Università degli Studi "Roma Tre"
Laurea Magistrale in Teoria della Comunicazione

TCI – Teoria della Computazione e dell'Interazione – A.A. 2011-2012

M. Pedicini

Scriba-notes di Giulio Pellitta e Giorgia Pestrin

Appunti di Informatica: Macchine, Linguaggi e Modelli

versione preliminare del 19 febbraio 2012

Indice

Il corso rivolto a chi si interessa agli aspetti matematici della Computer Science, fornirà una panoramica sui risultati e sui principali problemi aperti in questo campo.

Informazioni

Pagina Web: `http://logica.uniroma3.it/~marco/TCI.2011`
e-mail: `pedicini@mat.uniroma3.it`

Lavoro Guidato

- lavoro guidato (scadenza mensile);
- esercizi di programmazione (programmazione funzionale ed object oriented);
- esami/esoneri (esonero parziale (aprile) ed esonero finale (giugno)).

Valutazione

(1) (per chi segue il corso)
  - 35% esame finale (secondo esonero);
  - 65% media voti lavoro guidato (scartando il voto peggiore ad uno degli esercizi) + 1 esonero + progetto;
  - eventuale orale per "aggiustare" il voto finale.

(2) (per chi non segue il corso)
  - esercizi di programmazione;
  - esame scritto;
  - esame orale.

Nota: **validità temporale degli esoneri**, l'esonero ha validità per un numero di sessioni che dipende dal voto x:

- $15 \leq x < 18$: valido per una sessione,
- $18 \leq x < 24$: valido per due sessioni,
- $24 \leq x < 27$: valido per tre sessioni,
- $27 \leq x < 30$: valido per tutti gli appelli dell'anno accademico.

Bibliografia

- M. Pedicini, *Appunti di Informatica Teorica* (questi appunti) essenzialmente basati su: Patrick Dehornoy, *Calculabilite et Decidabilite* (1993), Springer-Verlag in francese;
- A. Bernasconi, B. Codenotti, *Introduzione alla complessità computazionale*, Springer-Verlag.
- J.-L. Krivine, *Lambda Calculus: Types and Models*, (Masson).
- P. H. Wintson, S. Narasimhan, On to Java, Addison-Wesley (1998).

Bibliografia Addizionale

- R. Sethi, *Programming Languages: concepts and constructs*, Addison-Wesley (ed. italiana Zanichelli).
- Aho, Hopcroft, Ullman, *Design and Analysis of Computer Programming.*
- H. Hermes, *Enumerability, Decidability, Computability*, Die Grundlehren der Mathematichen Wissenshaften in Einzeldarstellungen, n. 127, Springer-Verlag.
- N. D. Jones, *Computability and Complexity from a Programming Perspective*, MIT Press.
- G. Ausiello, G. Gambosi, F. d'Amore - Linguaggi, Modelli, Complessità, Franco Angeli Editore (2003).

Introduzione

Argomenti di studio in questo corso sono i *modelli di calcolo* utilizzati per definire e studiare il concetto di calcolo automatico. Gli aspetti che trattiamo sono essenzialmente quelli di natura matematica seppure osserviamo che il concetto di calcolo ha interessato ed interessa molto la filosofia della scienza. A questo scopo la prima parte del corso riguarda la presentazione di alcuni dei formalismi più utilizzati e dei legami che intercorrono tra essi.

Una prima osservazione che si può fare, è che i modelli proposti non necessariamente fanno riferimento all'idea ed alla forma che si è andata definendo per i calcolatori, e infatti alcuni dei modelli più usati sono stati proposti prima dell'avvento stesso dal calcolo meccanico, ed anzi ne hanno influenzato la struttura ed anticipato le proprietà.

Se dovessimo ordinare rispetto ad una qualche scala evolutiva i modelli matematici del concetto di computazione, dovremmo situare in basso i modelli che più somigliano ai calcolatori elettronici e più in alto situare i modelli più astratti. Una caratteristica dei modelli più semplici risiede nel fatto di catturare il concetto in maniera più *intensionale*, intendendo con questo termine che il modello cattura il concetto andando a descrivere i passi elementari di calcolo permessi alla macchina più che fornire una descrizione del calcolabile. I modelli più astratti sarebbero invece quelli che ripongono la propria espressività su una descrizione *estensionale*, ovvero che considera due oggetti identici se il loro comportamento è indistinguibile; in questo caso una descrizione funzionale risulta certamente più vicina alla pratica del linguaggio matematico.

I modelli che analizzeremo sono: il modello RAM (Random Access Machine) una formalizzazione dell'architettura di von Neumann alla base dei calcolatori elettronici, il modello di Macchina di Turing identificato con la definizione stessa di calcolabilità, il modello delle funzioni ricorsive e quello del calcolo delle lambda-espressioni alla base dei linguaggi di programmazione funzionale.

Un ultimo aspetto che affronteremo è dato dalla classificazione delle funzioni computabili in base alla loro complessià computazionale. La complessità computazionale è la disciplina che si occupa delle questioni fondamentali circa la potenza e le limitazioni dei calcolatori, o, più in generale dei sistemi di calcolo. L'origine di queste problematiche risale alla prima metà del ventesimo secolo e si sviluppa insieme al concetto di computazione sviluppato nell'ambito della logica matematica.

L'ordine temporale con cui queste teorie sono state introdotte è invertito rispetto alla presentazione che ne daremo in questo corso, questa inversione (dai modelli più astratti a quelli più concreti) è motivata con la difficoltà di mettere in luce i vari concetti atti a delimitare il significato di calcolo. In particolare, questa ricerca dei fondamenti del calcolo nel corso di tutto il secolo passato, si è dovuta misurare con una delle più stupefacenti scoperte scientifiche in questo ambito: il teorema

di di Gödel sull'incompletezza dell'aritmetica, che pone un limite ad ogni sistema di calcolo. Non è azzardato affermare che tra i risultati scientifici più importanti del ventesimo secolo vanno ricordati i lavori di K. Gödel di A. W. Turing e di A. Church, che hanno analizzato in maniera precisa il concetto di *algoritmo* e di *problema risolvibile per via algoritmica*, dimostrando anche l'esistenza di problemi non risolvibili (vedremo alcuni problemi *non-decidibili* più avanti).

Riassumendo, ci occuperemo della visione matematica della computer science, la scienza della programmazione, ponendo l'accento sui concetti, come:

- computabilità: cosa si può calcolare ?
- efficienza: quale è il modo più rapido per effettuare un calcolo ?
- rappresentabilità: quali sono i modi equivalenti per descrivere un problema ?
- astrazione: come fare astrazione dalle macchine per studiare le proprietà dei programmi ?

Parte 1

Teoria della Calcolabilità

CAPITOLO 1

Algoritmi e Macchine di Turing

Il concetto di *algoritmo* è l'elemento centrale della teoria della computazione, (o teoria della calcolabilità). Il punto di partenza di questa teoria è infatti l'esigenza di formalizzare l'idea intuitiva di funzione *calcolabile* tramite un algoritmo, ovvero di funzione algoritmica. Informalmente possiamo dire che un algoritmo è un procedimento di calcolo che consente di pervenire alla soluzione di un problema, mediante una sequenza finita di operazioni, completamente ed univocamente determinate. In altri termini, un algoritmo consiste di una serie di istruzioni (operazioni) la cui esecuzione consente di "trasformare" l'insieme finito di dati simbolici che descrivono un problema, nella soluzione del problema stesso (o meglio, di una sua rappresentazione).

Le caratteristiche essenziali della nozione informale di algoritmo sono le seguenti:

- l'insieme delle istruzioni che definiscono l'algoritmo è finito,
- l'insieme delle informazioni che rappresentano il problema e i dati richiesti per la sua soluzione possiedono una descrizione finita,
- esiste un agente di calcolo in grado di eseguire le operazioni,
- il procedimento di calcolo, o *computazione*, è suddiviso in passi discreti e non fa uso di dispositivi analogici,
- la computazione è *deterministica*, ossia la sequenza di passi computazionali è determinata senza ambiguità.

Una procedura si può dire algoritmica se riceve in ingresso una descrizione finita dei dati del problema, e restituisce dopo un numero finito di passi di calcolo, una descrizione (per forza) finita del risultato. La natura deterministica della procedura fa sì che l'algoritmo fornisca sempre lo stesso risultato in presenza degli stessi dati iniziali. In questo modo l'algoritmo stabilisce una relazione di tipo funzionale tra i dati in ingresso e quelli in uscita. Quindi è importante distinguere la funzione $f : X \to Y$ calcolata da un certo algoritmo e l'algoritmo stesso. Infatti, per la funzione f possono esistere diversi algoritmi e per questo più o meno efficienti.

1. Problemi di Decisione

In questo capitolo si fornirà un quadro preciso della nozione di algoritmo e verrà introdotta la particolare famiglia degli algoritmi associati alle macchine di Turing. Saranno sviluppati alcuni esempi e verrà dimostrato un risultato di complessità.

Cosa sono i problemi di decisione ? Per la determinazione della difficoltà intrinseca associata ad un certo oggetto matematico possiamo utilizzare una nozione di misura della sua difficoltà.

Sappiamo che gli oggetti di base della matematica sono gli insiemi, e per essi è la relazione di appartenenza che fornisce una struttura. Dunque è proprio la relazione

$x \in X$ che fornisce una nozione di complessità (nel senso per ora informale di ricchezza strutturale, o di contenuto informativo) all'insieme stesso.

E' proprio questa struttura che ci servirà per precisare la definizione di algoritmo ed arrivare ad una definizione formale. Del concetto di algoritmo si è soliti dare definizioni del tipo:

un algoritmo è un metodo generale e meccanico per risolvere un dato problema.

Nel seguito preciseremo la nozione di algoritmo in modo da poter studiare e confrontare diversi algoritmi dal punto di vista matematico, questa nozione dovrà mantenersi ad un livello di generalità il piú alto possibile, per poter esprimere la classe piú ampia di algoritmi. Per tutti i modelli di calcolo che verranno introdotti mostreremo, se esiste, un risultato di equivalenza[1].

Il problema che ci guiderà nella formalizzazione del concetto di algoritmo è quello di

problema di decisione,

la cui soluzione consiste nel determinare un metodo generale per stabilire se un elemento x appartiene ad un dato insieme X.

