- **b:** Esiste un agente di calcolo che porta avanti il calcolo eseguendo le istruzioni dell'algoritmo.
- **c:** L'agente di calcolo ha a disposizione una memoria dove vengono immagazzinati i risultati intermedi del calcolo.
- **d:** Il calcolo avviene per passi discreti.
- **e:** Il calcolo non è probabilistico.

I punti **a**–**c** hanno una ovvia interpretazione. Il punto **d** afferma che il calcolo non avviene mediante dispositivi analogici. Il punto **e** afferma che il calcolo non obbedisce a nessuna legge di probabilità. Associato al punto **a** parleremo di *espressioni simboliche* ovvero espressioni in un linguaggio di simboli che costituisce il linguaggio per decrivere un algoritmo.

Altre caratteristiche degli algoritmi sono:

- **f:** Non deve esserci alcun limite finito alla lunghezza dei dati di ingresso.
- **g:** Non deve esserci alcun limite alla quantità di memoria disponibile.

Mentre il punto **f** è ragionevole: ad esempio un algoritmo di somma deve poter funzionare per ogni possibile addendo, ovvero numero naturale, per il punto **g** è necessario un chiarimento. Per evidenziare la necessità di assumere una memoria illimitata facciano osservare che, limitandola, alcuni algoritmi noti per calcolare semplici funzioni non potrebbero funzionare. Ad esempio la funzione $\lambda x.\ x^2$ non sarebbe calcolabile poiché lo spazio di memoria necessario per calcolare il quadrato di x dipende da x, e per il punto **f**, esso deve essere illimitato.

Le seguenti osservazioni sono essenziali per comprendere la natura del calcolo e la sua complessità.

- **h:** Deve esserci un limite finito alla complessità delle istruzioni eseguibili dal dispositivo.
- **i:** Sono ammesse esecuzioni con un numero di passi finito ma illimitato.

Il punto **h**, in relazione al punto **a**, stabilisce la intrinseca finitezza del dispositivo di calcolo. Ad esempio, pensando ad un calcolatore, esso sarà in grado di calcolare solo indirizzando direttamente una parte finita della sua memoria (registri o memoria RAM) stabilendo quindi un limite nella complessità delle istruzioni eseguibili. Questo è un punto chiave nell'analisi che vedremo della nozione di effettiva calcolabilità. Un dispositivo di calcolo può effettivamente tenere traccia solo di un numero finito di simboli, ovvero esso può reagire (eseguendo un istruzione) tenendo conto di un numero finito di simboli ricordati. Questo venne intuitivamente giustificato da Turing che per primo analizzò formalmente il concetto di effettiva calcolabilità, con l'intrinseca limitazione della memoria umana. Tuttavia, non c'è limite alla memoria ausiliaria, come visto nel punto **g**. Per quanto riguarda il punto **i**, osserviamo che non essendo limitabile il numero di passi richiesti per eseguire un generico algoritmo (si pensi alla moltiplicazione o alla funzione $\lambda x.\ x^2$ discussa nel punto **g**), non è possibile stabilire a priori un limite massimo sul numero di passi nell'esecuzione di un algoritmo. Uno studio approfondito su come legare il numero di passi alla lunghezza dei dati è argomento trattato nella teoria della *complessità* degli algoritmi (si veda la Parte 4 del testo).

2. Funzioni calcolabili

Secondo quanto appena illustrato, un algoritmo definisce una funzione, ovvero una associazione $\{\mathtt{input}\} \rightsquigarrow \{\mathtt{output}\}$. Questa può essere rappresentata matematicamente come una funzione sui dati manipolati dall'algoritmo. Possiamo pertanto asserire che un algoritmo "calcola" funzioni. Diremo quindi che una funzione f è calcolabile se esiste un algoritmo ed un agente di calcolo tale che ad ogni input x restituisce come risultato del calcolo $f(x)$.

Tuttavia, una definizione intuitiva di questo tipo riguardante ciò che e' intuitivamente calcolabile non è soddisfacente. Supponiamo di trattare solo funzioni sui numeri naturali.[1] Supponiamo di poter descrivere in questo modo tutti i processi di calcolo sui numeri naturali, ovvero che una funzione $f : \mathbb{N} \longrightarrow \mathbb{N}$ sia calcolabile se esistono un algoritmo ed un agente di calcolo come descritti nei punti **a**–**i** sopra, tale che per ogni input $x \in \mathbb{N}$, l'algoritmo resituisce $f(x) \in \mathbb{N}$. Ovvero, secondo queste ipotesi, una funzione è calcolabile se esiste un algoritmo ed un agente di calcolo che terminano restituendo comunque un risultato, per ogni input. Pur non assumendo limiti nel tempo e spazio impiegati dall'agente di calcolo per eseguire i suoi conti, le precedenti ipotesi stabiliscono che ciò che è calcolabile lo sia comunque, per ogni input, a seguito di un insieme potenzialmente illimitato (ma finito) di passi elementari di calcolo eseguiti dall'agente. Ovvero le funzioni considerate come "calcolabili" secondo le ipotesi **a**–**i** sono tutte *totali*, ovvero sempre definite per ogni input.

Essendo gli algoritmi di lunghezza finita ed essendo possibile descrivere l'agente di calcolo mediante un insieme finito di simboli e regole che ne disciplinano il calcolo, è possibile enumerare gli algoritmi e gli agenti di calcolo. Sia dunque P_x l'x-esimo programma-agente che calcola una data funzione g_x. Per le ipotesi **a**–**i** la funzione g_x è totale, ovvero sempre definita. Definiamo la funzione $h(x) = g_x(x) + 1$. È chiaro che la funzione h è calcolabile: tutti noi conosciamo un algoritmo per incrementare un naturale, ed inoltre per ipotesi g_x è calcolabile, essendo calcolata dall'algoritmo-agente P_x. Quindi, essendo h calcolabile, deve esistere un algoritmo-agente che la calcola e che avrà un dato indice nella numerazione. Sia questo indice x_0, ovvero h sia calcolata dal programma-agente P_{x_0} ($h = g_{x_0}$). Si ottiene in

[1] Questa non è una limitazione, poiché, come vedremo in seguito, ogni struttura dati, anche la più complessa, può essere messa in corrispondenza biunivoca con un sottoinsieme dei numeri naturali, modulo una opportuna codifica dei dati in numeri.

questo modo una evidente contraddizione:

$$g_{x_0} = h(x_0) = g_{x_0} + 1$$

Non è possibile che una funzione sempre definita sui naturali sia uguale al successore di se stessa. La funzione h è quindi non calcolabile nel nostro formalismo scelto, che pertanto, essendo h intuitivamente calcolabile, non esprime tutto ciò che è intuitivamente calcolabile.

3. Algoritmi e Programmi

Con le suddette ipotesi si è dimostrato che non si è mai in grado di descrivere tutto ciò che è "intuitivamente calcolabile". Ovvero, sarà sempre possibile costruire funzioni, intuitivamente calcolabili e totali, non esprimibili nel sistema formale adottato. Questo chiaramente rappresenta un limite insormontabile alla nostra teoria, rendendola inadeguata a trattare la nozione di calcolabilità. Abbiamo bisogno di estendere i precedenti punti **a**–**i** con una nuova ipotesi fondamentale: si assumerà che un processo di calcolo possa *non terminare*. Questo corrisponde all'intuizione che, un programma chiaramente non terminante del tipo:

$$\mathit{read}(x);\ \textbf{while}\ \mathit{true}\ \textbf{do}\ \cdots\ \textbf{endw}$$

corrisponde ad una funzione non definita su tutti gli argomenti, per la quale sappiamo pensare ad un programma (quello sopra) ed un agente di calcolo che la calcola (un computer). Si noti ora la necessità di introdurre degli oggetti sintatticamente del tutto simili agli algoritmi, ma senza la proprietà di sicura terminazione. Sono questi i *programmi*.

È quindi necessario, per superare la precedente limitazione dei punti **a**–**i**, assumere che, oltre ai punti **a**–**i** vale la seguente ipotesi:

- **l:** Sono ammesse esecuzioni con un numero di passi infinito (cioè non terminanti).

Il punto **l** rappresenta una necessità intrinseca, e come abbiamo visto irrinunciabile, nella trattazione matematica di ciò che è effettivamente calcolabile. Secondo questa ipotesi, la funzione h descritta in precedenza può risultare indefinita su alcuni argomenti (ad esempio x_0), ovvero indicando con $\uparrow$ il simbolo di indefinito, la precedente uguaglianza può risultare valida (non contraddittoria):

$$\uparrow = g_{x_0} = h(x_0) = g_{x_0} + 1 = \uparrow .$$

CHAPTER 11

Macchine di Turing

In questo capitolo introduciamo le Macchine di Turing (MdT), ideate da Alan Turing negli anni '30 per rappresentare in modo formale una macchina in grado di eseguire algoritmi costituiti da passi elementari e discreti di calcolo, secondo la definizione informale data precedentemente. Turing, nella sua analisi del concetto di calcolabilità, introduce per primo la parola "computer", con il significato di una persona che fa un calcolo composto da passi elementari e discreti utilizzando un foglio di carta (memoria). Questa nozione di computer include tutte le caratteristiche di base dei calcolatori moderni (memoria, input/output, stato) e, come vedremo, modella in modo soddisfacente il concetto di effettiva calcolabilità.

1. Descrizione modellistica e matematica

Nell'eseguire un calcolo (manuale), dato un input, calcoliamo un risultato con l'ausilio della scrittura su carta. L'idea di Turing è quella di pensare ad una semplice macchina in grado di scrivere simboli su un nastro potenzialmente infinito, rappresentando il fatto (anche espresso nel punto **g**) che nei calcoli abbiamo solitamente a disposizione una quantità illimitata di carta dove registrare i risultati parziali del calcolo. L'organizzazione del nastro è fatta mediante *celle* che rappresentano la quantità di memoria unitaria. La macchina così pensata utilizzerà un alfabeto finito, col quale possiamo rappresentare una infinità di dati. Il corpo pensante della macchina (detto controllo) sarà realizzato mediante un *automa a stati finiti deterministico* (DFA). Questa restrizione modella perfettamente il punto **h** precedente, dove è stata assunta una complessità limitata nell'esecuzione delle istruzioni da parte della macchina. Tutto questo viene rappresentato matematicamente con una macchina detta *Macchina di Turing* (MdT in breve).

Una MdT è un dispositivo di calcolo rappresentabile con un *nastro* di lunghezza infinita nel quale vengono immagazzinati i dati o sequenze di simboli del calcolo. Uno di questi simboli è $ rappresentante l'assenza di simboli (spazio bianco). Il controllo della MdT ha accesso al nastro attraverso una *testina* di lettura e scrittura che permette di leggere o scrivere un simbolo alla volta. Una MdT è pertanto costituita da due parti: il programma finito secondo cui verrà eseguito il calcolo e gli organi meccanici per lo scorrimento del nastro e il comando della testina. La struttura a stati finiti della parte di controllo modella una macchina con una memoria che potremmo dire a *breve termine*, finita, mentre il nastro modella una memoria a *lungo termine*, potenzialmente infinita. Questa distinzione tra memoria a breve e lungo termine è riscontrabile anche nelle architetture dei moderni calcolatori.

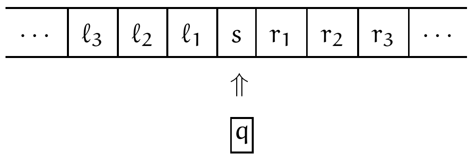


FIGURE 1. Il modello meccanico della Macchina di Turing

Ad ogni istante nel calcolo, il simbolo presente nella casella esaminata dalla testina rappresenta l'input alla macchina (in Figura 1, il simbolo s). In risposta la macchina può decidere di modificare il simbolo e/o spostare il nastro a sinistra o a destra della casella esaminata. Questo permette di avere in input un simbolo diverso per eseguire il *passo* successivo della computazione (il simbolo ℓ_1 se lo spostamento sarà a sinistra, r_1 se a destra). Inoltre la testina (che si trova nello stato q—memoria finita) può cambiare il suo stato scegliendolo da un insieme finito di stati possibili.

Il comportamento di una MdT è descrivibile mediante una tabella detta *matrice funzionale* della macchina in cui le righe rappresentano gli stati del controllo mentre le colonne rappresentano i simboli di ingresso. In una generica cella è descritta l'azione eseguita dalla macchina nello stato corrispondente in riga e leggendo dal nastro il simbolo corrispondente in colonna.

		s_j	
q_i		$q_r\ s_k\ L$	

L'azione compiuta, essendo il controllo nello stato q_i e leggendo il simbolo s_j, è in questo caso:

- portarsi in uno stato (interno) q_r
- scrivere il simbolo s_k
- spostare il nastro a destra R o a sinistra L.

Nel caso $s_j = s_k$, la macchina lascia inalterato il simbolo sul nastro. Se la casella corrispondente alla coppia $\langle s_j, q_i \rangle$ è vuota, la macchina si ferma. Una *computazione* di una MdT sarà dunque una sequenza di passi compiuti dalla macchina.

Più formalmente, definiamo una MdT come segue.

Definizione 11.1. Una *Macchina di Turing* M consiste di:

(1) un alfabeto finito $\Sigma = \{s_0, \dots, s_n\}$ con almeno due simboli distinti $s_0 = \$$ (blank) e $s_1 = 0$ (tally);
(2) un insieme finito di stati $Q = \{q_0, \dots, q_m\}$ tra i quali vi è lo stato iniziale q_0 (dunque Q è non vuoto);
(3) un insieme finito non vuoto di istruzioni (o quintuple) $P = \{I_1, \dots, I_p\}$ ognuna delle quali di uno dei seguenti 2 tipi base:
   - $q\ s\ q'\ s'\ R$
   - $q\ s\ q'\ s'\ L$

   tale che non esistono due istruzioni che iniziano con la medesima coppia q, s.

Si può dunque immaginare P come la descrizione "estensionale" (ovvero per casi) di una funzione (in generale) parziale $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{R, L\}$. Tale funzione δ è detta *funzione di transizione*.

Si osservi la caratteristica *locale* delle istruzioni: il loro significato è univocamente determinato dalla parte di nastro immediatamente adiacente alla testina e dallo stato in cui la macchina si trova. Lo stato in cui si trova l'intera MdT in un certo istante del calcolo è dunque esprimibile mediante la nozione di *descrizione istantanea* di una MdT, che rappresenta lo stato interno del controllo, il posizionamento della testina e il nastro.

Definizione 11.2. Una *descrizione istantanea* (ID) della macchina è una quadrupla

$$\langle q, v, s, w \rangle$$

dove $v, w \in \Sigma^\star$ rappresentano i caratteri significativi (cioè escludendo la sequenza illimitata di \$ sempre presenti a destra e a sinistra) presenti sul nastro a sinistra ed a destra della testina, s è il simbolo letto dalla testina, essendo la macchina nello stato q.

Ad esempio, se la situazione è

$$\begin{array}{c} \dots \$\,\$\,\$\; s_1 \dots s_{i-1} \quad s_i \quad s_{i+1} \dots s_j \;\$\,\$\,\$ \dots \\ \Uparrow \\ \boxed{q} \end{array}$$

allora $v = s_1 \dots s_{i-1}$, $s = s_i$ e $w = s_{i+1} \dots s_j$.

Vediamo ora come definire formalmente il generico passo di calcolo di una MdT. Definiamo una relazione tra descrizioni istantanee, ovvero una relazione $\vdash \subseteq \text{ID} \times \text{ID}$, che rappresenti l'esecuzione di un singolo passo di calcolo in una data MdT.

Definizione 11.3. Il *successore* $\vdash$ è una "mappa" tra descrizioni istantanee definita come:

- $\langle q, v, r, s\,w \rangle \vdash \langle q', v\,r', s, w \rangle$ se $q\ r\ q'\ r'\ R$ è una istruzione di P;

- $\langle q, \nu\, s, r, w\rangle \vdash \langle q', \nu, s, r'\, w\rangle$ se $q\ r\ q'\ r'\ L$ è una istruzione di P.

Una computazione sarà dunque definita come segue, ovvero come una sequenza finita di passi.

DEFINIZIONE 11.4. Una *computazione* è una sequenza *finita* di ID $\alpha_0, \dots, \alpha_n$ tale che

- $\alpha_0 = \nu\ s\ q_0\ w$ e,
- $\alpha_i \vdash \alpha_{i+1}$ per ogni $i \in \{0, \dots, n-1\}$.

Una computazione è detta *terminante* se esiste un $n \geq 0$ per cui $\alpha_n = \nu\ s\ q\ w$ e non vi è nessuna istruzione di P iniziante con q ed s (cioè $\delta(q, s)$ non è definita), i.e. $\alpha_n \not\vdash$.

In accordo con il punto **l** della precedente sezione, esistono MdT che non terminano, ovvero che non calcolano nulla a partire da un dato input.

Osserviamo come una MdT così descritta soddisfa i requisiti per la definizione ragionevole di algoritmo.

a: Le istruzioni della MdT sono finite.
b: La MdT è l'agente di calcolo che esegue l'algoritmo.
c: Il nastro rappresenta la memoria della macchina.
d: La MdT opera in modo discreto.
e: Ad ogni ID corrisponde una sola azione (determinismo della MdT).
f: Non esiste alcun limite all'input, essendo il nastro illimitato.
g: La capacità della memoria (nastro) è illimitata.
h: Le operazioni eseguibili sono semplici e quindi di complessità limitata.
i: Non esiste alcun limite al numero di istruzioni eseguite in quanto la medesima quintupla può essere usata più volte.
l: Possono esistere MdT che non calcolano nulla generando una sequenza infinita di ID.

TEOREMA 11.5 (Determinismo). *Se* $\alpha \vdash \beta$ *e* $\alpha \vdash \gamma$ *con* $\alpha, \beta, \gamma \in$ ID, *allora* $\beta \equiv \gamma$.

PROOF. Ovvio, poiché per definizione non esistono due quintuple $q\ s\ q'\ s'\ X$ aventi la medesima coppia q, s. □

ESEMPIO 11.6. Incremento di uno in binario.
Sia $\Sigma = \{\$, 0, 1\}$. Si vuole passare da una configurazione iniziale

$$\begin{array}{c} \dots \$\,\$\,\$\, s_n \dots s_0 \;\; \overset{\$}{\Uparrow} \;\; \$\,\$\,\$ \dots \\ \boxed{q_0} \end{array}$$

ad una finale

$$\begin{array}{c} \dots \$\,\$\,\$ \;\; \overset{\$}{\Uparrow} \;\; s'_m \dots s'_0\, \$\,\$\,\$ \dots \\ \boxed{q_2} \end{array}$$

ove

$$\sum_{i=0}^{m} 2^i \cdot s'_i = 1 + \sum_{i=0}^{n} 2^i \cdot s_i.$$

Una possibile definizione della funzione δ sarà la seguente:

δ	$\$$	0	1
q_0	$q_1\ \$\ L$		
q_1	$q_2\ 1\ L$	$q_2\ 1\ L$	$q_1\ 0\ L$
q_2		$q_2\ 0\ L$	$q_2\ 1\ L$

Esempio 11.7. Copia di una stringa unaria.
Sia $\Sigma = \{\$, 0\}$. Si vuole passare da una configurazione iniziale

$$\ldots \$\,\$\,\$\ \underbrace{0\ldots0}_{n}\ \underset{\substack{\Uparrow \\ \boxed{q_0}}}{\$}\ \$\,\$\,\$\ldots$$

ad una finale

$$\ldots \$\,\$\,\$\ \underbrace{0\ldots0}_{n}\ \underset{\substack{\Uparrow \\ \boxed{q_f}}}{\$}\ \underbrace{0\ldots0}_{n}\ \$\,\$\,\$\ldots$$

Una possibile definizione della funzione δ sarà la seguente:

δ	$\$$	0
q_0	$q_1\ \$\ L$	
q_1		$q_2\ \$\ R$
q_2	$q_3\ \$\ R$	$q_2\ 0\ R$
q_3	$q_4\ 0\ L$	$q_3\ 0\ R$
q_4	$q_5\ \$\ L$	$q_4\ 0\ L$
q_5	$q_6\ 0\ L$	$q_5\ 0\ L$
q_6	$q_7\ \$\ R$	$q_2\ \$\ R$
q_7		$q_7\ 0\ R$

Si provi, per esercizio, a definire δ supponendo di disporre di $\Sigma = \{\$, 0, 1\}$.

Esempio 11.8. Definiamo una MdT che calcola il successore in base 10 di un numero. $S = \{0, \ldots, 9\}$, $Q = \{q_0, q_1\}$ essendo q_1 lo stato di terminazione. La

δ	0	1	2	$\cdots$	7	8	9	$\$$
q_0	$q_1\ 1\ R$	$q_1\ 2\ R$	$q_1\ 3\ R$	$\cdots$	$q_1\ 8\ R$	$q_1\ 9\ R$	$q_0\ 0\ L$	$q_1\ 1\ R$
q_1				$\cdots$				

FIGURE 2. MdT per l'incremento in base 10

matrice funzionale in figura 2 definisce la MdT desiderata che calcola il successore di un numero scritto sul nastro supponendo che la MdT inizi il calcolo essendo la testina posizionata sulla cifra meno significativa del numero da incrementare.

ESERCIZIO 11.9. Si scriva la funzione di transizione per le macchine di Turing soddisfacenti i seguenti requisiti:

(1) si calcoli il successore di un numero decimale
(2) decisione di quale sia la più lunga tra due stringhe di 0 poste una a sinistra ed una a destra della testina. Si scelga una opportuna forma di risposta (che tenga conto anche della possibilità che le due stringhe siano uguali);
(3) si effettui la somma (in binario) di due stringhe non nulle di 0 e 1 poste entrambe a sinistra della testina ed inframmezzate da un \$;
(4) si sposti la testina di n posizioni a destra rispetto a quella di partenza, ove n è letto da un input binario;
(5) si calcoli il numero di occorrenze di 1 in una data sequenza binaria;
(6) si ordini, con ripetizioni, una stringa in input di 0 e 1.

2. Funzioni calcolabili da MdT

Ad ogni MdT può essere facilmente associata una funzione che diremo *calcolata* dalla MdT. Poiché in questa parte del corso trattiamo solo funzioni sui naturali, è necessario fissare una opportuna codifica dei naturali come stringhe nell'alfabeto della MdT. Ad esempio è possibile utilizzare una codifica binaria nel caso $\Sigma = \{\$, 0, 1\}$ o una codifica decimale. Nella seguente definizione, assumeremo invece una codifica *unaria*, ovvero un generico numero $n \in \mathbb{N}$ sarà rappresentato come una sequenza di $n+1$-simboli 1 consecutivi. Questa codifica è particolarmente conveniente, anche se del tutto inessenziale, per la seguente definizione di funzione calcolabile da una MdT.

DEFINIZIONE 11.10. Una funzione $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ è *Turing–calcolabile* se esiste una MdT tale che, partendo dalla configurazione iniziale

$$\begin{array}{ccc} \ldots \$\,\$\,\$\,\underline{x_1}\,\$ \ldots \$\,\underline{x_n} & \$ & \$\,\$\,\$ \ldots \\ & \Uparrow & \\ & \boxed{q_0} & \end{array}$$

termina nella configurazione

$$\begin{array}{c} \ldots \$\,\$\,\$\,\ldots\$\,\underline{\mathsf{f}(x_1,\ldots,x_n)} \quad \overset{\$}{\Uparrow} \quad \$\,\$\,\$\ldots \\ \boxed{q_f} \end{array}$$

se $f(x_1, \ldots, x_n)$ è definita, non termina altrimenti; dove per ogni $(x_1, \ldots, x_n) \in \mathbb{N}^n$, $\underline{x_1}, \ldots, \underline{x_n}, \underline{f(x_1, \ldots, x_n)}$ sono le rappresentazioni in unario rispettivamente di $x_1, \ldots, x_n, f(x_1, \ldots, x_n)$ (mentre $q_f \in Q$ è tale per cui $\delta(q_F, \$)$ è indefinito).

Chiaramente, una funzione sui naturali Turing-calcolabile è parziale, poiché può non essere definita su alcuni (o tutti) gli argomenti. Vedremo in seguito il significato più profondo di funzione parziale nella teoria della calcolabilità.

Esempio 11.11. Le seguenti funzioni sono Turing-calcolabili:

- $\lambda\, x.x$: $Q = \{q_0\}$; δ sempre indefinito.
- $\lambda\, x.0$:

δ	$\$$	0
q_0	$q_1\ \$\ R$	
q_1		

- $\lambda\, x.x + 1$:

δ	$\$$	0
q_0	$q_1\ 0\ R$	
q_1		

- $\lambda\, x_1 \ldots x_n.x_i$:
  for $j = 0, \ldots, i-1$
  $\quad\delta(q_j, 0) = (q_j, 0, L)$;
  $\quad\delta(q_j, \$) = (q_{j+1}, \$, L)$.

Esercizio 11.12. Si supponga di disporre delle definizioni per le macchine di Turing computanti le funzioni:
$\quad f : \mathbb{N}^k \longrightarrow \mathbb{N}$,
$\quad g_1, \ldots, g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$.
Come esercizio si mostri che la funzione
$\quad h : \mathbb{N}^n \longrightarrow \mathbb{N}$,
definita come
$\quad h(x_1, \ldots, x_n) \stackrel{\text{def}}{=} f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$,
è Turing calcolabile.

Nota 11.13. Si osservi che, se esiste una MdT computante la funzione f, allora ne esistono infinite calcolanti la stessa funzione (si aggiungano arbitrariamente stati o simboli 'inutili').

3. MdT generalizzate

Il modello di MdT che abbiamo visto è solo uno dei possibili modelli matematici per definire l'effettiva calcolabilità di funzioni. È infatti possibile definire una vasta gamma di MdT differenti ma equivalenti rispetto alla classe di funzioni Turing-calcolabili che esse possono indurre. Questo perchè qualunque MdT ottenuta anche con modifiche strutturali (per esempio aumentando il numero dei nastri o delle testine) se soddisfa le 10 condizioni che abbiamo definito per caratterizzare la nozione intuitiva di algoritmo, può essere simulata da una MdT definita come nella Definizione 11.1. I seguenti risultati, dei quali non riportiamo dimostrazione, forniscono un esempio di come si possa modificare il modello di MdT senza modificare l'insieme delle corrispondenti funzioni calcolabili. La potenza del formalismo delle MdT non viene dunque ridotta/aumentata modificando l'insieme dei simboli/stati o modificando la struttura stessa della macchina (numero di nastri e/o testine).

- Una MdT con n nastri ed m testine ($m \geq n$) può essere simulata da una MdT con 1 nastro ed una testina;
- Una MdT con n simboli ed m stati può essere simulata da una MdT con 2 stati, aumentando opportunamente il numero dei suoi simboli;
- Una MdT con n simboli ed m stati può essere simulata da una MdT con 2 simboli, aumentando opportunamente il numero dei suoi stati;
- Ogni MdT può essere simulata da una MdT che può solo scrivere e non rimpiazzare simboli sul nastro.

Per approfondimenti su queste tematiche, si consulti, ad esempio [**13**].

CHAPTER 12

Funzioni parziali ricorsive di Kleene & Robinson

Lo scopo primario di questa parte del corso è di studiare le funzioni sui numeri naturali, esprimibili in modo effettivo o algoritmico. La definizione intuitiva del concetto di funzione effettivamente calcolabile o funzione calcolabile mediante un algoritmo, ovvero mediante una sequenza discreta di passi elementari di calcolo di una MdT, fornita nel capitolo precedente, può sembrare limitativa rispetto a ciò che intuitivamente ci sembra effettivamente calcolabile. Per valutare la robustezza di questa formalizzazione del concetto di funzione calcolabile, nei seguenti due paragrafi studieremo diverse nozioni di calcolabilità che, come vedremo, si equivalgono e sono a loro volta tutte equivalenti alla calcolabilità mediante MdT.

Nel Paragrafo 1 studieremo la caratterizzazione di effettiva calcolabilità sui naturali nota con il termine di *funzione ricorsiva*. Arriveremo alla trattazione delle *funzioni parziali ricorsive*, dette anche *funzioni ricorsive generali*, di Kleene & Robinson, arricchendo via via con nuovi costrutti, o metodi di composizione, alcune funzioni elementari sui naturali, dette *funzioni ricorsive di base*.[1] Questi costrutti, come vedremo nel Cap. 16, corrispondono a costrutti noti ed ampiamente utilizzati nei moderni linguaggi di programmazione.

1. Funzioni primitive ricorsive

L'insieme dei numeri naturali è facilmente caratterizzabile mediante *induzione*, ovvero affermando che ogni numero naturale n è esprimibile come una iterazione della operazione elementare di *successore* applicata al valore costante 0: $n = S^n(0)$. Pertanto

$$\mathbb{N} = \{0, S(0), S(S(0)), S(S(S(0))), \dots\}$$

Nel seguito indicheremo spesso $S(x)$ con $x+1$. Questo procedimento induttivo per definire i numeri naturali si basa sull'idea che ogni numero naturale è definibile a partire da un numero più semplice (in questo caso minore nell'ordinamento usuale sui naturali) applicando l'operazione di successore. Questo modo di procedere è comune in matematica e informatica (si veda il paragrafo 2.6), e trova svariate applicazione nelle così dette *definizioni ricorsive* di funzioni. Una definizione ricorsiva di funzione è una definizione dove il valore della funzione per un dato argomento è direttamente correlato al valore della medesima funzione su argomenti

[1] Nel seguito per *funzione ricorsiva* intenderemo una funzione ricorsiva totale. Nel caso di funzioni parziali, parleremo di *funzioni parziali ricorsive*.

"più semplici" o al valore di funzioni "più semplici". Ad esempio, la seguente funzione definisce ricorsivamente la successione di Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

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

Al fine di comprendere meglio queste definizioni è pertanto dare un significato preciso al concetto informale di "più semplice". Identifichiamo le funzioni "più semplici" in assoluto con le funzioni costanti. Trattando i naturali, consideriamo dunque la costante 0 come funzione costante elementare. Supponiamo inoltre di avere a disposizione come funzione elementare l'operazione di successore, che ci permette di definire qualsiasi numero naturale, e la funzione identica. Queste assunzioni ci portano a definire il concetto di *funzione ricorsiva di base*.

Definizione 12.1. Si dicono *funzioni ricorsive di base* le seguenti funzioni:
- la funzione costante 0: $\lambda x.\, 0$;
- La funzione successore S: $\lambda x.\, x+1$;
- La funzione identità, o i-esima proiezione, $\pi_i : \lambda x_1 \cdots x_n.\, x_i$ con $1 \leq i \leq n$.

Formalizziamo ora il meccanismo di definizione di funzioni per composizione e ricorsione primitiva. Questo meccanismo generalizza i precedenti esempi.

Definizione 12.2. Una funzione $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ si dice:
- definita per *composizione* da $g_1, \ldots, g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ e $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ se $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$
- definita per *ricorsione primitiva* da $g : \mathbb{N}^{n-1} \longrightarrow \mathbb{N}$ e $h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ se
$$\begin{cases} f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1}) \\ f(x_1, \ldots, x_{n-1}, y+1) = h(x_1, \ldots, x_{n-1}, y, f(x_1, \ldots, x_{n-1}, y)) \end{cases}$$

Tutte e sole le funzioni definibili a partire dalle precedenti funzioni ricorsive di base mediante composizione e ricorsione primitiva definiscono l'insieme delle funzioni *primitive ricorsive*. L'idea è quella di costruire via via funzioni effettivamente calcolabili a partire dalle funzioni (banalmente) effettivamente calcolabili di base, ovvero 0 e S, costruendo da queste, per composizione e ricorsione primitiva, funzioni via via più complesse

Definizione 12.3. La classe $\mathcal{P}$ delle funzioni *primitive ricorsive* è la più piccola classe (ovvero l'intersezione di tutte le classi) di funzioni contenenti le funzioni ricorsive di base e chiuse per composizione e ricorsione primitiva.

Nota 12.4. Segue direttamente dalla definizione che, per ogni funzione f, $f \in \mathcal{P}$ sse esiste una sequenza finita di funzioni $f_1, \ldots, f_n$, tale che: $f_n = f$ e per ogni funzione f_j, con $j \leq n$, o f_j è una funzione ricorsiva di base, oppure è ottenuta mediante composizione o ricorsione primitiva a partire da funzioni $f_{i_1}, \ldots, f_{i_k}$ con $i_1, \ldots, i_k < j$ e per le quali vale la stessa proprietà. Queste non sono altro che le condizioni per fissare un sistema formale.

Esempio 12.5. Consideriamo la sequenza di funzioni ottenute come visto in precedenza:

$$\begin{aligned} f_1 &= \lambda x.\, x \\ f_2 &= \lambda x.\, x+1 \\ f_3 &= \lambda x_1x_2x_3.\, x_2 \\ f_4 &= f_2 \circ f_3 = \lambda x_1x_2x_3.\, x_2+1 \end{aligned}$$

$$\begin{aligned} f_5 &= \text{Tale che:} \\ & \quad \begin{aligned} f_5(0, x_2) &= f_1(x_2) \\ f_5(y+1, x_2) &= f_4(y, f_5(y, x_2), x_2) \end{aligned} \\ f_6 &= f_5(f_1, f_1) \end{aligned}$$

È facile vedere che $f_6 = \lambda x.\, 2x$ e che $f_5 = \lambda xy.\, x + y$. Dimostrare per esercizio.

Le funzioni primitive ricorsive sono molto frequenti in matematica ed informatica. Il seguente esempio elenca alcune tra le più note funzioni primitive ricorsive.

Esempio 12.6. Le seguenti funzioni sono ricorsive primitive (per semplicità si denoteranno numerali con numeri):

Zero: $0^n = \lambda x_1 \cdots x_n.\, 0$: definibile come $0(\pi_1(x_1, \ldots, x_n))$;
Costante: $c^n = \lambda x_1 \cdots x_n.c$: definibile come $\underbrace{S(\cdots S}_{c}(0^n(x_1, \ldots, x_n)\underbrace{)\cdots)}_{c}$;

Somma:

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

Rispetto alla notazione utilizzata:

- h è $+$, di arità 2,
- g è la funzione identica, di arità 1,
- $f(x, y, z)$, di arità 3, è la funzione ottenuta come $S(\pi_3(x, y, z))$.

Un esempio di 'computazione' nel formalismo che si sta definendo è il seguente:

$$\begin{aligned} +(S(S(0)), S(S(S(0)))) &= S(+(S(S(0)), S(S(0)))) \\ &= S(S(+(S(S(0)), S(0)))) \\ &= S(S(S(+(S(S(0)), 0))))) \\ &= S(S(S(S(S(0)))))) \end{aligned}$$

Moltiplicazione:
$$\begin{cases} \cdot(x,0) = 0 \\ \cdot(x,y+1) = +(\cdot(x,y),y)) \end{cases}$$
Potenza:
$$\begin{cases} x^0 = S(0) \\ x^{y+1} = \cdot(x^y,x)) \end{cases}$$
Iper-potenza:
$$\begin{cases} iper(x,0) = \pi_1(x) \\ iper(x,y+1) = iper(x,y)^x \end{cases}$$
Si osservi che $iper(x,y) = x^{\underbrace{x \cdots x}_{y}} = x^{(x^y)}$

Predecessore:
$$\begin{cases} pred(0) = 0 \\ pred(y+1) = \pi_1(y) \end{cases}$$
Differenza:
$$\begin{cases} -(x,0) = \pi_1(x) \\ -(x,y+1) = pred(-(x,y)) \end{cases}$$
Fattoriale:
$$\begin{cases} 0! = S(0) \\ (y+1)! = \cdot(S(y),y!) \end{cases}$$
Segno:
$$\begin{cases} sg(0) = 0 \\ sg(y+1) = S(0) \end{cases}$$
Definiamo inoltre $\overline{sg}(x) = -(1, sg(x))$.

Valore assoluto della differenza: $|-(x,y)| = +(-(x,y),-(y,x))$;

Minimo e massimo:
$$\left\{ \begin{array}{lcl} min(x,y) & = & -(x,-(x,y)) \\ max(x,y) & = & +(x,-(y,x)) \end{array} \right\}$$
Divisione intera: Assumiamo che $div(x,0) = mod(x,0) = 0$.
$$\begin{cases} mod(0,x) = 0 \\ mod(y+1,x) = S(mod(y,x)) \cdot sg(|-(x,S(mod(y,x)))|) \end{cases}$$
$$\begin{cases} div(0,x) = 0 \\ div(y+1,x) = div(y,x) + \overline{sg}(|-(x,S(mod(y,x)))|) \end{cases}$$

Produttoria:

Sia f una funzione ricorsiva primitiva binaria, allora definiamo la funzione $\Pi_{z<y} f(x,z) = \Pi_{z=0}^{y-1} f(x,z)$:

$$\begin{cases} \Pi_{z<0} f(x,z) = S(0) \\ \Pi_{z<y+1} f(x,z) = \cdot(f(x,y), \Pi_{z<y} f(x,z)) \end{cases}$$

Sommatoria:

Sia f una funzione ricorsiva primitiva binaria, allora definiamo la funzione $\Sigma_{z<y} f(x,z) = \Sigma_{z=0}^{y-1} f(x,z)$:

$$\begin{cases} \Sigma_{z<0} f(x,z) = 0 \\ \Sigma_{z<y+1} f(x,z) = +(f(x,z), \Sigma_{z<y} f(x,z)) \end{cases}$$

μ-operatore limitato di minimizzazione:

Sia f una funzione primitiva ricorsiva $n+1$ aria.

$$\begin{aligned} g(x_1, \dots, x_n, y) &= \mu z < y.\, (f(x_1, \dots, x_n, z) = 0) \\ &= \begin{cases} \text{il più piccolo } z \text{ minore di } y & \\ \text{tale che } f(x_1, \dots, x_n, z) = 0 & \text{se tale } z \text{ esiste} \\ y & \text{altrimenti} \end{cases} \end{aligned}$$

Tale funzione è primitiva ricorsiva:

$$g(x_1, \dots, x_n, y) = \Sigma_{v<y}(\Pi_{u\leq v} sg(f(x_1, \dots, x_n, u))).$$

Esercizio 12.7. Si definiscano le seguenti funzioni ricorsive primitive:

(1) la funzione che, dato x, fornisce il numero dei suoi divisori;
(2) la funzione che, dato x, restituisce 1 se x è primo, 0 altrimenti;
(3) la funzione p che, dato x, restituisce il p-esimo numero primo (si assuma $p(0) = 0$).
(4) la funzione che restituisce l'esponente di $p(y)$ nella fattorizzazione di x;
(5) la funzione che restituisce 1 se e solo se x è un cubo perfetto;
(6) la funzione che restituisce 1 se e solo se x è ottenibile come somma di due cubi.

