Elementi di Calcolabilità e Complessità

Luca De Paulis

30 settembre 2021

Indice

1

Introduzione alla Calcolabilità

Nella prima parte del corso, dedicata alla **Teoria della Calcolabilità**, cercheremo di studiare cosa significhi *calcolare* qualcosa e quali siano i limiti delle *procedure* a disposizione degli esseri umani per calcolare.

Per far ciò bisogna innanzitutto definire il concetto di **algoritmo** oppure procedura: lo faremo definendo dei vincoli che ogni algoritmo deve soddisfare per esser ritenuto tale.

1. Dato che gli uomini possono calcolare solo seguendo procedure finite, un algoritmo deve essere **finito**, ovvero deve essere costituito da un numero finito di istruzioni.
2. Inoltre devono esserci un numero **finito** di istruzioni distinte, e ognuna deve avere un **effetto limitato** su **dati discreti** (nel senso di non continui).
3. Una **computazione** è quindi una sequenza finita di passi discreti con durata finita, né analogici né continui.
4. Ogni passo dipende solo dai **passi precedenti** e viene scelto in modo **deterministico**: se ripetiamo due volte la stessa esatta computazione nelle stesse condizioni dobbiamo ottenere lo stesso risultato e la stessa sequenza di passi.
5. Non imponiamo un limite al numero di passi e alla memoria a disposizione.

Questi vincoli non definiscono precisamente cosa sia un algoritmo, anzi, vedremo che vi sono diversi modelli di computazione che soddisfano questi 5 requisiti. Le domande a cui vogliamo rispondere sono:

- Modelli diversi che rispettano questi vincoli risolvono gli stessi problemi?
- Un tale modello risolve necessariamente tutti i problemi?

1.1 MACCHINE DI TURING

Il primo modello di computazione che vedremo è stato proposto da Alan Turing nel 1936, ed è pertanto chiamato in suo nome.

Definizione 1.1.1 – Macchina di Turing

Una **Macchina di Turing** (MdT per gli amici) è una quadrupla (Q, Σ, δ, q_0) dove

- Q è un insieme finito, detto **insieme degli stati**. In particolare assumiamo che esista uno stato $h \notin Q$, detto stato terminatore o **halting state**.
- Σ è un insieme finito, detto **insieme dei simboli**. In particolare

  - esiste $\# \in \Sigma$ e lo chiameremo **simbolo vuoto**;
  - esiste $\triangleright \in \Sigma$ e lo chiameremo **respingente**.
- δ è una funzione

$$\delta : Q \times \Sigma \to (Q \cup \{ h \}) \times \Sigma \times \{ L, R, - \}$$

  detta **funzione di transizione**. È soggetta al vincolo

$$\forall q \in Q : \exists q' \in Q : \quad \delta(q, \triangleright) = (q', \triangleright, R).$$

- q_0 è un elemento di Q detto **stato iniziale**.

La definizione formale di Macchina di Turing può sembrare complicata, ma l'idea alla base è molto semplice: abbiamo una macchina che opera su un **nastro illimitato** (a destra) su cui sono scritti simboli (ovvero elementi di Σ). In ogni istante di tempo, la *testa* della macchina legge una casella del nastro, contenente il **simbolo corrente**. La macchina mantiene inoltre al suo interno uno stato (ovvero un elemento di Q), inizialmente settato allo stato iniziale q_0.

Un singolo passo di computazione è il seguente:

- la macchina legge il simbolo corrente σ;
- la macchina usa la funzione di transizione δ per effettuare la mossa: in particolare calcola $\delta(q, \sigma)$, dove q è lo stato corrente, e ne ottiene una tripla (q', σ', M);
- la macchina cambia stato da q a q';
- la macchina scrive al posto di σ il simbolo σ';
- la macchina si sposta nella direzione indicata da M: se $M = L$ si sposta di un posto a sinistra, se $M = R$ si sposta di un posto a destra, se $M = -$ rimane ferma.

Per formalizzare questi concetti abbiamo bisogno di altre definizioni.

Definizione 1.1.2 – Monoide libero, o Parole su un Alfabeto

Dato un insieme finito Σ, il **monoide libero** su Σ, anche chiamato **insieme delle parole su** Σ, è l'insieme Σ^* così definito:

$$\Sigma^* := \bigcup_{n \in \mathbb{N}} \Sigma^n$$

dove

- $\Sigma^0 := \{ \varepsilon \}$, dove ε è la parola vuota;
- $\Sigma^{n+1} := \{ \sigma \cdot w : \sigma \in \Sigma, w \in \Sigma^n \}$ è l'insieme delle parole di lunghezza $n + 1$, ottenute preponendo ad una parola di lunghezza n (ovvero $w \in \Sigma^n$) un simbolo $\sigma \in \Sigma$.

Tale insieme ammette un'operazione, ovvero la **concatenazione** di parole, e la parola vuota ε è l'identità destra e sinistra di tale operazione.

Osservazione 1.1.1. Un elemento di Σ^* è una stringa di caratteri di Σ di lunghezza arbitraria, ma sempre finita, in quanto ogni elemento di Σ^* deve essere contenuto in un qualche Σ^n.

Il nastro di una MdT può quindi essere formalizzato come un elemento di Σ^*. Questo tuttavia ancora non ci soddisfa per alcuni motivi:

- gli elementi di Σ^* sono illimitati a destra, ma non a sinistra, dunque la MdT potrebbe muoversi a sinistra ripetutamente fino a "cadere fuori dal nastro";
- non stiamo memorizzando da alcuna parte la posizione del cursore della MdT.

Per risolvere il primo problema possiamo assumere che ogni nastro inizi con il simbolo speciale $\triangleright$: per il vincolo sulla funzione di transizione ogni volta che la MdT si troverà nella casella più a sinistra (contenente $\triangleright$) sarà costretta a muoversi verso destra lasciando scritto il respingente.

Per quanto riguarda il secondo invece possiamo dividere il nastro infinito in tre parti:

- la porzione a sinistra del simbolo corrente, che è una stringa di lunghezza arbitraria che inizia per $\triangleright$ e quindi un elemento di $\triangleright\Sigma^*$;
- il simbolo corrente, che è un elemento di Σ;
- la porzione a destra del simbolo corrente, che è una stringa e quindi un elemento di Σ^*.

Quest'ultima porzione è una stringa che potrebbe terminare con un numero infinito di caratteri vuoti (#): dato che non siamo interessati (per il momento) a tenere tutti i simboli vuoti a destra dell'ultimo simbolo non-vuoto del nastro, considereremo la porzione a destra "eliminando" tutti i *blank* superflui.