Qui x costituisce il dato iniziale, a partire dal quale il metodo deve poter stabilire la risposta (affermativa o negativa), di appartenenza all'insieme X.

Alcune caratteristiche aggiuntive saranno richieste per ottenere un metodo effettivo (ovvero in qualche modo meccanizzabile):

- il metodo deve essere *deterministico* ovvero una volta fissati i dati iniziali non intervengono scelte che facciano appello a operazioni o dati non menzionati esplicitamente;
- i passi elementari del metodo costituiscono una successione *discreta* (in contrapposizione a continua) di passi elementari di natura *finitaria*, ovvero che mettono in gioco localmente un numero finito di parametri che variano su un insieme finito.

Quando il metodo di decisione soddisferà questi requisiti potrà essere considerato un *algoritmo*. Chiaramente questa definizione è ancora troppo informale per essere utile dal punto di vista matematico e sarà ulteriormente precisata nel seguito.

La misura della complessità di un insieme X sarà quindi fornita dalla complessità della procedura di decisione corrispondente.

E' chiaro che possono esistere insiemi X per i quali una procedura di decisione con le caratteristiche summenzionate non esiste (esempio ?).

Per quanto riguarda, la complessità uno degli aspetti di maggiore interesse è la possibilità di confrontare tra di loro differenti procedure di decisione al fine di stabilire quale di esse sia piú o meno efficace.

Per fare ciò si rende necessario l'introduzione di un formalismo *unificato* nel quale sia possibile esprimere il concetto generale di algoritmo, per questo introduciamo un linguaggio uniforme in cui esprimere ogni algoritmo come successione di passi elementari appartenenti ad un unico linguaggio di base, in questo modo è possibile dare una valutazione della complessità dell'algoritmo contando il numero di operazioni elementari che deve eseguire per ottenere il risultato.

[1]Tesi di Church: l'insieme dei risultati di equivalenza tra vari modelli effettivi di calcolo è noto come Tesi di Church, che afferma che tutti i modelli di calcolo si equivalgono.

Esempio: per quanto riguarda il concetto di complessità relativa al linguaggio di base si pensi di dover confrontare algoritmi per l'aritmetica sui numeri interi, in tal caso il linguaggio di base potrebbe essere considerato quello delle operazioni aritmetiche: somme, prodotti, sottrazioni, divisioni.

Intuitivamente, possiamo pensare ad una procedura deterministica, sequenziale, discreta e finitaria in termini di linguaggi di programmazione imperativi (come C o PASCAL), oppure in termini piú astratti come procedure rappresentabili in termini di Macchine di Turing (confronta il Capitolo **??**).

Una difficoltà che sorge nel manipolare e nel voler unificare il trattamento di diversi algoritmi è che gli oggetti di cui si occupa la matematica non sono tutti della stessa natura: numeri interi, numeri reali, punti, funzioni, etc... Allora, un algoritmo, o un qualsiasi altro metodo effettivo, non lavora mai direttamente sull'oggetto in se, ma su una sua *rappresentazione* concreta. Nella pratica queste rappresentazioni sono quasi esclusivamente costituite da "parole" ovvero *sequenze finite* di simboli ottenute concatenando elementi di un insieme finito detto *alfabeto*.

Tipicamente i numeri interi, l'insieme $\mathbb{N}$, sono nella pratica utilizzati attraverso la loro rappresentazione in un base fissata, ovvero un numero è rappresentato come sequenza finita di elementi (le cifre) presi nell'alfabeto $A = \{0, 1, \ldots, b-1\}$ dove b è la base utilizzata per la rappresentazione. Un ulteriore esempio è quello dell'insieme dei numeri razionali $\mathbb{Q}$ che essendo rappresentabili come coppie di numeri interi, sono esprimibili utilizzando un alfabeto $A' = A \cup \{/\}$ (con il simbolo / utilizzato per indicare la linea di frazione). In generale la descrizione di un algoritmo su un insieme X fa riferimento ad una particolare codifica finitaria degli elementi di X con parole di un opportuno alfabeto, ovvero di un insieme finito.

La codifica dei numeri reali comporta però delle problematiche ulteriori: lo sviluppo decimale infinito

$$\pi = 3.14926535897932384626433832795 02\ldots$$

non è una parola finita, e non si può sperare di codificare con le parole di un alfabeto finito ogni numero reale. Infatti, l'insieme delle parole su un alfabeto finito ha cardinalità del numerabile mentre sappiamo che la cardinalità dei numeri reali è più che numerabile, e quindi non vi può essere una funzione che "codifica" (ovvero rappresenta) i numeri reali con le parole di un alfabeto finito (poichè una codifica è necessariamente una funzione biunivoca).

Il punto importante di questa discussione emerge dal fatto che si può comunque pensare ad algoritmi applicati a numeri reali, in tal caso bisognerà trattare soltanto con numeri reali che in un modo o nell'altro sono stati specificati mediante una codifica finita, ovvero con numeri reali "definibili". Bisognerà specificare di volta in volta questa nozione di *numero reale definibile*[2], precisandone il quadro formale (definibile mediante una frase, mediante una formula matematica etc...) quindi per

[2]Ad esempio, mentre la rappresentazione decimale di un irrazionale algebrico induce una rappresentazione decimale che coinvolge un numero infinito di simboli, vedi la sezione aurea $A = (1+\sqrt{5})/2$, che è soluzione di $x^2 - x - 1 = 0$, ha una rappresentaione decimale infinita aperiodica

$$1.618033988749894848204\ldots$$

mentre ne ha una in frazione continua, infinita periodica e quindi rappresentabile con un numero finito di simboli $A = [1, 1, 1, \ldots] = [\overline{1}]$. N.B. che se includessimo anche il simbolo di $\sqrt{\ }$, la scrittura $(1+\sqrt{5})/2$ diventerebbe finita.

concludere: la rappresentazione dei numeri reali sarà ottenuta mediante la codifica su un certo alfabeto finito della definizione che lo specifica.

2. Sistemi di Transizione

La sezione precedente, ed in particolare l'osservazione conclusiva permettono di restringere la definizione di algoritmo a quelli che agiscono su insiemi di parole, indipendentemente da quello che queste parole rappresentano. Le notazioni che adotteremo, allora, nel seguito saranno le seguenti.

Se $A = \{a_1, \dots, a_n\}$ è un insieme finito scelto come alfabeto, $A^\star$ indicherà l'insieme delle parole su A, ovvero l'insieme delle successioni finite di elementi di A (compresa la parola vuota).

Se $u, v \in A^\star$ allora $uv \in A^\star$ indicherà la parola ottenuta concatenando u con v.

Volendo essere più formali possiamo introdurre le parole come applicazioni degli intervalli $\{1, \dots, k\}$ di numeri interi in A, e quindi:

$$A^* = \bigcup_{k=0}^{\infty} A^{\{1,\dots,k\}}.$$

Si noti che per convenzione l'insieme $\{1, \dots, k\}$ per $k = 0$ corrisponde all'insieme vuoto.

In questo modo, si può definire la funzione "lunghezza" di una parola $|u|$ come il più piccolo n tale che $u \in A^{\{1,\dots,n\}}$.

E in tal caso la concatenazione di due parole u e v sarà una funzione $uv \in A^{\{1,\dots,|u|+|v|\}}$ tale che

$$uv(k) = \begin{cases} u(k) & \text{se } k \leq |u| \\ v(k - |u|) & \text{se } |u| + 1 \leq k \leq |u| + |v| \end{cases}$$

Definiamo ora un linguaggio specifico adatto alla descrizione degli algoritmi: per prima cosa gli aspetti discreti e deterministici degli algoritmi saranno catturati permettendo agli algoritmi delle esecuzioni sequenziali indicizzate da numeri interi, un passo di esecuzione dell'algoritmo sarà trattato come l'applicazione di una *funzione* e tutta l'esecuzione sarà data dall'applicazione iterata di questa funzione che darà luogo ad una successione di passi intermedi di calcolo:

$$f(c_0), f(f(c_0)), \dots, f^n(c_0), \dots$$

ad ogni passo, un certo numero di parametri possono mutare in seguito all'applicazione della funzione di transizione: chiameremo *configurazioni* l'insieme dei valori assunti da questi parametri ad un dato momento della computazione.

In questo modo un algoritmo può essere visto essenzialmente come un insieme di configurazioni C (eventualmente infinito) munito di una *funzione di transizione* $\tau : C \to C$ (non necessariamente totale).

Il calcolo eseguito da un algoritmo (ovvero la sua *esecuzione* o *computazione*) sarà dato da una successione (finita o infinita) di configurazioni

$$(c_0, c_1, c_2 \dots)$$

tali che $\tau(c_{n+1}) = c_n$.

L'esecuzione costituisce la parte "interna" della computazione, e per avere un legame con il problema di decisione $x \in X$, bisogna collegare questa computazione con il dato iniziale e fornire il risultato della computazione in uscita. Il punto di

partenza, il dato iniziale, deve essere pensato come una parola di un opportuno alfabeto, adatto alla codifica di tutti gli elementi di X. La parola $w \in A^*$ costituisce l'argomento su cui sarà effettuata la computazione. Dunque per specificare l'algoritmo è necessario fornire un'applicazione $\epsilon : A^* \to C$ che associ ad una data parola la configurazione iniziale dell'esecuzione di τ. Questa funzione $\epsilon()$ sarà chiamata *funzione di input*, mentre le configurazioni di C che sono immagine di quache $w \in A^*$ attraverso ϵ saranno dette *configurazioni iniziali*.

In maniera del tutto simmetrica, introduciamo anche una funzione $\delta : C \to B^*$, e che fa corrispondere ad alcune configurazioni particolari, dette *terminali*, una parola di un opportuno alfabeto B, che costituisce lo spazio delle risposte possibili. Per quanto riguarda i problemi di decisione dobbiamo considerare due sole risposte possibili 1 per codificare l'appartenenza ($x \in X$) e 0 per la non-appartenenza ($x \notin X$).