Teorema 12.8. *Le funzioni primitive ricorsive sono totali.*

Proof. Per induzione strutturale sulla complessità (nel senso di numero di operazioni di composizione e ricorsione primitiva) delle funzioni definite in $\mathcal{P}$ osserviamo che le funzioni ricorsive di base sono totali, e la composizione di funzioni totali è totale. Resta da dimostrare che la composizione mediante ricorsione primitiva di funzioni totali è totale. Questo passo è lasciato per esercizio. $\square$

I precedenti esempi ci lascerebbero pensare che la nozione di funzione primitiva ricorsiva catturi esattamente il concetto intuitivo di funzione calcolabile mediante un algoritmo. Questo è falso! Innanzitutto, il Teorema 12.8 afferma che le funzioni primitive ricorsive sono totali e, come vedremo, questa è una limitazione, ovvero una funzione calcolabile può essere indefinita su alcuni (o tutti) i valori di input. Le funzioni primitive ricorsive hanno inoltre un'ulteriore limitazione anche all'interno delle funzioni totali sui naturali. Per dimostrare questo fatto, è sufficiente dare un testimone, ovvero una funzione evidentemente calcolabile e totale, ma non appartenente a $\mathcal{P}$. Una tale funzione è la *funzione di Ackermann*.

La funzione di Ackermann è una funzione in 3 argomenti, $\mathsf{ack} : \mathbb{N}^3 \longrightarrow \mathbb{N}$, definita come segue:

$$\begin{cases} \mathsf{ack}(0,0,y) = y \\ \mathsf{ack}(0,x+1,y) = \mathsf{ack}(0,x,y)+1 \\ \mathsf{ack}(1,0,y) = 0 \\ \mathsf{ack}(z+2,0,y) = 1 \\ \mathsf{ack}(z+1,x+1,y) = \mathsf{ack}(z,\mathsf{ack}(z+1,x,y),y). \end{cases}$$

Per esercizio si verifichi che la funzione è totale (ed è ben definita, cioè non vi sono chiamate ricorsive ad oggetti 'più grandi').

Vediamo di capire il significato della definizione di ack. Dalle prime due equazioni segue che $\mathsf{ack}(0,x,y) = y+x$. La terza e quarta defnzione costituiscono le condizioni iniziali per la quinta che è la vera e propria chiamata ricorsiva di ack. Questa afferma che $\lambda xy.\ \mathsf{ack}(z+1,x,y)$ è ottenuta calcolando

$$y_0 = \mathsf{ack}(z+1,0,y) = \begin{cases} 0 & \text{se } z+1=1 \\ 1 & \text{se } z+1>1 \end{cases}$$

e quindi applicando la funzione $\lambda wy.\ \mathsf{ack}(z,w,y)$ x volte, ovvero la sequenza:

$$\begin{array}{lcl} y_1 & = & \mathsf{ack}(z+1,1,y) & = & \mathsf{ack}(z,y_0,y) \\ y_2 & = & \mathsf{ack}(z+1,2,y) & = & \mathsf{ack}(z,y_1,y) \\ y_3 & = & \mathsf{ack}(z+1,3,y) & = & \mathsf{ack}(z,y_2,y) \\ \dots & & \dots & & \dots \\ y_x & = & \mathsf{ack}(z+1,x,y) & = & \mathsf{ack}(z,y_{x-1},y) \end{array}$$

Quindi abbiamo:

$$\begin{array}{rcl} f(0,x,y) & = & y+x \\ f(1,x,y) & = & y\cdot x \\ f(2,x,y) & = & y^x \\ f(3,x,y) & = & \left.y^{y^{\cdot^{\cdot^{y}}}}\right\} x\text{-volte} \\ \dots & & \dots \end{array}$$

Ad esempio, $\mathsf{ack}(3,3,3) = 3^{27} > 10^{14}$.

TEOREMA 12.9. ack *non è ricorsiva primitiva* [**25**].

La dimostrazione del Teorema 12.9 è piuttosto complessa ed è basata sul fatto che ack *cresce* più velocemente di qualunque funzione ricorsiva primitiva. Questa funzione, universalmente accettata come funzione calcolabile non è primitiva ricorsiva pur essendo totale. Inoltre esistono infinite funzioni totali e chiaramente calcolabili che non sono esprimibili mediante ricorsione primitiva: basta aggiungere alla funzione ack una generica costante.

2. Diagonalizzazione

In questa sezione studieremo uno degli strumenti fondamentali della teoria delle funzioni calcolabili: la *diagonalizzazione*. Il suo uso, già evidenziato nel Capitolo 10, sarà frequente nel seguito. La tecnica di diagonalizzazione fu per la prima volta utilizzata da Cantor nel 1874 per dimostrare uno dei risultati fondamentali nella teoria classica degli insiemi, ovvero la non enumerabilità dei numeri reali (si veda anche il Capitolo 2).

L'idea della diagonalizzazione è la seguente: Dato un insieme S numerabile, costruiamo una matrice nel modo seguente: in ogni riga i è presente una possibile enumerazione di S: $s_{i,0}, s_{i,1}, s_{i,2}, \dots$ che identifica univcocamente una funzione totale $f : \mathbb{N} \longrightarrow S$. Supponiamo che l'insieme di tali enumerazioni sia numerabile e tale enumerazione la leggiamo scandendo la matrice riga per riga.

$$\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & \dots \\ s_{1,0} & s_{1,1} & s_{1,2} & \dots \\ s_{2,0} & s_{2,1} & s_{2,2} & \dots \\ \dots & \dots & \dots & \dots \end{array}$$

Sia $d : S \longrightarrow S$ una funzione totale che non sia mai l'identità su S, ovvero tale che $\forall s \in S.\ d(s) \neq s$ (una tale funzione ovviamente esiste). Da questa costruzione, consideriamo la diagonale $s_{0,0}$, $s_{1,1}$, $s_{2,2}$, ...che viene trasformata da d nella sequenza:

$$d(s_{0,0}), d(s_{1,1}), d(s_{2,2}) \dots.$$

La caratteristica essenziale di questa sequenza è che essa differisce da ogni altra riga della matrice. Pertanto esiste una enumerazione di S non presente nella matrice. Ciò contraddice con l'ipotesi che l'insieme delle enumerazioni n(e dunque delle funzioni totali da $\mathbb{N}$ a S è numerabile. Pertanto, data una qualsiasi enumerazione degli oggetti di un sistema formale dato, in questo modo sarà sempre possibile "costruire" un oggetto che non appartiene alla numerazione, ovvero esterno al sistema formale dato.

Come esempio di applicazione di questa tecnica, dimostreremo l'inadeguatezza della classe delle funzioni primitive ricorsive nell'esprimere ciò che intuitivamente viene ritenuto effettivamente calcolabile. Ovvero mediante un "argomento diagonale" dimostreremo che è possibile definire effettivamente una classe di funzioni calcolabili non esprimibili mediante ricorsione primitiva. In realtà la diagonalizzazione ha un ben più ampio campo di utilizzo in quanto è applicabile ad ogni sistema formale le cui istruzioni definite su un alfabeto dato, sono effettivamente enumerabili. L'inadeguatezza della classe delle funzioni primitive ricorsive per esprimere la calcolabilità era comunque già chiara dal fatto che la funzione calcolabile di Ackermann non è primitiva ricorsiva.

Consideriamo l'insieme delle funzioni primitive ricorsive $\mathcal{P}$ ed un formalismo, ovvero un insieme finito di simboli con il quale sia possibile descrivere tutte le possibili derivazioni associate alle funzioni in $\mathcal{P}$. Un siffatto alfabeto può essere ad esempio

$$\Sigma = \{a, b, c, \dots, z, 0, 1, 2, \dots 9, (,)\}$$

Ogni derivazione associata ad una funzione in $\mathcal{P}$ è quindi descrivibile con una opportuna sequenza finita di simboli in Σ^*. È inoltre facile definire una procedura che verifichi se una data stringa $\sigma \in \Sigma^*$ corrisponde ad una derivazione legittima per una funzione in $\mathcal{P}$ (si veda anche la Nota 12.4). È pertanto possibile enumerare tutte le derivazioni legittime per funzioni primitive ricorsive esaminando le stringhe in Σ di lunghezza 1, quindi quelle di lunghezza 2 etc. Sia quindi Q_x l'$(x+1)$-esima derivazione in questa lista e sia g_x la funzione di cui Q_x è derivazione. Definiamo la funzione h come segue:

$$h(x) = g_x(x) + 1$$

Evidentemente abbiamo un algoritmo per calcolare h: dato x, generiamo la lista delle derivazioni fino a Q_x, quindi utilizziamo Q_x per calcolare $g_x(x)$ e sommiamo 1. h così costruita non è primitiva ricorsiva altrimenti, se lo fosse, avremmo $h = g_{x_0}$ per un qualche x_0, e quindi l'assurdo:

$$g_{x_0}(x_0) = h(x_0) = g_{x_0}(x_0) + 1$$

Si noti l'analogia di questa dimostrazione (informale) con la dimostrazione di Cantor sulla non enumerabilità di $\wp(\mathbb{N})$. È pertanto possibile "costruire", mediante diagonalizzazione, una funzione a tutti gli effetti calcolabile che non è primitiva ricorsiva.

Il problema posto dalla diagonalizzazione sembra limitare in modo intrinseco e insuperabile ogni definizione matematica di funzione calcolabile codificabile con un alfabeto finito di simboli. Infatti, per diagonalizzazione sembra possibile costruire funzioni calcolabili che sfuggono ad ogni definizione formale, basata ad esempio su un alfabeto finito, di funzione calcolabile.

L'idea fondamentale che permette di superare la difficoltà indotta dalla diagonalizzazione su un dato sistema formale è quella di ammettere algoritmi, o insiemi di istruzioni per calcolare sia funzioni totali che *parziali*. Per comprendere meglio questo aspetto fondamentale da cui è iniziato lo studio della teoria della calcolabilità, cerchiamo di applicare un argomento diagonale al caso di sistemi formali per esprimere funzioni parziali. Osserveremo che un sistema formale che caratterizzi un sottoinsieme opportuno di funzioni parziali, non è soggetto a limitazioni dovute ad argomenti diagonali.

Sia ψ_x la funzione parziale corrispondente all'$(x+1)$-esimo insieme di istruzioni P_x in un dato sistema formale per funzioni parziali, e sia x_0 tale che ψ_{x_0} è la funzione parziale φ definita dalle seguenti istruzioni: trova P_x, calcola $\psi_x(x)$ e, se e quando si ottiene un risultato, restituisci come output di φ il valore $\psi_x(x) + 1$. Trattandosi di funzioni parziali, l'equazione

$$\psi_{x_0}(x_0) = \varphi(x_0) = \psi_{x_0}(x_0) + 1$$

non genera contraddizioni, poiché $\varphi(x_0)$ può non essere definita.

Questo metodo di procedere, ovvero considerare la teoria della effettiva calcolabilità come una teoria di funzioni parziali è alla base degli approcci di Kleene, Church e Turing sviluppati negli anni '30. Tutti i sistemi formali che vedremo infatti hanno in comune una descrizione in un opportuno sistema formale, ovvero mediante un opportuno insieme di simboli e regole, di ciò che sono gli algoritmi ed una conseguente caratterizzazione del sottoinsieme delle funzioni parziali calcolabili in quel sistema formale, ovvero calcolabile dagli algoritmi così descritti. Abbiamo già visto un esempio di questo modo di procedere nella descrizione delle MdT come sistema formale per esprimere algoritmi e nella funzioni calcolabili da una MdT come descrizione di opportune funzioni parziali. Queste considerazioni generali sulla natura parziale delle funzioni calcolabili giustificano l'assunzione **l** nella descrizione intuitiva di ciò che è effettivamente calcolabile. Indicheremo nel seguito con $f(x)\downarrow$ il fatto che $f(x)$ è definita mentre con $f(x)\uparrow$ il fatto che $f(x)$ non è definita.

3. Funzioni parziali ricorsive

Introduciamo ora un metodo generale per costruire funzioni parziali a partire da funzioni totali (ad esempio primitive ricorsive).

DEFINIZIONE 12.10. Sia f una funzione totale $n+1$ aria, allora definiamo la funzione

$$\begin{aligned}\varphi(x_1,\ldots,x_n) &= \mu z.\ (f(x_1,\ldots,x_n,z)=0)\\ &= \begin{cases}\text{il più piccolo } z \text{ t.c. } f(x_1,\ldots,x_n,z)=0 & \text{se } z \text{ esiste}\\ \uparrow & \text{altrimenti}\end{cases}\end{aligned}$$

Tale operatore tra funzioni è detto μ-operatore ed una funzione così definita è detta definita per *minimizzazione* o μ-ricorsione da f.

Chiaramente una funzione definita per minimizzazione è in generale, per definizione, parziale.

NOTA 12.11. Analizziamo l'ipotesi che f sia una funzione totale. Questa ipotesi può essere rilasciata definendo la minimizzazione, o μ-ricorsione nel modo seguente: Sia ψ una funzione parziale; φ è definita per μ-ricorsione da ψ se

$$\varphi(x_1,\ldots,x_n) = \mu z.\ (\forall y \le z.\ \psi(x_1,\ldots,x_n,y)\downarrow\ \wedge\ \psi(x_1,\ldots,x_n,z)=0)$$

La rimozione dell'ipotesi di totalità nella Definizione 12.10 provocherebbe un chiaro non senso. Infatti, per determinare il minimo z tale che $f(x_1,\ldots,x_n,z)=0$ si ricorre al seguente algoritmo (facilmente descrivibile con una MdT) calcolando la sequenza

$$f(x_1,\ldots,x_n,0), f(x_1,\ldots,x_n,1),\ldots$$

fino a che non si trova il primo valore z tale che $f(x_1,\ldots,x_n,z)=0$. Se f fosse parziale, ovvero il suo calcolo potesse non convergere, la procedura così descritta non sarebbe effettiva. Inoltre, come dimostrato da Kleene nel '52, le funzioni parziali ricorsive non sono chiuse rispetto allo schema[2]

$$\varphi(x_1,\ldots,x_n) = \mu z.\ (\psi(x_1,\ldots,x_n,z)=0).$$

DEFINIZIONE 12.12. La classe delle funzioni *parziali ricorsive* è la minima classe $\mathcal{PR}$ di funzioni contenente $\mathcal{P}$ e chiusa per μ-ricorsione.

ESEMPIO 12.13. Si osservi come si possa utilizzare il μ-operatore per la definizione delle seguenti funzioni:

- logaritmo (intero):
  $\lfloor \log_a x \rfloor = \mu y.\ (\mathtt{leq}(x, a^{(y+1)})=0)$,
  ove $\mathtt{leq}$ è una funzione ricorsiva primitiva tale che $\mathtt{leq}(x,y)=0$ se e solo se $x<y$;

[2]Tale risultato si può mostrare facilmente ma utilizzando risultati e nozioni che vedremo in seguito nel Cap. 17. Anticipando alcune nozioni, se A è un insieme ricorsivamente enumerabile ma non ricorsivo, allora definendo:

$$\psi(x,y)=0 \iff (y=0 \wedge x\in A) \vee y=1$$

si ha che ψ è parziale ricorsiva; ma definendo $f(x)=\mu y.\ (\psi(x,y)=0)$, abbiamo che f non può essere parziale ricorsiva altrimenti, se lo fosse, essendo chiaramente totale, sarebbe ricorsiva e poiché $f(x)=0 \iff x\in A$ anche A sarebbe ricorsivo.

- radice n-esima (intera):
  $\lfloor \sqrt[n]{x} \rfloor = \mu y.\ (\mathtt{leq}(x, (y+1)^n) = 0)$;

4. Equivalenza tra MdT e funzioni parziali ricorsive

Il seguente risultato dimostra l'equivalenza delle funzioni calcolabili da MdT e le funzioni parziali ricorsive di Kleene & Robinson.

TEOREMA 12.14. $f : \mathbb{N} \longrightarrow \mathbb{N}$ *è parziale ricorsiva se e solo se è Turing-calcolabile.*

PROOF. Per dimostrare il verso $(\rightarrow)$, conviene dimostrare un teorema più forte: se φ è parziale ricorsiva, allora esiste una MdT che la calcola che:

(1) Funziona anche se il nastro a sinistra dell'input e a destra della posizione iniziale della testina non è una sequenza infinita di \$.
(2) Quando termina, sappiamo che deve terminare a destra dell'output. Richiediamo inoltre che subito a sinistra dell'output ci sia l'input (o una copia di esso).
(3) La parte del nastro che sta a sinistra della posizione iniziale della testina (in particolare l'input) non viene modificata.

Queste 3 assunzioni ci permetteranno di applicare l'ipotesi induttiva.

Si tratta ora di realizzare (o descrivere a parole) le MdT che calcolano le funzioni di base soddisfacendo alle restrizioni suddette e poi mostrare che disponendo delle MdT che soddisfano le condizioni sopra e che calcolano delle funzioni, siamo in grado di definire quelle necessarie per calcolare la funzione che fa la loro composizione (qui le proprietà sopra sono essenziali), ricorsione primitiva e minimizzazione [Completare per esercizio].

Mostriamo invece l'implicazione inversa e cioé che se φ è Turing-calcolabile allora φ è parziale ricorsiva.

Sia $\varphi : \mathbb{N} \longrightarrow \mathbb{N}$ una funzione calcolabile da una MdT Z. L'obiettivo è quello di codificare la MdT Z come una funzione sui naturali che sia parziale ricorsiva. Per semplicità assumiamo, senza incorrere in limitazioni, che la MdT Z sia definita con 2 simboli, che corrispondono ai numeri 0 e 1 rispettivamente:

$$\begin{aligned} \Sigma &= \{0, 1\} \\ Q &= \{q_0, \dots, q_k\} \end{aligned}$$

Con queste ipotesi possiamo rappresentare una generica ID come una tupla di numeri, associando un numero nell'intervallo $[0, k]$ a ogni simbolo di stato in Q. Definiamo $\mathtt{ar} : \mathrm{ID} \longrightarrow \mathbb{N}^4$ tale che, per ogni $\alpha \in \mathrm{ID}$ della forma:

$$\alpha = \cdots\ b_2\ b_1\ b_0\ s\ q_h\ c_0\ c_1\ c_2\ \cdots$$

con $s \in \{0,1\}$, $h \in [0,k]$ e $\{b_i, c_i\}_{i\in\mathbb{N}} \subseteq \Sigma$, allora: $ar(\alpha) = (h, m, s, n)$ dove

$$\begin{array}{rcl} m & = & \sum_{i=0}^{\infty} b_i 2^i = \sum_{i=0}^{k_m} b_i 2^i \\ n & = & \sum_{i=0}^{\infty} c_i 2^i = \sum_{i=0}^{k_n} c_i 2^i \end{array}$$

ove k_m (k_n) è il massimo intero i per cui $b_i \neq 0$ (rispettivamente $c_i \neq 0$). In questo modo, poiché il numero di 1 sul nastro non può essere infinito (siamo partiti con un numero finito di 1 ed abbiamo effettuato un numero finito di passi), per ogni ID in una computazione di Z, segue che ar associa in modo univoco ad ogni ID una quadrupla di naturali (aritmetizzazione di ID).

Una computazione di una MdT è definita come una sequenza (anche infinita) di ID: $\alpha_1 \vdash \alpha_2 \vdash \ldots$. Definiamo dunque una funzione che esprime un passo di transizione tra ID, ovvero una funzione che manipola quadruple di naturali. A tal proposito, definiamo le seguenti funzioni: $\Delta : Q \times \Sigma \longrightarrow Q$, $\nabla : Q \times \Sigma \longrightarrow \Sigma$ e $\mathcal{X} : Q \times \Sigma \longrightarrow \{0,1\}$ tali che:

- $\Delta(q,s)$ è lo stato che la MdT Z assume trovandosi nello stato q con simbolo in lettura s;
- $\nabla(q,s)$ è il simbolo che la MdT Z produce trovandosi nello stato q con simbolo in lettura s;
- $\mathcal{X}(q,s)$ è lo spostamento a destra ($= 1$) o sinistra ($= 0$) compiuto dalla MdT Z trovandosi nello stato q con simbolo in lettura s.

Per i valori non definiti si assegni il valore $k+1$ per renderle totali. Si noti che Δ, ∇ e $\mathcal{X}$ sono funzioni definite dalla matrice funzionale di Z, e sono chiaramente funzioni primitive ricorsive (lo si dimostri per esercizio). Possiamo quindi definire le trasformazioni compiute eseguendo un singolo passo della MdT Z sulle quadruple rappresentanti una generica ID di Z:

$$\begin{array}{rcl} q' & = & \Delta(q,s) \\ s' & = & (m \bmod 2)(1 - \mathcal{X}(q,s)) + (n \bmod 2)\mathcal{X}(q,s) \\ m' & = & (2m + \nabla(q,s))\mathcal{X}(q,s) + (m \operatorname{div} 2)(1 - \mathcal{X}(q,s)) \\ n' & = & (2n + \nabla(q,s))(1 - \mathcal{X}(q,s)) + (n \operatorname{div} 2)\mathcal{X}(q,s). \end{array}$$

Queste funzioni sono chiaramente primitive ricorsive, essendo la composizione di funzioni primitive ricorsive. Possiamo dunque rappresentare una transizione $\alpha \vdash \beta$ nel modo seguente: Se $ar(\alpha) = (q,s,m,n)$ allora $ar(\beta) = (q',s',m',n')$.

Per rappresentare l'effetto di t-transizioni o passi di calcolo della MdT Z, definiamo le seguenti funzioni di 5 argomenti:

- $P_\Delta(t,q,s,m,n)$ è lo stato ($\in [0,k]$) ottenuto dopo t-passi partendo da una ID α tale che $ar(\alpha) = (q,m,s,n)$;
- $P_\nabla(t,q,s,m,n)$ è il simbolo ($\in \{0,1\}$) ottenuto dopo t-passi partendo da una ID α tale che $ar(\alpha) = (q,m,s,n)$;

- $P_M(t, q, s, m, n)$ è il valore ($\in \mathbb{N}$) rappresentante il nastro a sinistra della testina ottenuto dopo t-passi partendo da una ID α tale che $ar(\alpha) = (q, m, s, n)$;
- $P_N(t, q, s, m, n)$ è il valore ($\in \mathbb{N}$) rappresentante il nastro a destra della testina ottenuto dopo t-passi partendo da una ID α tale che $ar(\alpha) = (q, m, s, n)$.

È facile vedere che, ad esempio, P_Δ può essere definita nel modo seguente:

$$\begin{aligned} P_\Delta(0, q, s, m, n) &= q \\ P_\Delta(t+1, q, s, m, n) &= P_\Delta(t, q', s', m', n') \end{aligned}$$

Supponiamo che la MdT Z abbia uno stato $q_1 \in Q$ di terminazione, ovvero tale che le uniche caselle vuote della matrice funzionale di z siano in corrispondenza della riga q_1. È chiaro che ogni MdT può essere trasformata in modo tale da avere uno stato di terminazione di questo tipo. Pertanto, la MdT Z termina il suo calcolo nel primo (minimo) t tale che: $P_\Delta(t, q_0, s_0, m_0, n_0) = 1$, essendo α_0 la configurazione iniziale tale che $ar(\alpha_0) = (q_0, m_0, s_0, n_0)$. Ovvero, il numero di passi necessario per terminare, è definibile mediante μ-ricorsione come:

$$\mu t.\ (P_\Delta(t, q_0, s_0, m_0, n_0) - 1 = 0).$$

Sia $x \in \mathbb{N}$ un numero naturale di input. Assumiamo che la MdT inizi il suo calcolo avendo il numero x sul nastro alla destra della sua testina codificato come una sequenza di x-zeri seguita da 1 (rappresentata matematicamente dal numero 2^x). Assumiamo anche che il risultato, se calcolato, sia rappresentato nel medesimo modo sul nastro della macchina. È dunque chiaro che, avendo un modo algoritmico primitivo ricorsivo dato da una funzione C per rappresentare come potenza di 2 il numero alla destra della testina nella ID terminale, la funzione φ calcolata da Z è esprimibile dalla seguente funzione parziale ricorsiva:

$$\varphi(x) = C(P_N(\mu t.\ (P_\Delta(t, 0, 0, 0, 2^x) - 1 = 0), 0, 0, 0, 2^x)).$$

Per esercizio si trovi la funzione C che chiude la dimostrazione e si dimostri che è primitiva ricorsiva. $\square$

Corollario 12.15 (Forma normale di Kleene). Per ogni funzione $\varphi \in \mathcal{PR}$ (o equivalentemente Turing calcolabile) esistono $f, g \in \mathcal{P}$ tali che

$$\varphi = \lambda x.\ f((\mu t.\ g(t, x) = 0), x)\,.$$

CHAPTER 13

Tesi di Church-Turing

Negli stessi anni in cui venivano proposti i vari formalismi per definire la effettiva calcolabilità, ad esempio le MdT, le funzioni ricorsive, il λ-calcolo etc., veniva dimostrata anche la loro l'equivalenza. Si dimostrava cioè che i diversi sistemi permettono di calcolare esattamente la stessa classe di funzioni. Abbiamo visto degli esempi nelle sezioni precedenti riguardanti le MdT e le funzioni parziali ricorsive di Kleene & Robinson.

Questi risultati (in particolare l'equivalenza del λ-calcolo con le funzioni parziali ricorsive) portarono Church e Turing nel '36 a formulare la seguente Tesi fondamentale:

> **Tesi di Church-Turing:** *La classe delle funzioni "intuitivamente calcolabili" coincide con la classe delle funzioni Turing calcolabili.*

Si tratta chiaramente di una tesi indimostrabile, vista l'impossibilità di maneggiare formalmente (= matematicamente) la nozione di "intuitivamente calcolabile". Accettando la Tesi di Church-Turing si osserva immediatamente la notevole importanza che assumono le funzioni calcolabili da MdT, o equivalentemente, le funzioni parziali ricorsive o λ-definibili. Queste classi di funzioni vengono a rappresentare, in virtù della Tesi di Church-Turing, la classe delle funzioni calcolabili effettivamente mediante un algoritmo. La Tesi di Church-Turing ha pertanto notevole interesse metamatematico. Lo stesso Gödel scrisse nel '46 le seguenti osservazioni riguardanti la Tesi di Church-Turing, in riferimento ad una presentazione di Tarski sul medesimo argomento:

> *Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness*[1] *(or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e. one not depending on the formalism chosen. [...] By a kind of miracle it is not necessary to distinguish orders and the diagonal procedure does not lead outside the defined notion.*

[1] ... che noi chiamiamo ricorsività *a la* Kleene & Robinson.

[K. Gödel, Princeton 1946]

La Tesi di Church-Turing permette di limitare l'estensione di ciò che è effettivamente calcolabile: se dimostriamo che una funzione non è parziale ricorsiva, allora, in virtù della Tesi di Church-Turing, essa non è calcolabile in nessun modo effettivo. Analogamente, dare un algoritmo per una funzione corrisponde a dimostrare, per la Tesi di Church-Turing, che essa è una funzione parziale ricorsiva. Questo utilizzo della Tesi di Church-Turing è fondamentale in questo corso. Infatti, sarà sufficiente dimostrare la realizzabilità di un algoritmo per affermare l'esistenza di una MdT o di una funzione parziale ricorsiva corrispondente, senza dover esibire una tale macchina o una funzione per calcolarlo. Questo ci permetterà di sviluppare un certo numero di strumenti a supporto della teoria delle calcolabilità, senza dover esibire esplicitamente complesse MdT o funzioni parziali ricorsive. Questa caratteristica della Tesi di Church-Turing darà un aspetto apparentemente informale alle dimostrazioni di effettiva calcolabilità di funzioni. Dimostrazioni di questo tipo, che si basano su una non banale applicazione della Tesi di Church-Turing, saranno dette *dimostrazioni mediante Tesi di Church-Turing*. Pertanto, ogni volta che una funzione risulterà "palesemente" calcolabile, la si accetterà come effettivamente calcolabile da uno dei formalismi prescelti, senza dover costruire la MdT o altra funzione corrispondente. Nel seguito dunque considereremo equivalenti i concetti di calcolabilità: "effettiva" (ovvero associata all'esistenza di una MdT), "algoritmica" (ovvero associata all'esistenza di un algoritmo che soddisfi i requisiti **a–l**) o "ricorsiva" (ovvero associata ad una funzione ricorsiva parziale che la calcola).

A questo punto della trattazione, è bene distinguere, per chiarezza, tra due nozioni diverse di calcolabilità. Infatti è possibile definire una funzione g per cui sappiamo esistere un algoritmo ma per la quale non sappiamo dare l'algoritmo che la calcola. Consideriamo le funzioni definite nel modo seguente:

$$f(x) = \begin{cases} 1 & \text{se esattamente } x \text{ `5' consecutivi appaiono nella} \\ & \text{espansione decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

$$g(x) = \begin{cases} 1 & \text{se almeno } x \text{ `5' consecutivi appaiono nella espansione} \\ & \text{decimale di } \pi \\ 0 & \text{altrimenti} \end{cases}$$

È chiaro che g è calcolabile: dato un algoritmo che genera l'espansione decimale di π, g sarà o la funzione costante $g(x) = 1$, nel caso vi sia un numero arbitrariamente grande di occorrenze successive di '5' in π, che è chiaramente una funzione calcolabile; oppure sarà la funzione calcolata da un qualche algoritmo che, fissato un k (ed esempio il massimo numero di '5' consecutivi in π), calcola:

$$g(x) = \begin{cases} 1 & \text{se } x \leq k \\ 0 & \text{se } x > k \end{cases}$$

In questo secondo caso esiste sicuramente un tale k che permette di individuare l'algoritmo cercato, ma ci è impossibile sapere quale sia il k giusto. In entrambi i casi la funzione g è calcolabile (ed é in particolare primitiva ricorsiva), ma in generale non sappiamo costruire effettivamente un algoritmo per calcolarla. Al contrario, per f non siamo in grado a tutt'oggi di affermare nulla sulla sua calcolabilità effettiva. Pertanto, mentre per g è possibile invocare la Tesi di Church-Turing, ed affermare che esiste una MdT che la calcola, anche se non si sa quale, per f questo non è possibile. Accettando quindi di considerare calcolabile ogni funzione per la quale <u>esiste</u> un algoritmo che la calcoli, possiamo nel seguito utilizzare la Tesi di Church-Turing ogni qual volta sia chiara l'esistenza di un algoritmo per calcolare una data funzione.

CHAPTER 14

Aritmetizzazione e universalità

Aritmetizzazione significa semplicemente traduzione nel linguaggio dell'aritmetica. Il primo utilizzo dell'aritmetizzazione è dovuto a Gödel nel 1931 nella dimostrazione del risultato fondamentale di incompletezza dell'aritmetica; da qui anche il termine equivalente di Gödelizzazione. Nel seguito considereremo le MdT come esempio ed applicheremo alle MdT il concetto di aritmetizzazione. Per la Tesi di Church-Turing, sappiamo che questa non è una restrizione, ed analoghe aritmetizzazioni possono essere definite per ogni sistema formale equivalente alle MdT.

1. Enumerazione delle MdT

Sia $\Sigma = \{\$, 0\}$. Quante sono le macchine di Turing 'distinte' descrivibili? Si supponga $Q = \{q_0\}$. Si tratterà di riempire la tabella

	$\$$	0
q_0		

in tutti i modi (significativi) possibili, ossia ogni singola casella potrà essere

(1) *vuota* (si immagini di scrivere $\$\,\$\,\$$);
(2) $q_0\,\$\,R$;
(3) $q_0\,\$\,L$;
(4) $q_0\,0\,R$;
(5) $q_o\,0\,L$.

Dunque ci sono <u>esattamente</u> $25 = 5 \times 5$ macchine di Turing distinte ad un solo stato.

Si può pensare a questo punto di definire un ordinamento $\lhd$ sulle macchine di Turing ad uno stato definibili; fissato un ordinamento su Σ (poniamo $\$ <_\Sigma 0$), sia inoltre $L <_M R$ e si consideri la cella a sinistra più importante di quella a destra, essendo M insieme dei possibili movimenti della testina, cioé $M = \{L, R\}$.

Si assuma che $\$\,\$\,\$$ sia minore di qualunque terna $q_0\ s\ m$, allora si avrà (in una estensione lessicografica i cui parametri sono, nell'ordine *cella* e *contenuto*):

$$\langle \$\$\$, \$\$\$\rangle \lhd \langle \$\$\$, q_0\$L\rangle \lhd \cdots \lhd \langle \$\$\$, q_0 0 R\rangle \lhd \langle q_0\$L, \$\$\$\rangle \lhd \cdots \lhd \langle q_0 0 R, q_0 0 R\rangle.$$

Più in generale, sia $Q = \{q_0, \ldots, q_{n-1}\}$, $\Sigma = \{s_0 = \$, s_1 = 0\}$, $m_0 = L, m_1 = R$; si definisca inoltre un ordinamento tra le $2 \cdot n$ celle presenti (per righe, colonne, a zig zag etc.). Definendo esplicitamente l'ordinamento per ogni cella nel modo seguente:

- $\$\,\$\,\$ \prec q_i\ s\ m$ per ogni $q_i \in Q$, $s \in \Sigma$, $m \in \{L, R\}$;
- $q_{i_1}\ s_{j_1}\ m_{k_1} \prec q_{i_2}\ s_{j_2}\ m_{k_2}$ se $i_1 < i_2$ oppure ($i_1 = i_2$ e $j_1 < j_2$) oppure ($i_1 = i_2$ e $j_1 = j_2$ e $k_1 < k_2$).

si ha un buon ordine sull'insieme delle macchine di Turing aventi n stati, che sono in tutto

$$(n \cdot 2 \cdot 2 + 1)^{2n} = (4 \cdot n + 1)^{2n},$$

dunque ci saranno 25 MdT ad uno stato, 9^4 MdT a due stati, 13^6 MdT a tre stati,.... Si può dunque pensare di associare un numero naturale (un *indice*) ad ogni macchina di Turing in maniera biunivoca:

$1 \div 25$	MdT ad uno stato
$26 \div 25 + 9^4$	MdT a due stati
$25 + 9^4 + 1 \div 25 + 9^4 + 13^6$	MdT a tre stati
...	...

all'interno di ogni gruppo, poi ci si basa sulla relazione di ordinamento. In generale, con un semplice algoritmo, dato $x \in \mathbb{N}$ si può risalire alla MdT x-esima.

Esempio 14.1. Assumendo un ordinamento delle celle per cui la più importante è quella in alto a sinistra, la macchina di Turing n. 4399 ha due stati ($Q = \{q_0, q_1\}$), e la sua funzione di transizione è la seguente:

δ	$\$$	0
q_0	$q_1\ \$\ R$	
q_1		

Si osservi che tale MdT computa la funzione $\lambda x.0$.

Una differente aritmetizzazione delle MdT si basa sulle proprietà della decomposizione in fattori primi dei numeri naturali. Analogamente al caso precedente, è possibile, in modo algoritmico, passare da una MdT ad un numero naturale che la aritmetizza, e viceversa è possibile determinare se un numero è o no l'artimetizzazione di una MdT. Consideriamo ancora una generica MdT definita su un alfabeto finito:

$$\begin{aligned} \Sigma &= \{s_0, \ldots, s_n\} \\ Q &= \{q_0, \ldots, q_m\} \\ X &= \{L, R\} \end{aligned}$$

ed associamo, ad ogni simbolo $x \in \Sigma \cup Q \cup X$ un numero dispari maggiore di 1: $a(x) = 2k+1$ per qualche $k \geq 1$, tale che se $x \neq y$ allora $a(x) \neq a(y)$. Ad esempio:

$$\begin{array}{ccccccc} R & L & s_0 & q_1 & s_1 & q_2 & \ldots \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \ldots \\ 3 & 5 & 7 & 9 & 11 & 13 & \ldots \end{array}$$

Definiamo *numero di Gödel* di una espressione $E \in (\Sigma \cup Q \cup X)^*$ il numero:

$$gn(E) = \prod_{i=1}^{k} p_i^{a(x_i)}$$

essendo k il numero dei simboli in E, p_i l'i-esimo numero primo e x_i l'i-esimo simbolo di E. Per l'unicità della decomposizione in fattori primi, per ogni espressione esiste uno ed un solo numero di Gödel corrispondente; inoltre tale numero è pari nell'ipotesi E sia una stringa di almeno un simbolo.

Poiché le quintuple di una MdT sono a tutti gli effetti espressioni in $(\Sigma \cup Q \cup X)^*$, è possibile in modo algoritmico associare ad ogni quintupla un corrispondente numero di Gödel, e viceversa verificare se un dato numero è numero di Gödel di una quintupla legittima.

Definiamo numero di Gödel di una MdT Z, ovvero di una sequenza di quintuple $Z = \{E_1, \ldots, E_h\}$ il numero:

$$\prod_{i=1}^{h} p_i^{gn(E_i)}$$

Questo numero è anch'esso unico, per l'unicità della decomosizione in fattori primi, e non risulta confondibile con $gn(E)$ per una qualche espressione $E \in (\Sigma \cup Q \cup X)^*$, poiché, al contrario di gn, è ottenuto mediante esponenziazione pari. Analogamente, due sequenze con lo stesso numero di Gödel sono identiche (verificare per esercizio).