In particolare indicando sempre con $\varepsilon \in \Sigma^*$ la stringa vuota e convenendo che

- $\#\varepsilon = \varepsilon\# = \varepsilon$ (la concatenazione della stringa vuota con il *blank* dà ancora la stringa vuota);
- $\sigma\varepsilon = \varepsilon\sigma = \sigma$ per ogni $\sigma \neq \#$ (la concatenazione della stringa vuota con un simbolo non-*blank* dà il simbolo)

possiamo considerare l'insieme

$$\Sigma^F := \Big(\Sigma^* \cdot (\Sigma \setminus \{\#\})\Big) \cup \{\varepsilon\},$$

ovvero l'insieme delle stringhe in Σ che finiscono con un carattere non-*blank*, più la stringa vuota.

Usando queste convenzioni, la stringa che definisce il nastro è finita: siamo pronti a definire la *configurazione* di una MdT in un dato istante.

Definizione 1.1.3 – Configurazione di una MdT

Sia $M = (Q, \Sigma, \delta, q_0)$ una MdT. Una **configurazione** è una quadrupla

$$(q, u, \sigma, v) \in Q \times \triangleright\Sigma^* \times \Sigma \times \Sigma^F.$$

Più nel dettaglio:

- q è lo stato corrente,
- u è la porzione del nastro che precede il simbolo corrente, ed inizia per $\triangleright$,
- σ è il simbolo corrente,
- v è la porzione del nastro che segue il simbolo corrente, ed è vuota oppure termina per un simbolo diverso da #.

Osserviamo che:

- il simbolo corrente può essere #;

- è possibile che il simbolo corrente sia # e $v = \varepsilon$ (cioè vuota),
- è possibile che u sia vuota solo nel caso in cui il simbolo corrente è $\triangleright$, poiché significherebbe trovarsi all'inizio del nastro.

Spesso indicheremo la quadrupla (q, u, σ, v) con $(q, u\underline{\sigma}v)$: la sottolineatura ci indicherà il simbolo corrente. In contesti in cui non sia necessario sapere la posizione del cursore scriveremo semplicemente (q, w) per risparmiare tempo.

Esempio 1.1.4. Ad esempio la configurazione

$$(q_0, \triangleright ab\#\#b\#a\underline{\#}b\#a)$$

indica che la MdT è nello stato q_0, sta leggendo il carattere #, a sinistra del simbolo letto ha la stringa $\triangleright ab\#\#b\#a$ e a destra $b\#a$.

1.2 LINGUAGGI FOR E WHILE

Introduciamo ora un secondo paradigma per il calcolo di algoritmi, ovvero quello dato dai linguaggi `FOR` e `WHILE`. In effetti anche se le MdT rispondono ai nostri requisiti formali per un algoritmo e sono il modello teorico delle **macchine di Von Neumann**, al giorno d'oggi non costruiamo una nuova macchina per ogni algoritmo che dobbiamo risolvere: usiamo dei **linguaggi di programmazione** che verranno interpretati o compilati e restituiranno il risultato del calcolo.

I linguaggi `FOR` e `WHILE` sono quindi la base teorica dei moderni linguaggi imperativi, e anche se sembrano mancare di espressività rispetto ad essi, vedremo che in realtà il linguaggio `WHILE` riesce a risolvere tutti e soli i problemi risolvibili da un linguaggio moderno.

Sintassi astratta

Definizione 1.2.1 – Sintassi astratta di `FOR` e `WHILE`

$$\begin{aligned}
\texttt{EXPR} &::= n \mid x \mid E_1 + E_2 \mid E_1 \cdot E_2 \mid E_1 - E_2 && \text{Espr. aritmetiche}\\
\texttt{BEXPR} &::= b \mid E_1 < E_2 \mid \neg b \mid B_1 \vee B_2 && \text{Espr. booleane}\\
\texttt{CMD} &::= \texttt{skip} \mid x := E \mid C_1; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 && \text{Comandi}\\
&\quad \mid \texttt{for } x = E_1 \texttt{ to } E_2 \texttt{ do } C \mid \texttt{while } B \texttt{ do } C
\end{aligned}$$

dove $n \in \mathbb{N}$, $x \in \texttt{Var}$ (che è un insieme *numerabile* di variabili), $b \in \mathbb{B} := \{\mathcal{T}, \mathcal{F}\}$.

Il linguaggio `FOR` contiene solo il comando `for`, il linguaggio `WHILE` contiene solo il comando `while`.

Semantica

Per definire la **semantica** dei linguaggi `FOR` e `WHILE` abbiamo bisogno di alcuni costrutti ausiliari. In particolare ogni nostro programma conterrà delle variabili che possono essere valutate oppure aggiornate (tramite il comando di assegnamento): dobbiamo *memorizzare* il loro valore.

Definizione 1.2.2 – Funzione memoria e funzione di aggiornamento

La funzione **memoria** è una funzione

$$\sigma : \texttt{Var} \to \mathbb{N}$$

definita solo per un sottoinsieme finito di `Var`.

La funzione di **aggiornamento** è una funzione

$$-[-/-] : (\texttt{Var} \times \mathbb{N}) \times \mathbb{N} \times \texttt{Var} \to (\texttt{Var} \times \mathbb{N})$$

definita da

$$\sigma[n/x](y) := \begin{cases} n & \text{se } y = x, \\ \sigma(y) & \text{altrimenti.} \end{cases}$$

Osservazione 1.2.1. La funzione di aggiornamento prende una memoria ($\sigma : \texttt{Var} \to \mathbb{N}$), un valore intero ($n \in \mathbb{N}$) e una variabile ($x \in \texttt{Var}$) e produce una nuova memoria $\sigma[n/x] : \texttt{Var} \to \mathbb{N}$ che si comporta come σ su tutte le variabili diverse da x, ma restituisce n quando l'input è x.

Tramite la memoria possiamo definire la funzione di valutazione delle espressioni aritmetiche, ovvero la loro **semantica**.

Definizione 1.2.3 – Funzione di valutazione semantica (aritmetica)

La **funzione di valutazione semantica (aritmetica)** è una funzione

$$\mathcal{E}[\![-]\!]- : \texttt{EXPR} \times (\texttt{Var} \to \mathbb{N}) \to \mathbb{N}$$

definita per induzione strutturale a partire da

$$\begin{array}{lcll} \mathcal{E}[\![n]\!]\sigma & := & n & \text{(val. dei naturali)} \\ \mathcal{E}[\![x]\!]\sigma & := & \sigma(x) & \text{(val. delle variabili)} \\ \mathcal{E}[\![E_1 + E_2]\!]\sigma & := & \mathcal{E}[\![E_1]\!]\sigma + \mathcal{E}[\![E_2]\!]\sigma & \text{(val. della somma)} \\ \mathcal{E}[\![E_1 \cdot E_2]\!]\sigma & := & \mathcal{E}[\![E_1]\!]\sigma \cdot \mathcal{E}[\![E_2]\!]\sigma & \text{(val. del prodotto)} \\ \mathcal{E}[\![E_1 - E_2]\!]\sigma & := & \mathcal{E}[\![E_1]\!]\sigma - \mathcal{E}[\![E_2]\!]\sigma & \text{(val. della sottrazione)} \end{array}$$

Dato che il nostro linguaggio modella solo numeri naturali (quindi positivi), l'operazione di sottrazione sarà quella data dal **meno limitato**:

$$a - b := \begin{cases} a - b, \text{se } a > b \\ 0, \text{altrimenti.} \end{cases}$$

Analogamente possiamo definire la semantica delle espressioni booleane.