Per non perdere la proprietà del determinismo della funzione è necessario restringere gli algoritmi a quelli per cui il dominio della funzione di transizione τ e quello della funzione di uscita δ sono disgiunti. Inoltre per comodità si prende una partizione di C, ovvero

$$dom(\tau) \cap dom(\delta) = \emptyset \qquad dom(\tau) \cup dom(\delta) = C.$$

Riassumendo, abbiamo che un algoritmo è completamente descritto una volta che sono fissati: un alfabeto A di input e un alfabeto B di uscita, uno spazio di configurazioni possibili C e tre funzioni

$$F : \begin{cases} \epsilon : A^* \to C \\ \tau : C \to C \\ \delta : C \to B^* \end{cases}$$

in tal caso la computazione associata ad una parola $w \in A^*$ sarà la successione

$$(\epsilon(w), \tau(\epsilon(w)), \tau^2(\epsilon(w)), \dots)$$

che potrà essere finita o infinita, in quest'ultimo caso si dirà che l'algoritmo *non termina*.

Se la sequenza termina in una configurazione $\tau^n(\epsilon(w))$ allora si dice che $\delta(\tau^n(\epsilon(w)))$ è il risultato; quindi ad ogni algoritmo resta associata la funzione $f : A^* \to B^*$ tale che

$$f(w) = \begin{cases} \delta(\tau^n(\epsilon(w))) & \text{se } F \text{ termina ed } n \text{ é la lunghezza della computazione,} \\ \perp & \text{se } F \text{ non termina.} \end{cases}.$$

Nella pratica non sarà mai necessario distinguere funzione e algoritmo e quindi utilizzeremo $F(w)$ direttamente per indicare $\delta(\tau^n(\epsilon(w)))$.

Quindi ad ogni algoritmo è associata una funzione $f : A^* \to B^*$ tale che

$$f(w) = \begin{cases} F(w) & \text{se la computazione di } F \text{ a partire dall'input } w \text{ termina,} \\ \perp & \text{altrimenti} \end{cases}$$

Diremo allora che:

DEFINIZIONE 1. *Dato un alfabeto A qualunque, e un sotto insieme $X \subset A^*$, l'*algoritmo F decide l'insieme X *se il suo alfabeto di ingresso contiene A e il suo alfabeto di uscita contiene $\{0, 1\}$ e se per ogni parola $w \in A^*$ si ha che se $w \in X$ allora $F(w) = 1$ e se $w \notin X$ allora $F(w) = 0$.*

*Inoltre, diremo che l'*algoritmo F semi-decide l'insieme X *se il suo alfabeto di ingresso contiene* A *e il suo alfabeto di uscita contiene* $\{1\}$ *e se per ogni parola* $w \in A^*$ *si ha che se* $w \in X$ *allora* $F(w) = 1$*, e se* $w \notin X$ *allora l'algoritmo non termina.*

L'insieme X *si dice* decidibile *(risp.* semi-decidibile*) se esiste un algoritmo* F *che lo decide (risp. lo semi-decide).*

Si puó effettuare la modifica di ogni algoritmo che decide un dato insieme X in uno che semi-decide X: è sufficiente aggiungere per ogni configurazione terminale c che produce un risultato $\delta(c) = 0$ una nuova transizione τ da c in se stessa $\tau(c) = c$ e modificare δ in modo che non sia definita su c. Questo significa che ogni insieme decidibile é anche semidecidibile.

Il quadro formale che abbiamo appena dato fornisce in prima approssimazione la specifica di un algoritmo (di tipo sequenziale) ma non costituisce per questo un quadro sufficiente: l'arbitrio dato dall'insieme delle configurazioni C, dalle funzioni ϵ, δ e τ, rendono troppo generale questa definizione e fanno corrispondere alla definizione che abbiamo dato realmente un algoritmo se esse stesse sono funzioni "effettive" ovvero "algoritmiche", in caso contrario il quadro formale che abbiamo introdotto risulta privo di un significato reale.

Ad esempio, per un qualsiasi sottoinsieme $X \subset A^*$, si può ricondurre la decidibilità di X alla definizione precedente prendendo come insieme delle configurazioni $C = A^*$, ponendo $\epsilon(w) = w$ e

$$\delta(w) = \begin{cases} 1 & \text{se } w \in X \\ 0 & \text{se } w \notin X. \end{cases}$$

É chiaro che la formalizzazione di un tale algoritmo non fornisce nessuna informazione aggiuntiva: il problema di decisione associato a X, poggia completamente sulla definizione esplicita della funzione δ.

2.1. Esempio. *Stabilire l'algoritmo che decide l'insieme dei numeri interi divisibili per* 9*.*

Scegliamo la rapprasentazione decimale (ovvero quella usuale) per i numeri interi, dunque resta fissato come alfabeto

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Affrontiamo la specifica dell'algoritmo avendo in mente il seguente criterio di divisibilità per nove: un numero $n = c_d c_{d-1} \dots c_0$ è divisibile per nove se la somma delle sue cifre lo è, ovvero

$$(\sum c_i) \equiv 0 \mod 9.$$

In pratica, non è necessario effettuare la somma in una sola volta, ma basterà considerare una ad una le cifre della rappresentazione in base 10, sommando la cifra corrente con la classe della somma delle cifre già considerate, ovvero

$$(\sum_{i=0}^{d} c_i) \mod 9 = c_d + (\sum_{i=0}^{d-1} c_i \mod 9) \mod 9.$$

Si introduce allora come spazio delle configurazioni

$$C = \{(w, k, n) \mid w \in A^\star, k \in \mathbb{N}, n \in \{0, \dots, 8\}\}$$

una data configurazione (w, k, n) sarà utilizzata per memorizzare

- l'elemento w conterrà la parola che stiamo analizzando,
- l'elemento k rappresenterà la posizione all'interno della parola w dell'ultimo simbolo analizzato,
- l'elemento n è la classe di apparteneza del numero intero corrispondente alla parola w troncata al k-esimo elemento.

Quindi nel nostro caso abbiamo, il seguente algoritmo:

- problema di decisione:
$$X = \{n \in \mathbb{N} | n \equiv 0 \mod 9\},$$
- alfabeto di ingresso: $A = \{0, 1, \ldots, 9\}$,
- alfabeto di uscita: $B = \{0, 1\}$,
- spazio delle configurazioni:
$$C = \{(w, k, s) | w \in A^*, k \in \mathbb{N}, s \in \{0, 1, 2 \ldots, 8\}\},$$
- funzione di ingresso: $\varepsilon(w) = (w, 0, 0)$,
- funzione di transizione:
$$\tau((w, k, s)) = (w, k + 1, (w_{k+1} + s) \mod 9),$$
per $k < |w|$,
- funzione di uscita:
$$\tau((w, k, s)) = \delta(w, |w|, s) = \begin{cases} 0 & s \neq 0, \\ 1 & s = 0. \end{cases}.$$

Ricordando la definizione di computazione e di risultato ad esempio abbiamo che la computazione associata a $n = 3271$ è:

$$\varepsilon(3271) = (3271, 0, 0) \tag{1}$$
$$\tau((3271, 0, 0)) = (3271, 1, 3) \tag{2}$$
$$\tau((3271, 1, 3)) = (3271, 2, 5) \tag{3}$$
$$\tau((3271, 2, 5)) = (3271, 3, 3) \tag{4}$$
$$\tau((3271, 3, 3)) = (3271, 4, 4) \tag{5}$$
$$\delta((3271, 4, 4)) = 0. \tag{6}$$

Da quello che abbiamo visto a proposito della definizione di algoritmo è chiaro che resta da analizzare il carattere "effettivo" delle funzioni ϵ, τ e δ: bisogna che queste funzioni rispondano alle caratteritiche richieste: algoritmo finito, il dato iniziale possiede una descrizione finita, la computazione è suddivisa in passi discreti, la computazione è deterministica, esiste un agente di calcolo.

In particolare nel nostro esempio, la funzione di transizione suppone la conoscenza delle 90 possibilità associate per il resto modulo 9 della somma di due cifre di cui la prima è compresa tra 0 e 9 e la seconda tra 0 e 8. Sembra dunque abbastanza naturale considerare questo esempio come tipico per il definire nel seguito una nozione più formale di algoritmo.

3. Automi Finiti

Le caratteristiche dell'esempio (criterio di divisibilità) definiscono una restrizione del concetto di algoritmo formale, che sarà utile per la chiarire gli aspetti deterministici e finitari richiesti dal concetto di algoritmo.

Infatti, ogni passo elementare, ogni transizione, consiste nel modificare il risultato ottenuto dai passi precedenti di calcolo, in funzione del carattere della parola di ingresso, e a ripetere l'operazione sui caratteri seguenti sino alla fine della parola.

Poichè ogni passo è determinato da uno stato finito e da un carattere dell'alfabeto finito, resta evidente il carattere finitario della operazione (nella descrizione astratta di algoritmo un carattere infinitario si potrebbe nascondere nella funzione di transizione). D'altra parte la parte di parola già analizzata non viene memorizzata completamente ma solo l'informazione necessaria per il calcolo viene utilizzata (lo stato, ovvero la classe della somma delle cifre già lette). Questo modo di calcolare porta alla nozione di problema decidibile mediante automa finito.

Supponiamo di avere un dato alfabeto A ed un insieme finito Q e consideriamo una qualsiasi funzione

$$T : Q \times A \to Q,$$

ad ogni funzione T si può associare un algoritmo formale nel senso che abbiamo visto prima: il funzionamento dell'automa si descrive in modo analogo a quello dell'algoritmo dell'esempio, ovvero, data la parola in ingresso, l'automa si posiziona sul primo carattere della parola e ad ogni passo di computazione si muove sul carattere successivo ed effettua una transizione dallo stato q allo stato $T(q, a)$ se a è il carattere della posizione corrente. Al termine della lettura della parola la decisione viene presa in base allo stato corrente, quindi l'algoritmo deve prevedere che alcuni stati siano in corrispondenza con l'ouput 0 e alcuni siano in corrispondenza con l'output 1. In maniera precisa:

Definizione 2. *Siano A e Q insiemi finiti, un* automa finito *di alfabeto A e insieme degli stati Q è una tripla (T, q_0, F) dove*

$$T : Q \times A \to Q$$

$$F : Q \to \{0, 1\}$$

e $q_0 \in Q$ è uno stato particolare, detto stato iniziale.

Esempio 1. Descrivere l'automa finito che decide l'insieme delle parole di lunghezza pari (risp. dispari) su un alfabeto dato A.