In generale è possibile definire infinite aritmetizzazioni diverse per MdT, e quindi funzioni parziali ricorsive. L'artimetizzazione scelta non è quindi sostanziale nello sviluppo della teoria della calcolabilità. Essenziale è che, come dimostrato in precedenza, esistano metodi algoritmici per codificare MdT all'interno dei numeri naturali (suoi argomenti di calcolo). La possibilità di vedere i numeri come al tempo stesso rappresentanti gli argomenti del calcolo e le MdT che eseguono il calcolo, permette di applicare ai sistemi formali analizzati fino ad ora, uno dei principi alla base dell'informatica moderna, ovvero quello di poter passare programmi come argomenti ad altri programmi. In tutto ciò la scelta della particolare aritmetizzazione è inessenziale.[1]

[1]La libertà di poter applicare MdT a MdT (autoapplicazione o referenziazione) condurrebbe immediatamente a paradossi tipo *paradosso di Russell* nella teoria che stiamo sviluppando. Ancora una volta, l'assunzione fondamentale che le funzioni calcolabili siano essenzialmente parziali,

Nel seguito dunque indicheremo con:

- P_x l'insieme di istruzioni di una MdT di indice x, ovex il corrispondente *indice* o *numero di Gödel* della macchina, in una data aritmetizzazione.
- φ_x la funzione calcolata dalla macchina P_x.

Il seguente risultato è banale in seguito ad una qualsiasi aritmetizzazione delle MdT.

TEOREMA 14.2. *Ci sono* $\aleph_0$ *macchine di Turing distinte o funzioni parziali ricorsive, e* $\aleph_0$ *funzioni ricorsive.*

PROOF. Segue banalmente dal fatto che tutte le funzioni costanti sono ricorsive. Pertanto ci sono almeno $\aleph_0$ funzioni ricorsive. □

Segue dunque per la Tesi di Church-Turing che:

TEOREMA 14.3. *Ci sono esattamente* $\aleph_0$ *funzioni calcolabili.*

Indicheremo nel seguito con φ_x la funzione parziale ricorsiva di indice o numero di Gödel x. Chiaramente sappiamo che esiste un algoritmo per passare da x a P_x e/o φ_x e viceversa. Pertanto, con φ_x indicheremo indifferentemente l'x-esima MdT o funzione parziale ricorsiva in una data aritmetizzazione.

Il seguente risultato dimostra che la cardinalità delle funzioni calcolabili è strettamente minore della cardinalità di tutte le possibili funzioni sui naturali. Si tratta di una rivisitazione del Teorema di Cantor, nel contesto delle funzioni calcolabili. Ne consegue che l'insieme delle funzioni calcolabili è strettamente contenuto (ne rappresenta una piccolissima parte) nell'insieme di tutte le funzioni, ovvero nell'insieme di tutti i possibili *problemi* esprimibili mediante funzioni sui naturali. In particolare, potendo selezionare una funzione in $\mathbb{N} \longrightarrow \mathbb{N}$, è facile dimostrare che la probabilità di selezionarne una calcolabile è 0.

TEOREMA 14.4. *Esistono funzioni (totali)* $f : \mathbb{N} \longrightarrow \mathbb{N}$ *non Turing calcolabili.*

PROOF. Sappiamo che, dal Teorema di Cantor (vedi Cap. 2) che:

$$\begin{aligned} |\{f : \mathbb{N} \longrightarrow \mathbb{N}\}| &\geq |\{f : \mathbb{N} \longrightarrow \{0,1\}\}| \\ &= |\wp(\mathbb{N})| \\ &> |\mathbb{N}| \\ &\geq \left|\left\{f : \mathbb{N} \longrightarrow \mathbb{N} \,:\, f \text{ è Turing calcolabile}\right\}\right| \end{aligned}$$

□

Rispetto a quanto visto nelle sezioni precedenti, abbiamo pertanto dimostrato le inclusioni tra classi di funzioni come evidenziate nella Figura 1.

---

permette di superare tali paradossi, rendendo la teoria della calcolabilità una teoria consistente. Si veda in proposito la Nota 15.4.

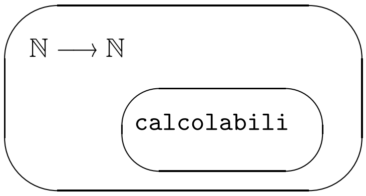


FIGURE 1. Diagramma di inclusione tra insiemi di funzioni

2. Macchina di Turing Universale

Per quanto si è illustrato nella sezione 1, dato un numero naturale x si può facilmente risalire ad una tabella definente la funzione di transizione della x-esima macchina di Turing. D'altro canto ogni singola macchina di Turing è un oggetto provvisto di un compito fisso. Una volta definita la funzione di transizione non vi è modo di modificarla.

Il passo immediatamente successivo (che fornisce la base teorica al concetto di calcolatore programmabile) è il seguente: si supponga di disporre della macchina di Turing con un unico input che, interpretiamo come una coppia $\langle x_1, x_2\rangle$, con il seguente comportamento:

- x_1 è l'indice di una macchina di Turing P_{x_1};
- x_2 è la rappresentazione di un input che va immaginato come input per la macchina P_{x_1}.

La macchina in questione dapprima calcola la funzione di transizione della macchina P_{x_1}, memorizzando questa funzione. Suppone poi (cioè lo memorizza opportunamente) di essere nello stato q_0. Si posiziona dunque in corrispondenza del primo carattere di input (chiamiamolo s) e va a vedere—sulla tabella memorizzata—cosa farebbe P_{x_1}, ossia computa $\delta_{P_{x_1}}(q_0, s)$. Procede dunque nella simulazione andando a modificare il carattere sulla zona di nastro dedicata all'input, lo stato nella opportuna zona in cui va memorizzato, e sposta idealmente la testina, in base al risultato di $\delta_{P_{x_1}}(q_0, s)$.

Una tale MdT si riesce ovviamente a costruire (anche se non può essere considerato un facile esercizio) e dunque avrà un suo indice nella enumerazione delle MdT; sia esso u. Si avrà pertanto:

$$\forall x_1\, x_2.\ P_u(\langle x_1, x_2\rangle) = P_{x_1}(x_2)$$

ove il simbolo $=$ in questo contesto sta a significare che, se una macchina termina, allora terminano entrambe fornendo lo stesso risultato, altrimenti entrambe ciclano. Come interessante ed immediata conseguenza avremo, ad esempio, $P_u(\langle u, u\rangle) = P_u(u)$.

Più in generale possiamo dimostrare il seguente risultato, che rappresenta una prima dimostrazione mediante Tesi di Church.

TEOREMA 14.5. *Esiste un indice* z *tale che per ogni* x *e* y,

$$\varphi_z(x, y) = \begin{cases} \varphi_x(y) & \textit{se } \varphi_x(y) \textit{ è definita} \\ \uparrow & \textit{altrimenti} \end{cases}$$

PROOF. Sia P_x l'insieme di istruzioni di una MdT di indice x. Sappiamo che esiste un metodo effettivo per ottenere P_x da x. Applichiamo P_x all'input y e, quando questa termina, prendiamo il risultato come output di una funzione di 2 argomenti $\psi(x, y)$. Pertanto abbiamo che:

$$\psi(x, y) = \begin{cases} \varphi_x(y) & \text{se } \varphi_x(y) \text{ converge} \\ \uparrow & \text{se } \varphi_x(y) \text{ diverge} \end{cases}$$

Applicando la Tesi di Church-Turing, possiamo concludere che ψ è parziale ricorsiva e pertanto esiste un indice z tale che $\psi = \varphi_z$. □

La funzione φ_z ottenuta nella dimostrazione del precedente teorema è detta *funzione parziale universale* e corrisponde alla MdT universale vista in precedenza. Chiaramente, il precedente teorema può essere generalizzato a funzioni di $k \geq 1$ variabili, in modo tale che, per ogni funzione di $k \geq 1$ variabili, esiste una funzione di $k + 1$ variabili che gioca il ruolo di funzione universale parziale.

ESERCIZIO 14.6. Formulare e dimostrare il precedente teorema per funzioni di $k \geq 1$ variabili.

3. Il Teorema s-m-n

Il seguente risultato, noto con il nome *Teorema s-m-n* e dovuto a Kleene, sarà fondamentale nel seguito. Anch'esso rappresnta un esempio di dimostrazione basata sulla Tesi di Church-Turing.

TEOREMA 14.7 (s-m-n). *Per ogni coppia di interi* $m, n \geq 1$ *esiste una funzione ricorsiva* s^m_n *di* $m + 1$ *variabili tale che per ogni* $x, y_1, \ldots, y_m$ *abbiamo che:*

$$\lambda z_1 \cdots z_n.\ \varphi_x(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s^m_n(x, y_1, \ldots, y_m)}$$

PROOF. Consideriamo per semplicità il caso $m = n = 1$. La dimostrazione per il caso più generale per $m, n \geq 1$ è simile. Fissati x_0 e y_0, la funzione $\lambda z.\ \varphi_{x_0}(y_0, z)$ è chiaramente una funzione calcolabile in una sola variabile. Per ogni x_0 e y_0, la funzione $\lambda z.\ \varphi_{x_0}(y_0, z)$ dipenderà costruttivamente da x_0 e y_0, pertanto esisterà per la Tesi di Church-Turing, una funzione ricorsiva (totale!) s tale che: $\lambda z.\ \varphi_{x_0}(y_0, z) = \varphi_{s(x_0, y_0)}$. □

Il Teorema s-m-n gioca un ruolo essenziale nello sviluppo della teoria della ricorsività, e spesso verrà utilizzato, come la Tesi di Church-Turing, tacitamente. Riportato alla programmazione 'tradizionale', il significato è più o meno il seguente. Se avete scritto un programma che implementa un certo algoritmo su $m + n$ dati in imput e da un certo momento in poi m di questi sono fissati, allora sapete ottenere un programma 'specializzato' che accetta solo n parametri in input. Questa è la giustificazione al lavoro delle software house che 'installano' del software gestionale generale alle varie ditte (che permettono di fissare, con i loro dati, gli m parametri).

Il seguente risultato è una semplice applicazione del Teorema s-m-n.

TEOREMA 14.8. *Esiste una funzione ricorsiva* g *in due argomenti tale che per ogni* x *e* y*:* $\varphi_{g(x,y)} = \varphi_x \circ \varphi_y$.

TRACCIA. Si definisca $\psi(x, y, z) = \varphi_x(\varphi_y(z))$ se $\varphi_y(z) \downarrow$ e $\varphi_x(\varphi_y(z)) \downarrow$, indefinita altrimenti. Concludere per esercizio. $\square$

CHAPTER 15

Problemi insolubili

In questo Capitolo analizzeremo alcuni problemi classici non risolvibili nella teoria della effettiva calcolabilità sviluppata fino ad ora. Il più classico problema insolubile è il *problema della terminazione* posto da Turing.

Esiste una procedura effettiva tale che dati x *e* y *determina se* $\varphi_x(y)$ *è definita o no?*

Il seguente risultato preliminare dimostra l'insolubilità di un problema correlato.[1]

LEMMA 15.1. *Non esiste una funzione ricorsiva* g *tale che per ogni* x*:*

$$g(x) = \begin{cases} 1 & \textit{se } \varphi_x(x)\downarrow \\ 0 & \textit{se } \varphi_x(x)\uparrow \end{cases}$$

PROOF. Supponiamo per assurdo che esista una MdT di indice i_0 tale che $g = \varphi_{i_0}$. Allora possiamo definire la funzione:

$$g'(x) = \begin{cases} \uparrow & \text{se } g(x) = 1 = \varphi_{i_0}(x) \\ 0 & \text{se } g(x) = 0 = \varphi_{i_0}(x) \end{cases}$$

Dunque anche g' sarà calcolabile e dunque esiste i_1 tale che $\varphi_{i_1} = g'$. Ma allora abbiamo che:

$$\begin{array}{ccccccc} \varphi_{i_1}(i_1)\downarrow & \Leftrightarrow & g'(i_1) = 0 & \Leftrightarrow & g(i_1) = 0 & \Leftrightarrow & \varphi_{i_1}(i_1)\uparrow \\ \varphi_{i_1}(i_1)\uparrow & \Leftrightarrow & g'(i_1)\uparrow & \Leftrightarrow & g(i_1) = 1 & \Leftrightarrow & \varphi_{i_1}(i_1)\downarrow \end{array}$$

Che è assurdo. Dunque non esiste un tale indice i_1, ovvero non esiste l'indice i_0 che calcoli g, ovvero g non è ricorsiva. □

In base al Lemma 15.1, segue il teorema:

TEOREMA 15.2. *Non esiste una funzione ricorsiva* ψ *tale che per ogni* x *e* y*:*

$$\psi(x, y) = \begin{cases} 1 & \textit{se } \varphi_x(y)\downarrow \\ 0 & \textit{se } \varphi_x(y)\uparrow \end{cases}$$

[1] Si ricorda che con $\varphi_x(y)\downarrow$ si intende che la macchina di Turing x-esima sull'input y termina o equivalentemente, che la funzione ricorsiva parziale x-esima è definita per l'input x. Con $\varphi_x(y)\uparrow$ si intende l'opposto di tale affermazione.

Proof. Se esistesse allora esisterebbe un indice x_0 tale che $\varphi_{x_0}(x) = \psi(x, x)$. $\psi(x, x)$ non è altro che la funzione $g(x)$ dimostrata non esistere nel Lemma 15.1. $\square$

Il problema della terminazione è stato, storicamente, uno dei primi problemi matematici dimostrati come insolubili, ovvero per i quali non esiste una procedura in grado di "decidere" il problema per ogni possibile input, e rappresenta, per la sua semplicità ed importanza pratica e teorica, uno dei risultati più importanti ottenuti in matematica nel ventesimo secolo.

Il seguente risultato, dovuto a Kleene, dimostra la non risolubilità di un altro semplice problema correlato alle funzioni ricorsive, ovvero non è possibile decidere se una funzione parziale ricorsiva data è totale. Il problema è chiaramente legato alla non decidibilità della terminazione.

Teorema 15.3. *Non esiste una funzione ricorsiva* f *tale che per ogni* x*:*

$$f(x) = \begin{cases} 1 & \textit{se } \varphi_x \textit{ è totale} \\ 0 & \textit{se } \varphi_x \textit{ non è totale} \end{cases}$$

Proof. La dimostrazione utilizza un argomento diagonale. Supponiamo che esista una tale funzione f. Allora definiamo una funzione che associa ad ogni naturale una corrispondente funzione ricorsiva.

$$\begin{aligned} g(0) &= \mu y.\,(f(y) = 1) \\ g(x+1) &= \mu y\,(y > g(x) \wedge f(y) = 1). \end{aligned}$$

Poiché sappiamo che esistono ω funzioni ricorsive totali, anche g sarà totale. Per la tesi di Church e questa osservazione è quindi ricorsiva. Definiamo una funzione h tale che $h = \lambda x.\ \varphi_{g(x)}(x) + 1$. Per la definizione di f segue che h è totale, e quindi, per la Tesi di Church-Turing, ricorsiva. Sia dunque z_0 l'indice per cui $h = \varphi_{z_0}$ e sia y_0 tale che $g(y_0) = z_0$. Questo indice esiste per definizione di g. Allora $h(y_0) = \varphi_{g(y_0)}(y_0) + 1$. Ma $\varphi_{g(y_0)}(y_0) = h(y_0)$ per definizione di y_0. Poiché h è totale, questa è una contraddizione. $\square$

Nota 15.4. Argomenti diagonali e insolubilità sono nozioni strettamente legate tra loro. Ritorniamo brevemente sulla questione delle funzioni parziali. Abbiamo visto che, ricorrendo alle funzioni parziali, si possono superare i limiti imposti da argomenti di tipo diagonale ai sistemi formali introdotti per definire il concetto di effettiva calcolabilità. La necessità di utilizzare funzioni parziali in luogo di funzioni totali segue anche dalla possibilità, dimostrata in precedenza, di definire macchine universali o funzioni ricorsive universali. Questa possibilità, associata alla ipotesi di totalità delle funzioni calcolabili, conduce infatti direttamente a paradossi simili al paradosso di Russell.[2] L'aritmetizzazione infatti non impone nessuna distinzione tra funzioni (MdT) e argomenti, come nella teoria degli insiemi classica non vi è distinzione tra insiemi e loro argomenti. Questa mancanza di struttura permette una indisciplinata possibilità di definizioni di insiemi o funzioni, che conduce ai ben noti paradossi.

È noto che è possibile codificare coppie, terne etc. con numeri naturali. Ad esempio la seguente funzione $g : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ è biiettiva (verificare per esercizio) e chiaramente calcolabile:

$$g(x, y) = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y).$$

Essa permette di associare in modo univoco, ad ogni coppia di naturali, un numero naturale corrispondente. Quindi per la Tesi di Church-Turing esisterà un indice x_0 tale che, se $\mathbb{U}(x, y)$ è la funzione parziale universale vista in precedenza, allora:

$$\mathbb{U}(x, y) = \varphi_{x_0}(g(x, y))$$

Inoltre, essendo sia g che φ_{x_0} calcolabili, esisterà un indice y_0 tale che $\varphi_{y_0}(x) = \varphi_{x_0}(g(x, x))$. Sarà quindi lecito definire la funzione:

$$f(x) = \begin{cases} 1 & \text{se } \varphi_{y_0}(x) = \varphi_x(x) = 0 \\ 0 & \text{se } \varphi_{y_0}(x) = \varphi_x(x) \neq 0 \end{cases}$$

Anche questa funzione avrà un opportuno indice z_0. Segue quindi che $\varphi_{z_0}(z_0) = 0$ se e solo se $\varphi_{z_0}(z_0) \neq 0$, che è chiaramente una contraddizione. Per superare questo paradosso, del tutto analogo al paradosso di Russell, analogamente a quanto visto per superare i limiti imposti dalla diagonalizzazione, basterà ancora assumere che la funzione calcolabile f, calcolata dalla z_0-esima MdT, diverga in z_0, essendo quindi parziale ricorsiva. La funzione

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

è dunque necessariamente parziale ricorsiva, ed infatti, l'algoritmo che la calcola è facilmente identificabile con la MdT universale vista in precedenza.

Esercizio 15.5. Dimostrare che i seguenti problemi non ammettono soluzione ricorsiva:

(1) Decidere se, per ogni x: $\varphi_x = \lambda x.0$. (Traccia: Sia f la funzione ricorsiva parziale definita come $f(x, y) = 0$ se $\varphi_x(y) \downarrow$, indefinita altrimenti. Per *s-m-n* si ha che $f(x, y) = \varphi_{r(x)}(y)$ per r totale ricorsiva. Se, per assurdo, esistesse g t.c. $g(x) = 1$ se $\varphi_x = \lambda x.0$, $g(x) = 0$ altrimenti, si definisca

[2] Il paradosso di Russell nella teoria classica degli insiemi è ottenuto considerando come definibile nella teoria stessa l'insieme $A = \{x \mid x \notin x\}$. Quindi $x \in A \Leftrightarrow x \notin x$, da cui segue che $A \in A \Leftrightarrow A \notin A$.

$h(x) = g(r(x))$ e si giunga ad un assurdo rispetto all'enunciato del Lemma 15.1).

(2) Decidere se, per ogni x: φ_x è costante;
(3) Decidere se per ogni x e y: $\varphi_x = \varphi_y$. (Traccia: si mostri che allora si saprebbe decidere il problema 1).
(4) Decidere se per ogni x, y e z: $\varphi_x(y) = z$.

CHAPTER 16

Calcolabilità e Linguaggi di Programmazione

In questo capitolo vedremo alcune applicazioni tra le più importanti della teoria elementare della calcolabilità vista nel precedente capitolo al progetto ed alla implementazione dei moderni linguaggi di programmazione. In particolare metteremo in evidenza come l'espressività (in senso lato) di un linguaggio dipenda dalla possibilità offerta da quest'ultimo di codificare uno dei formalismi affrontati per definire il concetto di effettiva calcolabilità. Questi problemi hanno dato origine, negli anni '60, alla ricerca di linguaggi di programmazione ad alto livello, in grado cioè di esprimere tutte le funzioni effettivamente calcolabili. Tale ricerca ha dato origine a una vasta famiglia di linguaggi come ALGOL e PASCAL, progenitori dei più moderni linguaggi C, C++, e Java. Inoltre vedremo come l'esistenza (almeno teorica) di strumenti ampiamente utilizzati nella moderna pratica informatica per manipolare programmi, come i compilatori, gli interpreti, e i programmi di specializzazione, dipenda dai risultati di universalità visti nella caratterizzazione delle funzioni calcolabili.

1. Il linguaggio While

Abbiamo fino ad ora definito il concetto di effettiva calcolabilità su una semplicissima struttura dati: i numeri naturali $\mathbb{N}$. Come vedremo, questa apparente restrizione in realtà si dimostra essere perfettamente equivalente ad ogni definizione di calcolabilità data su strutture dati più evolute, quali stringhe, alberi etc. I numeri naturali sono pertanto sufficienti per rappresentare ogni possibile struttura dati su cui si vogliono definire algoritmi.

Al fine di evitare l'aritmetizzare dei programmi, in questo capitolo studieremo un semplice linguaggio di programmazione imperativo chiamato While in grado di manipolare strutture dati più complesse. While è un linguaggio ad alto livello adeguato per rappresentare tutte le funzioni calcolabili che permette di manipolare semplici strutture dati. Le strutture dati impiegate in While sono molto simili a quelle impiegate nei linguaggi Scheme e LISP. Questa estensione ci permetterà di evitare l'aritmetizzazione (operazione di interesse preminentemente matematico) permettendo quindi di considerare direttamente i programmi come dati. Questo, come abbiamo visto, è il passo fondamentale per definire il concetto di funzione universale.

2. Strutture dati

Le strutture dati del linguaggio WHILE sono alberi binari. Questi permettono di rappresentare la *sintassi* concreta dei programmi come dati. Sia A un insieme *finito* di atomi, o espressioni elementari, e nil l'albero vuoto. L'insieme degli alberi $\mathbb{D}_A$ è definito ricorsivamente come il più piccolo insieme tale che:

$$
\begin{aligned}
&\mathsf{nil} \in \mathbb{D}_A \\
&A \subseteq \mathbb{D}_A \\
&\forall d_1, d_2 \in \mathbb{D}_A.\ (d_1.d_2) \in \mathbb{D}_A
\end{aligned}
$$

In altri termini, se $A = \{a_1, \ldots, a_n\}$, $\mathbb{D}_A$ è il linguaggio generato dalla grammatica CF: $D_A \longrightarrow \mathsf{nil} \mid a_1 \mid \cdots \mid a_n \mid (D_A.D_A)$.

Ad esempio, l'albero in Figura 1 è rappresentato come: $((a.((b.b).c)).d)$, dove $A = \{a, b, c, d\}$ è il corrispondente insieme di atomi.

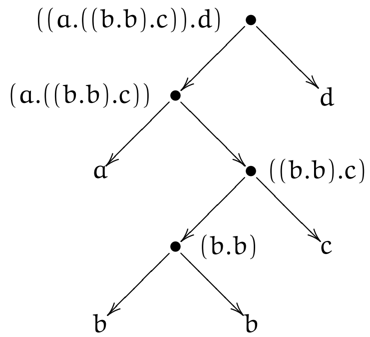


FIGURE 1. L'albero $((a.((b.b).c)).d)$

TEOREMA 16.1. $\mathbb{D}_A$ *è isomorfo a* $\mathbb{N}$

TRACCIA. È semplice definire una biiezione tra $\mathbb{N}$ e $\mathbb{D}_A$. Ad esempio si possono prima contare gli alberi di altezza 0, poi quelli di altezza 1 e così via. Ad esempio, se $A = \{a_1, \ldots, a_n\}$:

0	1	2	$\ldots$	n	$n+1$	$n+2$	$\ldots$	n^2+3n+2	n^2+3n+3	$\ldots$
nil	a_1	a_2	$\ldots$	a_n	$(\mathsf{nil}.\mathsf{nil})$	$(\mathsf{nil}.a_1)$	$\ldots$	$(a_n.a_n)$	$(\mathsf{nil}.(\mathsf{nil}.\mathsf{nil}))$	$\ldots$

$\square$

3. Sintassi

Nel seguito assumeremo di avere a disposizione un insieme infinito e numerabile di variabili Var. La sintassi di WHILE è definita da una grammatica CF nel modo seguente dove $x, y \in \mathsf{Var}$, e $d \in \mathbb{D}_A$ (per essere precisi, dovremmo scrivere D_A in luogo di d; tuttavia ci farà comodo nel seguito identificare con d un arbitrario elemento di $d \in \mathbb{D}_A$, pertanto usiamo questa notazione):

$$
\begin{array}{lcl}
\mathsf{Exp} & \to & x \mid d \mid \mathsf{cons}(\mathsf{Exp}_1, \mathsf{Exp}_2) \mid \mathsf{hd}(\mathsf{Exp}) \mid \mathsf{tl}(\mathsf{Exp}) \mid \mathsf{Exp}_1 = \mathsf{Exp}_2 \\
\mathsf{Com} & \to & x := \mathsf{Exp} \mid \mathsf{Com}_1; \mathsf{Com}_2 \mid \mathsf{skip} \mid \textbf{while}\ \mathsf{Exp}\ \textbf{do}\ \mathsf{Com}\ \textbf{endw} \\
\mathsf{Prog} & \to & \textbf{read}(\mathsf{Listavar}); \mathsf{Com}; \textbf{write}(\mathsf{Listavar}) \\
\mathsf{Listavar} & \to & x \mid x, \mathsf{Listavar}
\end{array}
$$

Assumiamo inoltre che in $\mathsf{Listavar}$ le variabili siano tutte distinte. Si osservi come in questo linguaggio non vi sia la dichiarazione di tipo per le variabili. Tutte le variabili infatti possono assumere solo valori di "tipo" $\mathbb{D}_A$.

ESEMPIO 16.2. Il seguente programma WHILE è ben definito [darne l'albero di derivazione per esercizio].

$$
\begin{array}{l}
\textbf{read}(x); \\
y := \mathsf{nil}; \\
\textbf{while}\ x\ \textbf{do} \\
\quad y := \mathsf{cons}(\mathsf{hd}(x), y); \\
\quad x := \mathsf{tl}(x) \\
\textbf{endw}; \\
\textbf{write}(y)
\end{array}
$$

4. Semantica

La semantica di un programma WHILE è data in termini di un sistema di transizione [**26, 31**]. Per definirla, dobbiamo prima definire il concetto di *stato*. Questo rappresenta la memoria della macchina preposta all'esecuzione di programmi WHILE. La memoria può essere vista come una nastro in cui ogni cella corrisponde esattamente ad una variabile in Var e può contenere esclusivamente valori in $\mathbb{D}_A$ oppure non contenere alcun valore (essere indefinita o cella vuota)—in tal caso diremo che contiene $\perp$. Pertanto, uno stato σ è rappresentabile come una funzione parzialmente definita da variabili in valori $\sigma : \mathsf{Var} \to \mathbb{D}_A \cup \{\perp\}$. L'insieme di tutti gli stati è detto State. Se $\sigma \in \mathsf{State}$, allora il valore dello stato (memoria) σ in corrispondenza alla variabile $x \in \mathsf{Var}$ è dato dal valore della funzione σ in x, ovvero $\sigma(x) \in \mathbb{D}_A$. Un comando ha lo scopo di modificare lo stato corrente.

$\mathcal{E}[\![x]\!]\sigma = \sigma(x)$	$\mathcal{E}[\![d]\!]\sigma = d$
$\mathcal{E}[\![\mathsf{cons}(E_1, E_2)]\!]\sigma = (\mathcal{E}[\![E_1]\!]\sigma.\mathcal{E}[\![E_2]\!]\sigma)$	$\mathcal{E}[\![E_1 = E_2]\!]\sigma = (\mathcal{E}[\![E_1]\!]\sigma = \mathcal{E}[\![E_2]\!]\sigma)$
$\mathcal{E}[\![\mathsf{tl}(E)]\!]\sigma = \begin{cases} c & \text{se } \mathcal{E}[\![E]\!]\sigma = (t.c) \\ \mathsf{nil} & \text{altrimenti} \end{cases}$	$\mathcal{E}[\![\mathsf{hd}(E)]\!]\sigma = \begin{cases} t & \text{se } \mathcal{E}[\![E]\!]\sigma = (t.c) \\ \mathsf{nil} & \text{altrimenti} \end{cases}$

TABLE 1. Semantica delle espressioni

La *semantica intuitiva* dei comandi di WHILE è data nel seguente modo:

> I programmi hanno variabili di ingresso (**read**) e uscita (**write**). Possono usare inoltre quante altre variabili si voglia, prese da Var. I comandi modificano lo stato nel seguente modo:
> - skip lascia lo stato invariato;
> - $x := \mathsf{Exp}$ memorizza il valore di Exp calcolato nello stato corrente nella cella di memoria rappresentata dalla variabile x;
> - $C_1; C_2$ corrisponde all'esecuzione del comando C_2 nello stato lasciato dall'esecuzione del comando C_1;
> - **while** E **do** C **endw** esegue il comando C fino a che l'albero rappresentato dalla espressione E non è vuoto.
>
> La semantica di un programma:
>
> $$\textbf{read}(x_1, \dots, x_n);\ C;\ \textbf{write}(y_1, \dots, y_m)$$
>
> è quindi data dalla funzione che associa ad ogni variabile in ingresso $x_1, \dots, x_n$ il valore delle variabili in uscita $y_1, \dots, y_m$ determinato nello stato modificato da C.

Possiamo quindi dare la semantica formale di WHILE induttivamente sulla sintassi come sistema di transizione:

Semantica delle espressioni: Ogni espressione rappresenta un albero. Al fine di valutare espressioni contenenti variabili è necessario per la loro valutazione ricorrere alla memoria. La semantica delle espressioni è data dunque da una funzione di interpretazione semantica:

$$\mathcal{E} : \mathsf{Exp} \times \mathsf{State} \to \mathbb{D}_A \cup \{\bot\}$$

definita come in Tabella 1. Il risultato dell'espressione $E_1 = E_2$ può essere false oppure true; queste entità sintattiche possono essere usate liberamente come abbreviazioni degli alberi nil e $(\mathsf{nil.nil})$, rispettivamente.

Semantica dei comandi: La semantica dei comandi è definita induttivamente sulla sintassi dal sistema di transizione in Tabella 2. Le *configurazioni* del sistema di transizione sono definite da coppie: $\mathtt{Com} \times \mathtt{State}$. Ogni configurazione $\langle \mathsf{C}, \sigma \rangle$ rappresenta il comando C da eseguire e lo stato σ in cui questo viene eseguito. Il sistema di transizione definisce una relazione $\longrightarrow \subseteq (\mathtt{Com} \times \mathtt{State}) \times (\mathtt{Com} \times \mathtt{State})$ che rappresenta il generico passo di calcolo. Nel seguito, se il comando è vuoto (stringa vuota ε), allora la configurazione corrispondente $\langle \varepsilon, \sigma \rangle$ è rappresentata semplicemente come σ.

$$\frac{\mathcal{E}[\![\mathsf{E}]\!]\sigma = \mathsf{d}}{\langle \mathsf{x} := \mathsf{E}, \sigma \rangle \longrightarrow \sigma[\mathsf{d}/\mathsf{x}]} \qquad \frac{}{\langle \mathsf{skip}, \sigma \rangle \longrightarrow \sigma}$$

$$\frac{\langle \mathsf{C}_1, \sigma \rangle \longrightarrow \sigma'}{\langle \mathsf{C}_1; \mathsf{C}_2, \sigma \rangle \longrightarrow \langle \mathsf{C}_2, \sigma' \rangle}$$

$$\frac{\mathcal{E}[\![\mathsf{E}]\!]\sigma = \mathsf{nil}}{\langle \textbf{while } \mathsf{E} \textbf{ do } \mathsf{C} \textbf{ endw}, \sigma \rangle \longrightarrow \sigma}$$

$$\frac{\mathcal{E}[\![\mathsf{E}]\!]\sigma \neq \mathsf{nil}}{\langle \textbf{while } \mathsf{E} \textbf{ do } \mathsf{C} \textbf{ endw}, \sigma \rangle \longrightarrow \langle \mathsf{C}; \textbf{while } \mathsf{E} \textbf{ do } \mathsf{C} \textbf{ endw}, \sigma \rangle}$$

TABLE 2. Semantica dei comandi

Semantica dei Programmi: La semantica di un programma WHILE è definita a partire dalla semantica dei comandi, o meglio dalla chiusura transitiva $\longrightarrow^*$ della relazione di transizione dei comandi. Essa definisce una funzione parziale da alberi in alberi, ovvero:

$$[\![\cdot]\!]^{\mathcal{W}} : \mathtt{Prog} \to (\mathbb{D}_{\mathsf{A}} \to \mathbb{D}_{\mathsf{A}} \cup \{\uparrow\})$$

La semantica $[\![\mathsf{P}]\!]^{\mathcal{W}}$ di un programma

$$\mathsf{P} = \textbf{read}(x_1, \ldots, x_n);\ \mathsf{C};\ \textbf{write}(y_1, \ldots, y_m)$$

è definita come segue, per ogni $d_1, \ldots, d_n \in \mathbb{D}_{\mathsf{A}}$:

$$[\![\mathsf{P}]\!]^{\mathcal{W}}(d_1, \ldots, d_n) = \begin{cases} e_1, \ldots, e_m & \text{se } \langle \mathsf{C}, [d_1/x_1, \ldots, d_n/x_n] \rangle \longrightarrow^* \sigma' \\ & \text{e } \sigma'(y_1) = e_1, \ldots, \sigma'(y_m) = e_m \\ \uparrow & \text{altrimenti} \end{cases}$$

ESEMPIO 16.3. Come esempio di semantica di un semplice programma WHILE, consideriamo la semantica del programma P dell'Esempio 16.2 che calcola l'inverso di una lista rappresentata come un albero sbilanciato a sinistra.

ESERCIZIO 16.4. Il caso del $\perp$, o meglio della propagazione del valore indefinito, è trattato in modo incompleto nella Tabella 1. Per esercizio, si pensi a come modificare la semantica delle espressioni in modo da migliorare questo aspetto.

ESERCIZIO 16.5. Si fornisca una semantica formale per i comandi:

- $\mathsf{if}\ \mathsf{Exp}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, dal significato intuitivo seguente: si valuti l'espressione Exp; se è diverso da nil si esegua l'istruzione C_1, altrimenti si esegua l'istruzione C_2.
- $\mathsf{for}\ x := \mathsf{Exp}\ \mathsf{do}\ C\ \mathsf{endfor}$. La semantica intuitiva di questo importante costrutto, ripreso nella Sezione 6, è la seguente: si valuta Exp sul valore dello stato iniziale. Essa sarà un albero che necessita di esattamente $n \geq 0$ operazioni di tipo tl per restituire l'espressione nil. L'istruzione C viene quindi ripetuta per n volte; x all'inizio ha il valore iniziale di Exp, poi viene via via ridotto fino a raggiungere $x = \mathsf{nil}$. x e le variabili che occorrono in Exp non possono essere modificate entro C.

ESERCIZIO 16.6. Si mostri, scrivendo frammenti di codice WHILE in cui sono introdotte (se servono) ulteriori variabili, che i due costrutti suddetti possono essere definiti all'interno del linguaggio WHILE (in altri termini che il codice scritto ha, sulle variabili originarie, la stessa semantica).

ESERCIZIO 16.7. Si mostri come il comando $\mathsf{if\text{-}then\text{-}else}$ dell'esercizio precedente possa essere simulato dal comando for definito nello stesso esercizio. Suggerimento: se α è l'espressione dell'$\mathsf{if\text{-}then\text{-}else}$, si assegnino le variabili u e v nel seguente modo: $u := (\alpha = \mathsf{nil}); v := (u = \mathsf{nil})$; Si eseguano dunque due cicli for: uno controllato da u che esegue C_2 e uno da v che esegue C_1.

ESERCIZIO 16.8. Sia $A = \{a, \dots, z\}$. Si scriva un Programma WHILE che verifica se un elemento x_1 di A è presente nell'albero x_2 (restituisce $y = 1$ in caso affermativo, $y = 0$ altrimenti).

5. Espressività di WHILE e Turing completezza

È possibile rappresentare i numeri naturali in $\mathbb{D}_A$. La rappresentazione in $\mathbb{D}_A$ del numero n è denotata $\underline{n}$. La seguente definizione illustra uno dei modi a nostra disposizione di codificare i numeri naturali:

$$\underline{0} = \mathsf{nil}$$
$$\underline{n+1} = (\mathsf{nil} \,.\, \underline{n})$$

Utilizzando il cons e la sua semantica, anche $\mathsf{cons}(\underbrace{\mathsf{nil}, \mathsf{cons}(\mathsf{nil}, \dots, \mathsf{cons}(\mathsf{nil}}_{n}, \mathsf{nil}) \cdots)$ rappresenta il numero n.

DEFINIZIONE 16.9. Una funzione $f : \mathbb{N}^k \to \mathbb{N}$ è WHILE-calcolabile se esiste un programma WHILE P tale che per ogni $x_1, \ldots, x_k \in \mathbb{N}$:

$$[\![P]\!]^W(\underline{x_1}, \ldots, \underline{x_k}) = \underline{f(x_1, \ldots, x_k)}$$

ESEMPIO 16.10. Il seguente programma WHILE calcola la funzione $f(x, y) = x + y$:

$$
\begin{array}{l}
\textbf{read}(x_1, x_2); \\
\textbf{while } x_2 \textbf{ do} \\
\quad x_1 := \mathsf{cons}(\mathsf{nil}, x_2); \\
\quad x_2 := \mathsf{tl}(x_2) \\
\textbf{endw}; \\
\textbf{write}(x_1)
\end{array}
$$

ESERCIZIO 16.11. Si scrivano i programmi WHILE per il calcolo della differenza tra numeri naturali, del prodotto, delle operazioni `div`, `mod`, e del test $x < y$.