Definizione 1.2.4 – Funzione di valutazione semantica (booleana)

La **funzione di valutazione semantica (booleana)** è una funzione

$$\mathcal{B}[\![-]\!]- : \texttt{BEXPR} \times (\texttt{Var} \to \mathbb{N}) \to \mathbb{N}$$

definita per induzione strutturale a partire da

$$\begin{array}{lcll} \mathcal{B}[\![t]\!]\sigma & := & \mathcal{T} & \text{(val. del } \texttt{true}\text{)} \\ \mathcal{B}[\![f]\!]\sigma & := & \mathcal{F} & \text{(val. del } \texttt{false}\text{)} \\ \mathcal{B}[\![E_1 < E_2]\!]\sigma & := & \mathcal{E}[\![E_1]\!]\sigma < \mathcal{E}[\![E_2]\!]\sigma & \text{(val. del minore)} \\ \mathcal{B}[\![\neg B]\!]\sigma & := & \neg\mathcal{B}[\![B]\!]\sigma & \text{(val. del } \texttt{not}\text{)} \\ \mathcal{B}[\![B_1 \vee B_2]\!]\sigma & := & \mathcal{B}[\![B_1]\!]\sigma \vee \mathcal{B}[\![B_2]\!]\sigma & \text{(val. della sottrazione)} \end{array}$$

Osserviamo che i simboli usati nel linguaggio (come $+, <, \neg$, ed altri) sono solo **simboli formali**: per essere più precisi dovremmo differenziarli dalle funzioni effettive (ovvero quelle che compaiono a destra del $:=$).

Osservazione 1.2.2. Le funzioni $\mathcal{E}$ e $\mathcal{B}$ si comportano come un **interprete**: ad esempio $\mathcal{E}$ prende un'espressione aritmetica, una memoria e restituisce la valutazione dell'espressione nella memoria data.

Tuttavia tramite il **currying** possiamo esprimere $\mathcal{E}$ come una funzione

$$\mathcal{E}[\![-]\!]- : \texttt{EXPR} \to \Big((\texttt{Var} \to \mathbb{N}) \to \mathbb{N}\Big)$$

ovvero come una funzione che prende un'espressione aritmetica e restituisce una *funzione* che a sua volta prenderà una memoria per restituire finalmente la valutazione dell'espressione nella memoria.

Anche se le due modalità in pratica ci portano allo stesso risultato, la seconda modella più l'azione di un **compilatore**: infatti nella seconda versione $\mathcal{E}$ prende un'espressione, cioè del codice, e restituisce un *eseguibile* che avrà bisogno dei dati (cioè della memoria) per dare il suo risultato.

Lo stile usato per definire la semantica delle espressioni viene chiamato **semantica denotazionale**: in questo stile cerchiamo di associare ad ogni costrutto del linguaggio una funzione che ne dà la semantica (ad esempio abbiamo associato al + del linguaggio la funzione che somma due naturali).

Per quanto riguarda i comandi adopereremo un altro stile, detto **semantica operazionale**. Come si evince dal nome, cercheremo di definire una *macchina astratta* che modifica il proprio *stato interno* valutando a piccoli passi il comando da eseguire.

Definizione 1.2.5 – Sistema di transizioni

Si dice **sistema di transizioni** una coppia $(\Gamma, \to)$ dove

- Γ è l'insieme delle **configurazioni** oppure stati;
- $\to : \Gamma \to \Gamma$ è una funzione, detta **funzione di transizione**.

Nel caso della nostra macchina astratta, le configurazioni saranno delle coppie

$$\langle c, \sigma \rangle \in \texttt{CMD} \times (\texttt{Var} \to \mathbb{N})$$

ovvero delle coppie "comando da valutare", "memoria".

Per definire la semantica operazionale dei comandi useremo un approccio **small-step**, in cui ogni transizione

$$\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$$

rappresenta un singolo passo dell'esecuzione del programma. Una **computazione** diventa allora una sequenza di passi, ovvero un elemento della chiusura transitiva e riflessiva di $\to$, che indicheremo come al solito come $\to^*$.

Analogamente alle MdT, una computazione **termina con successo** se

$$\langle c, \sigma \rangle \to^* \sigma',$$

ovvero se esauriamo la valutazione del comando c in un numero finito (anche se arbitrario) di passi.

La semantica operazionale dei comandi è dunque data attraverso una serie di assiomi e regole di inferenza, che insieme ci permettono di valutare ogni comando per induzione strutturale.

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma} \qquad \text{Assioma dello } \texttt{skip}$$

$$\frac{}{\langle x := E, \sigma\rangle \to \sigma[n/x]} \quad \text{se } \mathcal{E}[\![E]\!]\sigma = n \qquad \text{Assioma dell'assegnamento}$$

$$\frac{\langle C_1, \sigma\rangle \to \langle C_1', \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \to \langle C_1'; C_2, \sigma'\rangle} \qquad \text{Regola della sequenza 1}$$

$$\frac{\langle C_1, \sigma\rangle \to \langle C_2, \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \to \sigma'} \qquad \text{Regola della sequenza 2}$$

$$\frac{}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_1, \sigma\rangle} \quad \text{se } \mathcal{B}[\![B]\!]\sigma = \mathcal{T} \qquad \text{Assioma cond. 1}$$

$$\frac{}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_2, \sigma\rangle} \quad \text{se } \mathcal{B}[\![B]\!]\sigma = \mathcal{F} \qquad \text{Assioma cond. 2}$$

$$\frac{}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \langle i := n; C; \texttt{for } i = n_1 + 1 \texttt{ to } n_2 \texttt{ do } C, \sigma\rangle} \qquad \text{Assioma del } \texttt{for } 1$$
$$\text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = \mathcal{F}, \mathcal{E}[\![E_1]\!]\sigma = n_1, \mathcal{E}[\![E_2]\!]\sigma = n_2$$

$$\frac{}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \sigma} \quad \text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = \mathcal{T} \qquad \text{Assioma del } \texttt{for } 2$$

$$\frac{}{\langle \texttt{while } B \texttt{ do } C, \sigma\rangle \to \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, \sigma\rangle} \qquad \text{Assioma del } \texttt{while}$$

Dalla regola di transizione del `for` segue una proprietà fondamentale del linguaggio `FOR`: ogni suo programma termina in tempo finito. Infatti prima della prima iterazione la semantica ci impone di valutare le espressioni E_1 ed E_2, che verranno valutate a dei naturali n_1, n_2. A questo punto il ciclo `for` verrà eseguito esattamente $n_2 - n_1$ volte (dove il $-$ è sempre limitato, per cui se $n_1 > n_2$ non eseguiremo mai il ciclo) in quanto gli estremi di iterazione non possono essere modificati dai comandi del corpo del `for`.