A partire dall'alfabeto finito A, e da un insieme di due stati $Q = \{0, 1\}$ che corrisponderanno rispettivamente allo stato "abbiamo letto un numero pari (oppure dispari) di caratteri della parola"; costruiamo la funzione di transizione che ad ogni carattere letto cambia stato:

$$T : Q \times A \to Q$$

tale che

$$T(q, a) = \begin{cases} 1 & \text{se } q = 0 \\ 0 & \text{se } q = 1, \end{cases}$$

per ogni $a \in A$.

Avremo allora che lo stato iniziale $q_0 = 0$, poichè inizialmente non avremo ancora letto nessun carattere e che la funzione che stabilisce gli stati accettanti e quelli non accettanti sarà:

$$F(q) = \begin{cases} 1 & \text{se } q = 0 \\ 0 & \text{se } q = 1. \end{cases}$$

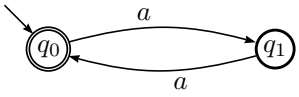

FIGURA 1. Automa dell'Esempio **??**, notare che $a \in \{0, 1\}$.

A partire da un automa finito (T, q_0, F) si può dare una descrizione grafica nel modo seguente, si considera un grafo i cui nodi sono gli stati Q con archi etichettati con elementi di A che collegano due stati (q_i, q_j) se esiste la transizione $T(q_i, a) = q_j$:

$$G_{(T,q_0,F)} = \{(q_i, q_j, a) | q_i, q_j \in Q, a \in A, T(q_i, a) = q_j\}$$

inoltre avremo una funzione che ci indica lo stato iniziale

$$\texttt{init}(G_{(T,q_0,F)}) = q_0$$

(di solito indicata da un arco entrante nel nodo iniziale q_0) ed una che distingue gli stati accettanti (risp. non accettanti)

$$\texttt{acc}(G_{(T,q_0,F)}) = F^{-1}(1) \qquad (\text{risp. } \texttt{ref}(G_{(T,q_0,F)}) = F^{-1}(0))$$

(di solito si indicano gli stati accettanti con un doppio cerchio, mentre tutti gli altri nodi sono considerati non accettanti).

Ad esempio il grafo associato all'automa dell'esempio precedente, è rappresentato in Figura **??**.

3.1. Esecuzione di un automa/rappresentazione matriciale. Come abbiamo visto un automa ha una rappresentazione in termini di grafo possiamo utilizzare questa rappresentazione come punto di partenza per fornire un'ulteriore formalismo legato al concetto di automa. Associato ad un grafo $G = (V, E)$ troviamo la sua presentazione come matrice di adiacenza:

$$M = (m_{ij})_{i,j \in V} \text{ dove } m_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E, \\ 0 & \text{altrimenti.} \end{cases}$$

Questa definizione si estende al caso di grafi non orientati (per i quali si costruiscono matrici simmetriche) e a quella di grafi etichettati (per i quali gli elementi della matrice sono le etichette degli archi).

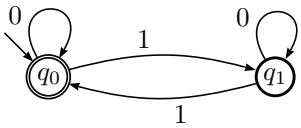

FIGURA 2. In modo simile all'automa dell'Esempio **??** si può dare l'automa che decide l'insieme di tutte le parole con un numero pari di 1.

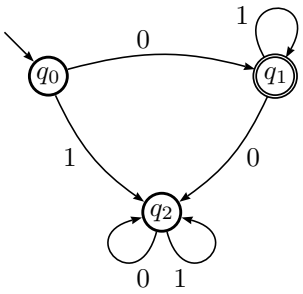

FIGURA 3. Automa che decide l'insieme $X = \{01^n | n \geq 0\}$.

In quest'ultimo caso, allo scopo di indicare gli archi assenti, sarà necessario introdurre un'etichetta "non definita" per segnalare l'assenza dell'arco. In particolare se l'insieme delle etichette ha una struttura di *monoide* (prodotto associativo tra etichette, con elemento neutro) si puó poi considerare una struttura con una somma libera (detta delle *serie formali*) che ci fornisce un'operazione additiva in aggiunta a quella moltiplicativa di monoide: abbiamo quindi il seguente semianello che é sufficiente per introdurre il prodotto righe per colonne tra matrici.

Introdurre semianello idempotente.

Considerre le matrici a coefficienti sul semianello.

DEFINIZIONE 3. *Dato un monoide* $(M, \cdot, 1)$ *si consideri l'insieme delle serie formali a coefficienti su un semi-anello* $\mathbb{K}$

$$\mathbb{K}[M] := \left\{ \sum_{m \in M} \alpha_m m | \alpha_m \in \mathbb{K} \right\}.$$

Su $\mathbb{K}[M]$ *resta definito un prodotto*

$$\sum_{m' \in M} \alpha_{m'} m' \sum_{m'' \in M} \beta_{m''} m'' = \sum_{m \in M} \left(\sum_{m'm''=m} \alpha_{m'} \beta_{m''} \right) m$$

L'algebra delle serie formali su un semianello $\mathbb{K}$ viene detta algebra di Kleene se ad ogni elemento $a \in \mathbb{K}$ si associa un elemento $a^* \in \mathbb{K}$ che soddisfa le seguenti proprietà:

(...)

[?]

Nel caso degli automi iterare le potenze successive della matrice di adiacenza corrisponde al calcolo dell'insieme deciso dall'automa:

$$E(M) = \sum_{k \geq 0} M^k$$

Si noti che M^0 corrisponde alla matrice diagonale con elemento neutro moltiplicativo sulla diagonale (la parola vuota nel caso del monoide delle parole) e l'elemento neutro additivo altrove (la somma formale vuota 0).

ESERCIZIO 1. *Dare la tabella per la decidibilità dei multipli di 9.*

Dato un automa finito si può descrivere l'algortimo di decisione associato: infatti, a partire dalla tripla (T, q_0, F) si costruisce:

- lo spazio delle configurazioni è $Conf_1^{A,Q} = A^* \times \mathbb{N} \times Q$,
- la funzione di ingresso $\varepsilon(w) = (w, 1, q_0)$,
- la funzione di transizione $\tau(w, p, q) = (w, p + 1, T(q, w_p))$,
- la funzione di uscita: $\delta(w, p, q) = F(q)$ se $p = |w| + 1$.

DEFINIZIONE 4. *Un insieme $X \subset A^*$, è decidibile per automa finito (AF-decidibile), se esiste un insieme finito Q e un automa finito con alfabeto A e insieme degli stati Q tale che l'algoritmo associato decide X.*

La restrizione a un insieme finito di stati Q è fondamentale. L'idea che lo stato corrisponde ad ogni passo all'informazione che bisogna riportare dai passaggi precedenti corrisponde all'idea di memoria del sistema di calcolo. Se eliminassimo questa restrizione, e quindi ammettessimo una capacità di memoria illimitata, usciremmo dal quadro fintario in cui ci siamo posti, facendo perdere alla nozione di decidibilità per automa tutto il significato: ogni insieme (discreto) è decidibile per "automa infinito".[3]

Infatti, per decidere un qualsiasi sottoinsieme X di A^* si potrebbe utilizzare l'automa che ha A^* come stati, con la parola vuota come stato iniziale e X come insieme degli stati accettanti. La tabella dell'automa sarebbe banalmente

$$(w, a) \mapsto wa,$$

per ogni $w \in A^*$ e $a \in A$. Ma in questo caso la nozione non avrebbe alcun interesse poichè non permetterebbe di distinguere gli insiemi (ad esempio decidibli/non-decidibili).

Un esempio di qualcosa di calcolabile in senso intuitivo che non è possibile decidere con un AF è dato dalle parole sull'alfabeto $\{0, 1\}$ che sono date da una sequenza di n volte 0 e di n volte 1 (formalmente sull'alfabeto $A = \{0, 1\}$, $X = \{0^n 1^n | n \in \mathbb{N}\}$). In questo caso, un automa finito non può decidere l'insieme perchè dovrebbe avere a disposizione una memoria non limitata (quindi un insieme infinito di stati) per memorizzare il numero di 0 letti.

allora bisogna modificare la nozione di algoritmo che è troppo restrittiva ... per questo si introducono i modelli di calcolo delle sezioni che seguono ed in particolare in sezione **??** *daremo un rilassamento della nozione di AF.*

3.2. Linguaggi regolari. Un insieme X AF-decidibile si dice anche *regolare*. Gli insiemi regolari (detti anche *linguaggi regolari* essendo insiemi di parole su un determinato alfabeto) soddisfano alcune proprietà di chiusura, che rendono semplice l'analisi dell'insieme stesso.

Per determinare se un determinato linguaggio è regolare: mentre la dimostrazione sufficiente di AF-decidibilità (esibendo esplicitamente un automa) puó risultare noiosa e complicata, è possibile fornire dimostrazioni alternative (possibilmente più semplici) mostrando che l'insieme è ottenibile a partire da insiemi AF-decidibili utilizzando operazioni "compatibili" con l'AF-decidibilità.

In questa sezione vedremo dunque che esistono operazioni fatte su insiemi AF-decidibili che permettono di ottenere un altro insieme AF-decidibile, più precisamente:

[3]semmai un'estensione che non è priva di significato consiste nella definzione di automi che riconoscono parole infinite ($w_{\mathbb{N}} \to A$) date su un alfabeto finito A, e che quindi decidono insiemi $X \subset A^{\mathbb{N}}$; in questo caso si parla di automi di Büchi.

- l'operazione di *concateazione*:
$$X_1 \cdot X_2 = \{w_1 w_2 \in A^* | w_i \in X_i\};$$
- l'operazione di *somma*:
$$X_1 + X_2 = \{w \in A^* | w \in X_1 \text{ oppure } w \in X_2\}$$
(di fatto, è l'unione insiemistica di X_1 con X_2);
- l'operazione di *star*:
$$X^* = \{w_1 w_2 \dots w_n | w_i \in X, \text{for } i = 1, \dots, n \text{ e } n \geq 0\}$$

PROPOSIZIONE 1. *Siano X_1 e X_2 regolari allora anche $X_1 + X_2$ è regolare.*

DIMOSTRAZIONE. Sia $\mathcal{A}_1 = (T_1, q_0, F_1)$ l'automa che decide l'insieme X_1 e sia $\mathcal{A}_2 = (T_2, q_0', F_2)$ l'automa che decide X_2.

L'automa che riconosce $X_1 + X_2$ è ottenuto utilizzando il prodotto cartesiano degli stati dei due automi e calcolando l'evoluzione di entrambi gli automi, in questo modo se una delle due componenti dello stato è accettante alla fine della computazione la parola sarà accettata poiché apparterrà ad uno dei due insiemi e quindi sarà nella loro unione.