L'ampiezza della classe delle funzioni WHILE-calcolabili è determinata dal seguente teorema, che conferma la Tesi di Church-Turing anche per i linguaggi di programmazione imperativa.

TEOREMA 16.12. *$f : \mathbb{N} \to \mathbb{N}$ è WHILE-calcolabile sse è Turing-calcolabile.*

PROOF. ($\to$) Per induzione sulla sintassi di WHILE, costruiamo una MdT che calcola la medesima funzione calcolata dal corrispondente programma WHILE. Poiché i programmi sono costituiti da sequenze di comandi, è sufficiente indurre sulla complessità sintattica dei comandi. Similmente a quanto fatto nella dimostrazione del Teorema 12.14, poniamo delle restrizioni alle MdT che costruiamo in modo tale da semplificare l'applicazione dell'ipotesi induttiva.

(1) Per semplicità, l'alfabeto delle MdT coincide con l'alfabeto della grammatica usata per costruire $\mathbb{D}_A$.
(2) Tali macchine hanno, prima dell'esecuzione, a sinistra dell'input la descrizione del contenuto delle variabili $x_1, x_2, x_3, \ldots$ (solo un numero finito di queste è diverso da 0). Alla fine della computazione (se termina) a sinistra della testina vi sarànno invece i valori finali di tali variabili (poiché all'inizio solo un numero finito di queste era diverso da 0 e la computazione ha per ipotesi lunghezza finita, anche alla fine avremo solo un numero finito di $x_1, x_2, x_3, \ldots$ diverse da 0).
(3) Ci serviranno MdT per il calcolo di espressioni. In tal caso assumiamo che la parte sinistra del nastro sia come sopra, mentre immediatamente a destra della testina sia memorizzata l'espressione calcolata.
(4) Inoltre le macchine sono definite in modo tale da essere indipendenti dal contenuto del nastro sulla parte destra rispetto alla posizione iniziale della testina.

Base: Vi sono 3 MdT di base:

- Calcolo di una espressione E: In E vi saranno dati da $\mathbb{D}_{\mathsf{A}}$ e variabili. Si costruisce l'espressione a partire da ciò. [Esercizio], mettendo il risultato subito a destra della posizione finale della testina.
- istruzione skip: una MdT che non fa nulla soddisfa a tutti i requisiti.
- $x_i := \mathsf{E}$. Si parte dal calcolo dell'espressione (vedi sopra). Una volta pronta, si mette il valore dell'espressione calcolata nella posizione della variabile x_i, dopo avere spostato a destra i contenuti delle variabili $x_1, \dots, x_{i-1}$.

Passo: Come caso induttivo, affrontiamo i 3 casi possibili:

(1) Supponiamo che la MdT M_1 simuli l'istruzione C_1 e la MdT M_2 simuli l'istruzione C_2:
   Si attiva la macchina M_1. Se questa termina, si attiva la macchina M_2.

(2) Analizziamo il caso del comando **while**. Supponiamo di avere, per ipotesi induttiva, una MdT M_C per il comando C ed una M_E per la valutazione della espressione E. La seguente MdT corrisponde alla esecuzione del comando **while** E **do** C **endw**:
   Si attiva M_E. Se sul nastro a destra della testina vi è nil, allora termina. Altrimenti si attiva M_C e, qualora termini, si reitera.

($\rightarrow$) Sia f una funzione Turing calcolabile. Dal Teorema 12.14, sappiamo che $f \in \mathcal{PR}$. La dimostrazione prosegue per induzione sulla struttura delle funzioni parziali ricorsive. Costruiamo ed utilizziamo solo programmi che *inizializzano* tutte le variabili che usano.
Base:

- $\lambda x.\, 0$: $\mathbf{read}(x); x := \mathsf{nil}; \mathbf{write}(x)$;
- $\lambda x.\, x+1$: $\mathbf{read}(x); x := \mathsf{cons}(\mathsf{nil}, x); \mathbf{write}(x)$;
- $\lambda x_1 \dots x_n.\, x_i$: $\mathbf{read}(x_1, \dots, x_n); \mathsf{skip}; \mathbf{write}(x_i)$.

Passo: Analizziamo i tre casi induttivi.

(1) Supponiamo per ipotesi induttiva che esista un programma P_h:

$$\mathbf{read}(x_1^h, \dots, x_k^h); C_h; \mathbf{write}(y^h)$$

che calcola la funzione h e $P_{g_1}, \dots, P_{g_k}$:

$$\mathbf{read}(x_1^{g_i}, \dots, x_n^{g_i}); C_{g_i}; \mathbf{write}(y^{g_i})$$

che calcolano le funzioni $g_1, \dots, g_k$. Supponiamo inoltre che tali programmi usino tutti variabili diverse (altrimenti le ridenominiamo). Sia μ il massimo indice di variabile usato in quei programmi. Allora la funzione $f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n))$ si calcola nel modo seguente:

$\textbf{read}(x_{\mu+1}, \ldots, x_{\mu+n});$

$x_1^{g_1} := x_{\mu+1}; \ldots ; x_n^{g_1} := x_{\mu+n}; C_{g_1};$

$\vdots$

$x_1^{g_k} := x_{\mu+1}; \ldots ; x_n^{g_k} := x_{\mu+n}; C_{g_k}; x_1^h := y^{g_1}; \ldots ; x_k^h := y^{g_k}; C_h;$

$\textbf{write}(y_h)$

(2) Sia ora f definita per ricorsione primitiva (per semplicità denotazionale usiamo x invece di $x_1, \ldots, x_n$):

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

e supponiamo, per ipotesi induttiva, che alle funzioni g ed h corrispondano i programmi:

$$\begin{array}{rll} \textbf{read}(x_1^g); & C_g; & \textbf{write}(y_g) \\ \textbf{read}(x_1^h, x_2^h, x_3^h); & C_h; & \textbf{write}(y_h) \end{array}$$

Supponiamo ancora che tali programmi usino variabili diverse e sia μ il massimo indice di variabile usato. Il seguente programma implementa la funzione f:

$\textbf{read}(x_{\mu+1}, x_{\mu+2});$

$x_g := x_{\mu+1}; C_g; x_{\mu+3} := y_g; x_{\mu+4} := nil;$

$\textbf{while } x_{\mu+2} \textbf{ do}$

$\quad x_1^h := x_{\mu+1}; x_2^h := x_{\mu+4}; x_h^3 := x_{\mu+3};$

$\quad C_h;$

$\quad x_{\mu+2} := \mathsf{tl}(x_{\mu+2}); x_{\mu+4} := \mathsf{cons}(\mathsf{nil}, x_{\mu+4});$

$\quad x_{\mu+3} := y_h;$

$\textbf{endw};$

$\textbf{write}(x_{\mu+3})$

(3) Sia infine φ definita per minimizzazione a partire da una funzione $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$, calcolata da:

$$\textbf{read}(x_1^f, \ldots, x_n^f, x_{n+1}^f); C_f; \textbf{write}(y_f).$$

e siano $x_1, \dots, x_n, y, \mathtt{temp}$ variabili non usate in esso. Il seguente programma calcola la funzione φ:

$$
\begin{array}{l}
\textbf{read}(x_1, \dots, x_n);\\
y := \mathtt{nil};\\
x_f^1 := x_1; \dots ; x_n^f := x_n; x_{n+1}^f := y;\\
C_f;\\
\textbf{while } y_f \textbf{ do}\\
\quad y := \mathsf{cons}(\mathsf{nil}, y);\\
\quad x_f^1 := x_1; \dots ; x_n^f := x_n; x_{n+1}^f := y;\\
\quad C_f;\\
\textbf{endw};\\
\textbf{write}(y)
\end{array}
$$

$\square$

È possibile generalizzare quanto visto per il linguaggio WHILE ad un arbitrario linguaggio di programmazione. Sia $\mathcal{L}$ un linguaggio di programmazione che definisce funzioni su un insieme di dati $\mathbb{D}$. Nel seguito con $\mathcal{L}$ indicheremo la classe dei programmi sintatticamente corretti scritti nel linguaggio $\mathcal{L}$. Supponiamo che esista una codifica univoca dei numeri naturali in $\mathbb{D}$, ovvero per ogni n: $\underline{n} \in \mathbb{D}$ e per ogni $n \neq m$: $\underline{n} \neq \underline{m}$. Sia $[\![\cdot]\!]^{\mathcal{L}}$ la semantica del linguaggio $\mathcal{L}$ definita in modo formale:[1]

$$[\![\cdot]\!]^{\mathcal{L}} : \mathcal{L} \to (\mathbb{D} \to \mathbb{D} \cup \{\uparrow\})$$

Una funzione $f : \mathbb{N}^k \to \mathbb{N}$ è $\mathcal{L}$-calcolabile se esiste un programma $P \in \mathcal{L}$ tale che per ogni $x_1, \dots, x_k \in \mathbb{N}$:

$$[\![P]\!](\underline{x_1}, \dots, \underline{x_k}) = \underline{f(x_1, \dots, x_k)}$$

DEFINIZIONE 16.13. Un linguaggio di programmazione $\mathcal{L}$ è detto *Turing-completo* se l'insieme delle funzioni $\mathcal{L}$-calcolabili coincide con la classe delle funzioni Turing-calcolabili.

Il linguaggio WHILE è dunque Turing-completo per il Teorema 16.12. In particolare, ogni linguaggio di programmazione sufficientemente espressivo per codificare i numeri naturali e comprendente i comandi di base di *assegnamento*, *composizione* ed *iterazione condizionata* tipo WHILE è Turing-completo. Infatti, per dimostrare la Turing-completezza di un linguaggio è sufficiente dimostrare che questo è in grado di *simulare* i costrutti di base di WHILE.

[1] È noto che ad ogni linguaggio di programmazione, anche complesso e di alto livello, è possibile associare una semantica formale [**31, 21**].

Esercizio 16.14. Una Macchina di Turing si può rappresentare mediante un insieme (lista) di quintuple (lista di 5 elementi). Si scriva un programma While in grado di simulare il comportamento (interpretare) una data macchina di Turing (questa è una dimostrazione alternativa che ogni funzione Turing-calcolabile è While-calcolabile).

6. For-calcolabilità e funzioni primitive ricorsive

In questa sezione presenteremo una variante al linguaggio While. Sostituiremo al comando **while** due comandi, uno di selezione ed uno iterativo molto usati in programmazione: l'if-then-else ed il ciclo for. Nell'esercizio 16.5 tali costrutti sono stati definiti e la loro simulazione utilizzando il **while** è stata richiesta come (semplice) esercizio. Mostreremo ora come un linguaggio che disponga di questi due costrutti più l'assegnamento permette una espressività pari a quella del formalismo delle funzioni primitive ricorsive.

Nelle stesse ipotesi generali dei programmi While, definiamo un programma For mediante una grammatica CF nel modo seguente dove $x, y \in \mathsf{Var}$, e $d \in \mathbb{D}_A$ (in particolare, d può essere nil):

$$\begin{array}{lcl} \mathsf{Exp} & \to & x \mid d \mid \mathsf{cons}(\mathsf{Exp}_1, \mathsf{Exp}_2) \mid \mathsf{hd}(\mathsf{Exp}) \mid \mathsf{tl}(\mathsf{Exp}) \mid \mathsf{Exp}_1 = \mathsf{Exp}_2 \\ \mathsf{Com} & \to & x := \mathsf{Exp} \mid \mathsf{Com}_1; \mathsf{Com}_2 \mid \mathsf{skip} \mid \\ & & \mathsf{if\ Exp\ then\ } C_1 \mathsf{\ else\ } C_2 \mid \mathsf{for\ } x := \mathsf{Exp\ do\ } C \mathsf{\ endfor} \\ \mathsf{Prog} & \to & \mathbf{read}(x_1, \ldots, x_n); \mathsf{Com}; \mathbf{write}(y_1, \ldots, y_m) \end{array}$$

La semantica intuitiva dei due costrutti è definita nell'esercizio 16.5. Inoltre, come mostrato nell'esercizio 16.7, il costrutto if-then-else è simulabile dal for e dunque superfluo (non ne parleremo nelle dimostrazioni). Una funzione $f : \mathbb{N}^k \to \mathbb{N}$ è For-calcolabile se se esiste un programma For P tale che per ogni $x_1, \ldots, x_k \in \mathbb{N}$:

$$[\![P]\!]^{\mathcal{L}}(\underline{x_1}, \ldots, \underline{x_k}) = \underline{f(x_1, \ldots, x_k)}$$

Nel teorema seguente si forniranno delle definizioni più tecniche e precise, ma la sua comprensione e definizione ne può fare a meno.

Teorema 16.15. *$f : \mathbb{N}^m \longrightarrow \mathbb{N}$ è For-calcolabile se e solo se f è primitiva ricorsiva.*

Traccia. Per mostrare che una funzione primitiva ricorsiva è For-calcolabile è sufficiente ripetere la seconda parte della dimostrazione del Teorema 16.12 fino al passo induttivo 1. Bisogna mostrare anche il passo induttivo 2 con il nuovo linguaggio. Supponiamo che alla funzione $h : \mathbb{N}^3 \longrightarrow \mathbb{N}$ corrisponda il programma: $\mathbf{read}(x_h^1, x_h^2, x_h^3); C_h; \mathbf{write}(y_h)$. Il seguente programma implementa la funzione f

definita per ricorsione primitiva:

$$
\begin{array}{l}
\textbf{read}(n, x); \\
x_g := x; \\
C_g; \\
temp := y_g; \\
\textsf{for } w := n \textsf{ do} \\
\quad x_h^1 := temp; x_h^2 := w; x_h^3 := x; \\
\quad C_h; \\
\quad temp := y_h; \\
\textsf{endfor;} \\
\textbf{write}(temp)
\end{array}
$$

Leggermente più tecnico è il viceversa. Dobbiamo mostrare che ogni funzione calcolata da un programma For è calcolabile con una funzione primitiva ricorsiva.

Innanzitutto partiamo dalle seguenti assunzioni:

- Assumiamo che ogni espressione usata nel programma sia tale che per ogni istanziazione delle variabili in essa in numerali si ottiene un numerale. Questa assunzione è eliminabile grazie all'isomorfismo tra $\mathbb{D}_A$ e $\mathbb{N}$, ma snellisce la dimostrazione.
- Con le stesse motivazioni, assumiamo che ogni variabile non presente tra quelle di input sia immediatamente inizializzata a nil ($\underline{0}$).
- Assumiamo che un programma usi tutte e sole le variabili $v_1, \dots, v_p$ e che le variabili $x_1, \dots, x_n$ di input siano le prime n variabili $v_1, \dots, v_p$. In generale si avrà che $m \leq p$ e $n \leq p$ (n è il numero di variabili di output).
- Ogni comando C è visto come un insieme di p funzioni da $\mathbb{N}^p \longrightarrow \mathbb{N}$.
- Le m funzioni calcolate da un programma P

  $$\textbf{read}(v_1, \dots, v_n); C; \textbf{write}(v_{i_1}, \dots, v_{i_m})$$

  saranno dunque le funzioni $f^P_{i_1}, \dots, f^P_{i_m}$ definite nel seguente modo: se $f^C_{i_1}, \dots, f_{i_m}$ sono le funzioni calcolate da C sulle variabili di output, allora, per $j = 1, \dots, m$

  $$f^P_{i_j}(x_1, \dots, x_n) = f^C_{i_j}(\underbrace{x_1, \dots, x_n, 0, \dots, 0}_{p})$$

  ovvero con le variabili del programma non presenti tra gli input inizializzate a 0. Ovviamente, se le f^C sono primitive ricorsive, lo saranno anche le f^P.

- Per brevità, con $\bar{x}$ si denota la lista $x_1, \ldots, x_p$.

Ci si concentra dunque sulle funzioni calcolate dai comandi.

Base:

skip**:** Per ogni $i = 1, \ldots, p$ si avrà che $f_i^C(\bar{x}) = x_i$ ovvero una proiezione.

$v_j := \mathsf{E}$**:** Per ogni $i = 1, \ldots, p$, $i \neq j$ si avrà che $f_i^C(\bar{x}) = x_i$ ovvero una proiezione. Per quanto riguarda $f_j^C(\bar{x})$, per l'assuzione sopra, o E è un numerale $\ell \in \mathbb{N}$ e dunque $f_i^C(\bar{x}) = \ell$ (ovviamente p.r.) oppure E è del tipo $\mathsf{cons}(\mathsf{nil}, \cdots, \mathsf{cons}(\mathsf{nil}, v_k) \cdots)$ ovvero $f_i^C(\bar{x}) = x_k + \ell$ per qualche ℓ intero. Anche in questo caso la funzione è p.r.

Passo:

$\mathsf{C}_1; \mathsf{C}_2$**:** Siano, per ipotesi induttiva, per $i = 1, \ldots, p$ le funzioni $f_i^{C_1}$ e $f_i^{C_2}$. Allora le funzioni associate alla composizione sono, per $i = 1, \ldots, p$:

$$f_i^{C_2}(f_1^{C_1}(\bar{x}), \ldots, f_p^{C_1}(\bar{x}))$$

$\mathsf{for}\ v_j := \mathsf{E}\ \mathsf{do}\ \mathsf{C};\ \mathsf{endfor}$**:** Siano, per ipotesi induttiva e per $i = 1, \ldots, p$, le funzioni $f_i^C(\bar{x})$ primitive ricorsive equivalenti al comando C e f_i^E quelle associate al comando $v_j := \mathsf{E}$. Definisco allora per $i = 1, \ldots, p$, le funzioni h_i^j (l'apice j ricorda che stiamo parlando della variabile v_j) nel seguente modo:

$$\begin{cases} h_i^j(\bar{x}, 0) & = & x_i & i \neq j \\ h_j^j(\bar{x}, 0) & = & 0 & \\ h_i^j(\bar{x}, n+1) & = & f_i^C(h_1^j(\bar{x}, n), \ldots, h_p^j(\bar{x}, n)) & i \neq j \\ h_j^j(\bar{x}, n+1) & = & h_j^j(\bar{x}, n) + 1 & \end{cases}$$

La funzione da associare al comando for, per ogni i, sarà:

$$g_i(\bar{x}) = h_i^j(\bar{x}, f_j^E(\bar{x}))$$

$\square$

7. Interpreti e Metaprogrammazione

In questa sezione introduciamo il concetto di interprete e di metaprogrammazione. Per metaprogrammazione si intende la costruzione di programmi che manipolano programmi. Questa possibilità, sancita dalla universalità dei sistemi di calcolo Turing-completi, risiede nel fatto che i programmi possono essere codificati in modo univoco nei dati manipolati dai programmi stessi. La MdT universale $\mathbb{U}$ è il prototipo di metaprogramma: essa prende in input l'indice x di una data MdT M ed un dato y, e restituisce in output il risultato ottenibile attivando la macchina M_x sul dato y:

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

Traslando questo ragionamento sui linguaggi di programmazione, si ottiene il concetto di *interprete*. Siano $\mathcal{L}$ ed $\mathcal{S}$ due linguaggi di programmazione Turing-completi; assumiamo che operino sul medesimo insieme di dati $\mathbb{D}$.

DEFINIZIONE 16.16. Un programma $\mathtt{int} \in \mathcal{L}$ tale che, per ogni $\mathcal{S}$-programma P e per ogni dato $\mathsf{d} \in \mathbb{D}$:

$$[\![\mathtt{int}]\!]^{\mathcal{L}}(\mathsf{P}, \mathsf{d}) = [\![\mathsf{P}]\!]^{\mathcal{S}}(\mathsf{d})$$

è un *interprete* in $\mathcal{L}$ di $\mathcal{S}$-programmi (o semplicemente di $\mathcal{S}$).

Ovviamente, se $[\![\mathsf{P}]\!]^{\mathcal{S}}(\mathsf{d}) \uparrow$, allora anche $[\![\mathtt{int}]\!]^{\mathcal{L}}(\mathsf{P}, \mathsf{d}) \uparrow$.

L'esistenza di un interprete è assicurata dalla Turing-completezza dei linguaggi in oggetto. In altri termini in $\mathcal{L}$ si deve simulare l'esecuzione del $\mathcal{S}$-programma P sull'input d istruzione per istruzione. Per far ciò ci si baserà sulla semantica (formalmente definita) del programma P nel linguaggio $\mathcal{S}$. Un interprete in $\mathcal{L}$ per $\mathcal{L}$ programmi è detto *metainterprete* del linguaggio $\mathcal{L}$.

Vedremo nel dettaglio come realizzare un metainterprete del linguaggio WHILE. Per altri linguaggi Turing-completi, il lavoro da fare sarà analogo. Innanzitutto è necessario rappresentare in $\mathbb{D}_\mathsf{A}$ i programmi WHILE. A tal fine definiamo un insieme finito di atomi:

$$\mathsf{A} = \{\text{“}:=\text{''}, \text{“}\mathtt{var}\text{''}, \text{“}\mathbf{while}\text{''}, \text{“}\mathtt{cons}\text{''}, \text{“}\mathtt{tl}\text{''}, \text{“}\mathtt{hd}\text{''}, \text{“}\mathtt{nil}\text{''}, \text{“};\text{''}, \text{“}\mathtt{quote}\text{''}\}$$

In $\mathbb{D}_\mathsf{A}$ possiamo codificare i numeri naturali, mediante sintassi concreta:

$$\underline{n} = \underbrace{(\mathsf{nil}.(\mathsf{nil}.\cdots(\mathsf{nil}.}_{n}\,\mathsf{nil})\cdots))$$

Assumendo che l'insieme delle variabili $\mathtt{Var} = \{v_0, v_1, v_2, \ldots\}$, useremo la seguente codifica per codificare un insieme infinito di variabili: la variabile v_i è codificata da $(\mathtt{var}.\underline{i})$. Useremo poi una parola chiave per ogni costrutto, espressione o comando, previsto nella sintassi di WHILE. Useremo $\mathtt{quote}$ per dire che l'espressione (un albero d privo di variabili) non necessita di essere ulteriormente codificata. La seguente funzione $\underline{\ \ } : \text{WHILE} \longrightarrow \mathbb{D}_\mathsf{A}$ di traduzione da programmi WHILE ad alberi $\mathbb{D}_\mathsf{A}$, è definita induttivamente sulla sintassi in tabella 3.

Se vi sono più variabili di ingresso/uscita si userà una lista del tipo: lista $((\mathtt{var}.\underline{1}).((\mathtt{var}.\underline{2}).\cdots(\mathtt{var}.\underline{n}))\cdots)$. Se P è un programma WHILE, allora $\underline{\mathsf{P}} \in \mathbb{D}_\mathsf{A}$ è detta *sintassi concreta* di P. Ad esempio, la sintassi concreta del programma

$$\begin{array}{rcl}
\underline{\mathbf{read}(v_i);C;\mathbf{write}(v_j)} & = & ((var.\underline{i}).(\underline{C}.(var.\underline{j})))\\
\underline{C_1;C_2} & = & (;.(\underline{C_1}.\underline{C_2}))\\
\underline{\mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{endw}} & = & (\mathbf{while}(\underline{E}.\underline{C}))\\
\underline{v_i := E} & = & (:=.((var.\underline{i}).\underline{E}))\\
\underline{v_i} & = & (var.\underline{i})\\
\underline{d} & = & (quote.d)\\
\underline{cons(E_1,E_2)} & = & (cons.(\underline{E_1}.\underline{E_2}))\\
\underline{hd(E)} & = & (hd.\underline{E})\\
\underline{tl(E)} & = & (tl.\underline{E})\\
\underline{E_1 = E_2} & = & (=.(\underline{E_1}.\underline{E_2}))
\end{array}$$

TABLE 3.

nell'Esempio 16.3, assumendo x come v_1 e y come v_2 è data dal seguente albero:

$$\begin{array}{l}
(\quad (var.\underline{1}).\\
\quad (\ (;.\qquad (\ (:=.((var.\underline{2}).(quote\ nil))).\\
\qquad\qquad\quad (\ (\mathbf{while}.((var.\underline{1}).\\
\qquad\qquad\qquad (;.\ (\ (:=.((var.\underline{2}).(cons\ (hd\ (var.\underline{1})(var.\underline{2})))).\\
\qquad\qquad\qquad\quad (:=.((var.\underline{1}).(tl\ (var.\underline{1}))))\\
\quad\ ).\\
\quad (var.\underline{2})\quad))
\end{array}$$

TEOREMA 16.17. *Esiste un interprete in* WHILE *per* WHILE.

PROOF. Scrivere l'interprete per esercizio, utilizzando i costrutti di WHILE, eventualmente estesi con costrutti condizionali tipo `case`. Suggerimento: utilizzare le strutture dati ad albero per rappresentare la sintassi concreta dei programmi ed una pila implementata con una lista per la valutazione delle espressioni. □

La possibilità offerta da un linguaggio Turing-completo della metaprogrammazione è alla base della esistenza di problemi algoritmicamente non risolvibili. I limiti di ciò che è calcolabile nascono quindi dalle potenzialità del sistema di calcolo stesso. Così come l'indecidibilità della terminazione è dimostrata utilizzando funzioni universali, daremo una dimostrazione della indecidibilità della terminazione utilizzando il linguaggio WHILE, senza ricorrere all'aritmetizzazione dello stesso.

Supponiamo esista un programma $\mathtt{halt} \stackrel{\text{def}}{=} \mathbf{read}(x_1, x_2); \mathsf{C}; \mathbf{write}(y) \in$ WHILE, tale che:

$$[\![\mathtt{halt}]\!]^{\mathcal{W}}(\mathsf{P}, \mathsf{d}) = \begin{cases} (\mathsf{nil.nil}) & [\![\mathsf{P}]\!]^{\mathcal{W}}(\mathsf{d}) \downarrow \\ \mathsf{nil} & [\![\mathsf{P}]\!]^{\mathcal{W}}(\mathsf{d}) \uparrow \end{cases}$$

Definiamo il programma $\mathsf{R} \in$ WHILE nel modo seguente:

$$\begin{array}{l} \mathbf{read}(x); \\ x_1 := x; x_2 := x; \\ \mathsf{C}; \\ \mathbf{while}\ y\ \mathbf{do}\ y := y\ \mathbf{endw}; \\ \mathbf{write}(y) \end{array}$$

Sia dunque $\underline{\mathsf{R}}$ la rappresentazione in $\mathbb{D}$ del programma R.

- Se $[\![\mathsf{R}]\!]^{\mathcal{W}}(\underline{\mathsf{R}}) \downarrow$, allora l'output di C, che termina sempre per ipotesi, è $y = (\mathsf{nil.nil})$. Ma allora il programma R entra in loop nel ciclo `while`. Questo implica che $[\![\mathsf{R}]\!]^{\mathcal{W}}(\underline{\mathsf{R}}) \uparrow$.
- Se $[\![\mathsf{R}]\!]^{\mathcal{W}}(\underline{\mathsf{R}}) \uparrow$ allora l'output di C è nil. Ma allora il programma R non entra nel ciclo **while** e dunque termina restituendo nil. Dunque $[\![\mathsf{R}]\!]^{\mathcal{W}}(\underline{\mathsf{R}}) \downarrow$.

Entrambi i casi portano ad un assurdo.

8. Specializzatori e Proiezioni di Futamura

Dalla Turing-completezza dei linguaggi di programmazione derivano altri strumenti importanti per la moderna programmazione di sistemi complessi. Abbiamo già visto che la Turing-completezza assicura l'esistenza di programmi universali, ovvero di interpreti. Vedremo ora che essa permette di implementare programmi che operano come specializzatori di programmi. L'esistenza di questi programmi segue dal Teorema s-m-n, la cui dimostrazione segue a sua volta dalla possibilità di implementare programmi che manipolano programmi.

Supponiamo che $\mathcal{L}$, $\mathcal{S}$ e $\mathcal{T}$ siano linguaggi di programmazione Turing-completi operanti sul medesimo insieme di dati $\mathbb{D}$.

DEFINIZIONE 16.18. Un programma $\mathtt{spec}$ scritto in un linguaggio di *implementazione* $\mathcal{L}$ è uno *specializzatore* $\mathtt{spec} \in \mathcal{L}$ se per ogni programma P scritto in un linguaggio di programmazione $\mathcal{S}$ e per ogni porzione di suo input $\mathsf{s} \in \mathbb{D}$, restituisce un $\mathcal{T}$-programma: $[\![\mathtt{spec}]\!]^{\mathcal{L}}(\mathsf{P}, \mathsf{s})$ tale che per ogni dato $\mathsf{d} \in \mathbb{D}$:

$$[\![[\![\mathtt{spec}]\!]^{\mathcal{L}}(\mathsf{P}, \mathsf{s})]\!]^{\mathcal{T}}(\mathsf{d}) = [\![\mathsf{P}]\!]^{\mathcal{S}}(\mathsf{s}, \mathsf{d}).$$

Dunque uno specializzatore non interpreta un programma ma ne restituisce uno nuovo sulla base di uno esistente (in un dato linguaggio) e di una parte del suo input. Il suo compito è simile a quello della funzione ricorsiva totale del Teorema s-m-n. Per dimostrare l'esistenza di specializzatori, dimostriamo il Teorema s-m-n nel caso dei programmi WHILE.

TEOREMA 16.19 (Teorema s-m-n). *Esiste un programma* WHILE $\mathtt{spec}$ *tale che per ogni programma* WHILE P *e per ogni porzione del suo input* $\mathsf{s} \in \mathbb{D}$, $[\![\mathtt{spec}]\!](\mathsf{P}, \mathsf{s})$ *è un programma* WHILE *e per ogni* $\mathsf{d} \in \mathbb{D}$ *vale che:*

$$[\![[\![\mathtt{spec}]\!](\mathsf{P}, \mathsf{s})]\!](\mathsf{d}) = [\![\mathsf{P}]\!](\mathsf{s}, \mathsf{d})$$

PROOF. Vediamo cosa dovrebbe fare il programma $\mathtt{spec}$. Dato P:

$$\mathbf{read}(x_1, x_2); \mathsf{C}; \mathbf{write}(y)$$

e $\mathsf{s} \in \mathbb{D}$ deve restituire il programma:

$$\mathbf{read}(x_2); x_1 := \mathsf{s}; \mathsf{C}; \mathbf{write}(y).$$

E' immediato scrivere un programma che fa ciò in WHILE (ma anche in ogni altro linguaggio di programmazione). □

Dunque esiste uno specializzatore all'interno della classe dei programmi WHILE. Cerchiamo di generalizzare la cosa a tre linguaggi generici $\mathcal{S}$, $\mathcal{T}$, e $\mathcal{L}$. Per la definizione e l'esistenza di un interprete in $\mathcal{T}$ per $\mathcal{L}$, so che:

$$[\![\mathsf{P}]\!]^{\mathcal{S}}(\mathsf{s}, \mathsf{d}) = [\![\mathtt{int}]\!]^{\mathcal{T}}(\mathsf{P}, \mathsf{s}, \mathsf{d})$$

Per il Teorema s-m-n (dimostrato in $\mathcal{T}$), vedendo P ed s come input per $\mathtt{int}$ so che esiste uno specializzatore $\mathtt{spec}$ tale che:

$$[\![\mathtt{int}]\!]^{\mathcal{T}}(\mathsf{P}, \mathsf{s}, \mathsf{d}) = [\![[\![\mathtt{int}]\!]^{\mathcal{T}}(\mathtt{int}, \mathsf{P}, \mathsf{s})]\!]^{\mathcal{T}}$$

Ora, poiché $\mathtt{int}$ è fissato a priori, lo posso anche togliere dai parametri di input. Inoltre tale programma di conversione (che nella dimostrazione sopra avevamo scritto in WHILE) lo si può sicuramente scrivere con un altro linguaggio Turing completo (quello con cui preferiamo scrivere gli algoritmi—p.es., il C o il PROLOG).

Dalla combinazione di interpreti $\mathtt{int}$ e specializzatori $\mathtt{spec}$ è possibile costruire una famiglia di programmi assai complessi e utili, quali i *compilatori* e *programmi che generano compilatori*. Per ottenere questi strumenti a supporto della programmazione, utilizziamo le *proiezioni di Futamura* [**15**]. Esse permettono di costruire semplici compilatori e programmi che generano compilatori a partire dai concetti e dall'esistenza di interpreti e specializzatori. Chiaramente, più è sofisticato lo specializzatore, es. programmi per la ottimizzazione del codice, più è possibile costruire in questo modo strumenti di compilazione sofisticati ed utili. Nei tre risultati sotto riportati assumiamo che $\mathcal{S}$, $\mathcal{T}$, $\mathcal{L}$, $\mathcal{I}$ siano linguaggi di programmazione Turing-completi operanti sul medesimo insieme di dati $\mathbb{D}$.

TEOREMA 16.20 (I proiezione di Futamura). *Dato un programma* $\mathtt{source} \in \mathcal{S}$ *possiamo generare un programma equivalente* $\mathtt{target} \in \mathcal{T}$.

PROOF. Dimostreremo che tale programma è:

$$\mathsf{target} \overset{\text{def}}{=} [\![\mathsf{spec}]\!]^{\mathcal{I}}(\mathsf{int}, \mathsf{source}) \in \mathcal{T}$$

Sia spec uno specializzatore, scritto in $\mathcal{I}$ di $\mathcal{L}$ programmi in $\mathcal{T}$ programmi. Allora:

$$\begin{aligned}
[\![\mathsf{source}]\!]^{\mathcal{S}}(\mathsf{d}) &= [\![\mathsf{int}]\!]^{\mathcal{L}}(\mathsf{source}, \mathsf{d}) && \text{Per Def. di Interprete in } \mathcal{L} \\
&= [\![[\![\mathsf{spec}]\!]^{\mathcal{I}}(\mathsf{int}, \mathsf{source})]\!]^{\mathcal{T}}(\mathsf{d}) && \text{Per Def. di Specializzatore} \\
&= [\![\mathsf{target}]\!]^{\mathcal{T}}(\mathsf{d}) && \text{Per Def. di } \mathsf{target}
\end{aligned}$$

□

Dunque, dato uno specializzatore spec, assegnandogli come input l'interprete di $\mathcal{S}$ programmi scritto in $\mathcal{L}$ e il programma source, restituisce il $\mathcal{T}$ programma desiderato.

TEOREMA 16.21 (II proiezione di Futamura). *Possiamo generare un compilatore da $\mathcal{S}$ a $\mathcal{T}$ scritto in $\mathcal{T}$.*

PROOF. Dimostreremo che: $\mathsf{comp} \overset{\text{def}}{=} [\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{spec}, \mathsf{int})$.

Sia $\mathsf{target} \in \mathcal{T}$ come stabilito dalla I proiezione. Supponiamo che $\mathcal{I} = \mathcal{L}$ (uso lo stesso linguaggio per implementare). Allora:

$$\begin{aligned}
\mathsf{target} &= [\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{int}, \mathsf{source}) \\
&= [\![[\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{spec}, \mathsf{int})]\!]^{\mathcal{T}}(\mathsf{source}) && \text{Per definizione di specializzatore} \\
&= [\![\mathsf{comp}]\!]^{\mathcal{T}}(\mathsf{source})
\end{aligned}$$

□

Dunque, per generare un compilatore, basta prendere uno specializzatore e mettere come input il suo codice e quello di un interprete.

Il terzo risultato di Futamura riguarda i generatori di compilatori:

TEOREMA 16.22 (III proiezione di Futamura). *Possiamo generare un generatore di compilatori scritto in $\mathcal{T}$.*

PROOF. Dimostreremo che il generatore di compilatori è:

$$\mathsf{compgen} \overset{\text{def}}{=} [\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{spec}, \mathsf{spec})$$

Anche in questo caso assumiamo che $\mathcal{I} = \mathcal{L}$. Dalla definizione di specializzatore e dalla II proiezione segue che:

$$\begin{aligned}
\mathsf{comp} &= [\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{spec}, \mathsf{int}) \\
&= [\![[\![\mathsf{spec}]\!]^{\mathcal{L}}(\mathsf{spec}, \mathsf{spec})]\!]^{\mathcal{T}}(\mathsf{int}) \\
&= [\![\mathsf{compgen}]\!]^{\mathcal{T}}(\mathsf{int})
\end{aligned}$$

□

Dunque $\mathtt{compgen}$ accetta come input un interprete e genera un compilatore.

ESERCIZIO 16.23. Dimostrare che $\mathtt{compgen} = [\![\mathtt{compgen}]\!](\mathtt{spec})$.

Part 3

Teoria matematica della ricorsione

CHAPTER 17

Insiemi ricorsivi e ricorsivamente enumerabili

Abbiamo visto nel Cap. 10 diverse formalizzazioni per definire ciò che noi riteniamo effettivamente calcolabile. Queste comprendono le MdT, le funzioni parziali ricorsive di Kleene & Robinson, ed i linguaggi di programmazione imperativa, tipo WHILE. In questo capitolo studieremo alcuni risultati classici della teoria della ricorsività, indipendentemente dal sistema formale adottato per esprimere algoritmi, o funzioni calcolabili. L'equivalenza tra le diverse definizioni di calcolabilità e la Tesi di Church-Turing ci permettono di studiare l'effettiva calcolabilità in un modo per così dire "più astratto". In particolare la Tesi di Church-Turing ci permette di astrarre dalla particolare MdT o programmi WHILE, e trattare in modo formale il concetto (informale) di funzione effettivamente calcolabile, indipendentemente dal formalismo adottato per rappresentarla. Nel seguito assumeremo di avere fissato una data enumerazione $\{\varphi_i\}_{i\in\mathbb{N}}$ delle MdT o delle funzioni parziali ricorsive o dei programmi WHILE. Il materiale contenuto in questa parte è tratto in larga parte dai testi di riferimento [**27, 2, 19, 15, 22**].