Questo ci dimostra immediatamente che il linguaggio `FOR` **non è equivalente** alle macchine di Turing, ovvero esistono macchine di Turing che codificano algoritmi non risolvibili dal linguaggio `FOR`. (Studieremo in seguito cosa significa *codificare algoritmi*.)

Il linguaggio `WHILE` invece può codificare algoritmi che non terminano. Ad esempio si vede subito che la configurazione

$$\langle \texttt{while } \mathcal{T} \texttt{ do skip}, \sigma\rangle$$

diverge a prescindere da σ.

1.3 CALCOLABILITÀ DI FUNZIONI

Dopo aver definito le computazioni per le MdT e per i linguaggi `FOR` e `WHILE`, vogliamo spiegare cosa significa che una MdT o un comando *calcola* una funzione.

Funzioni

Prima di tutto, ricordiamo le definizioni di base sulle funzioni.

Definizione 1.3.1 – Funzione

Dati A, B insiemi, una **funzione** f da A in B è un sottoinsieme di $A \times B$ tale che

$$(a, b), (a, b') \in f \implies b = b'.$$

Scriveremo

- $f : A \to B$ per indicare una funzione da A in B
- $b = f(a)$ per dire $(a, b) \in f$.

Notiamo inoltre che non abbiamo fatto assunzioni sulla **totalità** di f: per qualche valore di $a \in A$ potrebbe non esistere un valore $b \in B$ tale che $f(a)$, cioè f potrebbe *non essere definita* in a.

Definizione 1.3.2 – Funzioni totali e parziali

Sia $f : A \to B$.

- f **converge su** $a \in A$ (e lo si indica con $f(a)\downarrow$) se esiste $b \in B$ tale che $f(a) = b$;
- f **diverge su** $a \in A$ (e lo si indica con $f(a)\uparrow$) se f non converge su a;
- f è **totale** se $f(a)\downarrow$ per ogni $a \in A$;
- f è **parziale** se non è totale.

In generale le nostre funzioni saranno parziali.

Definizione 1.3.3 – Dominio ed immagine

Sia $f : A \to B$. Si dice **dominio** di f l'insieme

$$\operatorname{dom} f := \{ a \in A \ : \ f(a)\downarrow \}.$$

Si dice **immagine** di f l'insieme

$$\operatorname{Im} f := \left\{ b \in B \ : \ b = f(a) \text{ per qualche} a \in A \right\}.$$

Definizione 1.3.4 – Iniettività/surgettività/bigettività

Sia $f : A \to B$ una funzione.

- f è **iniettiva** se per ogni $a, a' \in A$, $a \neq a'$, allora $f(a) \neq f(a')$.
- f è **surgettiva** se $\operatorname{Im} f = B$.
- f è **bigettiva** se è iniettiva e surgettiva.

Calcolare funzioni

Definiamo ora quando una macchina/un comando *implementa* una funzione.

Definizione 1.3.5 – Turing-calcolabilità

Siano $\Sigma, \Sigma_0, \Sigma_1$ alfabeti, $\triangleright, \# \notin \Sigma_0 \cup \Sigma_1 \subsetneq \Sigma$.
Sia inoltre $f : \Sigma_0 \to \Sigma_1$, $M = (Q, \Sigma, \delta, q_0)$ una MdT.
Si dice allora che M **calcola** f (e che f è **Turing-calcolabile**) se per ogni $v \in \Sigma_0$

$$w = f(v) \ \text{ se e solo se } \ (q_0, \underline{\triangleright} v) \to^* (h, \triangleright w \underline{\#}).$$

Indicando con $M(v)$ il risultato della computazione della macchina M sulla configurazione iniziale $(q_0, \underline{\triangleright} v)$, questa definizione ci dice che gli output della funzione e della MdT sono

esattamente gli stessi. In particolare, dato che le funzioni possono essere *parziali* e le macchine di Turing possono *divergere*, $M(v)\downarrow$ se e solo se f è definita su v, cioè se esiste w tale che $f(v) = w$.

Definizione 1.3.6 – `WHILE`-calcolabilità

Sia C un comando `WHILE`, $g : \texttt{Var} \to \mathbb{N}$. Si dice allora che C **calcola** g (e che g è **`WHILE`-calcolabile**) se per ogni $\sigma : \texttt{Var} \to \mathbb{N}$

$$n = g(x) \text{ se e solo se } (C, \sigma) \to^* \sigma^* \text{ e } \sigma^*(x) = n.$$

Analogamente possiamo definire il concetto di funzione `FOR`-calcolabile.

È vero che le funzioni `WHILE`-calcolabili sono tutte e sole le funzioni `FOR`-calcolabili? **No**: infatti dato che non abbiamo fatto assunzioni sulla totalità di g, essa può essere `WHILE`-calcolabile ma non `FOR`-calcolabile.

Un possibile problema nella definizione di calcolabilità data è che abbiamo supposto che le funzioni abbiano una specifica forma: sono funzioni $\Sigma_0^* \to \Sigma_1^*$ nel caso delle MdT, $\texttt{Var} \to \mathbb{N}$ nel caso dei comandi. Scegliendo altri insiemi con le stesse caratteristiche (quindi di cardinalità numerabile) cambiano le funzioni calcolabili?

Fortunatamente la risposta è **no**. Consideriamo una funzione $f : A \to B$: se gli insiemi A, B sono numerabili possiamo scegliere delle **codifiche** $A \to \mathbb{N}$, $\mathbb{N} \to B$. A questo punto possiamo

- trasformare l'input $a \in A$ in un naturale tramite la prima codifica;
- fare il calcolo tramite una funzione $\mathbb{N} \to \mathbb{N}$,
- trasformare l'output in un elemento di B tramite la seconda codifica.

In questo modo possiamo limitarci a solo funzioni $\mathbb{N} \to \mathbb{N}$, a patto che la codifica sia **effettiva**, cioè sia calcolabile anch'essa. Vedremo in seguito che esistono codifiche per rappresentare macchine di Turing come numeri.

1.4 FUNZIONI RICORSIVE

Introduciamo ora un ultimo formalismo per rappresentare un modello di calcolo, ovvero quello delle funzioni ricorsive.