Costruzione dell'automa $\mathcal{A} = (T, q_0'', F)$ di alfabeto A e insieme degli stati $Q = Q_1 \times Q_2$: definiamo
$$q_0'' = (q_0, q_0'),$$
$$T((q_1, q_2), a) = (T(q_1, a), T(q_2, a))$$
e infine
$$F(q_1, q_2) = \begin{cases} 1 & \text{se } F_1(q_1) = 1 \text{ oppure } F_2(q_2) = 1, \\ 0 & \text{se } F_1(q_1) = 0 \text{ e } F_2(q_2) = 0, \end{cases}$$

□

Esercizio: dati gli automi ce decidono $X_1 = \{01^n | n \geq 0\}$ e $X_2 = \{\}$ (vedi Figura 1 e Figura 2), costruire l'automa che decide $X_1 + X_2$.

4. La macchina RAM

La RAM (macchina ad accesso casuale) è il modello matematico utilizzato per formalizzare il concetto di computabilità che più si avvicina alla architettura reale di calcolatore, esso riprende infatti l'architettura di Von Neumann. Una RAM è costituita da un programma di controllo che ha accesso ad una memoria ad accesso casuale composta da una sequenza di locazioni, ognuna delle quali può contenere un intero relativo *arbitrario*. L'aggettivo casuale si riferisce al fatto che il programma può accedere alle singole celle di memoria in un ordine non sequenziale. Completano la descrizione architetturale della RAM un dispositivo di ingresso ed uno di uscita, questi invece ad accesso sequenziale.

Il modello di RAM originale fu introdotto da S. Cook agli inizi degli anni '70 ed è dotato di un linguaggio macchina costituito da istruzioni per l'accesso alla memoria, per operazioni aritmetiche, per il trasferimento del controllo (operazioni di salto) e di arresto.

L'uso del linguaggio macchina permette di misurare il costo dell'esecuzione delle singole istruzioni impiegate nella descrizione del programma in base alle risorse fisiche necessarie per eseguire l'istruzione. Dal costo di ogni singola istruzione è possibile risalire al costo dell'intero algoritmo. In realtà esistono due modalità per

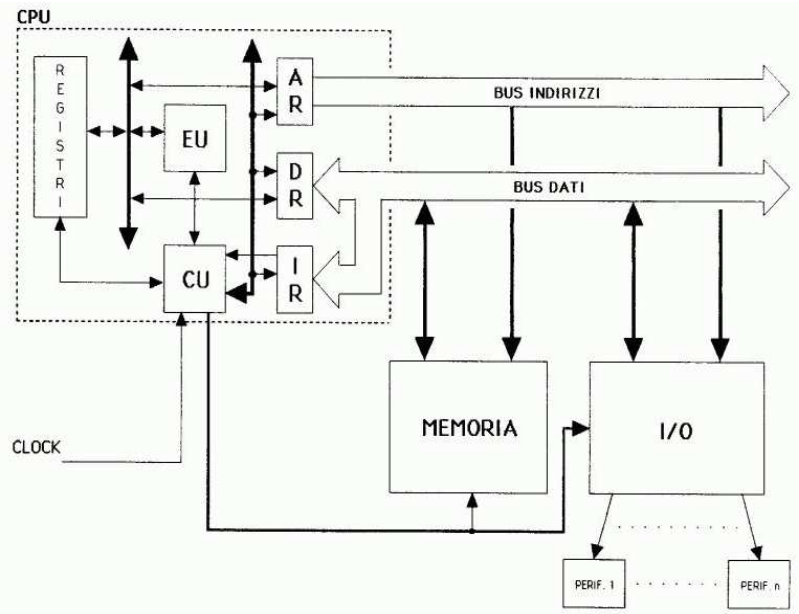

fig. 1.1 – Sistema basato sul modello Von Neumann

calcolare ed esprimere il costo dell'esecuzione di un programma, esse si distinguono poichè la pria considera costante il costo di ogni singola operazione, questa modalità non tiene però in conto come la dimensione del dato influenza il costo del circuito che effettua il calcolo. Il secondo modello allora misurerà il costo necessario a manipolare un dato in relazione alla sua lunghezza.

D'altra parte l'uso del linguaggio macchina non è ovviamente necessario ne' alla descrizione del programma ne' alla valutazione del suo costo computazionale. Infatti, è sempre possibile utilizzare un traduttore da un linguaggio diverso al linguaggio macchina, in questo modo i modelli che seguiranno saranno più orientati alla descrizione dell'algoritmo piuttosto che alla codifica.

Precisiamo ora in che modo funziona una RAM: una macchina ad accesso casuale è costitutito da un computer che esegue un programma che non può modificare se stesso e che utilizza un solo accumulatore A.

Le parti che costituiscono la RAM sono:

- un nastro di input in sola lettura,
- un nastro di output in sola scrittura,
- un programma,
- una memoria.

Il nastro di input è formato da celle che possono contenere numeri interi (eventualmente negativi).

Ogni volta che un simbolo viene letto dal nastro di input una testina di lettura che si trova sul nastro si sposta alla cella successiva.

Il nastro di output è anch'esso costitutito da celle che possono essere utilizzate per memorizzare numeri interi (eventualmente negativi) e che inizialmente contengono il carattere nullo (blank).

Quando una istruzione di scrittura viene eseguita, un intero viene memorizzato

Operazione	Indirizzamento	semantica
LOAD	operando	carica A con op.
STORE	operando	memorizza A in op.
ADD	operando	somma A con op.
SUB	operando	sottrae op. da A.
MULT	operando	moltiplica A con op.
DIV	operando	divide A con op.
READ	operando	carica il valore nella cella di input corrente in op.
WRITE	operando	scrive il valore di op. nella cella di output corrente.
JMP	label	salta alla riga di programma etichettata con lab.
JGTZ	label	se $A > 0$ salta alla riga di programma etichettata con lab.
JZERO	label	se $A = 0$ salta alla riga di programma etichettata con lab.
HALT		ferma l'esecuzione

TABELLA 1. Tabella del linguaggio di programmazione per le RAM.

in maniera indelebile nella cella corrente del nastro. La testina di scrittura si sposta poi alla cella successiva. In questo modo, quando un carattere in uscita è stato scritto non può più essere modificato.

La memoria, consiste di una sequenza registri che possono memorizzare interi di taglia arbitraria. Non viene posto nessun limite, né alla dimensione degli interi né al numero di registri necessari per eseguire un dato algoritmo (in questo il modello differisce da un'architettura reale, poichè la quantità di memoria di un calcolatore è finita e pure è finita la dimensione delle parole di memoria).

Il programma eseguito da una RAM, a differenza dei calcolatori con architettura di Von Neumann, non è memorizzato nella memoria del calcolatore, quindi il programma non ha modo di modificare se stesso. Il programma è una sequenza di istruzioni etichettate. Il tipo di istruzioni che possono essere utilizzate nel modello RAM sono di tipo aritmetiche, istruzioni di input/output, istruzioni di indirizzamento indiretto, istruzioni di salto condizionato.

Ad esempio si potrebbero utilizzare le istruzioni riportate nella Tabella **??**.

Un *operando* può avere una delle forme seguenti:

- valore: $= i$, significa che l'operando va preso per l'intero in se;
- indirizzamento diretto (o assoluto): i, significa che l'intero deve essere considerato il valore contenuto nel registro r_i,
- indirizzamento indiretto (o relativo): $*i$, significa che l'intero deve essere considerato l'indirizzo del registro $j = r_i$ da usare come indirizzo del registro che contiene il valore, ovvero r_j, se $j < 0$ la macchina si ferma.

Questo tipo di istruzioni corrispondono bene alle itruzioni che si trovano nei linguaggi di tipo assembler.

Il significato, o semantica, di un programma per RAM può facilmente essere descritto con l'aiuto di una coppia di funzioni $v : \mathbb{N} \to \mathbb{Z}$ la quale specifica il valore da associare ad un certo operando, e una funzione $c : \mathbb{Z} \to \mathbb{Z}$, che specifica il valore da assegnare o leggere tramite una certa istruzione.

Quindi abbiamo che per ogni operando a si ha che

- se $a = i$ allora $v(i) = c(i)$;
- se $a = *i$ allora $v(*i) = v(c(i))$;
- se $a = (= i)$ allora $v(= i) = i$.

ESEMPIO 2. *Criterio di divisibilitá per 9: il programma di questo esempio legge sul nastro di input una sequenza $(\alpha_d, \alpha_{d-1}, \ldots, \alpha_0)$ di cifre intere, terminate da -1, e valuta se l'intero che ha $\alpha_d\alpha_{d-1}\ldots\alpha_0$ come rappresentazione decimale è divisibile per 9 o meno, scrivendo in output 1 se lo è e 0 altrimenti.*

Il programma che segue fa uso di 3 registri della RAM: r_1 contiene ad ogni passo di esecuzione la classe delle cifre lette fino a quel momento, r_2 contiene l'ultima cifra letta, mentre r_3 contiene una copia della somma tra la classe calcolata e la nuova cifra letta, infatti la somma modulo 9 viene realizzata sottraendo il valore 8 dalla somma e controllando che il valore ottenuto sia maggiore di 0 o meno in caso affermativo viene ulteriormente sottratto 1 per ottenere la classe (ad es. classe 3 e cifra 7, si calcola 7+3=10, si sottrae 8, 10-8=2, poichè il valore è positivo si sottrae 1 per ottenere la classe), in caso negativo si ripristina il valore che era stato preventivamente memorizzato in r_3.

```
        LOAD       =0
        STORE       1
        READ        2
<ciclo> LOAD        2
        ADD        =1
        JZERO  <fine>
        SUB        =1
        ADD         1
        STORE       3
        SUB        =8
        JGTZ   <class>
        LOAD        3
        STORE       1
        JMP    <ciclo>
<class> SUB        =1
        STORE       1
        JMP    <ciclo>
<fine>  LOAD        1
        JZERO  <div>
        WRITE      =0
        HALT
<div>   WRITE      =1
        HALT
```

ESEMPIO 3. *Scrivere il programma per RAM che calcola la seguente funzione:*

$$f(n) = \begin{cases} n^n & \text{se } n \geq 1 \\ 0 & \text{altrimenti} \end{cases}$$

A differenza del caso precedente supponiamo di trovare l'intero nella prima cella del nastro di input e quindi di leggere un solo intero.