In questo capitolo studieremo le proprietà ricorsive di insiemi di numeri naturali. La scelta di studiare insiemi di numeri naturali non è affatto limitativa per gli scopi dell'informatica. Tutti i dati manipolabili da un algoritmo sono infatti numerabili e possono quindi essere messi in corrispondenza biunivoca con l'insieme $\mathbb{N}$. Un analogo ragionamento si può applicare ad insiemi di stringhe su di un alfabeto, ovvero ad un linguaggio, o ad insiemi di alberi sintattici, rappresentanti WHILE-programs. È infatti noto l'isomorfismo tra $\mathbb{N}$ e rispettivamente Σ^* e $\mathbb{D}_A$, essendo Σ un dato alfabeto e A un dato insieme di *atomi*. Da questo consegue che studiare proprietà degli elementi di $\wp(\mathbb{N})$ equivale a studiare le proprietà degli elementi di $\wp(\Sigma^*)$ e $\wp(\mathbb{D}_A)$, e che ogni definizione/proprietà che vedremo per insiemi di naturali, si applica senza restrizione alcuna a linguaggi (ovvero elementi di $\wp(\Sigma^*)$) e a insiemi di alberi sintattici. In questo senso, il passaggio da funzioni ad insiemi è naturale e rappresenta una astrazione successiva verso una più astratta definizione di problema ricorsivamente risolvibile. Questo ci permetterà di studiare, analogamente a quanto visto per le funzioni calcolabili, insiemi che risultano essere effettivamente costruibili, ovvero i cui elementi possono essere determinati in modo algoritmico.

Consideriamo un insieme $I \subseteq \mathbb{N}$. Ci chiediamo se I è costruibile in modo algoritmico, ovvero se è possibile generare gli elementi di I mediante una funzione che sia effettivamente calcolabile.

DEFINIZIONE 17.1. Un insieme $I \subseteq \mathbb{N}$ è detto *ricorsivamente enumerabile (r.e.)* se esiste una funzione parziale ricorsiva ψ tale che $I = \mathsf{dom}(\psi)$.

Nel seguito indicheremo con $\mathsf{RE} \subseteq \wp(\mathbb{N})$ la classe degli insiemi ricorsivamente enumerabili. Pertanto $A \in \mathsf{RE}$ sse A è r.e. Inoltre, nel seguito indicheremo con W_x l'insieme r.e. associato alla funzione φ_x secondo una opportuna Gödelizzazione, ovvero $W_x = \mathsf{dom}(\varphi_x)$ essendo φ_x l'x-esima MdT nella numerazione fissata.

Sia $A \in \mathsf{RE}$. La funzione parziale:

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

è detta *funzione semicaratteristica* dell'insieme A. È chiaro che $A = \mathsf{dom}(\psi_A)$. Inoltre, essendo per ipotesi A r.e. , la verifica della condizione $x \in A$ è semidecidibile, ovvero se y è tale che $A = W_y$, $x \in A$ sse $\varphi_y(x) \downarrow$. Sia dunque $P_y \stackrel{\text{def}}{=} \textbf{read } x; C; \textbf{write } z$ un programma WHILE equivalente a φ_y. È naturale definire il seguente programma: $\textbf{read}(x); C; z := 1; \textbf{write}(z)$. Questo programma corrisponde ad una MdT, ed implementa la funzione ψ_A. Pertanto abbiamo dimostrato che un insieme è detto r.e. sse ha una funzione semicaratteristica parziale ricorsiva.

All'interno della famiglia degli insiemi r.e. , esistono insiemi con caratteristiche più forti. Per questi insiemi vale la condizione aggiuntiva che la loro funzione caratteristica (che è sempre una funzione totale) è ricorsiva, ovvero esiste un algoritmo in grado di *decidere* l'appartenenza o meno di un elemento all'insieme.

DEFINIZIONE 17.2. Un insieme A è detto *ricorsivo* se esiste una funzione ricorsiva (totale) f_A tale che:

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

ESEMPIO 17.3. I seguenti sono insiemi ricorsivi:

- L'insieme $\{0, 2, 4, 6, \ldots\}$ dei numeri pari;
- $\mathbb{N}$ e $\varnothing$;
- Ogni insieme finito (Dimostrare per esercizio);
- Ogni insieme A tale che $\bar{A}$ è finito (Dimostrare per esercizio).

Esisteranno dunque $\aleph_0$ insiemi r.e. di cui una parte di uguale cardinalità sarà ricorsiva. Si nota infatti subito della definizioni che

$$\boxed{A \text{ ricorsivo} \Rightarrow A \in \mathsf{RE}}$$

Dato A insieme ricorsivo, basta infatti definire la funzione parziale e chiaramente ricorsiva:

$$\varphi(x) = \begin{cases} 1 & \text{se } f_A(x) = 1 \\ \uparrow & \text{altrimenti} \end{cases}$$

Esercizio 17.4. Dimostrare che i seguenti insiemi sono ricorsivi:

- $\{x \in \mathbb{N} : \exists n \exists p \text{ primo } . \, x = p^n\}$;
- $\{\langle x, y\rangle \in \mathbb{N} \times \mathbb{N} : x^4 + y^6 = 1238\}$;
- $\{x \in \mathbb{N} : \exists n. \, x = 2n + 1\}$.

Teorema 17.5 (Teorema di Post). *Un insieme* A *è ricorsivo se solo se* A *e* $\bar{A}$ *sono r.e.*

Proof. Sia $A \subseteq \mathbb{N}$ e $\bar{A} = \mathbb{N} \setminus A$.

($\Rightarrow$) Sia A ricorsivo e f_A la sua funzione caratteristica (totale) ricorsiva. Definiamo:

$$\psi(x) = \begin{cases} 1 & \text{se } f_A(x) = 1 \\ \uparrow & \text{altrimenti} \end{cases}$$

È chiaramente una funzione parziale ricorsiva. Ad esempio il seguente frammento di programma implementa ψ: Sia $F_A \stackrel{\text{def}}{=} \textbf{read } x; C_A; \textbf{write } y$.

$$\begin{aligned} G: \quad & \textbf{read } x; \\ & C_A; \\ & \textbf{while } y = \mathsf{nil} \textbf{ do } y := y \textbf{ endw}; \\ & \textbf{write } y \end{aligned}$$

A ha quindi una funzione semicaratteristica parziale ricorsiva. Analogo discorso si applica a $\bar{A}$, essendo $f_{\bar{A}} = 1 - f_A$ la corrispondente funzione caratteristica. Abbiamo pertanto dimostrato che A e $\bar{A}$ sono r.e.

($\leftarrow$) Supponiamo A e $\bar{A}$ r.e. , con funzioni semicaratteristiche ψ_A e $\psi_{\bar{A}}$. Definiamo:

$$f(x) = \begin{cases} 1 & \text{se } \psi_A(x) \downarrow \\ 0 & \text{se } \psi_{\bar{A}}(x) \downarrow \end{cases}$$

È banale osservare che, dato $x \in \mathbb{N}$: o $x \in A$ o $x \in \bar{A}$, allora eseguendo a turno un'istruzione della MdT che calcola ψ_A e un'istruzione della MdT che calcola $\psi_{\bar{A}}$, prima o poi una delle due MdT termina, rendendo la funzione f totale ricorsiva. $\square$

Si osservi come la ricorsività di $\mathbb{N}$ segua anche dal fatto che sia $\bar{\mathbb{N}} = \varnothing$ che $\mathbb{N}$ sono r.e.

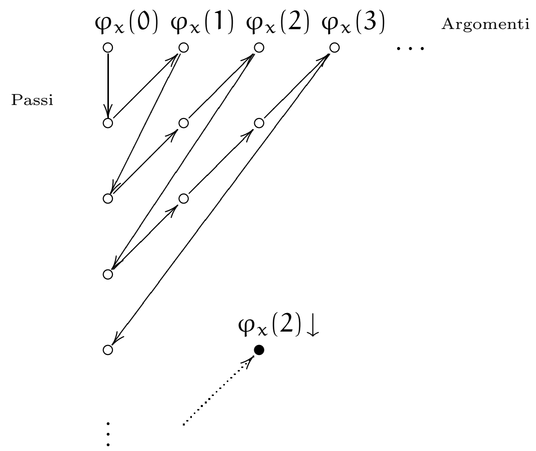


FIGURE 1. Dovetail: $\varphi_x(2)$ converge dopo 2 passi

Il seguente teorema caratterizza gli insiemi r.e. come la famiglia degli insiemi che possono essere effettivamente enumerati da un algoritmo, ovvero i cui elementi possono essere "listati" in modo effettivo come "output" di un programma. Gli insiemi r.e. possono dunque essere visti come liste, eventualmente infinite, di elementi generabili da un algoritmo.

Il teorema di caratterizzazione fa uso di una metodologia effettiva per esplorare il campo di convergenza di una funzione parziale, ovvero l'insieme degli elementi su cui una funzione parziale converge (è definita). Questa metodologia per esplorare il campo di convergenza di una funzione parziale è detta a *coda di colomba* (dovetail) e sarà spesso utilizzata nel seguito. In essa ha un ruolo essenziale il concetto di passo di calcolo o derivazione. Come abbiamo visto nel Cap. 10 ogni sistema formale introdotto definisce il concetto di calcolo come sequenza discreta di passi elementari. Pertanto, il procedimento a dovetail è perfettamente accettabile come metodo algoritmico per esplorare il campo di definizione (o convergenza) di una funzione parziale ricorsiva.

Sia dunque φ_x una funzione parziale ricorsiva. Applicare la tecnica dovetail a φ_x significa costruire l'insieme L in modo induttivo, tale che:

Passo 0: Si faccia un passo di calcolo per $\varphi_x(0)$, e se termina allora $L := \{\varphi_x(0)\}$;

Passo $n + 1$: Si faccia un passo nel calcolo di tutte le funzioni $\varphi_x(0), \dots, \varphi_x(n)$ ove il calcolo non sia già terminato al passo precedente, e per ogni m tale che $\varphi_x(m)$ converge, si pone $L := L \cup \{\varphi_x(m)\}$.

In pratica si esplora il grafico della funzione seguendo un andamento a zig-zag, come evidenziato nella figura 1.

L'insieme L è stato dunque effettivamente costruito ed i suoi valori corrispondono al codominio definito di φ_x. Siamo ora in grado di dimostrare il seguente teorema di Kleene di caratterizzazione degli insiemi r.e.

TEOREMA 17.6 (Caratterizzazione degli insiemi r.e.). *Le seguenti affermazioni sono equivalenti:*

(1) $A \in \mathsf{RE}$*;*
(2) *esiste* $\varphi \in \mathcal{PR}$*:* $A = \mathsf{range}(\varphi)$*;*
(3) $A = \varnothing$ *oppure esiste una funzione* $f \in \mathsf{R}$ *tale che* $A = \mathsf{range}(f)$*.*

PROOF. Consideriamo una data enumerazione delle MdT (o delle funzioni parziali ricorsive) $\{\varphi_x\}_{x\in\mathbb{N}}$.

($1 \Rightarrow 2$) Sia $A \in \mathsf{RE}$. Per definizione, esisterà φ_x parziale ricorsiva t.c. $A = \mathsf{dom}(\varphi_x)$. Definiamo:

$$\delta(y) = \begin{cases} y & \text{se } \varphi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

δ è parziale ricorsiva e $\mathsf{range}(\delta) = \mathsf{dom}(\varphi_x) = A$.

($2 \Rightarrow 3$) Sia $A = \mathsf{range}(\varphi_x)$ e $A \neq \varnothing$. Applicando il dovetail alla funzione:

$$\psi(y) = \begin{cases} \varphi_x(y) & \text{se } \varphi_x(y)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

poiché $A = \mathsf{range}(\varphi_x) \neq \varnothing$, esiste $n_0 \in \mathbb{N}$ tale che all'n_0-esimo stadio del dovetail si osserva una terminazione. Definiamo la funzione f induttivamente sullo stadio n-esimo del dovetail, in modo tale che f abbia valori tutti e soli nella lista L generata dal dovetail di ψ.

- Se ψ converge in $\{m_0, \dots, m_{j_0}\}$ con $0 \le j_0 \le n_0$ allora per ogni $i \in [0, j_0]$: $f(i) = \psi(m_i)$;
- Supponiamo che per definire $f(j_p)$ si sia usato lo stadio n_p-esimo del dovetail. Possiamo avere i seguenti casi:
  - Se nell'$n_p + 1$-esimo passo del dovetail non si rivelano nuove terminazioni, allora $f(j_p + 1) = f(j_p)$.
  - Se nell'n_p+1-esimo passo del dovetail ψ converge con input $\{p_0, \dots, p_k\}$ con $0 \le k \le n_p + 1$, allora per ogni $i \in [0, k]$: $f(j_p + 1 + i) = \psi(p_i)$.
- Iteriamo il precedente punto ponendo $j_{p+1} = j_p + 1 + k$.

La procedura che abbiamo descritto è effettiva e definisce una funzione ricorsiva <u>totale</u> f tale che $A = \mathsf{range}(f)$.

($3 \Rightarrow 1$). Se $A = \varnothing$ allora $A = \mathsf{dom}(\lambda x. \uparrow)$. Sappiamo che $\lambda x. \uparrow \in \mathcal{PR}$, e quindi A è r.e. Se $A = \mathsf{range}(f)$ con $f \in \mathcal{R}$, basta porre:

$$\varphi(x) = \mu z.(f(z) - x = 0)$$

$\varphi \in \mathcal{PR}$ ed inoltre è immediato osservare che $\varphi(x)\downarrow \Leftrightarrow \exists z.\ f(z) = x$, quindi $A = \mathsf{dom}(\varphi)$ che dimostra che A è r.e. $\square$

NOTA 17.7. Nel testo di Rogers [**27**] la definizione di insieme r.e. viene data con la caratterizzazione (3) del Teorema appena visto. Tale caratterizzazione spiega anche il nome "ricorsivamente enumerabile" ovvero enumerabile mediante l'impiego di una funzione ricorsiva. Nei testi più recenti (p. es., [**15, 6**]) si adotta invece la definizione 17.1. Il Teorema sancisce che le due sono del tutto equivalenti.

ESEMPIO 17.8. $\mathbb{N}$ è r.e. , poiché $\lambda x.\ x \in \mathcal{R}$ e $\mathbb{N} = \mathrm{range}(\lambda x.\ x)$.

COROLLARIO 17.9. Esiste una funzione totale ricorsiva f tale che $W_x = \mathrm{range}(\varphi_{f(x)})$.

È chiaro che non è detto che $\varphi_{f(x)}$ sia totale, pur essendolo f.

TEOREMA 17.10. *Le seguenti affermazioni sono equivalenti:*

(1) A *è ricorsivo;*
(2) $A = \varnothing$ *oppure* $A = \mathrm{range}(f)$ *con* $f \in \mathcal{R}$ *non decrescente.*

PROOF. ($\Rightarrow$). Sia $A \neq \varnothing$ e A ricorsivo. Poiché $A \subseteq \mathbb{N}$, esiste $a = \min(A)$ minimo valore tra quelli contenuti in A. Definiamo:

$$\begin{array}{rcl} f(0) & = & a \\ f(n+1) & = & \begin{cases} n+1 & \text{se } n+1 \in A \\ f(n) & \text{altrimenti} \end{cases} \end{array}$$

È chiaro che f è totale ricorsiva non decrescente e che $A = \mathrm{range}(f)$.

($\Leftarrow$). Se A è finito, allora A è banalmente ricorsivo. Supponiamo che A sia infinito e $A = \mathrm{range}(f)$ con f totale ricorsiva non decrescente. Sia $x \in A$ e $z_x = \mu y.\ (f(y) > x)$. È chiaro che un tale z_x esiste (perchè?) e che

$$x \in A \;\Leftrightarrow\; x \in \{f(0), \ldots, f(z)\}$$

È quindi decidibile l'appartenenza di x ad A, ovvero A è ricorsivo (definire la funzione caratteristica di A). □

TEOREMA 17.11. *Ogni insieme r.e. infinito ha un sottoinsieme ricorsivo infinito.*

PROOF. Sia $A \subseteq \mathbb{N}$ tale che $A = \mathrm{range}(f)$ con f funzione ricorsiva totale e $|A| = \omega$. Definiamo la funzione g tale che:

$$\begin{array}{rcl} g(0) & = & f(0) \\ g(n+1) & = & f(\mu y.\ f(y) > g(n)). \end{array}$$

$g(n+1)$ è dunque il più piccolo elemento di A più grande di $g(n)$. g è crescente, quindi $\mathrm{range}(g)$ è ricorsivo per il Teorema 17.10. Inoltre $\mathrm{range}(g) \subseteq A$, poiché per definizione di g: $\mathrm{range}(g) \subseteq \mathrm{range}(f) = A$. □

ESERCIZIO 17.12. Si dica se i seguenti insiemi sono ricorsivi o r.e. :

(1) $A = \mathit{range}(f)$ con

$$\begin{aligned} f(0) &= n_0 \\ f(n+1) &= 2f(n). \end{aligned}$$

(2) $B = \mathit{range}(g)$ con

$$\begin{aligned} g(0) &= n_0 \\ g(n+1) &= h(n)g(n). \end{aligned}$$

essendo h una generica funzione ricorsiva (totale).

(3) Data una funzione ricorsiva f, dimostrare che l'immagine inversa di f e W_x, con $x \in \mathbb{N}$, ovvero l'insieme $f^{-1}(W_x) = \{y : f(y) \in W_x\}$, è r.e.

(4) Sia $D = \mathit{range}(\psi)$ con:

$$\begin{aligned} \psi(0) &= n_0 \\ \psi(n+1) &= \varphi_n(n)\psi(n). \end{aligned}$$

D è ricorsivo?

Fino a questo momento abbiamo visto insiemi ricorsivi e r.e. Tuttavia non abbiamo ancora visto un esempio di un insieme che sia r.e. ma non ricorsivo. Questo insieme (se esiste) rappresenta l'insieme che discrimina le due classi di insiemi che abbiamo definito. È chiaro che un insieme di questo tipo deve essere r.e. ma non ricorsivo, ovvero l'appartenenza ad esso deve essere un predicato semidecidibile. È naturale chiedersi se questo insieme possa essere in qualche modo correlato alla non decidibilità della terminazione. Al solito, fissiamo una numerazione delle MdT o delle funzioni parziali ricorsive $\{\varphi_x\}_{x\in\mathbb{N}}$.

Definizione 17.13. Definiamo $K = \{x : \varphi_x(x)\downarrow\}$.

È chiaro per la definizione sopra che $K = \{x : x \in W_x\}$[1].

Teorema 17.14. K *è r.e. ma non ricorsivo.*

[1]Questo insieme ricorda l'insieme utilizzato per contraddire la teoria classica degli insiemi, ovvero il *Paradosso di Russell* [**28**]. Se infatti fosse possibile definire un insieme del tipo: $M = \{x : x \notin x\}$ allora otterremmo immediatamente un paradosso poiché $M \in M$ sse $M \notin M$. L'idea utilizzata per risolvere questo paradosso nella teoria degli insiemi è quella di non ammettere M come insieme, ovvero di definire una "disciplina" per la costruzione di insiemi che vieti una costruzione tipo M. Si osservi come la teoria della ricorsione è immune da questo tipo di paradossi. Infatti $\bar{K} = \{x : x \notin W_x\}$, l'insieme dei numeri non appartenenti agli insiemi r.e. da essi generati, non può essere r.e. , altrimenti, essendolo K per il Teorema 17.14, K sarebbe ricorsivo (Teorema di Post) il che per il Teorema 17.14 è assurdo. Quindi gli insiemi r.e. non includono $\bar{K}$, ovvero non esiste y_0 tale che $\bar{K} = W_{y_0}$

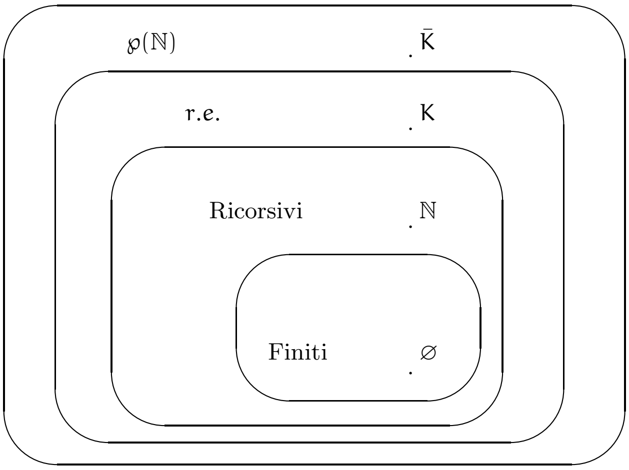


FIGURE 2. La gerarchia degli insiemi Ricorsivi

PROOF. definiamo la funzione ψ_K tale che:

$$\psi_K = \begin{cases} 1 & \text{se } \varphi_x(x)\downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

ψ_K è parziale ricorsiva (perchè?) e semicaratteristica di K, quindi K è r.e.

Supponiamo K ricorsivo. Allora $\bar{K}$ è r.e. Sia y_0 tale che $\bar{K} = W_{y_0}$. Segue che:

$$y_0 \in W_{y_0} = \bar{K} \;\Leftrightarrow\; y_0 \in K$$

che è chiaramente assurdo. □

ESERCIZIO 17.15.

(1) Dimostrare che $K_2 = \left\{\langle x, y\rangle \,:\, x \in W_y\right\}$ è r.e. ma non ricorsivo;
(2) Dimostrare che le seguenti proprietà caratterizzano gli insiemi finiti;
  - tutti i sottoinsiemi sono r.e.
  - tutti i sottoinsiemi sono ricorsivi
(3) Dimostrare che esistono funzioni ricorsive (totali) f e g tali che: $W_x \cap W_y = W_{f(x,y)}$ e $W_x \cup W_y = W_{g(x,y)}$;
(4) Dimostrare che A è ricorsivo infinito sse esiste una funzione f ricorsiva crescente (i.e. $\forall n.\ f(n) < f(n+1)$) tale che $A = range(f)$.

Abbiamo quindi la seguente rappresentazione in "classi" di risolvibilità degli insiemi di numeri naturali. La classe degli insiemi ricorsivi corrisponde alla classe degli insiemi *decidibili*, quella degli insiemi r.e. corrisponde alla classe degli insiemi semidecidibili, mentre gli insiemi non r.e. (che sono la maggior parte di $\wp(\mathbb{N})$) sono gli insiemi per cui non è possibile dare alcun metodo effettivo per verificare sia l'appartenenza che la non appartenenza di un elemento a questi insiemi. Valgono quindi le seguenti inclusioni, mostrate anche in Figura 2, tra le varie classi di insiemi viste fino ad ora:

$$\boxed{\mathit{ricorsivi} \subset \mathit{r.e.} \subset \mathit{non} - \mathit{r.e.}}$$

CHAPTER 18

I Teoremi di Ricorsione

I teoremi di ricorsione, detti anche di punto fisso, sono tra i risultati più importanti che si incontrano nello studio della teoria della ricorsività. In questo capitolo studieremo i teoremi di ricorsione e le loro conseguenze nello studio di proprietà di programmi.

1. Il primo Teorema di ricorsione

TEOREMA 18.1 (Teorema di ricorsione di Kleene). *Sia* $t \in \mathcal{R}$ *una funzione ricorsiva (totale). Allora esiste* $n \in \mathbb{N}$ *tale che* $\varphi_n = \varphi_{t(n)}$.

PROOF. Sia u l'indice di una generica MdT φ_u. Definiamo la seguente funzione parziale:

$$\psi(u, x) = \begin{cases} \varphi_{\varphi_u(u)}(x) & \text{se } \varphi_u(u) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

È chiaro che la funzione così definita è parziale ricorsiva (si descriva a parole una MdT che "calcola" ψ). Quindi, per il Teorema s-m-n, esiste una funzione ricorsiva (totale) g tale che: $\psi(u, x) = \varphi_{g(u)}(x)$. Sia ora t una funzione ricorsiva (totale). Poiché la composizione di funzioni ricorsive è ricorsiva, $t \circ g$ è ricorsiva. Quindi $t \circ g = \varphi_v$ per qualche v. Pertanto $t \circ g(v) = \varphi_v(v)$. Vista la genericità di u, sostituendo u con v nella definizione di $\psi(u, x)$ otteniamo:

$$\psi(v, x) = \varphi_{g(v)}(x) = \begin{cases} \varphi_{t(g(v))}(x) & \text{se } \varphi_v(v) = t(g(v)) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Poiché $t \circ g$ è totale, $\varphi_{g(v)}(x) = \varphi_{t(g(v))}(x)$. Posto quindi $n = g(v)$, abbiamo che $\varphi_n = \varphi_{t(n)}$. □

Il significato intuitivo del teorema di ricorsione di Kleene sarà più chiaro in seguito, in particolare nella Sezione 4. Per ora ci basti utilizzarlo come potente metodo per dimostrare proprietà di ricorsività sugli elementi di $\wp(\mathbb{N})$. Vediamo qualche esempio:

ESEMPIO 18.2. Esiste un $n \in \mathbb{N}$ t.c. $\mathtt{range}(\varphi_n) = \{n\}$? Utilizziamo il Teorema di Ricorsione per stabilire ciò. Sia $g(x, y) = x$. g è calcolabile e totale. Per il

Teorema s-m-n avremo che esiste h ricorsiva tale che per ogni y:

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

Per definizione, $range(\varphi_{h(x)}) = \{x\}$. Per il Teorema di ricorsione, si ha che esiste $n \in \mathbb{N}$ t.c. $\varphi_{h(n)}(y) = \varphi_n(y)$. Pertanto $range(\varphi_n) = range(\varphi_{h(n)}) = \{n\}$.

ESEMPIO 18.3. Esiste un $n \in \mathbb{N}$ t.c. $W_n = \{n\}$? Sia $g(x, y)$ definita come:

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

g è calcolabile. Per il Teorema s-m-n avremo che esiste h ricorsiva tale che per ogni y:

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

Per definizione, $W_{h(x)} = \{x\}$. Per il Teorema di ricorsione, si ha che esiste $n \in \mathbb{N}$ t.c. $\varphi_{h(n)}(y) = \varphi_n(y)$. Pertanto $W_n = W_{h(n)} = \{n\}$.

ESERCIZIO 18.4. Sia, per $i > 0$, p_i l'i-esimo numero primo ($p_1 = 2, p_2 = 3, \dots$). Si dimostri che esiste un indice u tale che $W_u = \{p_1, p_2, \dots, p_{(u!)}\}$ e $E_u = \{1, 2, \dots, u\}$.

ESERCIZIO 18.5.
Si mostri che esiste $v \in \mathbb{N}$ tale che:

$$\varphi_v(y) \begin{cases} 2^y & y \leq v \\ 5 & y > v \end{cases}$$

2. Il secondo Teorema di ricorsione

In questo paragrafo enunceremo il secondo teorema di ricorsione che estende il risultato del primo. Utilizzeremo questo teorema per la dimostrazione del teorema di Myhill nel Capitolo 19.

TEOREMA 18.6. *Sia* $f : \mathbb{N}^2 \longrightarrow \mathbb{N} \in \mathcal{R}$ *una funzione ricorsiva (totale). Allora esiste una funzione* $v : \mathbb{N} \longrightarrow \mathbb{N} \in \mathcal{R}$ *ricorsiva totale tale che:*

$$\forall y \in \mathbb{N} : \varphi_{f(v(y),y)} = \varphi_{v(y)}.$$

PROOF. Definiamo la seguente funzione parziale:

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

Per il Teorema s-m-n applicato al primo argomento, esiste una funzione calcolabile totale s tale che $\psi(x, y, z) = \varphi_{s(x,y)}(z)$. Si noti che, se $\varphi_x(x) \downarrow$, allora

$$\varphi_{s(x,y)} = \varphi_{f(\varphi_x(x),y)} \tag{2.1}$$

Per il Teorema s-m-n applicato al secondo argomento di s sappiamo che esiste una funzione calcolabile totale t tale che:

$$s(x, y) = \varphi_{t(y)}(x) \tag{2.2}$$

Definiamo la funzione

$$v(y) \stackrel{\text{def}}{=} \varphi_{t(y)}(t(y))$$

Poiché sia t che $\varphi_{t(y)}$ sono ricorsive e totali, lo sarà anche v (dunque in particolare $\varphi_v(v)\downarrow$). Dunque:

$$\begin{aligned} \varphi_{v(y)} &= \varphi_{\varphi_{t(y)}(t(y))} && \text{Def di } v \\ &= \varphi_{s(t(y),y)} && \text{(2.2), con } x = t(y) \\ &= \varphi_{f(\varphi_{t(y)}(t(y)),y)} && \text{Poiché } \varphi_v(v)\downarrow \text{ e (2.1)} \\ &= \varphi_{f(v(y),y)} && \text{Def di } v \end{aligned}$$

□

3. Il Teorema di Rice

Uno dei risultati più importanti dimostrabili mediante il teorema di ricorsione è il Teorema di Rice. Esso afferma che ogni proprietà non banale che rappresenti ciò che viene calcolato dalle MdT (o dai programmi, vedi Sezione 4) non è ricorsiva. Definiamo innanzitutto cosa si intende per *proprietà* sulle MdT. Sia al solito $\{\varphi_i\}_{i\in\mathbb{N}}$ una enumerazione delle MdT.

DEFINIZIONE 18.7. Una *proprietà* sulle MdT è un qualsiasi sottoinsieme di $\{\varphi_i\}_{i\in\mathbb{N}}$ (con abuso di notazion, qualsiasi sottoinsieme di $\mathbb{N}$).

Una proprietà sulle MdT è pertanto univocamente identificata dall'insieme degli indici delle MdT in essa contenuti. Nel seguito quindi non faremo distinzione tra proprietà intesa come insieme di indici (e quindi di numeri) di MdT o proprietà intesa come insieme di MdT. Nel primo caso, ovviamente potremo parlare di proprietà ricorsive (o decidibili), r.e. (o semidecidibili), e proprietà non r.e.

DEFINIZIONE 18.8. Sia dunque Π una proprietà sulle MdT, $\Pi \subseteq \mathbb{N}$. Π è *estensionale* (o chiusa per eguaglianza estensionale) se per ogni $x, y \in \mathbb{N}$:

$$x \in \Pi \wedge \varphi_x = \varphi_y \ \Rightarrow y \in \Pi$$

Intuitivamente, una proprietà è estensionale quando questa "parla" di cosa calcolano le MdT in essa contenute, e non di come queste macchine sono "fatte". Il seguente risultato sancisce l'esistenza di un numero arbitrario di MdT tra loro equivalenti.

LEMMA 18.9 (Tecnica del Padding). $\forall i, k.\ \exists j : (\varphi_j = \varphi_i \wedge j > k)$.

PROOF. Siano dati i e k e sia φ_i l'i-esima MdT. Se $i > k$ allora ponendo $j = i$ si ha la tesi. Se $i \leq k$ allora è sufficiente aggiungere a φ_i un numero congruo di quintuple (con stati mai raggiungibili dal calcolo) in modo tale che si ottenga φ_j con $j > k$. $\square$

ESERCIZIO 18.10. Si dimostri una proprietà analoga per i programmi WHILE.

Mediante la tecnica del padding si osserva che esistono infinite MdT equivalenti ad una MdT data. Pertanto, una proprietà estensionale sulle MdT non può che essere un insieme infinito.

ESERCIZIO 18.11. Si dimostri che le seguenti proprietà sono estensionali:

- $\varnothing$ e $\mathbb{N}$. In particolare, tali proprietà sono dette *banali*;
- Dato $F \subseteq \mathcal{PR}$, $\{i : \varphi_i = \psi \ \wedge \psi \in F\}$;
- $\{i : \varphi_i = \lambda x.\, x\}$;
- $\{i : \varphi_i \text{ è definitivamente } \geq 3\}$.

Al contrario, la proprietà $\{157\}$ che include solo la MdT di indice 157 non è estensionale. Infatti esistono infinite MdT equivalenti ad una data MdT (Lemma 18.9), e pertanto l'insieme $\{157\}$ non può essere estensionale.

TEOREMA 18.12 (Teorema di Rice). *Sia Π una proprietà estensionale. Π è ricorsiva sse è banale (ovvero $\Pi = \varnothing$ oppure $\Pi = \mathbb{N}$).*

PROOF. La direzione $(\leftarrow)$ segue per definizione.
$(\rightarrow)$ Sia Π ricorsivo e g la sua funzione caratteristica:

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

Supponiamo per assurdo che Π non sia banale. Allora esistono $a, b \in \mathbb{N}$ tali che $a \in \Pi$ e $b \notin \Pi$. Definiamo la funzione

$$h(x) = \begin{cases} b & \text{se } g(x) = 1 \\ a & \text{se } g(x) = 0 \end{cases}$$

Essendo g ricorsiva per ipotesi, chiaramente h è ricorsiva (e totale). Per il Teorema di ricorsione di Kleene, esiste n_0 tale che $\varphi_{n_0} = \varphi_{h(n_0)}$. Da questo fatto deriviamo l'assurdo:

- Se $n_0 \in \Pi$ allora essendo Π estensionale, $h(n_0) \in \Pi$, ma questo è assurdo poiché $h(n_0) = b$ e $b \notin \Pi$.
- Se $n_0 \notin \Pi$ allora essendo Π estensionale, $h(n_0) \notin \Pi$, ma questo è assurdo poiché $h(n_0) = a$ e $a \in \Pi$.

Quindi Π non può essere ricorsivo e non banale. $\square$

Il Teorema di Rice dunque afferma che ogni funzione parziale ricorsiva o MdT ammette un insieme infinito non ricorsivo di indici di funzioni o macchine equivalenti. In particolare osserviamo che i seguenti problemi, di chiaro interesse informatico, non sono decidibili in seguito al Teorema di Rice.

ESEMPIO 18.13. Data $\psi \in \mathcal{PR}$, ed una MdT M, non è in generale decidibile se M "calcola" ψ. Infatti se j è l'indice della MdT M, allora questo problema equivale a decidere se $j \in \{i : \varphi_i = \psi\}$ che è una proprietà estensionale non banale ($\{i : \varphi_i = \psi\} \neq \varnothing$ poiché esiste una MdT che calcola la funzione parziale ricorsiva ψ essendo $\psi \in \mathcal{PR}$ e $\{i : \varphi_i = \psi\} \neq \mathbb{N}$ poiché non tutte le MdT calcolano la stessa funzione) delle MdT, e quindi per il Teorema di Rice è una proprietà non ricorsiva.

ESEMPIO 18.14. Date M_1 e M_2, non è decidibile se esse sono equivalenti (ovvero calcolano la stessa funzione). Dimostrare questo per esercizio applicando il teorema di Rice.

È necessario fare molta attenzione ad utilizzare il Teorema di Rice. Esso infatti ha come ipotesi il fatto che l'insieme di cui si vuole studiare la ricorsività sia *estensionale*. Alcuni insiemi possono trarre in inganno, come nel seguente esempio.

ESEMPIO 18.15. Il seguente esempio, ideato da G. Longo [**19**], non può essere risolto invocando il Teorema di Rice. Definiamo una nozione *debole* di equivalenza tra MdT (o programmi) come segue:

$$\varphi_i \sim \varphi_j \iff (\forall x. (\varphi_i(x) \downarrow \wedge \varphi_j(x) \downarrow \Rightarrow \varphi_i(x) = \varphi_j(x)))$$

Ci chiediamo se l'insieme $D = \mathtt{range}(\psi)$ con $\psi(i) = \mu j. (\varphi_i \sim \varphi_j)$ è ricorsivo.

Si osservi che la funzione sempre indefinita $\lambda x. \uparrow$ è debolmente equivalente ad ogni altra funzione parziale ricorsiva, ovvero, per ogni $i \in \mathbb{N}$: $\varphi_i \sim \varphi_u$ con u indice della più piccola MdT nella numerazione delle MdT $\{\varphi_n\}_{n\in\mathbb{N}}$, tale che $\varphi_u = \lambda x. \uparrow$. Quindi, $D \subseteq \{0, \dots, u\}$ è un insieme finito e quindi ricorsivo.

Tuttavia interpretando erroneamente la proprietà come estensionale, l'utilizzo del Teorema di Rice porterebbe ad affermare che D non è ricorsivo. L'insieme D non caratterizza una proprietà estensionale: infatti, $= \Rightarrow \sim$, se $i \in D$ e $\varphi_k = \varphi_i$ con $k > i$ (k è facilmente ottenibile con la tecnica del Padding), allora $k \notin D$ (dimostrare per esercizio).

ESERCIZIO 18.16.

(1) Dire (giustificando formalmente) per quali $A \subseteq \mathbb{N}$ il seguente insieme $\{i : W_i = A\}$ è ricorsivo.
(2) Dire (giustificando formalmente) se $\{i : \forall x. \varphi_i(x) \uparrow\}$ è ricorsivo, r.e. o non r.e.
(3) Dimostrare che $\{i : \varphi_i \text{ è definitivamente } \geq 3\}$ è non ricorsivo.
(4) Dimostrare che $\{i : \exists x. \varphi_i(x) = x\}$ è non ricorsivo.

Il Teorema di Rice è assai disarmante di fronte a proprietà estensionali che riguardano MdT (quindi programmi). Di fatto, ogni proprietà non banale che tratti ciò che calcolano le MdT è non ricorsiva (non decidibile). Tuttavia, è possibile rappresentare alcune proprietà estensionali non ricorsive (ma r.e.) in modo decidibile, ricorrendo a proprietà su MdT non estensionali decidibili.

Definizione 18.17. Sia Π una proprietà. $A \subseteq \mathbb{N}$ è una *rappresentazione* di Π se per ogni $\varphi_i \in \Pi$ esiste $j \in A$ tale che $\varphi_i = \varphi_j$ e, viceversa, per ogni $j \in A$ esiste $\varphi_i \in \Pi$ tale che $\varphi_i = \varphi_j$.

Ogni insieme $A \subseteq \mathbb{N}$ può essere esteso in modo estensionale ad una proprietà $\Pi(A)$ tale che $A \subseteq \Pi(A)$ e A rappresenta $\Pi(A)$. Basta definire $\Pi(A) = \big\{i : \exists j \in A.\ \varphi_i = \varphi_j\big\}$.

Teorema 18.18. *Sia* A *un insieme r.e. . Allora esiste una rappresentazione ricorsiva* B *della proprietà* $\Pi(A)$.