Per semplificare la notazione useremo la λ-notazione per le funzioni anonime: la funzione $\lambda x.\, f(x)$ è la funzione che prende un unico parametro di ingresso x e restituisce $f(x)$.

Definizione 1.4.1 – Funzioni primitive ricorsive

La classe delle **funzioni primitive ricorsive** $\mathcal{PR}$ è la minima classe di funzioni che contenga gli schemi

I. **Zero:** $\lambda x_1, \ldots, x_n.\, 0$ per ogni $n \in \mathbb{N}$

II. **Successore:** $\lambda x.\, x + 1$

III. **Proiezione:** $\lambda x_1, \ldots, x_n.\, x_i$ per ogni $n \in \mathbb{N}$, $i = 1, \ldots, n$

e che sia chiusa per gli schemi

IV. **Composizione:** se $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$, $h : \mathbb{N}^k \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

$$\lambda x_1, \ldots, x_n.\, h\Big(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)\Big)$$

appartiene ancora a $\mathcal{PR}$;

V. **Ricorsione Primitiva:** se $h : \mathbb{N}^{n+1} \to \mathbb{N}$, $g : \mathbb{N}^{n-1} \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

$$f : \mathbb{N}^n \to \mathbb{N}$$

$$\begin{cases} f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n) \\ f(n+1, x_2, \dots, x_n) = h(n, f(n, x_2, \dots, x_n), x_2, \dots, x_n) \end{cases}$$

appartiene ancora a $\mathcal{PR}$.

Una funzione è quindi primitiva ricorsiva se può essere costruita dagli schemi della definizione, ovvero se esiste una successione di funzioni $f_1, \dots, f_n$ tale che

- $f_n = f$,
- per ogni $i = 1, \dots, n$, f_i è definita come uno dei casi base (ovvero è la funzione zero, successore o proiezione) oppure è ottenuta dagli schemi induttivi (composizione e ricorsione primitiva) e *solamente* dalle funzioni $f_1, \dots, f_{i-1}$.

Osserviamo che il calcolo di ogni funzione ricorsiva primitiva *termina sempre*: infatti i casi base (I, II e III) terminano in un singolo passo e per induzione strutturale si vede che anche i casi IV e V terminano in tempo finito, poiché la ricorsione si fa sempre diminuendo il valore di x_1 nello schema V.

Facciamo degli esempi di funzioni ricorsive primitive.

Esempio 1.4.2. Consideriamo la seguente sequenza di funzioni:

$$f_1 = \lambda x.\, x, \qquad f_2 = \lambda x.\, x + 1, \qquad f_3 = \lambda x_1, x_2, x_3.\, x_2,$$

$$f_4 = f_2(f_3(x_1, x_2, x_3)), \qquad f_5 = \begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(n+1, x_2) = f_4(n, f_5(n, x_2), x_2) \end{cases}$$

Ognuna delle funzioni è primitiva ricorsiva: infatti

1. la prima corrisponde alla proiezione per $i = n = 1$;
2. la seconda corrisponde al successore;
3. la terza corrisponde alla proiezione con $n = 3, i = 2$;
4. la quarta è la composizione di f_2 ed f_3;
5. l'ultima è ottenuta per ricorsione primitiva dalle prime quattro funzioni.

Possiamo porci il problema di come si effettui il calcolo di una funzione primitiva ricorsiva quando ci vengono forniti degli argomenti.

Consideriamo due **regole di valutazione**.

Call by value

Nel **call by value** si valutano prima gli operandi di una funzione e poi la funzione esterna: la **redex** (ovvero la prossima espressione da valutare) è la funzione più interna più a sinistra.

$$\begin{aligned}
\underline{f_5(2,3)} &= f_4(1, \underline{f_5(1,3)}, 3) \\
&= f_4(1, f_4(0, \underline{f_5(0,3)}, 3), 3) \\
&= f_4(1, f_4(0, \underline{f_1(3)}, 3), 3) \\
&= f_4(1, \underline{f_4(0,3,3)}, 3) \\
&= f_4(1, f_2(\underline{f_3(0,3,3)}), 3) \\
&= f_4(1, \underline{f_2(3)}, 3) \\
&= \underline{f_4(1,4,3)} \\
&= f_2(\underline{f_3(1,4,3)}) \\
&= \underline{f_2(4)} \\
&= 5.
\end{aligned}$$

▸ CALL BY NEED

Nel **call by need** valutiamo un'espressione *solo quando è necessario*: questa regola ci impone di valutare sempre l'espressione più esterna per prima. Per far ciò eviteremo di essere eccessivamente pignoli nella sintassi delle funzioni primitive ricorsive.

$$\begin{aligned}
\underline{f_5(2,3)} &= \underline{f_4(1, f_5(1,3), 3)} \\
&= \underline{f_2(f_3(1, f_5(1,3), 3))} \\
&= \underline{f_3(1, f_5(1,3), 3)} + 1 \\
&= \underline{f_5(1,3)} + 1 \\
&= \underline{f_4(0, f_5(0,3), 3)} + 1 \\
&= \underline{f_2(f_3(0, f_5(0,3), 3))} + 1 \\
&= \underline{f_3(0, f_5(0,3), 3)} + 1 + 1 \\
&= \underline{f_5(0,3)} + 1 + 1 \\
&= \underline{f_1(3)} + 1 + 1 \\
&= 3 + 1 + 1 = 5.
\end{aligned}$$

In entrambi i casi è chiaro che la funzione f_5 così definita è la somma.

1.4.1 Enumerazione di Gödel

Vogliamo ora mostrare che le macchine di Turing sono numerabili e esiste una loro enumerazione fatta da funzioni ricorsive primitive.

Definizione 1.4.3 – Relazione primitiva ricorsiva

Una relazione $P \subseteq \mathbb{N}^k$ è detta **primitiva ricorsiva** se lo è la sua funzione caratteristica $\chi_P : \mathbb{N}^k \to \{0,1\}$, definita da

$$\chi_P(x_1, \ldots, x_k) := \begin{cases} 1 & \text{se } (x_1, \ldots, x_k) \in P \\ 0 & \text{altrimenti.} \end{cases}$$

La funzione caratteristica di un sottoinsieme di $\mathbb{N}^k$ è quindi una funzione a valori booleani (ovvero è un predicato) che restituisce 1 sugli elementi che appartengono al sottoinsieme e 0

altrimenti. In futuro confonderemo senza porci troppi problemi le relazioni e le loro funzioni caratteristiche.

Per costruire l'enumerazione delle MdT ci serve un predicato molto importante, che è il predicato $P : \mathbb{N} \to \{0, 1\}$ definito da

$$P(p) = 1 \text{ se e solo se } p \text{ è un numero primo.}$$

È possibile dimostrare che tale predicato è primitivo ricorsivo: i numeri primi saranno quindi una parte fondamentale dell'enumerazione.