In questo caso abbiamo bisogno di iterare per n volte l'operazione di moltiplicazione tra interi, utilizzeremo allora 3 registri: r_1 conterrà il numero di iterazioni

mancanti, r_2 conterrà la potenza di n calcolata fino a quel momento r_3 conterrà una copia di n da usare come fattore nei prodotti.

Una verifica iniziale serve a trattare il caso in cui il valore letto n fosse minore o uguale di 0:

```
        READ        3
        LOAD        3
        JGTZ   <main>
        WRITE      =0
        HALT
<main>  STORE       2
        SUB        =1
<ciclo> STORE       1
        LOAD        2
        MULT        3
        STORE       2
        LOAD        1
        SUB        =1
        JGTZ   <ciclo>
        WRITE       2
        HALT
```

Il codice C corrispondente al suddetto programma sarebbe:

```
#include <stdio.h>
main()
 {
   int r1,r2,r3;

   scanf("%d",&r3);

   if (r3>0)
     {
       r2=r3;
       for(r1=r3-1;r1>0;r1--)
                        r2 = r2 * r3;
       printf("%d",r2);
       exit(1);
     }
   else
     {
      printf("0");
      exit(0);
     }
 }
```

Esempio 4. *Considerare la macchina RAM che decide tutte le parole su un alfabeto di input $A = \{0, 1\}$, composte dello stesso numero di* 0 *e* 1 *(ad esempio* 011001 *è accettata mentre* 01111100 *non lo è).*

Il programma legge un carattere e nel secondo registro dovrebbe memorizzare la differenza tra il numero di 0 *letti e il numero di* 1 *letti.*

Quando viene incontrato un marcatore di fine input (ad esempio -1*), allora il programma legge la differenza e stampa la risposta* 1 *se la differenza è* 0*, e* 0 *altrimenti.*

```
        LOAD       =0
        STORE       2
        READ        1
<ciclo> LOAD        1
    ADD        =1
        JZERO  <esci>
        LOAD        1
        SUB        =1
        JZERO  <one>
        LOAD        2
        SUB        =1
        STORE       2
        JUMP   <endif>
<one>   LOAD        2
        ADD        =1
        STORE       2
<endif> READ        1
        JUMP   <ciclo>
<esci>  LOAD        2
        JZERO  <print>
    WRITE      =0
        HALT
<print> WRITE      =1
        HALT
```

A questo punto il modello RAM appare molto più potente degli automi finiti e dunque sorge naturale la questione se rientri nei casi formalizzati dalla nostra nozione di algoritmo formale.

La codifica di un particolare programma per RAM nel formalismo dei sistemi di transizione risulta alquanto lungo e noioso affronteremo per ora alcune istruzioni lasciando al lettore ls parte rimanente della verifica.

Come prima cosa ci restringiamo ai problemi di decisione, quindi supporremo di avere un programma che decide un certo problema su un alfabeto finito A opportunamente codificato da interi non negativi e con -1 come marcatore di fine parola.

Osservazione: già a questo punto abbiamo tralasciato di considerare il problema che ogni cella di input può contenere un intero arbitrario; in questo modo la codifica di un intero arbitrario o necessita di una sequenza di cifre di lunghezza arbitraria (cf. il commento alla def. [**?**]) oppure necessita di un alfabeto infinito. Nel seguito supporremo risolti questi problemi supponendo di avere un linguaggio finito A che codifica gli interi, e un linguaggio finito P per codificare i programmmi per la RAM.

Lo spazio delle configurazioni è dato da

$$Conf = \{(w_i, w_p, c, i, f) \text{ dove } w_p \in P,\ w_i \in A^*\ c, i \in \mathbb{N},\ f : \mathbb{N} \to \mathbb{Z}\}$$

w_p e' la stringa che rappresenta il programma; w_i rappresenta l'input c è il contatore di programma, i è il puntatore alla cella di input corrente, f è la funzione che

rappresenta lo stato corrente della memoria.

La funzione di ingresso, associa alla coppia input-programma (w_i, w_p) la configurazione iniziale

$$\varepsilon((w_i, w_p)) = (w_i, w_p, 1, 1, f)$$

con f funzione arbitraria (ad esempio si può prendere la funzione costante a 0.

La funzione di transizione $\tau : C \to C$ deve modellare ogni possibile istruzione del programma, dunque in qualche modo deve prevedere un caso per ogni possibile interpretazione delle istruzioni del programma:

$$\tau((w_i, w_p, c, i, f)) = (w_i, w_p, c', i', f')$$

dove

- Se $w_p(c) =$ `READ x` allora
  - $c' = c + 1$,
  - $i' = i + 1$,
  - $f'(n) = \begin{cases} f(n) & \text{se } n \neq x \\ w_i(i) & \text{se } n = x \end{cases}$
- Se $w_p(c) =$ `READ *x` allora
  - $c' = c + 1$,
  - $i' = i + 1$,
  - $f'(n) = \begin{cases} f(n) & \text{se } n \neq f(x) \\ w_i(i) & \text{se } n = f(x) \end{cases}$
- Se $w_p(c) =$ `LOAD =x` allora
  - $c' = c + 1$,
  - $i' = i$,
  - $f'(n) = \begin{cases} f(n) & \text{se } n \neq 0 \\ x & \text{se } n = 0 \end{cases}$
- Se $w_p(c) =$ `LOAD x` allora
  - $c' = c + 1$,
  - $i' = i$,
  - $f'(n) = \begin{cases} f(n) & \text{se } n \neq 0 \\ f(x) & \text{se } n = 0 \end{cases}$
- Se $w_p(c) =$ `LOAD *x` allora
  - $c' = c + 1$,
  - $i' = i$,
  - $f'(n) = \begin{cases} f(n) & \text{se } n \neq 0 \\ f(f(x)) & \text{se } n = 0 \end{cases}$

  (...continuare la codifica per esercizio)

Dopo avere fatto la verifica, dovrebbe risultare chiaro che tutte le istruzioni si mappano nell'algoritmo formale. D'altra parte nasce l'esigenza di avere una idea piú precisa della nozione di calcolo effettivo. Il modello della sezione precedente (gli automi finiti) sembrano richiedere molto meno risorse di calcolo, ed è chiaro che la macchina RAM può simulare un automa finito. D'altra parte sembra difficile potere credere che ogni programma per RAM si possa "tradurre" in un automa finito che computa lo stesso algoritmo.

A questo punto nasce la problematica di quale modello di calcolo sia più potente, inteso nel senso della espressivitá (ovvero in base a quanti e quali algoritmi

permette di codificare). É legittimo inoltre rilassare la nozione di automa finito per catturare una classe più generale di algoritmi.[4]

Allo scopo di ottenere un modello vicino alla RAM per potere espressivo, nella prossima sezione, rilasseremo la nozione di automa finito in due direzioni:

- permettere di leggere un carattere dalle celle di input, ma anche di scrivere;
- permettere di muoversi in tutte le direzioni lungo la parola.

[4]L'Esempio **??** costituisce problema decidibile per RAM ma non decidibile per automa finito.

Parte 2

I linguaggi funzionali

Parte 3

La programmazione Object Oriented

Parte 4

Esercizi

CAPITOLO 2

Esercizi

1. Algoritmi Formali

ESERCIZIO 2. *Definire l'algoritmo formale costante a zero che per ogni intero $x \in \mathbb{Z}$ restituisce il valore zero.*

ESERCIZIO 3. *Definire l'algoritmo formale* successore *che per ogni intero $x \in \mathbb{Z}$ restituisce la rappresentazione di $x + 1$.*

ESERCIZIO 4. *Definire l'algoritmo formale* proiezionem k-esima *che per ogni n-upla di interi $(x_1, x_2, \ldots, x_n)$ restituisce il k-esimo elemento x_k.*

ESERCIZIO 5. *Definire gli algoritmi formali per il calcolo delle operazioni aritmetiche:*

$$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \quad \textit{(somma)}$$
$$\cdot : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \quad \textit{(prodotto)}$$

ESERCIZIO 6. *Dati due algoritmi formali*

$$F_1 : \begin{cases} \epsilon_1 : A^\star \to C_1 \\ \tau_1 : C_1 \to C_1 \\ \delta_1 : C_1 \to B^\star \end{cases}$$

ed

$$F_2 : \begin{cases} \epsilon_2 : B^\star \to C_2 \\ \tau_2 : C_2 \to C_2 \\ \delta_2 : C_2 \to D^\star \end{cases}$$

Definire la composizione dei due algoritmi come

$$F_2 \circ F_1 : \begin{cases} \epsilon_1 : A^\star \to C_1 \cup^\cdot C_2 \\ \tau : C_1 \cup^\cdot C_2 \to C_1 \cup^\cdot C_2 \\ \delta : C_1 \cup^\cdot C_2 \to D^\star \end{cases}$$

dare le funzioni ϵ, τ e δ in modo che per ogni $x \in X$ se la computazione $F_1(x) = y$ e $F_2(y) = z$ allora $F_2 \circ F_1(x) = F_2(F_1(x)) = z$.

ESERCIZIO 7. *Utilizzare gli esercizi precedenti per costruire un algoritmo che calcola il valore di un polinomio $P(x) \in \mathbb{Z}[x]$ in corrispondenza di un dato $x_0 \in \mathbb{Z}$.*

ESERCIZIO 8. *Definire un automa finito definito su un alfabeto con i quattro simboli*

$$\mathcal{A} = \{t, f, \to, \equiv\}$$

accetta tutte e sole le seguenti parole di $\mathcal{A}^*$:

$$\begin{array}{llll} f & \to f & \equiv & t \\ f & \to t & \equiv & t \\ t & \to f & \equiv & f \\ t & \to t & \equiv & t \end{array}$$

(tabella di verità dell'implicazione).

ESERCIZIO 9. *Dato l'automa finito sull'insieme degli stati*

$$Q = \{q_0, q_1, q_2, q_3\}$$

con q_0 stato iniziale, q_3 unico stato accettante e funzione di transizione specificata dalla tabella seguente:

$(q_0, \mathtt{k})$	q_3
$(q_1, \mathtt{a})$	q_1
$(q_1, \mathtt{b})$	q_2
$(q_2, \mathtt{a})$	q_0
$(q_2, \mathtt{n})$	q_3
$(q_3, \mathtt{c})$	q_1

Trovare almeno tre parole accettate dall'automa.