Proof. Se $A = \varnothing$ allora basta porre $B = A = \varnothing$. B è chiaramente una rappresentazione ricorsiva di $\Pi(A) = \varnothing$. Supponiamo che $A \neq \varnothing$. Per il Teorema 17.6, esiste una funzione ricorsiva (totale) f tale che $A = \mathrm{range}(f)$. Definiamo g tale che:

$$\begin{aligned} g(0) &= f(0) \\ g(n+1) &= j_n \qquad \text{dove } \varphi_{j_n} = \varphi_{f(n+1)} \text{ e } j_n > g(n) \end{aligned}$$

Mostriamo che g è ben definita: per il Lemma del Padding j_n così costruito esiste sempre, basta porre $i = f(n+1)$ e $k = g(n)$ nel Lemma del Padding. g è inoltre crescente e ricorsiva. Quindi per il Teorema 17.10 $B = \mathrm{range}(g)$ è ricorsivo. Inoltre, è chiaro che B è una rappresentazione per $\Pi(A)$ (segue banalmente per costruzione, verificare per esercizio). $\square$

Esercizio 18.19. Si mostri che esiste una rappresentazione ricorsiva della proprietà $\Pi(A)$ per i seguenti insiemi:

(1) $A = \{i : 2 \in W_i\}$
(2) $A = \{i : \varphi_i(2) = \varphi_{2 \cdot i}(4)\}$
(3) $A = \{i : \exists j \leq i. j \in W_i\}$
(4) $A = \{i : \forall j \in \{1900, \ldots, 2100\} : j \in W_i\}$

Una ulteriore importante caratterizzazione di proprietà r.e. o delle rappresentazioni è dato dal seguente Teorema:

Teorema 18.20 (Rice e Shapiro). *Sia* Π *una proprietà estensionale. Se* Π *è r.e. allora per ogni* $i \in \mathbb{N}$

$$i \in \Pi \textit{ sse } \exists j \in \Pi. \varphi_j \subseteq \varphi_i \wedge W_j \textit{ è finito}$$

Proof. ($\rightarrow$) Sia $i \in \Pi$. Se W_i è finito, si prenda $j = i$. Supponiamo che $i \in \Pi$, W_i è infinito, ma $j \notin \Pi$ ogni qual volta $\varphi_j \subseteq \varphi_i$ e W_j è finito. Sia P una

MdT t.c. $P(z)\downarrow$ sse $\varphi_z(z)\downarrow$ e sia g definita come segue:

$$g(z,t) = \begin{cases} \varphi_i(t) & \text{se } P(z) \text{ non termina in } \leq t \text{ passi} \\ \uparrow & \text{altrimenti} \end{cases}$$

Per il Teorema s-m-n esiste s ricorsiva tale che: $g(z,t) = \varphi_{s(z)}(t)$. Ora:

- $z \in K$ implica che $W_{s(z)}$ è finito e $\varphi_{s(z)} \subseteq \varphi_i$. Per ipotesi, $s(z) \notin \Pi$.
- $z \notin K$ implica che $\varphi_{s(z)} = \varphi_i$. Pertanto $s(z) \in \Pi$.

Dunque l'appartenenza a Π è equivalente all'appartenenza a $\bar{K}$ che sappiamo non essere r.e.: assurdo.

($\leftarrow$) Supponiamo che esista $i \in \mathbb{N}$ tale che: $\exists j.\varphi_j \subseteq \varphi_i$ e inoltre W_j è finito e $i \notin \Pi$. Sia g definita come segue:

$$g(z,t) = \begin{cases} \varphi_i(t) & \text{se } t \in W_j \text{ oppure } z \in K \\ \uparrow & \text{altrimenti} \end{cases}$$

Per il Teorema s-m-n esiste $s \in \mathcal{R}$ tale che $g(z,t) = \varphi_{s(z)}(t)$. Abbiamo che:

- $z \in K$ implica che $\varphi_{s(z)} = \varphi_i$. Per l'ipotesi su i, $s(z) \notin \Pi$.
- $z \notin K$ implica che $\varphi_{s(z)}|_{W_j} = \varphi_i$. Dunque $s(z)$ è estensionalmente equivalente a φ_j e pertanto $s(z) \in \Pi$.

Anche in questo caso l'appartenenza a Π è equivalente all'appartenenza a $\bar{K}$ che sappiamo non essere r.e.: assurdo. □

ESERCIZIO 18.21. Si mostri che i seguenti insiemi non sono r.e.:

(1) $\{i : W_i = \mathbb{N}\}$
(2) $\{i : W_i$ è infinito $\}$
(3) $\{i : \forall x \in \mathbb{N}(x \text{ è pari } \rightarrow x \in W_i)\}$

4. Proprietà di programmi

Il Teorema di Rice fornisce un importante risultato per comprendere i limiti dei sistemi automatici per dimostrare o verificare proprietà di programmi. Consideriamo un linguaggio di programmazione Turing-completo $\mathcal{L}$. π è una proprietà dei programmi di $\mathcal{L}$ se $\pi \subseteq \mathcal{L}$, ovvero π è l'insieme dei programmi che verificano la proprietà. Possiamo distinguere tra proprietà sintattiche e semantiche dei programmi. Le prime riguardano come i programmi sono scritti, mentre le seconde riguardano il comportamento. Esempi di proprietà di programmi sono:

$$\begin{array}{ll} \{P \in \mathcal{L} : \forall x \in \mathbb{D}.\ P \text{ termina}\} & \text{proprietà semantica} \\ \{P \in \mathcal{L} : P \text{ contiene un ciclo } \textbf{while}\} & \text{proprietà sintattica} \\ \{P \in \mathcal{L} : |P| < 1000\} & \text{proprietà sintattica} \\ \{P \in \mathcal{L} : P \text{ è un programma sicuro}\} & \text{proprietà semantica} \end{array}$$

È evidente che le proprietà dei programmi di maggior interesse sono quelle semantiche, ovvero quelle che riguardano il comportamento dei programmi, ovvero ciò che questi fanno/calcolano.

Il concetto di proprietà estensionale caratterizza le proprietà semantiche dei programmi. Una proprietà $\pi \subseteq \mathcal{L}$ è estensionale se per ogni $\mathsf{P}, \mathsf{G} \in \mathcal{L}$:

$$\mathsf{P} \in \pi \ \wedge \ [\![\mathsf{P}]\!]^{\mathcal{L}} = [\![\mathsf{Q}]\!]^{\mathcal{L}} \ \Rightarrow \ \mathsf{Q} \in \pi$$

Data una proprietà $\pi \subseteq \mathcal{L}$ è sempre possibile estendere π alla più piccola proprietà estensionale che contiene π. Sia $\mathsf{P} \in \mathcal{L}$ un programma. Definiamo:

$$\mathsf{P}^{\ni} \stackrel{\text{def}}{=} \left\{ \mathsf{Q} \in \mathcal{L} : [\![\mathsf{P}]\!]^{\mathcal{L}} = [\![\mathsf{Q}]\!]^{\mathcal{L}} \right\}$$

È chiaro che $\pi^{\ni} \stackrel{\text{def}}{=} \bigcup_{\mathsf{P} \in \pi} \mathsf{P}^{\ni}$ è la più piccola proprietà estensionale che contiene π. Una proprietà π è estensionale sse $\pi = \pi^{\ni}$. È importante osservare che, per il Lemma del Padding, una proprietà estensionale non vuota è infinita.

Il Teorema di Rice afferma dunque che una proprietà estensionale non vuota di programmi, ovvero in grado di caratterizzare almeno un programma, è ricorsiva, ovvero decidibile, se e solo se questa è banale, ovvero $\pi = \mathcal{L}$. La proprietà $\pi = \mathcal{L}$ corrisponde ad affermare che *un programma è un programma*. Solo quindi questa banale affermazione sui programmi è decidibile, ovvero esiste un algoritmo (programma) in grado di verificarla automaticamente.

Diamo di seguito una dimostrazione alternativa del Teorema di Rice per le proprietà del linguaggio WHILE. Questa dimostrazione mette in luce il legame diretto tra il problema della terminazione e la decidibilità di proprietà estensionali.

PROOF. Supponiamo che π sia estensionale ($\pi = \pi^{\ni}$) non banale ($\pi \neq \mathcal{L}$ e $\pi \neq \varnothing$) e decidibile. Dimostriamo che questo implicherebbe la decidibilità del problema della terminazione.

Consideriamo il programma W sempre divergente:

$$\textbf{read}(\mathsf{x});\ \textbf{while}\ (\mathsf{nil.nil})\ \textbf{do}\ \mathsf{x} := \mathsf{x}\ \textbf{endw};\ \textbf{write}(\mathsf{y})$$

È chiaro che per ogni $\mathbf{d} \in \mathbb{D}$, $[\![W]\!](\mathbf{d}) \uparrow$. Supponiamo che $W \in \pi$. Poiché π è non banale ($\pi \neq \mathcal{L}$), esiste un programma $\mathsf{R} \in \mathcal{L}$ tale che $\mathsf{R} \notin \pi$:

$$\textbf{read}(\mathsf{h});\ \mathsf{C}_{\mathsf{R}};\ \textbf{write}(\mathsf{k})$$

Consideriamo un *generico* programma P:

$$\textbf{read}(\mathsf{x});\ \mathsf{C}_{\mathsf{P}};\ \textbf{write}(\mathsf{y})$$

e sia $e \in \mathbb{D}$. Possiamo assumere che i programmi P ed R non condividano variabili. Costruiamo il seguente programma Q:

$$
\begin{array}{ll}
\textbf{read}(x); & \\
h := x; & [\text{memorizzo l'input } x \text{ in } h] \\
x := e;\ C_P;\ \mathsf{ResP} := y; & [\mathsf{ResP} := [\![\mathsf{P}]\!](e)] \\
C_R;\ \mathsf{ResR} := k; & [\mathsf{ResR} := [\![\mathsf{R}]\!](x)] \\
\textbf{write}(\mathsf{ResR}) &
\end{array}
$$

Se $[\![\mathsf{P}]\!](e) =\uparrow$ allora $\forall d \in \mathbb{D}.\ [\![\mathsf{Q}]\!](d) =\uparrow$. Se $[\![\mathsf{P}]\!](e) =\downarrow$ allora $\forall d \in \mathbb{D}.\ [\![\mathsf{Q}]\!](d) = [\![\mathsf{R}]\!](d)$. Pertanto abbiamo che:

- Se $[\![\mathsf{P}]\!](e) \uparrow$ allora $[\![\mathsf{Q}]\!] = [\![W]\!]$ che per estensionalità di π e poiché $W \in \pi$ implica che $\mathsf{Q} \in \pi$.
- Se $[\![\mathsf{P}]\!](e) =\downarrow$ allora $[\![\mathsf{Q}]\!] = [\![\mathsf{R}]\!]$ che per estensionalità di π e poiché $\mathsf{R} \notin \pi$ implica che $\mathsf{Q} \notin \pi$.

Pertanto, avremo che $[\![\mathsf{P}]\!](e) \downarrow$ se e solo se $\mathsf{Q} \in \pi$: la decidibilità di π implicherebbe la decidibilità della terminazione di $[\![\mathsf{P}]\!](e)$. Vista la genericità di P e dell'argomento $e \in \mathbb{D}$, avremmo dimostrato la decidibilità della terminazione per un linguaggio Turing-completo, il che è assurdo.
Un assurdo analogo si dimostra se $W \notin \pi$ [completare per esercizio]. $\square$

L'indecidibilità delle proprietà estensionali deriva quindi direttamente dalla indecidibilità della terminazione. È importante osservare che questa indecidibilità <u>non</u> deriva necessariamente da complicati programmi di raro interesse pratico. La terminazione del seguente semplice programma, ad esempio, costituisce da anni un enigma per gli scienziati, enigma che a tutt'oggi non risulta essere stato risolto (utilizziamo un linguaggio WHILE esteso con semplici operazioni aritmetiche e con l'istruzione **if-then-else**):

$$
\begin{array}{l}
\textbf{while}\ (n \geq 2)\ \textbf{do} \\
\quad \texttt{if}\ n\ \mathtt{mod}\ 2 = 0\ \texttt{then}\ n := n\ \mathtt{div}\ 2 \\
\qquad \texttt{else}\ n := 3n + 1 \\
\textbf{endw}
\end{array}
$$

L'impatto del Teorema di Rice sullo sviluppo di strumenti automatici di verifica non è tuttavia così drammatico. Lo sviluppo crescente di software di grandi dimensioni (dell'ordine di 10Mlinee) rende impossibile la verifica manuale di importanti proprietà quali la correttezza, la sicurezza, l'assenza di guasti, la fault-tolerance, etc. Queste proprietà, tutte chiaramente estensionali, risultano di vitale interesse per l'utilizzo del software in ambienti safety-critical, quali i sistemi di controllo del traffico aereo, delle transazioni bancarie, di centrali per la produzione di energia.

Risulta quindi necessario sviluppare strumenti automatici (programmi) che analizzano altri programmi. Il Teorema di Rice, pur imponenedo severe limitazioni a ciò che è effettivamente possibile dimostrare sui programmi mediante i programmi stessi (eventualmente definiti in linguaggi diversi), non deve scoraggiare la ricerca di strumenti e metodi per verificare proprietà di programmi. Mentre infatti questo risultato impedisce definitivamente la realizzazione di algoritmi di verifica automatica per <u>tutti</u> i programmi, esso non impedisce che strumenti di verifica automatica siano realizzabili per classi significative di programmi, ovvero che per alcune classi di programmi, proprietà estensionali significative siano decidibili. Il Teorema di Rice stabilisce che non esiste una soluzione universale in grado di verificare in tempo finito se un generico programma soddisfa o meno una data proprietà estensionale. Tuttavia esistono potenti strumenti e metodi di *analisi statica* ovvero a tempo di compilazione, quindi decidibili, per verificare ampie classi di programmi significativi. Questo ambito di studio è competenza dell'area dei *metodi formali* e costituisce uno degli ambiti di ricerca sia teorica che applicata più attivi in questi anni.

CHAPTER 19

Riducibilità funzionale e gradi di risolvibilità

L'inclusione insiemistica, intesa come relazione tra insiemi, non dà nessuna informazione circa la ricorsività di un insieme. In particolare, il fatto che un insieme sia contenuto in un dato insieme ricorsivo non implica in alcun modo la ricorsività del primo: la ricorsività di $\mathbb{N}$ non implica la ricorsività di ogni suo sottoinsieme (e.g. $K \subset \mathbb{N}$). Anzi, come si è visto, la maggior parte dei sottoinsiemi di $\mathbb{N}$ non è nemmeno r.e. In questa sezione studieremo il concetto di *riducibilità funzionale*. Questo concetto permette di correlare tra loro insiemi in modo tale che la ricorsività (o la non ricorsività) di un insieme possa essere propagata agli insiemi ad esso correlati mediante riduzione funzionale.

Consideriamo l'insieme $K_2 = \{\langle x, y\rangle : y \in W_x\}$. Con $\langle x, y\rangle$ si intende in questo contesto una possibile codifica delle coppie di naturali in numero naturale (ad esempio $2^x \cdot 3^y$). In questo modo ci occupiamo solo di relazioni tra insiemi di numeri naturali e non di insiemi di tuple generiche di numeri naturali.

LEMMA 19.1. K_2 *è r.e.* .

PROOF. Si definisca g nel modo seguente: $g(z) = 1$ se $z = \langle x, y\rangle$ e $\varphi_x(y)\downarrow$, indefinita altrimenti. g è la funzione semicaratteristica di K_2 ed è chiaramente calcolabile. □

K_2 ha una struttura molto simile a K, che sappiamo essere un insieme <u>non</u> ricorsivo. Ci chiediamo se anche K_2 sia non ricorsivo. Si osservi che K_2 rappresenta una generalizzazione di K:

$$x \in K \text{ sse } \langle x, x\rangle \in K_2$$

Pertanto, se K_2 fosse ricorsivo, avremmo un modo per decidere l'appartenenza a K (ovvero K sarebbe ricorsivo) che sappiamo essere assurdo. In altri termini, l'esistenza di una funzione totale e calcolabile f tale che

$$x \in K \text{ sse } f(x) \in K_2$$

ci ha permesso di dire qualcosa (di negativo) su K_2.

Quello che abbiamo visto è un metodo per *ridurre* la decidibilità dell'appartenenza ad un insieme alla decidibilità del'appartenenza ad un altro insieme. Questo esempio conduce alla seguente definizione.

DEFINIZIONE 19.2. Siano $A, B \subseteq \mathbb{N}$. A è (funzionalmente) *riducibile* a B ($A \preceq B$) se esiste una funzione (totale) ricorsiva f tale che:

$$x \in A \;\Leftrightarrow\; f(x) \in B$$

Nel caso $A \preceq B$ con funzione ricorsiva f, diremo che A si riduce a B via f. L'ordine $A \preceq B$ fa pensare che A non è più difficile di B. La relazione $\preceq$ è denominata $<_m$ (many-one reducibility) nel testo di Rogers [**27**].

1. La relazione di riducibilità $\preceq$

In questo paragrafo vedremo alcuni esempi e proprietà della nozione di riducibilità funzionale.

ESEMPIO 19.3. Sia $T = \{x : \varphi_x = 0\}$. Mostriamo che $K \preceq T$. Sia $f(x, y) = 0$ se $\varphi_x(x)\downarrow$, indefinita altrimenti. f è calcolabile e per s-m-n esiste g totale ricorsiva tale che $\varphi_{g(x)}(y) = f(x, y)$. Si osserva che $x \in K$ se e solo se $g(x) \in T$.

LEMMA 19.4. *La relazione* $\preceq \subseteq \wp(\mathbb{N}) \times \wp(\mathbb{N})$ *è riflessiva e transitiva.*

ESERCIZIO. $\square$

Pertanto, possiamo definire una relazione di equivalenza $\equiv$ tra insiemi di numeri, in modo tale che $A \equiv B$ sse $A \preceq B$ e $B \preceq A$. Denoteremo con $[A]_\equiv$ la classe di equivalenza di A, ovvero l'insieme $\big\{B \subseteq \mathbb{N} : A \equiv B\big\}$.

In questa relazione l'insieme vuoto $\varnothing$ e l'insieme $\mathbb{N}$ giocano un ruolo particolare e forse inatteso:

LEMMA 19.5. (1) *Sia* A *un insieme.* $\mathbb{N} \preceq A$ *se e solo se* $A \neq \varnothing$.
(2) *Sia* A *un insieme.* $\varnothing \preceq A$ *se e solo se* $A \neq \mathbb{N}$.

PROOF. (1) Sia $A \neq \varnothing$, sia $a \in A$ e sia $f = \lambda x.a$. Si osservi che $x \in \mathbb{N} \leftrightarrow f(x) \in A$.
Viceversa, per assurdo $\mathbb{N} \preceq \varnothing$. Allora esisterebbe f tale che $x \in \mathbb{N} \rightarrow g(x) \in \varnothing$. Ma questo è assurdo in quanto, ad esempio $0 \in \mathbb{N}$, ma $f(0)$ non appartiene a $\varnothing$.

(2) Mostriamo che se $A \subset \mathbb{N}$, allora $\varnothing \preceq A$. Sia $b \in \mathbb{N} \setminus A$ e sia $g = \lambda x.b$. Dobbiamo mostrare che $x \in \varnothing \leftrightarrow g(x) \in B$. Poiché nessun x appartiene ad $\varnothing$ devo solo mostrare che per $x \notin \varnothing$ (ovvero $x \in \mathbb{N}$) vale che: $g(x) \notin A$. Ciò è immediato per la definizione di g.
Viceversa, supponiamo per assurdo che $\varnothing \preceq \mathbb{N}$. Allora esisterebbe g tale che $x \in \varnothing \rightarrow f(x) \in \mathbb{N}$. Dunque dovrebbe valere in particolare per $x = 0$: $0 \notin \varnothing$ e $g(0) \notin \mathbb{N}$. Ma g è totale e dunque $g(0) \in \mathbb{N}$. Assurdo. $\square$

Pertanto $\varnothing$ e $\mathbb{N}$ sono degli insiemi *minimali* rispetto alla relazione $\preceq$ ma tra loro incomparabili (si veda Figura 1—l'insieme S sarà illustrato nel Teorema 19.24).

Vediamo alcune proprietà fondamentali della nozione di riduzione funzionale.

TEOREMA 19.6. *Sia* $A, B \subseteq \mathbb{N}$.
(1) $A \preceq B \;\Rightarrow\; \bar{A} \preceq \bar{B}$;

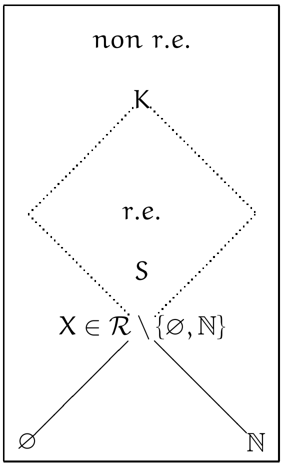


FIGURE 1. Diagramma della relazione $\preceq$

(2) $A \preceq B$ *e* $B \in \mathsf{RE} \;\Rightarrow\; A \in \mathsf{RE}$.
(3) $A \preceq B$ *e* B *ricorsivo* $\;\Rightarrow\;$ A *ricorsivo.*

PROOF. Il primo punto è banale. Dimostriamo che se $A \preceq B$ e $B \in \mathsf{RE}$ allora $A \in \mathsf{RE}$. Sia ψ_B la funzione semicaratteristica parziale ricorsiva di B e supponiamo che $A \preceq B$ via f. Consideriamo la funzione parziale ricorsiva ψ_A definita come: $\psi_A(x) = \psi_B(f(x))$. Allora abbiamo che

$$\psi_A = \begin{cases} 1 & \text{se } f(x) \in B \\ \uparrow & \text{se } f(x) \notin B \end{cases}$$

Poiché $x \in A$ sse $f(x) \in B$, ψ_A è la funzione semicaratteristica di A, e quindi $A \in \mathsf{RE}$. Il terzo punto ha una dimostrazione del tutto analoga al secondo punto, considerando la funzione caratteristica di B. □

Abbiamo quindi dimostrato che la proprietà di *essere ricorsivo* o *ricorsivamente enumerabile* si propaga verso il basso secondo l'ordinamento $\preceq$ tra insiemi. Il seguente lemma evidenzia una importante classe di insiemi $\preceq$ equivalenti.

LEMMA 19.7. B *ricorsivo e* $A \neq \varnothing$ *e* $A \neq \mathbb{N}$ *implica* $B \preceq A$.

PROOF. Sia $a \in A$ e $b \in \mathbb{N} \setminus A$ e sia ψ_B la funzione caratteristica di B. Definisco

$$g(x) = \begin{cases} a & \psi_B(x) = 1 \\ b & \psi_B(x) = 0 \end{cases}$$

g è ricorsiva e totale e vale che $x \in B$ se e solo se $g(x) \in A$. □

Come corollario, tutti gli insiemi ricorsivi diversi da $\varnothing$ e da $\mathbb{N}$ (insiemi ricorsivi non banali) sono tra loro $\preceq$-equivalenti.

Analizziamo ora gli insiemi r.e. .

TEOREMA 19.8. *Per ogni* $A \in RE$*,* $A \preceq K_2$*.*

PROOF. Sia $A \in RE$. Allora esiste y_0 tale che $A = W_{y_0}$. Per definizione di K_2, $x \in A = W_{y_0}$ sse $\langle y_0, x\rangle \in K_2$. Definiamo dunque $f_{y_0} = \lambda x.\ \langle y_0, x\rangle$. È chiaro che $x \in A$ sse $f_{y_0}(x) \in K_2$. □

Pertanto, ogni insieme r.e. è riducibile a K_2. Questo significa che K_2 ha un ruolo speciale all'interno della classe degli insiemi r.e. : ogni elemento di questa classe è riducibile a K_2.

DEFINIZIONE 19.9. Un insieme $C \in RE$ è *completo* (più precisamente, r.e. completo) se per ogni $A \in RE$: $A \preceq C$.

K_2 è dunque completo. Il seguente teorema dimostra che anche K è completo. Pertanto, $K \equiv K_2$, ed entrambi giocano un ruolo speciale (massimale) all'interno della classe degli insiemi r.e. È inoltre banale osservare che A è completo sse $A \in [K]_\equiv$.

TEOREMA 19.10. K *è completo.*

PROOF. Basta dimostrare che $K_2 \preceq K$. Infatti, in questo modo, poiché ogni insieme r.e. è riducibile a K_2 (vedi Teorema 19.8), allora per transitività, ogni insieme r.e. è anche riducibile a K, dimostrando la completezza di K.

Sia $\langle x, y\rangle \in K_2$. Per definizione, ciò è vero sse $y \in W_x$. Definiamo la funzione

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

È chiaro che ψ è parziale ricorsiva, quindi, per il teorema s-m-n, esiste una funzione ricorsiva (totale) g tale che: $\psi(x,y,z) = \varphi_{g(x,y)}(z)$. Si noti come, fissato $x, y \in \mathbb{N}$, la funzione $\lambda z.\ \varphi_{g(x,y)}(z)$ sia indipendente dal valore dell'argomento z. Si osserva che $\langle x, y\rangle \in K_2$ sse $\varphi_{g(x,y)}(z)\downarrow$ per $z \in \mathbb{N}$ qualsiasi. Quindi, a maggior ragione $\langle x, y\rangle \in K_2$ sse $\varphi_{g(x,y)}(g(x,y))\downarrow$. Questo dimostra che $\langle x, y\rangle \in K_2$ sse $g(x,y) \in K$, con g funzione ricorsiva. □

COROLLARIO 19.11. $C \in RE$ è completo se e solo se $K \leq C$.

PROOF. Immediato dalla definizione. □

ESEMPIO 19.12. Sia $F = \{x : |W_x| \text{ è finito}\}$. Dimostriamo che $K \preceq F$. Definiamo la funzione:

$$\psi(x,y) = \begin{cases} 1 & \text{se } \varphi_x(x) \text{ non converge in meno di } y \text{ passi} \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione è parziale ricorsiva (si descriva a parole una possibile MdT che calcola ψ). Per il teorema s-m-n, esiste g ricorsiva (totale) tale che $\psi(x,y) = \varphi_{g(x)}(y)$.

- Sia $x \in K$. Allora, per definizione di K: $\varphi_x(x)\downarrow$. Dunque esiste n tale che $\varphi_x(x)\downarrow$ in n passi. Dunque $W_{g(x)} = \{0,\dots,n-1\}$. Pertanto $g(x) \in F$.
- Sia $x \notin K$. Allora $\varphi_x(x)\uparrow$. Dunque $W_{g(x)} = \mathbb{N}$ e dunque $g(x) \notin F$.

Si osservi che l'esempio non dimostra che F è completo. Manca infatti la proprietà che F sia RE. Per mostrare che F non è r.e. si mostri per esercizio che $\bar{K} \preceq F$.

Si osservi inoltre che se A è r.e. e non ricorsivo, allora non vale né che $A \preceq \bar{A}$ né che $\bar{A} \preceq A$. Se fosse $A \preceq \bar{A}$ allora avremmo un modo per semidecidere $\bar{A}$ che è assurdo poiché A non è ricorsivo. Se fosse $\bar{A} \preceq A$ allora, in virtù del Teorema 19.6(1), varrebbe anche $A \preceq \bar{A}$. Dunque in particolare $K \not\preceq A$ per ogni A che sia complemento di un insieme r.e. . Nel seguente esercizio si mostra comunque che esistono insiemi non r.e. che stanno sopra a K (ad esempio $K \,\mathsf{join}\, \bar{K}$).

Esercizio 19.13. Consideriamo due insiemi A e B. Chi è un insieme (possibilmente il minimo) X tale che $A \preceq X$ e $B \preceq X$? Non può essere $A \cup B$. Si giunge infatti ad un assurdo partendo da $B = \bar{A}$, con $A \neq \varnothing$ e $A \neq \mathbb{N}$. Verificare che tale insieme è:

$$A \,\mathsf{join}\, B = \{y \,:\, (y = 2x \wedge x \in A) \vee (y = 2x+1 \wedge x \in B)\}$$

Si mostri inoltre che se $A \leq B$ allora $A \,\mathsf{join}\, B \equiv B$.

Esercizio 19.14. Dimostrare che:

- Se $A \in \mathsf{RE}$, $A \neq \varnothing$ e $A \neq \mathbb{N}$, allora A ricorsivo $\Leftrightarrow$ $A \preceq \bar{A}$.
- $A \in [B]_\equiv$ e A ricorsivo allora per ogni $X \in [B]_\equiv$, X è ricorsivo.

2. Insiemi creativi e produttivi

Abbiamo visto che all'interno della classe degli insiemi r.e. , esistono insiemi (e.g. K) che hanno la proprietà per cui ogni altro insieme r.e. è ad essi riducibile. Questi insiemi, detti completi, rappresentano quindi la classe degli insiemi r.e. , quasi fossero dei rappresentanti “canonici” di questa classe.

In questa sezione caratterizzeremo gli insiemi completi. Caratterizzare gli insiemi completi significa dare una condizione che determini in modo univoco se un dato insieme è o meno completo, senza dovere ricorrere ad altri insiemi r.e.

$\bar{K}$ non è r.e. . Ma questa proprietà può essere mostrata anche in modo costruttivo e convincente: ogni volta che un insieme r.e. W_x è incluso in $\bar{K}$ siamo in grado di trovare un elemento che appartiene a $\bar{K} \setminus W_x$. Tale elemento, in questo caso è proprio x. La seguente definizione generalizza questa proprietà.

Definizione 19.15. Un insieme $A \subseteq \mathbb{N}$ è detto *produttivo* se esiste una funzione (totale) ricorsiva f_A, detta funzione produttiva di A, tale che:

$$\forall x \in \mathbb{N}.\ (W_x \subseteq A \ \Rightarrow f_A(x) \in A \setminus W_x)$$

Un insieme A è detto *creativo* se A è r.e. ed il suo complemento è produttivo, ovvero se:

$$A \in \mathsf{RE} \text{ e } \forall x \in \mathbb{N}. \, (W_x \subseteq \bar{A} \; \Rightarrow f_A(x) \in \bar{A} \setminus W_x)$$

Intuitivamente un insieme è produttivo se per ogni tentativo di enumerarlo in modo effettivo mediante un algoritmo di indice x (ovvero per ogni $W_x \subseteq A$), esiste una trasformazione effettiva di x in un punto di A che sfugge all'enumerazione. È evidente che un insieme produttivo non può essere r.e. , altrimenti avremmo l'assurdo che $A = W_{x_0}$ per un certo x_0 e al tempo stesso $A \setminus W_{x_0} \neq \varnothing$, poiché $f(x_0) \in A \setminus W_{x_0}$. Al contrario, un insieme creativo è sempre, per definizione, r.e., ma il suo complemento non lo può essere. Riassumendo:

TEOREMA 19.16. *Sia* $A \subseteq \mathbb{N}$.

(1) A *produttivo* $\Rightarrow$ A *non r.e.*
(2) A *creativo* $\Rightarrow$ A *non ricorsivo*

PROOF. Immediato per definizione. □

L'insieme $\mathsf{K} = \{x : x \in W_x\}$ ancora una volta è rappresentativo per la classe di insiemi creativi, come dimostrato nel seguente teorema.

TEOREMA 19.17. K *è creativo e* $\bar{\mathsf{K}}$ *è produttivo.*

PROOF. Dimostriamo che $\bar{\mathsf{K}}$ è produttivo. Consideriamo la funzione ricorsiva di base $\mathtt{id} = \lambda x.\, x$. Ci chiediamo in quale regione del diagramma di Venn possa stare x:

$$x \in W_x? \qquad x \in \mathsf{K}? \qquad x \in \bar{\mathsf{K}} \setminus W_x?$$

Mostreremo che le prime due ipotesi portano ad un assurdo.

Se $W_x \subseteq \bar{\mathsf{K}}$ allora $\mathtt{id}(x) = x \notin W_x$. Infatti, se $x \in W_x$, allora per definizione di K: $x \in \mathsf{K}$. Poiché $W_x \subseteq \bar{\mathsf{K}}$, allora $x \in \bar{\mathsf{K}}$ che sarebbe assurdo. Quindi $\mathtt{id}(x) = x \notin W_x$.

Se $x \in \mathsf{K}$ allora avremmo $x \in W_x \subseteq \bar{\mathsf{K}}$, ovvero $x \in \bar{\mathsf{K}}$, che è assurdo poiché x non può essere al tempo stesso in K e $\bar{\mathsf{K}} = \mathbb{N} \setminus \mathsf{K}$. Quindi se $W_x \subseteq \bar{\mathsf{K}}$ allora $\mathtt{id}(x) = x \in \bar{\mathsf{K}} \setminus W_x$, ovvero $\bar{\mathsf{K}}$ è produttivo. La dimostrazione che K è creativo è immediata poiché $\mathsf{K} \in \mathsf{RE}$ e $\bar{\mathsf{K}}$ è produttivo. □

Il seguente teorema dimostra una proprietà fondamentale degli insiemi creativi e produttivi, ovvero che la produttività e la creatività di un insieme viene ereditata per riduzione funzionale.

TEOREMA 19.18. *Siano* $A, B \subseteq \mathbb{N}$.

(1) A *produttivo e* $A \preceq B$ $\Rightarrow$ B *produttivo.*
(2) A *creativo*, $A \preceq B$, *e* $B \in \mathsf{RE}$ $\Rightarrow$ B *creativo.*

PROOF. Dimostriamo il punto (2) e lasciamo il punto (1) per esercizio. La dimostrazione è del tutto analoga al caso creativo.

(2). Sia A creativo e $B \in \mathsf{RE}$ tale che $A \preceq B$ via f. Sia h la funzione produttiva di $\bar{A}$. Nella dimostrazione faremo uso della notazione $f^{-1}(X) = \{x : f(x) \in X\}$. E' immediato verificare che f^{-1} è monotona, ovvero $X \subseteq Y \to f^{-1}(X) \subseteq f^{-1}(Y)$.

f è ricorsiva, dunque esiste una funzione parziale ricorsiva $\psi(x, y)$ tale che

$$\psi(x, y) = \begin{cases} 1 & \text{se } f(y) \in W_x \\ \uparrow & \text{altrimenti} \end{cases}$$

Per il teorema s-m-n, esiste una funzione totale ricorsiva g tale che: $\psi(x, y) = \varphi_{g(x)}(y)$. Inoltre, per definizione di ψ, fissato $x \in \mathbb{N}$: $W_{g(x)} = f^{-1}(W_x)$.

Dimostriamo ora che B è creativo. Per fare ciò basta dimostrare che $\bar{B}$ è produttivo. Mostreremo che se $W_x \subseteq \bar{B}$, allora $f(h(g(x))) \in B \setminus W_x$.

- Mostriamo che $f(g(g(x))) \in B$.

  Se $W_x \subseteq \bar{B}$, dunque $f^{-1}(W_x) \subseteq f^{-1}(\bar{B})$. Poiché $A \preceq B$ via f, $f^{-1}(\bar{B}) = \bar{A}$. Inoltre sappiamo già che $f^{-1}(W_x) = W_{g(x)}$. Pertanto si ha che $W_{g(x)} \subseteq \bar{A}$. Per la produttività di $\bar{A}$ si ha dunque che:

$$h(g(x)) \in \bar{A} \setminus W_{g(x)} \tag{2.1}$$

  Poiché $A \preceq B$ via f, e $h(g(x)) \in \bar{A}$ si ha che $f(h(g(x))) \in \bar{B}$.
- Mostriamo che $f(h(g(x))) \notin W_x$. Se per assurdo fosse $f(h(g(x))) \in W_x$, per la definizione di ψ si avrebbe che $h(g(x)) \in W_g(x)$ che contraddice (2.1).

□

Il seguente teorema caratterizza gli insiemi completi come tutti e soli gli insiemi creativi, ovvero aventi un complemento produttivo. Gli insiemi completi hanno dunque complementi non r.e. . Nella dimostrazione del Teorema seguente faremo uso del secondo Teorema di ricorsione (Teorema 18.6).

TEOREMA 19.19 (Myhill). *Sia* $A \in \mathsf{RE}$.

$$A \textit{ completo} \Leftrightarrow A \textit{ creativo}$$

PROOF. ($\Rightarrow$). Sia $A \in \mathsf{RE}$. A è completo sse per ogni $X \in \mathsf{RE}$: $X \preceq A$. Poiché K è creativo e A completo, $K \preceq A$. Per il Teorema 19.18, anche A è creativo.

($\leftarrow$). Sia A creativo con funzione produttiva per $\bar{A}$, f. Sia B un generico insieme in RE che vogliamo ridurre ad A. Definiamo la funzione:

$$\psi(x, y, z) = \begin{cases} 0 & \text{se } y \in B \text{ e } z = f(x) \\ \uparrow & \text{altrimenti} \end{cases}$$

Per il teorema s-m-n, esiste una funzione calcolabile totale g tale che $\psi(x, y, z) = \varphi_{g(x,y)}(z)$. Quindi:

$$W_{g(x,y)} = \begin{cases} \{f(x)\} & \text{se } y \in B \\ \varnothing & \text{se } y \notin B \end{cases}$$

Per il secondo teorema di ricorsione (Teorema 18.6) esiste una funzione calcolabile ν tale che per ogni $y \in \mathbb{N}$

$$\varphi_{g(\nu(y),y)} = \varphi_{\nu(y)}$$

Allora, per ogni y si ha che:

$$W_{g(\nu(y),y)} = W_{\nu(y)} = \begin{cases} \{f(\nu(y))\} & \text{se } y \in B \\ \varnothing & \text{se } y \notin B \end{cases}$$

Mostriamo che $f \circ \nu$ è la funzione di riduzione da B ad A.