Gli altri due ingredienti della ricetta sono i seguenti.

- Il **Teorema di Esistenza e Unicità della Fattorizzazione in Primi**, che dice che se $P = p_0, p_1, \ldots$ è l'insieme dei numeri primi (considerato questa volta come relazione unaria!) allora per ogni $n \in \mathbb{N}$ esiste un numero finito di esponenti $e_i \neq 0$ tali che
$$n = \prod_{i \in \mathbb{N}} p_i^{e_i}.$$
- La funzione che prende un $n = \prod_{i \in \mathbb{N}} p_i^{e_i}$ e restituisce l'esponente e_k del k-esimo fattore della fattorizzazione è primitiva ricorsiva.

In particolare se considerando una sequenza finita di naturali $(n_0, \ldots, n_k)$ essa può essere codificata in modo *unico* come il numero

$$N := p_0^{n_0+1} \cdot p_k^{n_k+1}$$

e dalla codifica possiamo ricavare la sequenza originale grazie alle funzioni che danno gli esponenti della fattorizzazione.

Possiamo vedere finalmente una *quasi*-codifica delle macchine di Turing: in particolare delineeremo una funzione iniettiva che mappa le macchine di Turing ai naturali. La codifica vera e propria, chiamata **enumerazione di Gödel**, è in realtà una funzione bigettiva e primitiva ricorsiva.

Data una macchina $M := (Q, \Sigma, \delta, q_0)$, siccome Q e Σ sono finiti possiamo numerare i loro elementi:

$$Q = \{ q_0, \ldots, q_n \}, \qquad \Sigma = \{ \sigma_0, \ldots, \sigma_m \}.$$

Ogni transizione specificata da δ può essere rappresentata come una quintupla

$$(q_i, \sigma_j, q_k, \sigma_l, D) \in Q \times \Sigma \times Q \times \Sigma \times \{ L, R, - \}$$

e pertanto possiamo codificarla come il naturale

$$p_0^{i+1} \cdot p_1^{j+1} \cdot p_2^{k+1} \cdot p_3^{l+1} \cdot p_4^{m_D}.$$

Dobbiamo risolvere alcune piccole inesattezze:

- q_k potrebbe essere lo stato terminatore h, dunque numereremo h come lo stato q_{k+1};
- non abbiamo definito come numerare i simboli $L, R, -$, ma possiamo semplicemente considerarli come ulteriori simboli rispetto a quelli di Σ, dunque numereremo
$$L \mapsto \sigma_{m+2}, \quad R \mapsto \sigma_{m+3}, \quad - \mapsto \sigma_{m+4}.$$

Abbiamo quindi associato un numero ad ogni quintupla. Ma una macchina di Turing è semplicemente un insieme di quintuple: ordiniamole lessicograficamente (ovvero prima le ordiniamo per i, poi per j, ecc) e quindi abbiamo una sequenza finita di numeri naturali $g_0, \ldots, g_r$, ognuno dei quali codifica una quintupla.

Definiremo quindi **numero di Gödel** della macchina M il numero $N_M := p_0^{g_0+1} \cdots p_r^{g_r+1}$. Questa pseudo-codifica è sicuramente iniettiva grazie al Teorema di Fattorizzazione Unica,

ma non è detto che sia surgettiva: l'idea della vera enumerazione di Gödel è simile ma la realizzazione è molto più complessa.

Osserviamo che una volta numerate le quintuple possiamo anche enumerare intere computazioni con un singolo numero: infatti una computazione (terminante) è una successione finita di configurazioni $\gamma = (q_i, w_j) \in Q \times \Sigma^*$ e le configurazioni possono certamente essere enumerate.

Infine ogni passaggio fatto finora è primitivo ricorsivo, dunque il procedimento di enumerazione delle MdT e delle computazioni è primitivo ricorsivo.

1.4.2 Funzione di Ackermann

La maggior parte delle funzioni che usiamo comunemente è primitiva ricorsiva: i logici di inizio '900 si chiesero perciò se le funzioni primitive ricorsive potessero modellare *tutti* gli algoritmi che *terminano sempre*, ovvero tutte le funzioni totali.

La risposta purtroppo è **no** e ci è data dalla **funzione di Ackermann**, definita da

$$A(z,x,y) := \begin{cases} A(0,0,y) & = y \\ A(0,x+1,y) & = A(0,x,y)+1 \\ A(1,0,y) & = 0 \\ A(z+2,0,y) & = 1 \\ A(z+1,x+1,y) & = A(z,A(z+1,x,y),y). \end{cases}$$

Questa funzione è totale: basta mostrarlo per induzione sui casi ricorsivi, che sono il secondo e il quinto. Il primo termina in tempo finito poiché il secondo parametro decresce ad ogni applicazione; il quinto termina poiché l'applicazione interna avviene col secondo parametro diminuito di uno, mentre nell'applicazione esterna il primo parametro decresce.

Tuttavia l'ultimo caso non può essere espresso in termini di funzioni ricorsive primitive: la doppia ricorsione innestata non rientra nei cinque schemi e non è neanche ricavabile dagli schemi. Inoltre questa funzione è *intuitivamente calcolabile*: ad ogni passo restituiamo un risultato oppure calcoliamo ricorsivamente la funzione diminuendo il valore degli argomenti, quindi è certamente scrivere un programma che la calcola.

Segue che le funzioni primitive ricorsive **non sono tutte le funzioni calcolabili totali**.

1.4.3 Non esiste un formalismo capace di esprimere tutte e sole le funzioni calcolabili totali

La speranza è quindi che esista un formalismo, diverso dalle funzioni primitive ricorsive, capace di esprimere *tutte e sole* le funzioni calcolabili totali.

Teorema 1.4.4

Non esiste un formalismo capace di esprimere tutte e soli le funzioni calcolabili totali.

Dimostrazione. Supponiamo per assurdo che esista un formalismo $\mathcal{F}$ capace di esprimere tutte e sole le funzioni calcolabili totali. Sicuramente $|\mathcal{F}| = |\mathbb{N}|$: infatti sono al più numerabili poiché ogni funzione calcolabile è un algoritmo, e quindi è una stringa su un alfabeto finito; d'altro canto sono almeno $|\mathbb{N}|$ poiché le funzioni costanti sono certamente calcolabili totali.

Consideriamo allora una numerazione $\mathbb{N} \to \mathcal{F}$, $n \mapsto f_n$ che sia *bigettiva e calcolabile*.[a]
Costruiamo la funzione $g : \mathbb{N} \to \mathbb{N}$, $g(n) = f_n(n) + 1$. Tale funzione è

- calcolabile, in quanto è la composizione della numerazione di $\mathcal{F}$ che è calcolabile, del calcolo di $f_n(n)$ (che si può fare in quanto f_n è calcolabile) e del successore;
- totale, in quanto composizione di funzioni totali.

