Elementi di Calcolabilità e Complessità

Luca De Paulis

24 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.

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

- ZERO $\lambda x_1, \dots, x_n.\, 0$ per ogni $n \in \mathbb{N}$
- SUCCESSORE $\lambda x.\, x + 1$
- PROIEZIONE $\lambda x_1, \dots, x_n.\, x_i$ per ogni $n \in \mathbb{N}$, $i = 1, \dots, n$

e che sia chiusa per le seguenti operazioni:

- COMPOSIZIONE se $g_1, \dots, g_k : \mathbb{N}^n \to \mathbb{N}$, $h : \mathbb{N}^k \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

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

appartiene ancora a $\mathcal{PR}$;

▸ **RICORSIONE PRIMITIVA** se $h : \mathbb{N}^{n+1} \to \mathbb{N}$, $g : \mathbb{N}^{n-1} \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

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

$$f(x_1, \ldots, x_n) := \begin{cases} f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \\ f(n+1, x_2, \ldots, x_n) = h(n, f(n, x_2, \ldots, x_n), x_2, \ldots, x_n) \end{cases}$$

appartiene ancora a $\mathcal{PR}$.