- Se $y \in B$ allora $W_{\nu(y)} = \{f(\nu(y))\}$. Dunque, poiché f è la funzione produttiva di $\bar{A}$, se fosse $f(\nu(y)) \notin A$, allora varrebbe che $W_{\nu(y)} \subseteq \bar{A}$ e pertanto $f(\nu(y)) \in \bar{A} \setminus W_{\nu(y)}$ ovvero che $f(\nu(y)) \notin W_{\nu(y)}$. Assurdo.
- Se $y \notin B$, allora $W_{\nu(y)} = \varnothing$. Dunque $W_{\nu(y)} \subseteq \bar{A}$, e pertanto $f(\nu(y)) \in \bar{A}$.

□

Come conseguenza di questo fatto, osserviamo che tutti gli insiemi creativi sono tra loro equivalenti secondo $\equiv$. Inoltre, se A è creativo, allora la classe di equivalenza $[A]_\equiv$ è massima nell'ordinamento indotto da $\preceq$ sulle classi di equivalenza di insiemi r.e.

Esercizio 19.20.

(1) Studiare la relazione $\preceq$ o $\equiv$ esistente tra i seguenti insiemi:
   - $\{x : x \text{ è primo}\}$;
   - $\{x : x \text{ è pari}\}$;
   - $\{x : W_x = \varnothing\}$;
   - $\{x : |W_x| = \omega\}$;
   - $\{x : |W_x| < \omega\}$;
   - $\{x : \varphi_x \text{ è totale}\}$.

(2) Dimostrare che $\{x : \varphi_x \text{ è totale}\}$ è produttivo (Suggerimento: dimostrare che $\bar{K} \preceq \{x : \varphi_x \text{ è totale}\}$).

(3) Dimostrare che $\{i : W_i \text{ è ricorsivo}\}$ è produttivo.

(4) Dimostrare che $\{i : \varphi_i(i) = 0\}$ è creativo.

(5) Dimostrare che $\{x : \exists y.\, x \in W_y \ \wedge \ y \in W_x\}$ è creativo.

(6) Dimostrare che $\{x : \exists u \exists v. x = \langle u, v\rangle \wedge \varphi_{u \cdot v}(u + v) = 0\}$ è creativo.

Il seguente teorema stabilisce che ogni insieme produttivo contiene un insieme r.e. infinito.

Teorema 19.21. *Sia A un insieme produttivo. Esiste $n_0 \in \mathbb{N}$ tale che*

$$W_{n_0} \subseteq A \ \wedge \ |W_{n_0}| = \omega$$

ovvero esiste un sottoinsieme r.e. e infinito di A.

PROOF. Sia f la funzione di produttività di A. Costruiamo una funzione totale ricorsiva g tale che:

$$\mathrm{range}(g) \subseteq A \;\wedge\; |\mathrm{range}(g)| = \omega$$

La definizione di g è data per induzione. $\varnothing$ è r.e. , e so costruire una MdT M tale che $M(x) \uparrow$ per ogni x. Sia z_0 il suo indice (dunque $\varnothing = W_{z_0}$).

- Poniamo $g(0) \stackrel{\text{def}}{=} \psi(z_0)$. Poiché $W_{z_0} = \varnothing \subseteq A$ per definizione di produttività abbiamo che: $\psi(z_0) \in A$ e $\psi(z_0) \notin W_{z_0}$.
- Supponiamo di aver già calcolato $g(0), \ldots, g(n)$: so costruire una MdT M tale che $M(g(0)) = \cdots = M(g(n)) = 0$ e $M(x) \uparrow$ per ogni x diverso da $g(0), \ldots, g(n)$. Sia z_{n+1} il suo indice: dunque $W_{z_{n+1}} = \{g(0), \ldots, g(n)\}$. Poniamo $g(n+1) \stackrel{\text{def}}{=} f(z_{n+1})$. Poiché $W_{z_{n+1}} = \{g(0), \ldots, g(n)\} \subseteq A$ per definizione di produttività otteniamo: $f(z_{n+1}) \in A$ e $f(z_{n+1}) \notin W_{z_{n+1}}$.

La iniettività di g deriva dal fatto che, per produttività di A, $g(n) = f(z_n) \in A \setminus W_{z_n} = \{g(0), \ldots, g(n-1)\}$. Da questo segue che $|\mathrm{range}(g)| = \omega$. La ricorsiva enumerabilità di $\mathrm{range}(g) \subseteq A$ deriva dal Teorema di caratterizzazione di Kleene degli insiemi r.e. . □

3. Insiemi semplici

Da quanto abbiamo visto: creativi $\cup$ ricorsivi $\subsetneq$ r.e. . Ci chiediamo se tutti gli insiemi r.e. sono o ricorsivi o creativi. La risposta è no. Per stabilire questo fatto, definiamo il concetto di insieme *semplice*.

DEFINIZIONE 19.22. $A \subseteq \mathbb{N}$ è *semplice* se

- A è r.e. ,
- $|\bar{A}| = \omega$, ovvero $\bar{A}$ è infinito,
- $\forall x \in \mathbb{N}.\ |W_x| = \omega \;\rightarrow\; A \cap W_x \neq \varnothing$.

TEOREMA 19.23.

(1) A *semplice* $\rightarrow$ A *non ricorsivo*
(2) A *semplice* $\rightarrow$ A *non creativo*

PROOF. Per dimostrare il punto (1), supponiamo che A sia ricorsivo. Per il Teorema di Post, $\bar{A}$ è ricorsivo. Pertanto, esiste $x_0 \in \mathbb{N}$ tale che $\bar{A} = W_{x_0}$ e, essendo $|\bar{A}| = \omega$, $|W_{x_0}| = \omega$. Per definizione di insieme semplice, $A \cap W_{x_0} = A \cap \bar{A} \neq \varnothing$ che è assurdo.
Per dimostrare il punto (2), supponiamo che A sia creativo. Per definizione, $\bar{A}$ è produttivo e, per il Teorema 19.21, esiste x_0 tale che $W_{x_0} \subset \bar{A}$ e $|W_{x_0}| = \omega$. Per definizione di insieme semplice, questo implica che $A \cap W_{x_0} \neq \varnothing$, ma questo è assurdo poiché $W_{x_0} \subset \bar{A}$. □

Resta ora da vedere se esiste un insieme di numeri con le caratteristiche degli insiemi semplici. Il seguente teorema, dovuto a Post, dimostra che esistono insiemi semplici, che non sono né ricorsivi né creativi.

Teorema 19.24. *Esiste un insieme semplice.*

Proof. definiamo il seguente insieme:

$$A \stackrel{\text{def}}{=} \big\{(x, y) : y \in W_x \wedge y > 2x\big\} = \big\{(x, y) : \varphi_x(y) \downarrow \wedge y > 2x\big\}$$

È ovvio che A è r.e. e che $A \neq \varnothing$. Pertanto, esiste $x_0 \in \mathbb{N}$ tale che $A = W_{x_0}$ e $A = \mathtt{range}(\varphi_{h(x_0)})$. Definiamo una relazione di precedenza $\triangleleft$ sulle coppie di numeri tale che:

$$(x', y') \triangleleft (x, y) \quad \text{sse} \quad \exists n \leq m.\ (x', y') = \varphi_{h(x_0)}(n) \wedge (x, y) = \varphi_{h(x_0)}(m).$$

ovvero, $(x', y') \triangleleft (x, y)$ se (x', y') viene generata prima nella enumerazione di A da parte di $\varphi_{h(x_0)}$ rispetto a (x, y). Definiamo l'insieme:

$$B \stackrel{\text{def}}{=} \big\{(x, y) \in A : \forall z \neq y.\ (x, z) \in A \rightarrow (x, y) \triangleleft (x, z)\big\}$$

Si noti che B è r.e. [perchè?].

Definiamo dunque l'insieme S:

$$S \stackrel{\text{def}}{=} \big\{y : \exists x.\ (x, y) \in B\big\}$$

e mostriamo che è semplice.

- S è r.e. . (in quanto lo è B)
- $|\bar{S}| = \omega$. Infatti, sia $n \in \mathbb{N}$. Al più n elementi di $\{0, \dots, 2n\}$ sono in S poiché per ogni numero pari x c'è al più un elemento y tale che (x, y) è la prima coppia di A.
- Inoltre, sia $|W_x| = \omega$. Allora esiste $y > 2x$ tale che $y \in W_x$ (essendo W_x infinito) e $y \in W_x \cap S$, da cui segue la tesi.

□

Corollario 19.25. Vale che $S \preceq K$ ma $K \not\preceq S$.

Proof. S è semplice e pertanto r.e.; dunque $S \preceq K$ per completezza di K.

Se fosse $K \preceq S$ allora per il Teorema di Myhill S sarebbe creativo. Questo condraddirebbe il Teorema 19.23. □

Siamo dunque in grado di collocare S nel diagramma di Figura 1.

Part 4

Complessità Computazionale

CHAPTER 20

Classi di complessità e principali risultati

Una delle linee guida del testo è quella di investigare la nostra conoscenza di sottoinsiemi di $\mathbb{N}$ o, equivalentemente, di Σ^* (ovvero di insiemi di stringhe di caratteri di lunghezza finita costruiti a partire da un dato alfabeto Σ). Più precisamente, il criterio di conoscenza richiesto riguarda la capacità di stabilire se un elemento appartenga o meno ad uno di questi insiemi. Si è mostrato come per alcuni di essi tale problema sia decidibile (insiemi ricorsivi). In particolare si sono caratterizzati mediante grammatiche e automi alcune importanti famiglie di insiemi ricorsivi quali i linguaggi regolari, quelli liberi, e quelli dipendenti dal contesto. Si è mostrato come per altri il problema sia solo semidecidibile (insiemi r.e.), e che vi siano insiemi per cui l'appartenenza non è nemmeno semidecidibile (ad esempio per gli insiemi produttivi).

Quando un programmatore è chiamato a risolvere un problema, il primo obiettivo è quello di scrivere un programma che lo risolva e che *termini* per tutte le possibili istanze di input, al fine di evitare spiacevoli attese infinite all'utilizzatore del programma stesso. Dunque, per quanto non esista un programma universale in grado di stabilire se un programma goda della suddetta proprietà di terminazione, il programmatore deve scrivere il codice sincerandosi, mediante opportune dimostrazioni spesso di tipo induttivo, che la proprietà di terminazione per quel particolare programma valga. L'impiego di linguaggi di programmazione ad alto livello è particolarmente di aiuto in questa fase.

Tuttavia nella pratica questa proprietà è necessaria ma non sufficiente. Sapere che il programma *prima o poi* terminerà può non essere d'aiuto a chi necessita della risposta entro pochi minuti. Un passo avanti si potrebbe avere fornendo all'utilizzatore una ulteriore informazione: se l'input consta di k bits, allora il programma su quell'input terminerà entro $f(k)$ secondi. L'utente ora sa quanto aspettare. Ma se $f(k) = 2^{(2^k)}$ allora per un input di 1 Byte avremo bisogno di $2^{256}\mathsf{s} \sim 3.7 \cdot 10^{69}$ anni. Se questo programma decide l'appartenenza ad un insieme, sapere che quell'insieme è ricorsivo, non pare aiutarci troppo. Ci rimane però un dubbio: quel programma è il più veloce possibile per decidere l'appartenenza a quell'insieme e dunque quella funzione doppiamente esponenziale è una misura di complessità dell'insieme stesso, oppure è solo basato un algoritmo male architettato per risolvere un problema più facile?

Rispondere a domande di questo tipo è il principale scopo della teoria della complessità. In particolare si cerca di suddividere i problemi, gli insiemi in *classi di complessità*. Ciò permette di partizionare la classe degli insiemi ricorsivi in vare famiglie a complessità crescente o, meglio, non decrescente. Sono infatti numerosi e stimolanti i problemi aperti ancora esistenti riguardanti l'inclusione stretta o meno tra classi di complessità, tra cui il più famoso è $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, aperto ufficialmente da Cook nei primi anni 70 [**5**] ma esistente già nella sostanza nel contenuto di lettere private intercorse nei primi anni 50 tra Gödel e Von Neumann [**29**].

1. Problemi, insiemi, linguaggi

Il concetto di *problema*, il concetto di *insieme*, inteso come sottoinsieme di $\mathbb{N}$, e quello di *linguaggio* vengono utilizzati in modo equivalente nella teoria della complessità. Un *problema* (decisionale) viene tipicamente posto fornendo una certa caratterizzazione degli input possibili e una condizione da verificarsi su ciascuna *istanza* dell'input. Vi saranno delle istanze con risposta affermativa (istanze *yes*) e istanze con risposta negativa (istanze *no*). Ad esempio, prendiamo il problema: *dato un grafo* $\mathsf{G} = \langle \mathsf{N}, \mathsf{E} \rangle$*, il grafo è connesso?* Vi saranno pertanto istanze yes come $\mathsf{G} = \langle \{1,2\}, \{\{1,2\}\} \rangle$ e istanze no come $\mathsf{G} = \langle \{1,2\}, \varnothing \rangle$. Possiamo tranquillamente assumere che ogni istanza (o input) x sia una stringa di un alfabeto Σ fissato (nel caso sopra $\Sigma = \{`0', \ldots, `9', `\{', `\}', `\langle', `\rangle', `,'\}$). Il *linguaggio* associato al problema sarà l'insieme delle istanze *yes* ovvero $\mathsf{L} = \{x \in \Sigma^* :$ x verifica la condizione del problema$\}$. Tale linguaggio è per definizione stessa un insieme. Si osservi che in base a tale definizione ogni stringa che non descrive correttamente un'istanza dell'input (nell'esempio sopra potrebbe essere: $\langle \{\{\{, , , 112 \rangle \rangle$) finisce nelle istanze no. Ovviamente, poiché, dato Σ finito, Σ^* è numerabile, potremmo senza perdita di generalità studiare semplicemente sottoinsiemi di $\mathbb{N}$, come fatto nella parte 3 del presente testo.

Si osservi che la scelta di definire un problema nel modo suddetto permette di stabilire in modo naturale e rigoroso il concetto di *dimensione* dell'input come la lunghezza della stringa x, al solito denotata come $|x|$.

Un'ulteriore precisazione va fatta relativamente alla scelta dei problemi decisionali. Abbiamo appena mostrato come questo permetta di concentrarsi sul problema di appartenenza ad un insieme che è stata anche la guida per la classificazione di insiemi ricorsivi e riscorsivamente enumerabili. E' inoltre vero che ciò non rappresenta una limitazione significativa. Ragioniamo su un esempio. Si considerino i seguenti due problemi, il primo funzionale, il secondo, ottenuto a partire dal primo, decisionale:

(1) Dato un grafo completo $\mathsf{G} = \langle \mathsf{N}, \mathsf{E} \rangle$ e una funzione $\mathsf{c} : \mathsf{E} \longrightarrow \mathbb{N}$ che assegna un costo ad ogni arco, calcolare il circuito di costo minimo che parte e arriva nel nodo 1 e passa una sola volta per tutti gli altri nodi.
(2) Dato un grafo completo $\mathsf{G} = \langle \mathsf{N}, \mathsf{E} \rangle$, una funzione $\mathsf{c} : \mathsf{E} \longrightarrow \mathbb{N}$ che assegna un costo ad ogni arco, e un intero t, dire se esiste un circuito che parte e

arriva nel nodo 1 e passa una sola volta per tutti gli altri nodi, di costo minore o uguale a t?

E' chiaro che sapendo risolvere (1) avremmo immediatamente un metodo per risolvere (2). Viceversa, assumiamo di disporre dell'algoritmo **due** che risolve (2). Procederemo per bisezione:

$m := 0;$

$M := \sum_{e \in E} c(e);$

while $(M - m > 1)$ **do**

  $t := (M + m)/2;$

  if **due**$(G, c, t) = yes$

  then $m := t$

  else $M := t$

endw

Quando $M = m + 1$ ci si ferma. Il valore t_f del circuito di lunghezza minima per (1) va cercato solo tra M e m. Si tratta dunque di ripetere l'esecuzione di **due** un numero di volte massimo pari a $\log(\sum_{e \in E} c(e))$ che è proporzionale al numero di caratteri necessari per descrivere la funzione c ovvero $|c|$. Pertanto il numero di iterazioni risulta limitato linearmente dalle dimensioni dell'input.

Per scoprire quali sono gli archi utilizzati nella soluzione con valore t_f, si parta dal nodo iniziale. Uno ad uno, si assegni ad un arco in uscita da lui il valore t_f. Si lanci dunque **due** con limite t_f. Se la risposta è *yes*, quell'arco non stava nel cammino minimo. Altrimenti ci stava e dunque riassegnamo a lui il suo costo indicato da c e passiamo al nodo successivo.

Dunque, dalla soluzione al problema decisionale si può ottenere la soluzione a quello funzionale con un numero di operazioni che dipende polinomialmente dall'algoritmo per il problema decisionale moltiplicato per le dimensioni dell'input.

2. Classi di complessità in tempo e Tesi di Church computazionale

Seguendo il testo [**23**], a cui rimandiamo il lettore per approfondimenti, useremo il seguente modello computazionale:

DEFINIZIONE 20.1. Una k-*MdT* è una Macchina di Turing $M = \langle Q, \Sigma, q_0, P \rangle$ con k-nastri. Più precisamente:

(1) Q è l'insieme finito degli stati e $q_0 \in Q$ è lo stato iniziale.
(2) Σ è l'alfabeto dei simboli presenti sui nastri. Assumiamo che i simboli $\triangleright$ e \$ appartengano a Σ.

(3) L'insieme delle istruzioni P rappresenta una funzione di transizione δ che assumiamo essere una funzione *totale*

$$\delta : Q \times \Sigma^k \longrightarrow (Q \cup \{h, yes, no\}) \times (\Sigma \times \{L, R, F\})^k$$

(4) h, yes, no sono degli stati finali non presenti in Q, rispettivamente, di terminazione, accettazione, e refutazione. δ non è definita su di essi.
(5) Si osservi che l'insieme dei movimenti (per ogni nastro) consta di tre possibilità: L (sinistra), R (destra), e F (fermo).
(6) Assumiamo inoltre che il simbolo $\rhd$ non possa essere cancellato e quando viene letto su un nastro, l'effetto è che la testina, relativamente a quel nastro, si sposta a destra.
(7) La *configurazione iniziale* è del tipo:

$$(q_0, \underbrace{\varepsilon, \rhd, x}_{\text{nastro } 1}, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro } 2}, \dots, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro } k})$$

ove per ogni nastro $1, \dots, k$ i tre parametri sono: la stringa significativa a sinistra della testina, il simbolo letto e la stringa significativa a destra della testina, mentre x è l'input (sul nastro 1).

Si osservi che l'assumere δ totale permette di parlare di δ anziché dell'insieme delle produzioni P. Le assunzioni (6) e (7) fanno sí che le k MdT lavorino su nastri semi-infiniti (non si vai mai più a sinistra del $\rhd$ iniziale presente su ogni nastro). Vi è inoltre la possibilità del movimento nullo (F). Sebbene esso possa essere facilmente mimato da due movimenti L–R consecutivi, avere a disposizione anche il movimento F permette di scrivere del codice più snello.

Nota 20.2. Si osservi che per $k = 1$ la nozione di MdT appena definita risulta essere una restrizione della definizione generale (salvo per il movimento nullo che però, come appena detto, può facilmente essere simulato). La restrizione riguarda la presenza del simbolo $\rhd$ e le sue proprietà, la totalità di δ nonché gli stati h, yes, no. E' comunque facile convincersi che queste restrizioni non pregiudicano la Turing completezza del modello delle 1-MdT (richieste analoghe sono state fatte nelle dimostrazioni di equivalenza tra funzioni Turing-calcolabili e gli altri formalismi nei Capitoli 13 e 16).

Le nozioni già viste di relazione di successore tra configurazioni istantanee, computazioni eccetera, si adattano immediatamente a questo formalismo. In particolare, la macchina *termina* se e solo se raggiunge una configurazione con lo stato h, yes o no.

Definizione 20.3. Una k-MdT M è di tipo *decisionale* se ogni qual volta essa termina, raggiunge uno degli stati finali yes o no. L'output della macchina è, in questo caso, lo stato raggiunto.

Una k-MdT M invece *calcola* una funzione se ogni qual volta termina, essa raggiunge lo h. In questo caso l'output è il contenuto del k-esimo nastro.

Nello studio classico della complessità si è interessati alle macchine decisionali. Per completezza, tuttavia, abbiamo fornito anche la definizione nel caso più generale, che permette di definire le classi di complessità funzionali (si veda [**23**]).

DEFINIZIONE 20.4. Data una k-MdT M e un input x, il *tempo richiesto da* M *per* x è il numero di passi di computazione necessari a M con input x per terminare.

Sia $f : \mathbb{N} \longrightarrow \mathbb{N}$ una fuzione totale. Si dice che una k-MdT M *opera in tempo* $f(n)$ se per ogni input x il tempo richiesto da M per x è minore o uguale a $f(|x|)$ (ove, al solito, $|x|$ denota la lunghezza della stringa x).

Sia Σ un alfabeto. Assumiamo che $\triangleright \notin \Sigma$ e $\$ \notin \Sigma$. Tali simboli invece occorrono nell'alfabeto delle k-MdT di cui parleremo. Useremo tali k-MdT per decidere l'appartenenza di un elemento a un dato linguaggio $L \subseteq \Sigma^*$.

DEFINIZIONE 20.5 (Classi in tempo). Un linguaggio $L \subseteq \Sigma^*$ è *deciso* da una k-MdT M se per ogni $x \in \Sigma^*$ vale che:

- Se $x \in L$ allora M con input x termina nello stato yes (in breve $M(x) = yes$).[1]
- Se $x \notin L$ allora M con input x termina nello stato no (in breve $M(x) = no$).

Se $L \subseteq \Sigma^*$ è deciso da una k-MdT M e M opera in tempo $f(n)$, allora $L \in \mathsf{TIME}(f(n))$.

$\mathsf{TIME}(f(n))$ cosí definita è una *classe di complessità in tempo*. Essa rappresenta l'insieme di linguaggi (o, equivalentemente, di insiemi) che possono essere *decisi* (dunque si tratta di insiemi ricorsivi) in tempo limitato da una funzione nota a priori.

Il seguente Teorema mostra che il formalismo ora descritto non permette di descrivere algoritmi sensibilmente più efficienti rispetto alle MdT a un nastro.

TEOREMA 20.6. *Se* M *è una* k*-MdT che opera in tempo* $f(n)$*, possiamo costruire una* 1*-MdT* M' *equivalente e che opera in tempo* $O(nf(n) + f(n)^2)$.

L'idea della dimostrazione è semplicemente quella della simulazione di una computazione su k nastri in un solo nastro. Il contributo $nf(n)$ si può omettere qualora, come spesso accade, $f(n) \geq n$.

Per quanto riguarda la notazione O-grande, date due funzioni $f, g : \mathbb{N} \longrightarrow \mathbb{N}$, si dice che $f(n)$ è $O(g(n))$ se esiste $c > 0$ e $n_0 \in \mathbb{N}$ tali che $(\forall n \geq n_o)(f(n) \leq cg(n))$. In altri termini, che $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq k$ per qualche $k \geq 0$. Dire dunque che una MdT opera in tempo $O(g(n))$ significa che opera in tempo $f(n)$ con $f(n)$ che è $O(g(n))$.

Il seguente Teorema giustifica l'utilizzo della notazione O-grande per la caratterizzazione delle classi di complessità, mostrando come le 'costanti' non contino.

[1] Se vale solo questa condizione allora L è *semi-deciso* da M.

TEOREMA 20.7 (Speed-up). *Sia* $L \in \mathrm{TIME}(f(n))$. *Allora:*

$$\forall \epsilon > 0 : L \in \mathrm{TIME}(\epsilon f(n) + n + 2).$$

L'idea della dimostrazione è la seguente: data una k-MdT M che decide L. Costruiamo una h-MdT (ove $h = k$ se $k \geq 2$, 2 altrimenti) il cui linguaggio codifica m-uple di Σ. In tal modo ad ogni passo di computazione di M' corrispondono m passi di computazione di m. Scegliendo m in funzione di ϵ ($m > \frac{6}{\epsilon}$) si ottiene il risultato.

NOTA 20.8. Come corollario 'pratico' del risultato suddetto, si mostra che un potenziamento dell'HW può permettere solo un miglioramento lineare dei tempi di esecuzione. Algoritmi inefficienti rimangono tali anche se il calcolatore diventa 10, 100, 1000 volte più veloce.

Verificato che il formalismo delle k-MdT e quello delle 1-MdT sono sostanzialmente equivalenti a meno di un fattore polinomiale, può a questo punto sorgere il dubbio che altri formalismi (per esempio, le URM [**6**], le RAM [**23**], i programmi WHILE [**15**]) permettendo di descrivere algoritmi più veloci, rendano la MdT non adatta a ciò. Vale invece una versione *computazionale* della tesi di Church:

> *Tutti i formalismi di calcolo ragionevoli sono computazionalmente equivalenti a meno di fattori polinomiali.*

Ad esempio, nel caso delle RAM, vale il seguente risultato [**23**]:

TEOREMA 20.9. *Sia* $L \in \mathrm{TIME}(f(n))$. *Allora esiste un programma RAM che calcola la funzione caratteristica di* L *in tempo* $O(f(n))$.

Sia P *un programma RAM che calcola la funzione* ϕ *in tempo* $f(n)$. *Allora esiste una 7-MdT che calcola* ϕ *in tempo* $O(f(n)^3)$.

Le dimostrazioni sono, ovviamente, basate su simulazione di un formalismo Turing completo in un altro.

In Figura 1 sono definite le importanti classi P ed $\mathsf{EXPTIME}$ relative a quanto abbiamo ora descritto.

3. Il non determinismo

In questa sezione presenteremo un modello di computazione *non ragionevole* (nel senso della tesi di Church sopra). Il concetto che si vuol catturare è quello del non-determinismo. Diversamente da quanto accade per DFA e NFA in cui sostanzialmente si è provata l'equivalenza tra i formalismi, in questo caso i risultati saranno meno ovvi e alcuni non sono noti.

DEFINIZIONE 20.10. Una MdT $M = \langle Q, \Sigma, q_0, P \rangle$ (a un nastro) è *non-deterministica* (in breve, una ND-MdT) se P è una *relazione*:

$$P \subseteq (Q \times \Sigma) \times ((Q \cup \{h, yes, no\}) \times \Sigma \times \{L, R, S\})$$

$$
\begin{array}{rcl}
\mathsf{P} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{TIME}(n^k)\\
\mathsf{EXPTIME} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{TIME}(2^{n^k})\\
\mathsf{NP} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{NTIME}(n^k)\\
\mathsf{NEXPTIME} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{NTIME}(2^{n^k})\\
\mathsf{L} & \stackrel{\mathrm{def}}{=} & \mathsf{SPACE}(\log n)\\
\mathsf{PSPACE} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{SPACE}(n^k)\\
\mathsf{NL} & \stackrel{\mathrm{def}}{=} & \mathsf{NSPACE}(\log n)\\
\mathsf{NPSPACE} & \stackrel{\mathrm{def}}{=} & \bigcup_{k\geq 0}\mathsf{NSPACE}(n^k)
\end{array}
$$

FIGURE 1. Sommario delle classi di complessità introdotte

Per il resto questo modello eredita le definizioni fornite per il caso deterministico. La differenza sta che i successori a una data configurazione non sono solo 0 (quando è terminante) o 1 (tutti gli altri casi). Una configurazione (q, ℓ, s, r) ha un numero di successori pari all'insieme delle quintuple in P che iniziano con q, s. Data una configurazione iniziale $(q_0, \varepsilon, \rhd, x)$ esiste una famiglia di possibili computazioni non deterministiche. La relazione di successore $\longrightarrow_*$ dunque è una vera e propria relazione simile alla $\overset{G}{\Rightarrow}_*$ usata nelle produzioni possibili per un linguaggio CF.

DEFINIZIONE 20.11. Un linguaggio $L \subseteq \Sigma^*$ è *deciso* da una ND-MdT M se per ogni $x \in \Sigma^*$:

- Se $x \in L$, allora esiste una computazione non deterministica tale che:
$$(q_0, \varepsilon, \rhd, x) \longrightarrow_* (yes, u, s, v)$$
- Se $x \notin L$, allora non esiste una computazione non deterministica tale che:
$$(q_0, \varepsilon, \rhd, x) \longrightarrow_* (yes, u, s, v)$$

Si noti la grande asimmetria tra $x \in L$ e $x \notin L$. Quando $x \notin L$ tutte le computazioni potrebbero addirittura essere non terminanti. Con la sola definizione sopra, il linguaggio L appare solo semideciso.

DEFINIZIONE 20.12. Una ND-MdT M *opera in tempo* $f(n)$ se ogni computazione non deterministica sull'input x ha al più lunghezza $f(|x|)$.

Se una ND-MdT M che opera in tempo $f(n)$ decide un linguaggio $L \in \Sigma^*$, allora siamo in grado anche di stabilire in modo effettivo se $x \notin L$. Più precisamente:

DEFINIZIONE 20.13. Se un linguaggio $L \in \Sigma^*$ è deciso da una ND-MdT M che opera in tempo $f(n)$, allora $L \in \mathsf{NTIME}(f(n))$.

In Figura 1 sono definite le importanti classi non deterministiche NP ed $\mathsf{NEXPTIME}$.

Uno dei problemi aperti più affascinanti dell'informatica teorica è collocare insiemisticamente P e NP. Ovviamente vale:

LEMMA 20.14. $\mathsf{P} \subseteq \mathsf{NP}$.

PROOF. Se $L \in \mathsf{P}$, allora è deciso da una k-MdT deterministica M in tempo $O(n^h)$ per qualche $h \in \mathbb{N}$. Per il Teorema 20.6 sappiamo che esiste una 1-MdT M' che opera in tempo $O(n^{2h})$ equivalente a M. M' non è altro che una ND-MdT nel cui insieme delle istruzioni vi è esattamente una quintupla in P per ogni coppia q, s. □

Non si sa ancora né se $\mathsf{P} = \mathsf{NP}$ né se $\mathsf{P} \neq \mathsf{NP}$. Vale tuttavia il seguente:

TEOREMA 20.15. $\mathsf{NTIME}(f(n)) \subseteq \bigcup_{c\geq 1} \mathsf{TIME}(c^{f(n)})$.

PROOF. Sia $L \in \mathsf{NTIME}(f(n))$ e $M = \langle Q, \Sigma, q_0, P\rangle$ una ND-MdT che decide L. Sia c il massimo grado di non determinismo di M, ovvero

$$c = \max_{\langle q,s\rangle \in Q\times\Sigma} |\{\langle q, s, q', s', M\rangle \in P\}|.$$

Si tiene un vettore di lunghezza $f(n)$ che rappresenta un numero in base c di $f(n)$ caratteri inizializzato a $[0, 0, \dots, 0]$. Ad ogni passo di computazione il numero $i \in \{0, \dots, c-1\}$ indica quale delle scelte non deterministiche debba essere selezionata. Si simula dunque la computazione che effettua, ad ogni passo, le scelte illustrate dal vettore. Se alla fine si raggiunge yes, si accetta x. Altrimenti si incrementa il vettore di 1 (in base c) e si riparte. Se dopo il tentativo con il vettore a $[c-1, c-1, \dots, c-1]$ non si è ancora raggiunto yes, si restituisce no. □

COROLLARIO 20.16. $\mathsf{NP} \subseteq \mathsf{EXPTIME}$.

4. Una inclusione stretta

In letteratura si trovano svariate classi di complessità che permettono di partizionare l'insieme dei linguaggi (insiemi) ricorsivi. Tuttavia, sono molto pochi i risultati di inclusione *stretta* finora dimostrati. Uno di questi sarà presentato in questo paragrafo.

Innanzitutto una premessa. Quali sono le funzioni che si prestano ad essere funzioni *proprie* per il calcolo della complessità? Sicuramente tra queste ci sono $\log n, \sqrt{n}, n, n^2, n^3, 2^n, \dots$. Non sembra accettabile una funzione f del tipo:

$$f(n) = \begin{cases} 2^n & \text{Se } n \text{ è primo} \\ \prod_{i=1}^{n} \pi_i & \text{altrimenti} \end{cases}$$

dove π_i è l'i-esima cifra nell'espansione decimale di π.

Si può formalizzare la nozione di essere una buona funzione per la complessità.

DEFINIZIONE 20.17. f è una funzione *propria* di complessità se è debolmente crescente ($f(n+1) \geq f(n)$) ed esiste una una macchina di Turing a $k > 1$ nastri M tale che:

- non modifica mai il contenuto del primo nastro
- il k-esimo nastro è a sola scrittura e la testina in esso può solo (a) stare ferma o (b) scrivere e spostarsi a destra.
-
$$(q_0, \underbrace{\varepsilon, \rhd, x}_{\text{nastro } 1}, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro } 2}, \ldots, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro } k}) \longrightarrow_t$$
$$(h, \underbrace{\varepsilon, \rhd, x}_{\text{nastro } 1}, \underbrace{\varepsilon, \rhd, 0^{j_2}}_{\text{nastro } 2}, \ldots, \underbrace{\varepsilon, \rhd, 0^{j_{k-1}}}_{\text{nastro } k-1}, \underbrace{\rhd 1^{f(|x|)}, \$, \varepsilon}_{\text{nastro } k})$$
- $t = O(|x| + f(|x|))$ e,
- per $i = 2, \ldots, k-1$, $j_i = O(f(|x|))$.

Dunque una macchina di Turing per il calcolo delle funzioni proprie calcola f in unario e non usa spazio inutile (in particolare è una I/O-k-MdT—si confronti la Def. 20.23).

Sia $f(n) \geq n$ una funzione di complessità propria. Definiamo l'insieme H_f nel modo seguente:

$$H_f \stackrel{\text{def}}{=} \left\{M; x : M \text{ accetta } x \text{ in } \leq f(|x|) + 5|x| + 4 \text{ passi}\right\} \tag{4.1}$$

H_f è una versione 'limitata' dell'*Halting problem.*

TEOREMA 20.18. $H_f \in TIME(f(n)^3)$.

Idea della dimostrazione. Si costruisce una 4-MdT M che decide H_f. Si userà la MdT universale, la macchina che permette lo speed-up per togliere le costanti e la macchina per calcolare f che è propria.

A parte il limite cubico, per il quale è necessario sviluppare la dimostrazione in dettaglio, è abbastanza facile convincersi della possibilità di realizzare una M che decide H_f e che opera in tempo polinomialmente legata a $f(n)$. E questo è quello che veramente importa in questo contesto.

TEOREMA 20.19. $H_f \notin TIME(f(\lfloor \frac{n}{2} \rfloor))$.

PROOF. Supponiamo per assurdo che esista M_{H_f} che decide H_f in tempo $f(\lfloor \frac{n}{2} \rfloor)$. Allora possiamo costruire una macchina di Turing D con la seguente funzionalità:

$$D \stackrel{\text{def}}{=} \textbf{if } M_{H_f}(M; M) = yes$$
$$\textbf{then } no$$
$$\textbf{else } yes$$

Quanti passi impiega D sull'input M? So che

$$M_{H_f}(M; M) = f\left(\left\lfloor \frac{|M; M|}{2} \right\rfloor\right) = f(|M|)$$

D deve, partendo dall'input M, duplicarlo e aggiungere un ; in mezzo. Questo può essere fatto in tempo $5|\mathsf{M}|+4$. Inoltre va modificato P di $\mathsf{M}_{\mathsf{H}_\mathsf{f}}$ per terminare in modo opposto. Dunque

$$\mathsf{D}(\mathsf{M}) \quad \text{impiega tempo} \quad \leq \mathsf{f}(|\mathsf{M}|)+5|\mathsf{M}|+4 \tag{4.2}$$

Cerchiamo dunque di giungere ad una contraddizione:

- Se $\mathsf{D}(\mathsf{D}) = \mathsf{no}$ allora $\mathsf{M}_{\mathsf{H}_\mathsf{f}}(\mathsf{D};\mathsf{D}) = \mathsf{yes}$, ovvero D accetta D in $\leq \mathsf{f}(|\mathsf{D}|)+5|\mathsf{D}|+4$ passi. Dunque $\mathsf{D}(\mathsf{D}) = \mathsf{yes}$: una contraddizione.
- Se $\mathsf{D}(\mathsf{D}) = \mathsf{yes}$ allora $\mathsf{M}_{\mathsf{H}_\mathsf{f}}(\mathsf{D};\mathsf{D}) = \mathsf{no}$, ovvero D su D non termina in $\leq \mathsf{f}(|\mathsf{D}|)+5|\mathsf{D}|+4$ passi. Ma questo contraddice (4.2).

□

Corollario 20.20. Se $\mathsf{f}(n) \geq n$ è una funzione di complessità propria, allora:

$$\mathsf{TIME}(\mathsf{f}(n)) \subset \mathsf{TIME}(\mathsf{f}(2n+1)^3)$$

Corollario 20.21. $\mathsf{P} \subset \mathsf{EXPTIME}$.

Proof.

$$\mathsf{P} \subseteq \mathsf{TIME}(2^n) \subset \mathsf{TIME}((2^n)^3) \subseteq \mathsf{EXPTIME}$$

□

Dunque sappiamo che: $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXPTIME}$ e che almeno una delle due inclusioni è propria. Ma non sappiamo quale, né sappiamo se lo siano entrambe!

Se rimuoviamo l'ipotesi di essere propria, vale il seguente Teorema

Teorema 20.22 (GAP). *Esiste* f *ricorsiva tale che* $\mathsf{TIME}(\mathsf{f}(n)) = \mathsf{TIME}(2^{\mathsf{f}(n)})$.

Idea: si definisce f in modo tale che nessuna MdT su un input di lunghezza n termini in un numero di passi compreso tra $\mathsf{f}(n)$ e $2^{\mathsf{f}(n)}$.