Segue che g è una funzione calcolabile totale, e quindi $g \in \mathcal{F}$.

Tuttavia per ogni $n \in \mathbb{N}$ vale che $g(n) = f_n(n) + 1 \neq f_n(n)$, e dunque in particolare $g \neq f_n$ per ogni $n \in \mathbb{N}$. Ma le funzioni di $\mathcal{F}$ sono tutte e sole della forma f_n per qualche $n \in \mathbb{N}$, da cui segue che $g \notin \mathcal{F}$, che è assurdo. □

[a] Dovremmo dimostrare che ne esiste una, ma faremo finta di niente.

La tecnica usata nella dimostrazione di questo teorema viene chiamata **diagonalizzazione** ed è una tecnica usata molto frequentemente nella logica e nella teoria della calcolabilità.

1.5 FUNZIONI GENERALI RICORSIVE

Il problema messo in luce dal Teorema 1.4.4 ci impedisce di creare un formalismo che esprima solo funzioni totali: per risolverlo dobbiamo estendere la classe di funzioni di nostro interesse alle *funzioni parziali*. Questo non è assurdo: molte funzioni di nostro interesse (anche aritmetiche) sono non definite su alcuni input, e abbiamo anche visto che esistono MdT e funzioni `WHILE` che non convergono.

Per farlo, introduciamo un ultimo formalismo.

Definizione 1.5.1 – Operatore di minimizzazione

Dato un predicato P, ovvero una funzione $P : \mathbb{N} \to \{0, 1\}$, possiamo costruire l'**operatore di minimizzazione** μ tale che

$$\mu y.\, P(y) \coloneqq \min\{\, y \;:\; P(y) = 1 \,\}.$$

Osserviamo che $\mu y.\, P(y)$ potrebbe non essere definito: se P è il predicato sempre falso, non esiste un minimo y che lo renda vero.

Definizione 1.5.2 – Funzioni generali ricorsive

L'insieme delle funzioni **generali ricorsive** $\mathcal{R}$ è il minimo insieme di funzioni contenenti gli schemi I, II, III delle funzioni primitive ricorsive, chiuso per gli schemi IV e V e per lo schema

VI. **Minimizzazione:** se $\varphi : \mathbb{N}^{n+1} \to \mathbb{N} \in \mathcal{R}$, allora la funzione $\psi : \mathbb{N}^n \to \mathbb{N}$ definita da

$$\psi(x_1, \ldots, x_n) \coloneqq \mu y.\, \Big(\varphi(\overline{x}, y) = 0 \text{ e } \big(\forall z \leq y.\, \varphi(\overline{x}, z)\!\downarrow\big)\Big)$$

appartiene ancora a $\mathcal{R}$.

Una funzione ottenuta per minimizzazione è intuitivamente calcolabile: si calcola $\varphi(\overline{x}, y)$ per $y = 0, 1, 2, \ldots$ e ci si ferma al primo valore che soddisfi entrambe le proprietà. Osserviamo che una ψ ottenuta in tale modo *può essere parziale* in due casi:

- se $\varphi(\overline{x}, y) \neq 0$ per ogni y allora non c'è minimo, e quindi la computazione non termina;
- se $\varphi(\overline{x}, z)\!\uparrow$ per qualche valore z precedente al primo zero, nel calcolare $\varphi(\overline{x}, z)$ non ci fermeremo mai, e quindi il calcolo di ψ non termina.

Esempio 1.5.3. Sia $\varphi \coloneqq \lambda x, y.\, 3$. φ è primitiva ricorsiva, e quindi è anche totale, ma ψ ottenuta per minimizzazione su φ è **sempre indefinita**.

Infatti $\psi(x)$ è il minimo y per cui $\varphi(x, y) = 0$ (e l'altra condizione) ma questo non accade mai.

1.5.1 Tesi di Church-Turing

Abbiamo quindi visto diversi formalismi capaci di esprimere funzioni intuitivamente calcolabili, ovvero algoritmi. In che relazione sono?

Negli anni '20 e '30 i principali ideatori di questi formalismi (come Church, Turing, Gödel) dimostrarono l'equivalenza di tutti i formalisti che abbiamo visto: in particolare Turing e Church congetturarono che tutti i modelli capaci di esprimere funzioni calcolabili fossero **Turing-equivalenti**.

Teorema 1.5.4 – Tesi di Church-Turing

Le funzioni (intuitivamente) calcolabili sono tutte e sole le funzioni Turing-calcolabili.

Dato che le funzioni Turing-calcolabili sono tutte e sole quelle `WHILE`-calcolabili oppure tutte e sole le funzioni generali ricorsive, da questo momento in poi non specificheremo più il formalismo usato (tanto sono tutti equivalenti!) e parleremo semplicemente di **funzioni calcolabili**.

1.6 PRIMI TEOREMI SULLE FUNZIONI CALCOLABILI

Avendo stabilito il *framework* in cui lavoreremo, possiamo finalmente iniziare a dimostrare alcune proprietà delle funzioni calcolabili.

Teorema 1.6.1 – Esistenza di funzioni non calcolabili

Le funzioni calcolabili sono in quantità numerabile. In particolare esistono funzioni non calcolabili.

Per dimostrare questo teorema useremo il fatto che le funzioni $\mathbb{N} \to \mathbb{N}$ sono in quantità più che numerabile, quindi dimostriamolo.

Teorema 1.6.2 – Diagonalizzazione di Cantor

Siano A, B insiemi con $|A| \geq |\mathbb{N}|$, $|B| \geq 2$. Allora

$$\left|\{ f : A \to B \}\right| > |\mathbb{N}|.$$

Dimostrazione. È sufficiente dimostrare il Teorema nel caso in cui A sia numerabile e B abbia esattamente due elementi. Allora senza perdita di generalità sia $A = \mathbb{N}$, $B = \{0, 1\}$.

Sia $\mathcal{F} := \left|\{ f : \mathbb{N} \to \{0, 1\} \}\right|$ e supponiamo per assurdo $|\mathcal{F}| = |\mathbb{N}|$. Allora esiste una numerazione bigettiva degli elementi di $\mathcal{F}$, ovvero una funzione $n \mapsto f_n \in \mathcal{F}$.

Consideriamo allora $g : \mathbb{N} \to \{0, 1\}$ (e quindi $g \in \mathcal{F}$) definita da

$$g(n) := \begin{cases} 0 & \text{se } f_n(n) = 1, \\ 1 & \text{altrimenti.} \end{cases}$$

Ma allora $g(n) \neq f_n(n)$ per ogni n, dunque $g \neq f_n$ per ogni n. Ma gli elementi di $\mathcal{F}$ sono tutti e soli della forma f_n al variare di $n \in \mathbb{N}$, dunque $g \notin \mathcal{F}$, che è assurdo. □

Possiamo dimostrare il Teorema 1.6.1.