Soluzione. Ad esempio: $w_1 = \mathtt{k}$, $w_2 = \mathtt{kcbn}$, e $w_3 = \mathtt{kcaaaabak}$. □

2. Macchine di Turing

Attenzione: una macchina di Turing resta definita una volta che siano stati fissati l'alfabeto A, un insieme finito di stati Q e la tabella delle transizioni

$$M : Q \times \tilde{A} \to \tilde{Q} \times \tilde{A} \times \{+1, -1\},$$

quindi se non è altrimenti richiesto bisogna definire la funzione M esplicitamente.

ESERCIZIO 10. *Definire una macchina di Turing che presa una parola costruita sull'alfabeto contenente il solo simbolo 0: $A = \{0\}$ restituise l'intero $|w|$ (lunghezza di w) rappresentato in base 2 sull'alfabeto binario $B = \{0, 1\}$.*

Valutare il tempo di arresto $T(n)$ della macchina trovata.

ESERCIZIO 11. *Definire una macchina di Turing che presa una parola costruita sull'alfabeto binario $A = \{0, 1\}$ che rappresenta la codifica binaria di un intero n restituisce la rappresentazione binaria del successore $n + 1$.*

Valutare il tempo di arresto $T(n)$ della macchina trovata.

ESERCIZIO 12. *Sia l'insieme $A = \{0, 1\}$ e sia*

$$X_0 = \{0^p 1^p | p \in \mathbb{N}\} \subset A^*$$

ci proponiamo di studiare la complessità dell'insieme X_0.

Rispondere ai seguenti quesiti:

1. *Stabilire l'insieme di stati Q e descrivere una macchina di Turing che decide X_0.*
2. *Stabilire un limite superiore della complessità per X_0.*
3. *Considerare il seguente lemma:*

   LEMMA 1. *Se $X \in DTIME_1^A(cn)$ allora esiste k per cui per ogni $w \in X$ se $|w| \geq k$ esistono $u, x, v \in A^*$ per cui $w = uxv$ e $|ux| \leq k$ con x parola non vuota, per cui per ogni $m \in \mathbb{N}$ si ha $ux^m v \in X$.*

   Dimostrare che per ogni $c > 0$ si ha

   $$X_0 \notin DTIME_1^A(cn).$$

   (trovare l'assurdo con l'aiuto del lemma).

(4) Dimostrare che esiste c per cui

$$X_0 \in DTIME_1^A(cn \log_2 n).$$

(senza scrivere esplicitamente la tabella di transizione descrivere una macchina che converte la sequenza di 0 nella sua rappresentazione binaria, e che utilizza la rappresentazione trovata per "contare" gli 1, etc...)

3. Funzioni Ricorsive

ESERCIZIO 13. *(1) Dimostrare che la funzione* $\texttt{sqrt} : \mathbb{N} \to \mathbb{N}$ *definita come*

$$\texttt{sqrt}(x) = \begin{cases} y & \text{se } x = y^2 \\ \perp & \text{negli altri casi} \end{cases}$$

che computa la radice quadrata ristretta ai naturali è una funzione ricorsiva.

(2) Dimostrare che la funzione $\texttt{mod} : \mathbb{N}^2 \to \mathbb{N}$ *che computa* $\texttt{mod}(n, p)$*, la classe resto di n modulo p, è una funzione ricorsiva.*

(3) (facoltativo) Dimostrare che la funzione $\texttt{prime} : \mathbb{N} \to \mathbb{N}$ *che computa* $\texttt{prime}(n) = p$ *l'n-esimo numero primo è una funzione ricorsiva (ad esempio utilizzando il punto precedente).*

Soluzione. A partire dalla ricorsività della funzione `mod` e dalla funzione `iszero` (funz. caratteristica dell'insieme $\{0\}$) si può ricavare quella della funzione $\texttt{divide}(\texttt{x}, \texttt{y})$ dove

$$\texttt{divide}(x, y) = \texttt{iszero}(\texttt{mod}(y, x)) = \begin{cases} 1 & \text{se } x \text{ divide } y \\ 0 & \text{in tutti gli altri casi} \end{cases}$$

□

ESERCIZIO 14. *Un numero reale* ρ *si dice* ricorsivo *se esistono due funzioni ricorsive* f *e* g *per le quali, per ogni naturale* n *si ha*

$$\left| |\rho| - \frac{f(n)}{g(n)} \right| \leq \frac{1}{n}$$

nel qual caso si dirà che la coppia f *e* g *approssima* ρ*.*

Verificare che, per un qualsiasi ρ *ricorsivo e per ogni* $n_0 \neq 0$ *se*

$$0 \leq \rho \leq \frac{1}{n_0}$$

allora si può scegliere l'approssimazione in modo che sia $f(m) = 0$ *e* $g(m) = 1$ *per* $m = 0, \ldots, n_0$*.* **Soluzione.** Per l'ipotesi

$$0 \leq \rho \leq \frac{1}{n_0}$$

abbiamo che

$$0 \leq \rho \leq \frac{1}{n_0} \leq \frac{1}{n_0 - 1} \frac{1}{n_0 - 2} \leq \cdots \leq 1$$

quindi f e g come nel testo danno un'approssimazione di ρ per tutti gli $n \leq n_0$, essendo

$$\left| |\rho| - \frac{f(n)}{g(n)} \right| = \left| \rho - \frac{0}{1} \right| = \rho \leq \frac{1}{n}$$

per $n \leq n_0$.

Resta da far vedere che se esiste un'approssimazione di ρ allora se ne può trovare sempre una soddisfa i requisiti. Per ipotesi f e g sono funzioni ricorsive quindi

$$f'(n) = \chi_{\geq}(n, n_0)f(n) \qquad g'(n) = \chi_{\geq}(n, n_0)g(n) + \chi_{\leq}(n, n_0)$$

dove f' e g' sono funzioni ricorsive per composizione di funazioni ricorsive. □

ESERCIZIO 15. *Sia X la più piccola classe di funzioni parziali $f : \mathbb{N}^k \to \mathbb{N}$ contenente le seguenti funzioni*

- *la funzione costante a zero,*
- *il successore* $\texttt{succ} : \mathbb{N} \to \mathbb{N}$ *tale che* $s(n) = n + 1$,
- *la moltiplicazione* $\texttt{mult} : \mathbb{N}^2 \to \mathbb{N}$, *tale che* $\texttt{mult}(n, m) = n\,m$;
- *l'addizione* $\texttt{add} : \mathbb{N}^2 \to \mathbb{N}$ *tale che* $\texttt{add}(n, m) = n + m$;
- *la funzione caratteristica della relazione di minore o uguale*

$$\chi_{\leq}(x, y) = \begin{cases} 1 & \text{se } x \leq y \\ 0 & \text{altrimenti} \end{cases}$$

- *le proiezioni* $\Pi^k_n : \mathbb{N}^k \to \mathbb{N}$ *definite come* $\Pi^k_n(x_1, \ldots, x_n) = x_k$.

e chiusa per composizione e per operatore di minimizzazione μ.

Ricordiamo che una funzione f si ottiene per operatore μ da g e per notazione si scrive $f(x_1, \ldots, x_k) = \mu y g(x_1, \ldots x_k, y)$ se e solo se $f(x_1, \ldots, x_k) = y$ se per ogni $y' < y$ esiste $m \neq 0$ per cui $g(x_1, \ldots x_k, y') = m$ e $g(x_1, \ldots x_k, y) = 0$.

Dimostrare che la classe X coincide con la classe delle funzioni ricorsive.

ESERCIZIO 16. *Sia $A : \mathbb{N}^2 \to \mathbb{N}$ la funzione così definita*

$$A(x, y) = \begin{cases} y + 1 & \text{se } x = 0 \\ A(x - 1, 1) & \text{se } y = 0 \\ A(x - 1, A(x, y - 1)) & \text{altrimenti} \end{cases}$$

(1) programmare in ocaml la funzione $A(x, y)$;
(2) calcolare una tabella con i valori calcolati per $n!$ e $A(n, n)$ (utilizzare il programma);
(3) (falcoltativo) dimostrare ricorrendo alla definizione che $A(x, y)$ è una funzione ricorsiva.

4. Complessità

ESERCIZIO 17. **(*)** *Si consideri la seguente nozione modificata di macchina di Turing di alfabeto A:*

- *il nastro $f : \mathbb{Z} \to A$ nella configurazione iniziale può contenere una quantità arbitraria (eventualmente infinita) di simboli di A (non blank).*
- *per ogni configurazione iniziale del nastro la macchina termina.*

Dimostrare che allora esiste un n per cui per ogni configurazione iniziale si ha che per ogni configurazione c della computazione se $c = (f, p, q)$ allora $|p| \leq n$.

Soluzione. Fissiamo un macchina M modificata, senza perdere in generalità possiamo supporre che M sia una macchina speciale (macchina che si autoconfina al seminastro positivo). L'equivalenza è facilmente ottenibile per simulazione ripetendo la costruzione per l'equivalenza tra macchina a nastro e macchina a seminastro. e tra macchina speciale e macchina a seminastro.

Sia $f : \mathbb{N} \to A$ una configurazione iniziale del nastro con $||f||$ lo spazio di arresto di M a partire da f.

Sappiamo per assurdo che per ogni n esiste una configurazione f_n per cui $||f_n|| = k_n > n$. Allora possiamo costruire una configurazione $\tilde{f}$ per cui M non termina, infatti basta considerare la funzione periodica $\tilde{f} = f|^*_{[0,k_n]}$, e sostituire lo stato finale con uno stato di "continuazione" che riprenda la computazione M dallo stato iniziale nell'intervallo $[k_n + 1, 2K_n]$.

Questo contraddice evidentemente l'ipotesi di terminazione di M su tutte le configurazioni f. □

ESERCIZIO 18. *Considerare la funzione*

$$T'(n) = \begin{cases} n & \text{se } n \text{ è dispari} \\ \\ n^3 & \text{se } n \text{ è pari} \end{cases}$$

(1) Dimostrare che $T'(n)$ non è una funzione di complessità.