5. Complessità in spazio

Sebbene il tempo di computazione sia considerato la risorsa più importante da tenere bassa nello sviluppo di algoritmi (lo spazio, a differenza del tempo, può essere riutilizzato), la quantità di memoria (nastro) utilizzata durante la computazione ci fornisce una misura interessante e meritevole di essere studiata. Per studiare queste nozioni viene prima definito un ulteriore raffinamento del modello k-MdT che permette di escludere, nel conteggio di tale misura, lo spazio necessario per l'input e l'output (input e output saranno infatti ugualmente presenti in ogni k-MdT che risolve lo stesso problema e limitano dunque le possibilità di discriminazione.

Definizione 20.23. Una macchina di Turing a k nastri e con input e output (*I/0-*k*-MdT*) M è una k-MdT con almeno due nastri. Il primo, che contiene l'input, è un nastro a sola lettura. L'ultimo, che conterrà l'output, è a sola scrittura e la testina in esso può solo (a) stare ferma o (b) scrivere e spostarsi a destra.

Pertanto, se si volesse modificare il nastro che contiene l'input, sarebbe necessario prima ricopiarlo in un altro nastro e modificare quest'ultimo. Si osservi che questo non è un nuovo modello di MdT ma semplicemente una restrizione al formalismo della k-MdT definita in Def. 20.1.

Definizione 20.24 (Classi in spazio deterministico). Data una I/0-k-MdT M e un input x, se

$$(q_o, \underbrace{\varepsilon, \rhd, x}_{\text{nastro 1}}, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro 2}}, \dots, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro k}}) \longrightarrow_* (q, \underbrace{u_1, s_1, v_1}_{\text{nastro 1}}, \underbrace{u_2, s_2, v_2}_{\text{nastro 2}}, \dots, \underbrace{u_k, s_k, v_k}_{\text{nastro k}})$$

con $q \in \{h, yes, no\}$ allora diremo che lo *spazio richiesto da* M *per* x è $\sum_{i=2}^{k-1} |u_i| + |v_i| + 1$.

M *opera in spazio* $f(n)$ se per ogni input x lo spazio richiesto da M per x è minore o uguale a $f(|x|)$.

Se esiste una I/0-k-MdT M che decide un linguaggio $L \subseteq \Sigma^*$ e opera in spazio $f(n)$, allora $L \in \mathsf{SPACE}(f(n))$.

Le classi di spazio deterministiche L e PSPACE sono definite in Figura 1.

Vediamo ora l'importante teorema che collega le classi deterministiche di spazio e tempo:

Teorema 20.25. $\mathsf{SPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$.

Proof. Sia $L \in \mathsf{SPACE}(f(n))$ e sia M la I/O-k-MdT che decide L. Se durante la computazione M si trovasse nella medesima configurazione $(q, \ell_1, s_1, r_1, \dots, \ell_k, s_k, r_k)$ per due volte, il determinismo garantirebbe che la situazione si ripeterebbe all'infinito e dunque M non terminerebbe (assurdo). Il numero di configurazioni diverse (l'ultimo nastro non conta in quanto non viene letto) è dell'ordine:

$$\underbrace{(Q+3)}_{\text{stati}} \quad \underbrace{\Sigma^{O(f(|x|))}}_{\text{stringhe sui nastri}} \quad \underbrace{(|x|+1)f(|x|)^k}_{\text{posizione testina}}$$

□

Corollario 20.26. $\mathsf{L} \subseteq \mathsf{P}$ e $\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$.

Anche nel caso del non-determinismo si possono definire le classi in spazio. Una I/0-k-MdT M è non deterministica se l'insieme P delle istruzioni descrive una relazione anziché una funzione δ. Valgono dunque le stesse definizioni delle ND-MdT e delle I/0-k-MdT.

Definizione 20.27 (Classi in spazio non deterministico). Una ND-I/0-k-MdT M decide $L \subseteq \Sigma^*$ in spazio $f(n)$ se M decide L e per ogni $x \in \Sigma^*$, se

$$(q_o, \underbrace{\varepsilon, \rhd, x}_{\text{nastro 1}}, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro 2}}, \dots, \underbrace{\varepsilon, \rhd, \varepsilon}_{\text{nastro k}}) \longrightarrow_* (q, \underbrace{u_1, s_1, v_1}_{\text{nastro 1}}, \underbrace{u_2, s_2, v_2}_{\text{nastro 2}}, \dots, \underbrace{u_k, s_k, v_k}_{\text{nastro k}})$$

allora $\sum_{i=2}^{k-1} |u_i| + |v_i| + 1 \leq f(|x|)$.

In tal caso si dice che $L \in \mathsf{NSPACE}(f(n))$.

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \underbrace{\mathsf{P} \subseteq \mathsf{NP} \subseteq \begin{array}{c} \mathsf{PSPACE} \\ \mathsf{NPSPACE} \end{array}}_{\subset \ \mathsf{stretta}} \subseteq \mathsf{EXPTIME} \subseteq \mathsf{NEXPTIME}$$

FIGURE 2. Classi in spazio e tempo presentate e loro posizioni reciproche

Si osservi che tale definizione permette che M che decide L in spazio $\mathsf{f}(n)$ sia potenzialmente non terminante! Le classi NL e $\mathsf{NPSPACE}$ sono definite in Figura 1. Ovviamente vale:

TEOREMA 20.28. $\mathsf{SPACE}(\mathsf{f}(n)) \subseteq \mathsf{NSPACE}(\mathsf{f}(n))$

Pertanto si avrà:

COROLLARIO 20.29. $\mathsf{NL} \subseteq \mathsf{L}$ e $\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$.

Tuttavia vale il risultato (di cui si omette del tutto la dimostrazione):

TEOREMA 20.30 (Savitch). *Se* $\mathsf{f}(n) \geq n$ *ed è una funzione propria di complessità, allora* $\mathsf{NSPACE}(\mathsf{f}(n)) \subseteq \mathsf{SPACE}(\mathsf{f}(n)^2)$.

Come corollario, si ottiene il risultato:

COROLLARIO 20.31. $\mathsf{NPSPACE} = \mathsf{PSPACE}$

In realtà un sospetto su questo risultato lo si ha simulando una ND-MdT mediante una MdT deterministica (Teorema 20.15). Prima di iniziare ciascuna delle possibili (esponenziali in numero) simulazioni si può riportare prima le testine nella configurazione iniziale. Si userà così lo stesso spazio richiesto dalla ND-MdT di partenza. Similmente, si può dimostrare:

TEOREMA 20.32. $\mathsf{NP} \subseteq \mathsf{PSPACE}$.

Il riassunto delle inclusioni discusse in questo capitolo si trova in Figura 2.

CHAPTER 21

Riduzioni e NP-completezza

Nel capitolo 19 è stata introdotta la riducibilità funzionale tra insiemi. Ridurre A a B mediante una funzione ricorsiva f ci permetteva di dire che A non poteva essere più difficile di B. In questo capitolo si rafforza la nozione di riduzione aumentando le richieste per la funzione f. f deve essere non solo ricorsiva, ma calcolabile in modo semplice rispetto alla complessità degli insiemi A e B che si vuole correlare. In questo modo potremmo dire che sapendo calcolare B efficientemente, la proprietà positiva ricade su A. Viceversa, che se la decisione di A richiede algoritmi inefficienti, lo stesso varrà per B.

1. Riduzioni tra problemi

DEFINIZIONE 21.1. Dati due linguaggi L_1 e L_2 si dice che $L_1 \subseteq \Sigma_1^*$ *è riducibile a* $L_2 \subseteq \Sigma_2^*$ (in breve, $L_1 \preceq L_2$) se esiste una funzione $f : \Sigma_1^* \longrightarrow \Sigma_2^*$ tale che:

- f è calcolabile da una I/O-k-MdT (deterministica) in spazio $O(\log n)$ e
- per ogni $x \in \Sigma_1^*$ si ha che:

$$x \in L_1 \text{ sse } f(x) \in L_2$$

f è chiamata una *riduzione* da L_1 a L_2.

In letteratura si può trovare la richiesta per f di essere calcolabile polinomialmente. Seguendo [**23**] facciamo una richiesta più forte (si ricorda che: $L \subseteq P$). Qual'è il senso di questa scelta? Similmente a quanto fatto per la riducibilità usando funzioni ricorsive, si vorrebbe avere che $L_2 \in \mathcal{C}$ e $L_1 \preceq L_2$ implica che $L_1 \in \mathcal{C}$. Se ciò è vero, allora $\mathcal{C}$ è detta *chiusa per riduzione.*

Si consideri il seguente esempio:

- $y \in L_2$ si decide in tempo $|y|^2$ (ovvero $L_2 \in P$)
- f riduce L_1 a L_2 nel senso che $x \in L_1$ sse $f(x) \in L_2$, ma
- f si calcola in tempo $2^{|x|}$

Malgrado ci sia la *riduzione* non possiamo usarla per dire qualcosa sulla classe di complessità di L_1. Pertanto la funzione di riduzione deve poter essere calcolata in modo semplice rispetto alla complessità di L_2. La scelta della complessità—spazio $O(\log n)$—permette di usarla per tutte le classi da noi introdotte in Fig. 1: esse sono tutte *chiuse per riduzione.*

Anche per questo tipo di riduzioni si può introdurre il concetto di completezza:

Definizione 21.2. Data una classe $\mathcal{C}$, un problema L è $\mathcal{C}$-*completo* se ogni problema della classe può essere ridotto a L.

Quando si riducono problemi in P o in classi che contengono P, si può tranquillamente usare la richiesta più debole che f sia calcolabile in tempo polinomiale, che alle volte è più semplice da dimostrare.

Vedremo subito un esempio di riduzione:

Definizione 21.3 (SAT). **Input:** Una formula logica proposizionale φ data come congiunzione di disgiunzioni (clausole) di letterali, costruita su un insieme di variabili $\mathcal{V}$, ovvero:

$$\varphi \equiv (\ell_1^1 \vee \cdots \vee \ell_{n_1}^1) \wedge \cdots \wedge (\ell_1^m \vee \cdots \vee \ell_{n_m}^m)$$

ove ogni ℓ_j^i è o una variabile $X \in \mathcal{V}$ o la negazione di una variabile $\neg X$ con $X \in \mathcal{V}$. **Problema:** Esiste un assegnamento per le variabili in $\mathcal{V}$ che renda vera la formula?

SAT sta, ovviamente, per SATisfiability.

Definizione 21.4 (HP). **Input:** Un grafo (non diretto) $G = \langle N, E \rangle$. **Problema:** Esiste un cammino che visita ogni nodo esattamente una volta?

HP sta per Hamilton Path. In entrambi i casi, come visto, la *dimensione* dell'istanza del problema è la lunghezza dela stringa che descrive l'input.

Lemma 21.5. $\mathsf{HP} \preceq \mathsf{SAT}$.

Proof. Dato un grafo $G = \langle N, E \rangle$, costruiremo una formula $f(G)$ su un insieme di variabili $\mathcal{V}$. Assumiamo, per semplicità notazionale, che $N = \{1, \ldots, n\}$.

- Definiamo $\mathcal{V} = \{X_{ij} : 1 \leq i, j \leq n\}$. Informalmente, X_{ij} ha il significato: il nodo j è l'i-esimo nodo del cammino Hamiltoniano (proprietà Booleana).
- Descriviamo ora le varie clausole di $f(G)$ con cui vogliamo modellare il problema del cammino Hamiltoniano:
  - Prima di tutto, ogni nodo j deve stare nel cammino, anche se non sappiamo in quale posizione. Questo può essere espresso come:
  $$\phi_j \stackrel{\text{def}}{=} X_{1j} \vee X_{2j} \vee \cdots \vee X_{nj}$$
  - Ogni nodo j non può apparire sia come i-esimo nodo del cammino che come k-esimo, con $i \neq k$. Questo si può esprimere con la clausola:
  $$\psi_j(i,k) \stackrel{\text{def}}{=} \neg X_{ij} \vee \neg X_{kj}$$
  - Inoltre, per ogni $i = 1, \ldots, n$ vi dev'essere un nodo che sia l'i-esimo del cammino:
  $$\eta_i \stackrel{\text{def}}{=} X_{i1} \vee \cdots \vee X_{in}$$

– Per ogni coppia (i, j) tale che $\{i, j\}$ che non sia in E devo escludere la possibilità che i sia successore di j nel cammino o viceversa. Questo può essere espresso ripetendo, per $k = 1, \ldots, n-1$:[1]

$$\rho_{ij}(k) \stackrel{\text{def}}{=} \neg X_{ki} \vee \neg X_{k+1j}$$

- Possiamo dunque definire $f(G)$:

$$\begin{aligned} f(G) \quad \stackrel{\text{def}}{=} \quad & \textstyle\bigwedge_{j=1,\ldots,n} \phi_j \wedge \bigwedge_{i,j,k=1,\ldots,n:i\neq k} \psi_j(i,k) \wedge \\ & \textstyle\bigwedge_{i=1,\ldots,n} \eta_i \wedge \bigwedge_{i,j,k=1,\ldots,n:k\neq n,\{i,j\}\notin E} \rho_{ij}(k) \end{aligned}$$

Per costruzione, f trasforma un'istanza di HP (ovvero un grafo) in un'istanza di SAT.

(1) Dobbiamo mostrare che f si calcola in spazio logaritmico. L'idea della dimostrazione è la seguente: sostanzialmente si tratta di eseguire dei cicli for (annidati) ma limitati da $n = O(|G|)$. n si rappresenta in spazio logaritmico rispetto al suo valore.
(2) Dobbiamo inoltre mostrare che se π (una permutazione di $\{1, \ldots, n\}$) è un cammino Hamiltoniano in G, allora l'assegnamento che corrisponde a π rende vera la formula $f(G)$. Questo è immediato per costruzione di $f(G)$.
(3) Viceversa, se π non lo è, dobbiamo mostrare che l'assegnamento non rende vera la formula. Equivalentemente, mostriamo che se θ è un assegnamento che rende vera la formula, da esso si desume un cammino Hamiltoniano per G. Ogni assegnamento θ che renda vere le tre prime famiglie di clausole garantisce che per $\forall j \exists! i. X_{ij}$ = **true** e $\forall i \exists! j. X_{ij}$ = **true**. Dunque θ descrive una permutazione. L'ultima famiglia di clausole garantisce che tra nodi successivi nella permutazione vi sia un arco.

□

Avendo ridotto HP a SAT via f, quello che si può dire è che HP non può essere più difficile di SAT. Infatti, per stabilire se $x \in \mathsf{HP}$ effettuo la riduzione (in spazio logaritmico e dunque tempo polinomiale) e mi riduco così al problema di appartenenza a SAT.

2. I Teoremi di Cook

Studieremo ora l'esistenza di problemi completi per la classe P e per la classe NP.

Definizione 21.6 (CIRCUIT VALUE). **Input:** Un circuito logico con porte `and`, `or`, e `not` con una sola uscita e i valori per gli ingressi.
Problema: Stabilire se l'output è `true`.

[1]Per una dimostrazione analoga relativa al cammino Hamiltoniano nei grafi diretti questo è l'unico punto da modificarsi leggermente (Esercizio).

DEFINIZIONE 21.7 (CIRCUIT SAT). **Input:** Un circuito logico con porte `and`, `or`, e `not` con una sola uscita e n porte di ingresso.
Problema: Stabilire se esiste un assegnamento per le porte di ingresso che rende l'output `true`.

TEOREMA 21.8. *CIRCUIT VALUE è* P*-completo.*

PROOF. Innanzitutto va osservato che CIRCUIT VALUE sta in P. Dati i valori degli input, si percorre il circuito (aciclico) valutando gli output di ogni porta in funzione dei suoi input. Ciò può ovviamente essere fatto in tempo polinomiale.

Mostriamo ora che è P-completo, riducendo ogni problema L in P a lui. Sia $L \in \mathsf{P}$. Allora esiste una 1-MdT M che decide $L \subseteq \Sigma^*$ e che opera in tempo $p(n)$ con $p(n)$ un polinomio in n. Dobbiamo mostrare che esiste una f tale che, dato $x \in \Sigma^*$, mi permette di ottenere un circuito $f(x)$ tale che $f(x)$ ha output **true** se e solo se $x \in L$. Il circuito lo si ottiene dalla forma delle computazioni di M. In al più $p(|x|)$ passi M termina su $|x|$. Sia $k = p(|x|) + 1$: al più k celle del nastro possono essere utilizzate. Simuliamo la computazione con una matrice $k \times k$ in cui ogni riga i rappresenta la configurazione dopo l'i-esimo passo. La cella j rappresenta dunque il simbolo presente sul nastro oppure un simbolo che identifica che la testina si trova nello stato q e legge il simbolo s (una sola cella per riga avrà memorizzato questa informazione aggiuntiva). Assumiamo, senza perdita di generalità, che M termini sempre a destra del $\rhd$ iniziale. Inoltre, se M termina prima di $p(|x|)$ passi, si ricopia l'ultima configurazione. Sia $x = a_1 \dots a_n$.

$(\rhd, q_0)$	a_1	a_2	$\dots$	a_n	\$	$\dots$	\$
$\rhd$	(a_1, q_1)	a_2	$\dots$	a_n	\$	$\dots$	\$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\rhd$	$(y_1, yes/no)$	y_2	$\dots$	y_n	y_{n+1}	$\dots$	y_k

Quello che si osserva è che ogni elemento $M_{i,j}$ può essere calcolato a partire dalla matrice di transizione di M e dai valori di sole tre celle: $M_{i-1,j-1}, M_{i-1,j}, M_{i-1,j+1}$. A partire da una codifica binaria dei simboli di Σ e di $\Sigma \times Q$ è facile scrivere le funzioni booleane che calcolano la codifica del contenuto della cella $M_{i,j}$.

Il tutto viene poi composto per ottenere un circuito che simula l'evoluzione della tabella per righe, ovvero la computazione di M. L'output si otterrà dalla cella $M_{k,2}$ con un circuito che restituisce 1 se (y_1, yes), 0 altrimenti. $\square$

TEOREMA 21.9. *CIRCUIT SAT è* NP*-completo.*

PROOF. Bisogna innanzitutto mostrare che CIRCUIT SAT sta in NP. Scrivere una ND-MdT che risolve SAT è semplice. In una prima fase si aprono le strade non deterministiche per assegnare i possibili valori per le variabili di input. A questo punto in ogni strada non deterministica ci troviamo di fronte ad una istanza di CIRCUIT VALUE che è in P.

La dimostrazione di NP-completezza ricalca quella di P-completezza di CIRCUIT VALUE. Sia $\mathsf{L} \in \mathsf{NP}$. Assumiamo senza perdita di generalità che L sia deciso da una ND-k-MdT che opera in tempo $\mathsf{p}(\mathsf{n})$ e tale che il suo grado di non determinismo sia 2. Si ripete il ragionamento sopra solo che in ogni porta costruita si aggiunge un input che tiene conto del grado di non-determinismo. La lista di tali variabili di input è l'input di CIRCUIT SAT. □

TEOREMA 21.10. *SAT è* NP*-completo.*

PROOF. La dimostrazione che $\mathsf{SAT} \in \mathsf{NP}$ è identica a quella per CIRCUIT SAT, vista nel Teorema 21.9.

Per concludere la dimostrazione, riduciamo CIRCUIT SAT a SAT. Dato un circuito G, si tratta di trovare una formula logica in forma clausale equivalente al circuito dato. Grazie ai teoremi di De Morgan, possiamo assumere che il circuito abbia solo porte **or** e **not**. Vi sono tre tipi di nodi; ogni nodo `or` ha 2 archi entranti, ogni nodo `not` ha 1 arco entrante, mentre ogni nodo di input ha 0 archi entranti. Assegnamo una variabile di output ad ogni nodo del circuito.

or: la variabile Y di output rappresenta una funzione $\mathsf{Y} \leftrightarrow \mathsf{A} \vee \mathsf{B}$ ove A e B sono le variabili di output dei due nodi da cui partono gli archi entranti. $\mathsf{Y} \leftrightarrow \mathsf{A} \vee \mathsf{B}$ è equivalente a:

$$(\neg\mathsf{Y} \vee \mathsf{A} \vee \mathsf{B}) \wedge (\neg\mathsf{A} \vee \mathsf{Y}) \wedge (\neg\mathsf{B} \vee \mathsf{Y})$$

not: la variabile Y di output rappresenta una funzione $\mathsf{Y} \leftrightarrow \neg\mathsf{A}$ ove A è la variabile di output del nodo da cui parte l'arco entrante: $\mathsf{Y} \leftrightarrow \neg\mathsf{A}$ è equivalente a:

$$(\neg\mathsf{Y} \vee \mathsf{A}) \wedge (\mathsf{Y} \vee \neg\mathsf{A})$$

input: la variabile di output è la stessa dell'input.

La congiunzione delle formule ottenute è equisoddisfacibile al circuito finale. La riduzione si può fare in spazio $\log \mathsf{n}$ (guardo i nodi uno ad uno, dobbiamo solo scrivere temporaneamente degli indici che vanno da 1 a $|\mathsf{G}|$ e che dunque si rappresenta in spazio $\log |\mathsf{G}|$). □

3. Problemi NP-completi

La gran parte dei problemi pratici che un informatico deve affrontare (problemi su grafi, scheduling, problemi di logica) presenta molte caratteristiche simili a quelle di SAT, che riassumiamo: facilità di verifica di una soluzione; ampiezza esponenziale dello spazio di ricerca della stessa (delle stesse). In base al Teorema 21.10 e al fatto che la classe NP è chiusa per riduzione, è possibile cercare di dimostrare formalmente la NP-completezza di un problema L che ci sembra appartenere a questa famiglia. Per far ciò si tratta di dimostrare che:

(1) $\mathsf{L} \in \mathsf{NP}$; e
(2) $\mathsf{SAT} \preceq \mathsf{L}$.

Una volta mostrato che L è NP-completo, si potrà utilizzarlo, come si è fatto per SAT nel punto (2), per dimostrare la NP-completezza di altri problemi.

Per quanto riguarda l'appartenenza a NP, si può utilizzare la seguente definizione alternativa (ed equivalente): $L \in \mathsf{NP}$ se per ogni $x \in L$ esiste una certificazione 'succinta' (un testimone polinomiale) di questa proprietà che può essere *verificata* in tempo polinomiale. Nel caso di SAT, il testimone è un assegnamento di verità. Nel caso di HP, è proprio il cammino. In entrambi i casi la dimensione del testimone è *succinta*, nel senso che è di dimensioni polinomiali rispetto all'input. La verifica che si tratti di una soluzione è parimenti polinomiale. Ovviamente quello che si cerca di fare è di rendere esplicita la parte di non-determinismo necessaria per la definizione di una MdT che decida non deterministicamente il linguaggio: in modo non deterministico va generato il testimone.

Per quanto riguarda la proprietà di riduzione, bisogna invece avere sufficiente fantasia/esercizio. Questa proprietà (ovvero che esiste un NP-completo riducibile a lui) è detta *NP-hardness*.

Vedremo ora alcune riduzioni famose, alcuni problemi NP-completi fondamentali.

3.1. Varianti di SAT.

Definizione 21.11 (3SAT). **Input:** Istanze di SAT in cui ogni clausola consta di esattamente 3 elementi.
Problema: Come per SAT, stabilire se esiste un assegnamento che rende vera la formula.

Teorema 21.12. *3SAT è NP-completo.*

Proof. L'appartenenza ad NP si mostra in questo caso banalmente in quanto ogni istanza di 3SAT è anche istanza di SAT.

Mostriamo ora che $\mathsf{SAT} \preceq \mathsf{3SAT}$. Sia x input di SAT. Traduciamo ogni clausola $\ell_1 \vee \cdots \vee \ell_m$ di x in una equisoddisfacibile clausola di 3SAT.

- Se $m = 1$: la trasformo in $\ell_1 \vee \ell_1 \vee \ell_1$
- Se $m = 2$: la trasformo in $\ell_1 \vee \ell_2 \vee \ell_2$
- Se $m = 3$ non devo fare nulla.
- Se $m > 3$, introduco una nuova variabile X: restituisco la clausola $\ell_1 \vee \ell_2 \vee X$ e riapplico la riscrittura alla clausola $\neg X \vee \ell_3 \vee \cdots \vee \ell_m$.

□

Definizione 21.13 (NAESAT). **Input:** Tutti gli input di SAT.
Problema: Stabilire se esiste un assegnamento delle variabili di input che renda vera la formula e tale che in ogni clausola non tutti i letterali sono a **true**.

In altri termini, non ci vanno bene gli assegnamenti che mettono tutti i letterali di almeno una clausola a **false** o tutti a **true**: da cui il nome Not All Equal.

Teorema 21.14. *NAESAT è NP-completo.*

PROOF. Come per SAT, l'appartenenza a NP si può mostrare usando la definizione Alternativa fornita in questa sezione. Fornito l'input e un assegnamento, la verifica di 'bontà' di quell'assegnamento è chiaramente polinomiale.

Per mostrare la NP-hardness in modo semplice, basta modificare appena la dimostrazione del Teorema 21.8. Sia Z un'unica variabile aggiuntiva. $Y \leftrightarrow A \vee B$ viene trasformata in:

$$(\neg Y \vee A \vee B) \wedge (\neg A \vee Y \vee Z) \wedge (\neg B \vee Y \vee Z)$$

mentre $Y \leftrightarrow \neg A$ viene trasformata in:

$$(\neg Y \vee A \vee Z) \wedge (Y \vee \neg A \vee Z)$$

Sia σ che soddisfa l'istanza di CIRCUIT SAT. Consideriamo la traduzione non estesa con Z.

Supponiamo, per assurdo, che nella prima clausola dell'**or** tutti i tre letterali siano **true**. Allora la seconda e la terza non sarebbero soddisfatte. Dunque σ non rende veri tutti e tre i letterali delle prime clausole dell'**or**. Ma allora con $\sigma' = \sigma \circ [Z/\textbf{false}]$ si ha una soluzione per NAESAT.

Viceversa, sia σ una soluzione per NAESAT. E' immediato verificare che σ è una soluzione sse $\bar{\sigma}$ lo è. In una delle due Z sarà **true**: quella sarà la soluzione per CIRCUIT SAT. $\square$

La versione più estrema di SAT, ove ogni clausola ha esattamente 2 letterali e nota come 2SAT, è invece polinomiale. Si provi per esercizio questa proprietà.

3.2. Problemi su Grafi. Dato un grafo non diretto $G = \langle N, E \rangle$ e un insieme $I \subseteq N$, si dice che I è un *insieme indipendente (Independent Set) per* G se per ogni $i, j \in I$ non vi è in E l'arco $\{i, j\}$.

DEFINIZIONE 21.15 (IS). **Input:** Un grafo non diretto $G = \langle N, E \rangle$ e un intero k.
Problema: Stabilire se esiste un insieme indipendente I per G tale che $|I| = k$.

TEOREMA 21.16. *IS è NP-completo.*

PROOF. L'appartenenza a NP è immediata.
Mostriamo che $3SAT \preceq IS$. La dimostrazione è semplice. Tuttavia, per non renderla troppo complicata da apici e pedici annidati, mostriamo la riduzione su una istanza. Sia

$$\varphi = (X_1 \vee X_2 \vee \neg X_3) \wedge (X_1 \vee \neg X_2 \vee X_4) \wedge (\neg X_1 \vee X_4 \vee X_5) \wedge (\neg X_1 \vee \neg X_4 \vee \neg X_5)$$

Per ogni clausola costruisco un sottografo completo triangolare A ogni nodo associo (intuitivamente) un letterale (ad esempio, per il primo X_1, X_2, e $\neg X_3$. Avremo tanti triangoli quante sono le clausole in gioco (in questo caso 4), come raffigurato in Fig. 1. Si aggiungano poi degli archi tra tutti (e solo tra quelli) i nodi che rappresentano letterali complementari. Ebbene, φ è soddisfacibile se e solo se il grafo ottenuto ammette un insieme indipendente di 4 (in generale, il numero delle clausole)

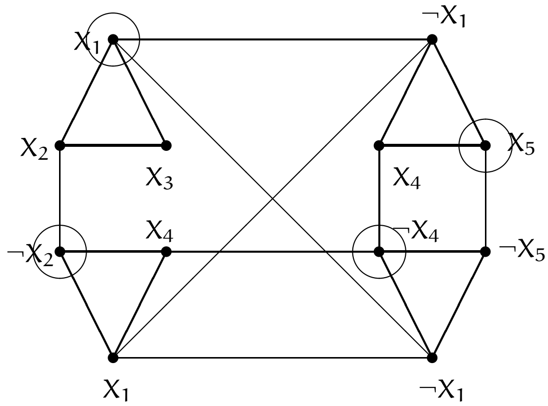


Figure 1. Grafo usato per la riduzione $3SAT \preceq IS$

elementi. I nodi cerchiati in figura costituiscono un insieme indipendente di 4 nodi e forniscono una soluzione per φ: $X_1 = \textbf{true}, X_2 = \textbf{false}, X_4 = \textbf{false}, X_5 = \textbf{true}$. Il valore di X_3 non ha importanza. □

Elenchiamo altri problemi NP-completi molto noti.

Definizione 21.17 (CLIQUE). **Input:** Un grafo non diretto $G = \langle N, E \rangle$ e un intero k.
Problema: Stabilire se esiste un sottografo di G di k elementi che sia un grafo completo.

Si osservi che CLIQUE su $G = \langle N, E \rangle$ corrisponde a IS su $G' = \langle N, (N \times N) \setminus E \rangle$

Definizione 21.18 (NODE COVER). **Input:** Un grafo non diretto $G = \langle N, E \rangle$ e un intero k.
Problema: Stabilire se esiste un insieme di nodi $C \subseteq N$ di k elementi tale che per ogni arco $\{i, j\}$ di E $i \in C$ o $j \in C$.

Si osservi che I è un IS di $G = \langle N, E \rangle$ se e solo se $C = N \setminus I$ è un NODE COVER di G.

Esercizio 21.19. Si formalizzi la dimostrazione di NP-completezza di CLIQUE e NODE COVER.

Anche il problema HP già discusso in precedenza è NP-completo (difficile riduzione da 3SAT—omessa). Vediamo ora la versione decisionale del noto problema del commesso viaggiatore (Traveling Salesman Problem).

Definizione 21.20 ($\mathsf{TSP}(k)$). **Input:** Un grafo non diretto $G = \langle N, E\rangle$, una funzione di costi $c : N \longrightarrow \mathbb{N}$, e un intero k.
Problema: Stabilire se esiste un circuito che passa esattamente una volta per ogni nodo e di *costo complessivo* $\leq k$.

Esercizio 21.21. Assumendo la NP-completezza di HP, si formalizzi la dimostrazione di NP-completezza di $\mathsf{TSP}(k)$.

Definizione 21.22 (k-COLORING). **Input:** Un grafo non diretto $G = \langle N, E\rangle$ e un intero k.
Problema: Stabilire se esiste un assegnamento $\mathtt{col} : \mathbb{N} \longrightarrow \{1, \dots, k\}$ tale che per ogni arco $\{i, j\}$ di E $\mathtt{col}(i) \neq \mathtt{col}(j)$.

Per $k = 2$ il problema sta in P (verificare per esercizio). Per $k = 4$ e grafi planari, il problema ha sempre soluzione. Per grafi qualunque, invece:

Teorema 21.23. *3-coloring è NP-completo.*

Proof. L'appartenenza a NP è immediata.

Riduciamo NAESAT a 3-coloring. Come nella dimostrazione del Teorema 21.16, mostriamo la riduzione su un esempio. Sia

$$\varphi = (X_1 \vee X_2 \vee \neg X_3) \wedge (X_1 \vee \neg X_2 \vee \neg X_3) \wedge (\neg X_1 \vee X_2 \vee \neg X_3) \wedge (\neg X_1 \vee \neg X_2 \vee X_3)$$

Costruiamo un grafo nel seguente nodo. Sia μ un nodo. Per ogni variabile vengono introdotti due nodi (uno per X_i uno per $\neg X_i$). Si inseriscono archi per avere i triangoli $\mu, X_i, \neg X_i$. Il nodo μ è comune a tutti i triangoli.

Per ogni clausola vengono introdotti tre nodi C_1, C_2, C_3 ed archi tra loro per ottenere un triangolo per ogni clausola. Ogni nodo è associato ad uno dei letterali della clausola.

Si aggiungono dunque degli archi tra tutti (e solo tra quelli) i nodi dei triangoli sopra i nodi relativi ai letterali corrispondenti.

Si mostra che σ è una soluzione di φ, allora la funzione che mette il colore 0 ai nodi dei triangoli variabile corrispondenti a **false**, il colore 1 a quelli corrispondenti a **true** e il colore 2 al nodo μ e a un nodo per ogni triangolo clausola è un 3-coloring. Viceversa, se non esiste un 3-coloring, non esiste un assegnamento di verità per σ. □

Esercizio 21.24. Si consideri il seguente problema $\mathsf{ST}(k)$: *Siano dati un grafo non diretto* $G = \langle N, E\rangle$ *e un intero* $k \in \mathbb{N}$. *Si vuol sapere se esiste uno SPANNING TREE per* G *tale che nessun nodo dell'albero abbia grado maggiore di* k. In altri termini, se esiste $E' \subseteq E$ t.c.:

- $|E'| = |N| - 1$,
- il grafo $\langle N, E'\rangle$ è connesso, e
- nessun nodo di N è incluso in più di k archi di E'.

Si mostri che $\mathsf{ST}(k)$ è NP-completo.

Esercizio 21.25. Si dimostri la NP-completezza del *problema del cruciverba*, ovvero, dato un insieme di parole $p_1, \dots, p_k$ in un dato alfabeto A (N.B. potrebbe essere un alfabeto con più delle 26 lettere che usiamo di solito!) e uno schema di cruciverba C (una matrice rettangolare di caselle bianche e nere) con le caselle nere già fissate, stabilire se esiste un modo per scrivere le k parole nello schema, con le regole di incrocio standard dei cruciverba, che riempia esattamente tutte le caselle bianche.

Per l'esercizio sopra può far comodo conoscere ed utilizzare il seguente problema, che è NP-completo:
Exact Cover by 3 sets: Dato un insieme $U = \{1, \dots, 3 \cdot m\}$, e dati n insiemi $S_1, \dots, S_n \subseteq U$, con $|S_1| = |S_2| = \cdots = |S_n| = 3$ stabilire se esistono m insiemi (disgiunti) tra $S_1, \dots, S_n$ tali che la loro unione sia U.

Esercizio 21.26. Si consideri il problema che chiamiamo ZSAT:
Input: Una congiunzione di clausole $C_1 \wedge \cdots \wedge C_n$ ove ogni C_i è una disgiunzione di esattamente i letterali (non necessariamente diversi tra loro).
Problema: Stabilire se esiste un assegnamento di verità per le variabili che compaiono nella formula che rende l'istanza vera.
Si dimostri che ZSAT è NP-completo.

Esercizio 21.27. Si consideri il problema della fila tra detenuti (FTD): vi sono n detenuti ed è nota una relazione x *odia* y tra coppie di detenuti. Il problema è: esiste un modo per mettere in fila gli n detenuti in modo tale che non vi sia mai un detenuto con uno che lo odia alle spalle?
Si mostri che il problema FTD è NP-completo.

Bibliography

[1] P. Aczel. *Non-well-founded sets*, volume 14 of *CSLI Lecture Notes*. Stanford University Press, 1988.

[2] M. Aiello, A. Albano, G. Attardi, and U. Montanari. *Teoria della Computabilità, logica, teoria dei linguaggi formali*. Materiali didattici ETS, Pisa, 1976.

[3] H. P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier North-Holland, 1984.

[4] A. Church. A note on the Entscheidungsproblem. *J. of Symbolic Logic*, 1:40–41, 1936.

[5] S. A. Cook. The Complexity of Theorem Proving Procedures. In *Proceedings of ACM Symposium on the Theory of Computing*, pages 151–158. ACM Press, 1971.

[6] N. J. Cutland. *An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[7] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 1973. 2nd printing.

[8] G. Frege. —. In J. Van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard Univ Press, 2001.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[10] K. Gödel. —. In Solomon Feferman, Stephen C. Kleene, John W. Dawson, and Robert M. Solovay, editors, *Kurt Gödel Collected Works: Publications 1929-1936*, volume 1. Oxford University Press, 1999.

[11] C. Hankin. *Lambda Calculi: A Guide for Computer Scientists*, volume 3 of *Graduate Texts in Computer Science*. Springer-Verlag, 1996.

[12] M. Hirvensalo. *Quantum Computing*. Springer-Verlag, 2001.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[14] N. D. Jones and Y. E. Lien. New problems complete for nondeterministic log space. *Mathematical Systems Theory*, 10:1–17, 1976.

[15] N.D. Jones. *Computability and Complexity*. MIT Press, 1997.

[16] W. Just and M. Weese. *Discovering modern set theory. I: The basics*, volume 8 of *Graduate studies in mathematics*. American mathematical society, 1996.

[17] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1996.

[18] K. Kunen. *Set Theory. An Introduction to Independence Proofs*. Studies in Logic. North Holland, Amsterdam, 1980.

[19] G. Longo. *Metodi per il trattamento dell'informazione*. Servizio Editoriale Universitario di Pisa, 1984.

[20] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, Princeton, N. J., 1979.

[21] J. C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, 1996.

[22] P. Odifreddi. *Classical Recursion Theory*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1989.

[23] C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[24] G. Paun, G. Rozenberg, A. Salomaa, and W. Brauer. *DNA Computing: New Computing Paradigms*. Texts in Theoretical Computer Science. Springer-Verlag, 1998.

[25] R. Péter. *Rekursive funktionen*. Akadémiai Kiadó, Budapest, 1953.

[26] G. Plotkin. A structural approach to operational semantics. DAIMI-19 Aarhus University, Denmark, 1981.

[27] H. J. Rogers. *Theory of Recursive Functions and Effective Computability*. The MIT Press, 1988.

[28] J.R. Shoenfield. Axioms of set theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 321–344. North-Holland, Amsterdam, 1977.

[29] M. Sipser. The History and Status of the P versus NP Question. In *Proceedings of ACM Symposium on the Theory of Computing*, pages 603–618. ACM Press, 1992.

[30] A. M. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Math. Society*, 42(2):230–265, 1936-7.

[31] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.