Dimostrazione del Teorema 1.6.1. Sia $\mathcal{C}$ l'insieme delle funzioni calcolabili. Mostriamo che $|\mathcal{C}| \geq |\mathbb{N}|$ e $|\mathcal{C}| \leq |\mathbb{N}|$.

▸ $|\mathcal{C}| \geq |\mathbb{N}|$ Le funzioni $\lambda x.\, n$ al variare di $n \in \mathbb{N}$ sono in quantità numerabile e sono tutte calcolabili.

▸ $|\mathcal{C}| \leq |\mathbb{N}|$ Ogni funzione calcolabile è calcolata da una macchina di Turing, dunque $|\mathcal{C}| \leq |\mathcal{M}|$ dove $\mathcal{M}$ è l'insieme delle MdT; inoltre $|\mathcal{M}| \leq |\mathbb{N}|$ poiché possiamo numerarle tramite l'enumerazione di Gödel, dunque $|\mathcal{C}| \leq |\mathbb{N}|$.

Segue che $|\mathcal{C}| = |\mathbb{N}|$.

Per dimostrare che esistono funzioni non calcolabili osserviamo che $\mathcal{C}$ è un sottoinsieme di tutte le funzioni $\mathbb{N} \to \mathbb{N}$. Tuttavia per il Teorema 1.6.2 le funzioni $\mathbb{N} \to \mathbb{N}$ sono in quantità più che numerabile, dunque $\mathcal{C}$ deve essere un sottoinsieme proprio delle funzioni $\mathbb{N} \to \mathbb{N}$: in particolare devono esistere funzioni che non sono in $\mathcal{C}$. □

1.6.1 Enumerazioni effettive

Dato che abbiamo dimostrato che le funzioni calcolabili sono numerabili possiamo porci il problema di come *enumerarle*. Per far ciò non enumereremo direttamente le funzioni, ma solo le macchine di Turing.

In particolare considereremo sono **enumerazioni effettive**, ovvero enumerazioni che dipendono solo dalla **sintassi** della macchina di Turing, cioè soltanto dai simboli che compaiono al suo interno, e non dal comportamento. [1]

Si può dimostrare che tutti i teoremi che vedremo sono indipendenti dal formalismo e dall'enumerazione scelta, purché quest'ultima sia effettiva: fissiamo quindi una enumerazione effettiva $n \mapsto M_n$. Tale enumerazione induce un'enumerazione sulle funzioni calcolabili: indicheremo con φ_i la funzione calcolata dalla macchina M_i.

Osserviamo però che dati due indici i, j diversi sicuramente vale che $M_i \neq M_j$ ma è possibile che $\varphi_i = \varphi_j$, ovvero è possibile che due macchine diverse calcolino la stessa funzione.

In realtà vale un teorema molto più forte, solitamente conosciuto come **Padding Lemma**.

Teorema 1.6.3 – Padding Lemma

Per ogni indice i esiste un insieme numerabile A_i di indici tale che per ogni $j \in A_i$

$$\varphi_j = \varphi_i,$$

ovvero ogni funzione calcolabile è calcolata da infinite MdT.

Dimostrazione. Consideriamo l'algoritmo che calcola φ_i: aggiungendo uno `skip` alla fine si ottiene un algoritmo diverso (e quindi una macchina di Turing diversa) che calcola la stessa funzione. Aggiungendo altri `skip` otteniamo una quantità numerabile di algoritmi che calcolano φ_i. □

1.6.2 Equivalenza tra MdT e funzioni generali ricorsive

Ora mostriamo che ogni funzione calcolabile può essere scritta in una forma standard, detta **Forma Normale di Kleene**.

Teorema 1.6.4 – Forma Normale di Kleene

Esistono un predicato $T : \mathbb{N}^3 \to \{0, 1\}$ (detto **predicato di Kleene**) e una funzione $U : \mathbb{N} \to \mathbb{N}$ entrambi calcolabili totali tali che

$$\forall i, x : \varphi_i(x) = U\big(\mu y.\, T(i, x, y)\big).$$

Inoltre T e U sono primitivi ricorsivi.

[1] In realtà una enumerazione si dice effettiva se è ottenuta post-componendo l'enumerazione di Gödel con una bigezione $\mathbb{N} \leftrightarrow \mathbb{N}$.

Dimostrazione. Definiamo $T(i, x, y) = 1$ se e solo se la *computazione* $M_i(x)$ converge ad una configurazione $(h, \triangleright z)$ e y è la codifica di tale computazione.

T è calcolabile: in effetti basta recuperare la macchina M_i dalla lista (che è un'operazione calcolabile), decodificare y come una computazione $c_1 \dots c_n$ (dove tutte queste sono configurazioni) e verificare che $M_i(x)$ converga effettivamente ad una configurazione $c_n = (h, \triangleright z)$ tramite la computazione codificata da y.[a]

Allora possiamo definire U in modo che $U(y) =$ "la codifica di z"++. La computazione del membro destro procede quindi in questo modo: si scorrono i valori di y e si trova il primo valore di y che corrisponde alla computazione di $M_i(x)$. Se esiste, ritorniamo la codifica di z, altrimenti il procedimento non termina.

È facile vedere che ciò è uguale a $\varphi_i(x)$ per ogni x: distinguiamo due casi.

- Se $\varphi_i(x) = n$ la computazione di $M_i(x)$ termina e dà come configurazione finale $(h, \triangleright z)$, dove n codifica z. Ma allora esiste un $y \in \mathbb{N}$ che codifica la computazione terminante di $M_i(x)$ (e tale y è anche unico, poiché la computazione è unica) e dunque y è il minimo valore del terzo parametro per cui vale che $T(i, x, y) = 1$. Per definizione di U, $U(y)$ è la codifica di z e dunque è n.
- Se $\varphi_i(x)$ diverge, allora non esiste una codifica della computazione di $M_i(x)$ (poiché la computazione diverge) e pertanto non esiste un y per cui $T(i, x, y) = 1$. In particolare $U(y)$ diverge.

Infine T ed U sono primitivi ricorsivi in quanto lo sono le codifiche e i controlli effettuati, e composizione di funzioni primitive ricorsive è ancora primitiva ricorsiva. □

[a] Osserviamo che questo procedimento termina sempre poiché le computazioni sono di lunghezza finita, dunque possiamo verificare in tempo finito se il calcolo di $M_i(x)$ corrisponde alla computazione cercata oppure no.

Osservazione 1.6.1. Il teorema ci dice che ogni funzione calcolabile φ_i è esprimibile come composizione di due funzioni primitive ricorsive (T, U) e una funzione generale ricorsiva (data dalla minimizzazione), o equivalentemente da due comandi `FOR` e un comando `WHILE`.