Soluzione. $T'(n)$ non è una funzione di complessità poiché non é crescente infatti per ogni n pari $T(n) > T(n+1)$, ad esempio per $n = 2$ $T(2) = 8$ e $T(3) = 3 < 8$. □

(2) Trovare la più piccola[1] funzione di complessità $T(n)$ che maggiora $T'(n)$.

Soluzione. Considerare

$$T(n) = \begin{cases} (n-1)^3 & \text{se } n \text{ è dispari} \\ \\ n^3 & \text{se } n \text{ è pari} \end{cases}$$

$T(n)$ è crescente: infatti, per ogni n_1, n_2 si ha che $n_2 = n_1 + k$ con $k > 1$ quindi bisogna mostrare che $n_1^3 \leq (n_1+k)^3$ oppure $n_1^3 \leq (n_1+k-1)^3$, il primo caso discende dalla crescenza del cubo, il secondo caso sempre dalla crescenza del cubo per $k > 1$ e per $k = 1$ poiché $n_1^3 = (n_2 - 1)^3$. □

(3) Mostrare quale tipo di inclusione sussiste tra le due classi

$$\mathrm{DTIME}(n^2) \subset \mathrm{DTIME}(T(n))$$

oppure

$$\mathrm{DTIME}(n^2) \supset \mathrm{DTIME}(T(n)).$$

Soluzione. Vale la prima inclusione, per questo basta mostrare che definitivamente si ha che $n^2 \leq T(n)$, questa disuguaglianza si riduce a

$$n^2 \leq n^3 \text{per } n \text{ pari}$$

che è banalmente verificata, mentre nel caso n dispari si riduce a

$$n^2 \leq (n-1)^3 = n^3 - 3n^2 + 3n - 1$$

che è verificata per tutti gli n dispari a partire da $n = 5$ in poi. □

ESERCIZIO 19. *(1) Dato un alfabeto finito A, e due insiemi $X_1, X_2 \subset A^*$, si definisca la seguente relazione:*

$$X_1 \leq_P X_2$$

se esiste una macchina di Turing M con tempo di arresto $T(n^k)$ per qualche k tale che M computa la funzione $\tau : A^* \to A^*$ e per ogni parola $w \in X_1$, $\tau(w) \in X_2$ (in tal caso si dice che X_1 si riduce a X_2 in tempo polinomiale).

[1] Considerare l'ordine puntuale $f_1 \leq f_2$ se e solo se per ogni x si ha $f_1(x) \leq f_2(x)$.

Dimostrare che $\leq_P$ costituisce un pre-ordine (relazione riflessiva e transitiva) sull'insieme delle parti $P(A^*)$.

Soluzione. La verifica della proprietà riflessiva $X \leq_P X$ infatti la funzione $\tau(w) = w$ (la funzione identità) è Turing-computabile in tempo costante (funzione di complessità $T(n) = n$).

Abbiamo poi la transitività della relazione di riducibilità polinomiale infatti, se τ_1 riduce X_1 in X_2 e τ_2 riduce X_2 in X_3 allora sappiamo che la funzione composta $\tau_2 \circ \tau_1$ è Turing computabile, e inoltre esisteranno k_1 per cui la computazione di $\tau_1(w_1)$ termina in tempo $|w_1|^{k_1}$ mentre e $\tau_2(w_2)$ termina in tempo $|w_2|^{k_2}$ quindi la computazione di $\tau_2(\tau_1(w_1))$ terminerà in tempo $(|w_1|^{k_1})^{k_2} = |w_1|^{k_1\, k_2}$, il che dimostra che la composizione è pure del tipo n^k per $k = k_1\, k_2$. □

(2) Si definisca

$$EXP = \bigcup_{d,c} \mathrm{DTIME}_1^A(c^{n^d})$$

dimostrare che se $X_1 \leq_P X_2$ e $X_2 \in EXP$ allora anche $X_1 \in EXP$.

Soluzione. Poiché $X_1 \leq_P X_2$, esiste una macchina M che computa $\tau(w)$ in tempo $|w|^k$ inoltre $\tau(w) \in X_2$ è decidibile in tempo $c^{(|\tau(w)|^d)}$ per l'appartenenza di X_2 ad EXP quindi X_1 può essere deciso componendo la macchina che computa τ e quella che decide X_2 in tempo $c^{(|w|^k)^d} = c^{|w|^{k\, d}}$ e quindi $X_1 \in \mathrm{DTIME}(c^{n^{d'}})$ con $d' = k\, d$. □

Esercizio 20. *Una macchina di Turing si dice avere spazio di arresto logaritmico $S(n)$ se soddisfa le seguenti condizioni:*

- *è una macchina a tre nastri (detti di input, di lavoro, di uscita),*
- *la macchina non modifica mai il contenuto del primo nastro, e nell'ultimo nastro puo' solo scrivere una sola volta per ogni cella in modo incrementale (si deve spostare a destra dopo avere scritto e non puó tornare indietro),*
- *il secondo nastro deve avere la proprietá di non andare mai a leggere o scrivere su una cella i con $|i| > S(n) = \log n$ se n è la lunghezza della parola di input.*

Dato un alfabeto finito A, e due insiemi $X_1, X_2 \subset A^*$, si definisca la seguente relazione:

$$X_1 \leq_L X_2$$

se esiste una macchina di Turing M con spazio di arresto logaritmico $S(n) = k \log n$ per qualche k tale che M computa la funzione $\tau : A^* \to A^*$ e per ogni parola $w \in X_1$, $\tau(w) \in X_2$ (in tal caso si dice che X_1 si riduce a X_2 in spazio logaritmico).

- *Dimostrare che $\leq_L$ costituisce un pre-ordine (relazione riflessiva e transitiva) sull'insieme delle parti $P(A^*)$.*
- *Definito*

$$LOGSPACE = \bigcup_{c \geq 0} DSPACE(c \log n)$$

dire (giustificando formalmente la risposta) se è vero che se $X_1 \leq_L X_2$ e $X_2 \in LOGSPACE$ allora anche $X_1 \in LOGSPACE$.

5. Lambda Calcolo

ESERCIZIO 21. *Sia K^∞ un punto fisso chiuso del lambda termine $K = \lambda x \lambda y x$, ovvero un termine t_0 tale che*

$$(K)t_0 \simeq_\beta t_0$$

Dimostrare che:

a) per ogni termine u si ha $(K^\infty)u \simeq_\beta K^\infty$,
b) K^∞ non è un termine risolubile,
c) non esiste una relazione di equivalenza $\sim$ non banale che contiene la β equivalenza, tale che $I \sim K^\infty$ e che passa al contesto,
d) esiste una relazione di equivalenza $\sim$ non banale che contiene la β equivalenza, tale che $u \sim K^\infty$ per u termine non risolubile, e che inoltre passa al contesto.

ESERCIZIO 22. *Disegnare l'albero sintattico e l'albero di Böhm (se esiste) per i seguenti lambda termini:*

(1) $(c)((\lambda x \lambda y (y) y x) \lambda z z) v$,
(2) $\lambda x \lambda y (m)(\lambda z z) y z$,
(3) $(r)\lambda y (y) x r$,
(4) $\lambda x (y) x) \lambda p q$.

ESERCIZIO 23. *Utilizzando la definizione di alpha equivalenza mostrare che $\lambda x (p) \lambda z (p) z =_\alpha \lambda x (p) \lambda q (p) q$.*

ESERCIZIO 24. *Ricordando la definizione dei numerali di Church (la rappresentazione degli interi nel lambda calcolo) $\underline{n} = \lambda f \lambda x (f)^n x$.*
Dare i lambda termini che rappresentano le seguenti funzioni:

(1) la funzione predecessore

$$\texttt{pred}(n) = \begin{cases} 0 & \text{se } n = 0 \\ n - 1 & \text{altrimenti} \end{cases}$$

Soluzione. L'idea per costruire il predecessore $n-1$ consiste nell'utilizzare il lambda termine $\lambda f \lambda x (f)^n x$ che rappresenta n per iterare una funzione Φ che a partire da una coppia di numerali (i, j) costruisce la coppia $(i+1, i)$, in questo modo a partire da un termine t_0 che rappresenta la coppia $(0, 0)$, si ha che

$$(\underline{n})\Phi t_0$$

si riduce alla coppia $(n, n-1)$ da cui si estrae il predecessore selezionando la seconda componente.

Per questo abbiamo bisogno di rappresentare una coppia e le due proiezioni. Infatti la coppia (a, b) si rappresenta con il lambda termine $\lambda z (z) a b$ e il primo e il secondo elemento della coppia viene selezionato dai termini $\texttt{true} = \lambda x \lambda y x$ e $\texttt{false} = \lambda x \lambda y y$; infatti $(\lambda z (z) a b)\texttt{true} \to_{\beta_0} (\texttt{true}) a b = (\lambda x \lambda y x) a b \to_{\beta_0} a$.

Il termine Φ si ottiene costruendo la coppia formata dal successore del primo elemento e dal primo elemento ovvero, se il parametro p rappresenta una coppia:

$$\lambda p \lambda z ((z)(\texttt{succ})(p)\texttt{true})(p)\texttt{true},$$

e $t_0 = \lambda z (z) \underline{0}\underline{0}$.

Quindi

$$\texttt{pred} = \lambda n(((n)\Phi)t_0)\texttt{false}.$$

Ricordiamo infine che la funzione successore di un intero di Church si può rappresentare con il lambda termine:

$$\texttt{succ} = \lambda n\lambda f\lambda x(f)(n)fx.$$

□

(2) la funzione esponenziale

$$\texttt{exp}(n) = 2^n$$

Soluzione. La funzione esponenziale si ottiene per applicazione diretta di due interi tra loro, ovvero se $\underline{n}$ ed $\underline{m}$ sono i due lambda termini abbiamo che $(\underline{n})\underline{m} = \underline{m^n}$, infatti

$$\begin{aligned}
(\underline{n})\underline{m} = \quad & (\lambda f\lambda x(f)^n x)\underline{m} \to_{\beta_0} \\
& \lambda x(\lambda s\lambda p(s)^m p)^n x \to_{\beta_0} \\
& \lambda x(\lambda s\lambda p(s)^m p)^{n-1}\lambda p(x)^m p \to_{\beta_0} \\
& \lambda x(\lambda s\lambda p(s)^m p)^{n-2}(\lambda s\lambda p(s)^m p)\lambda p(x)^m p \to_{\beta_0} \\
& \lambda x(\lambda s\lambda p(s)^m p)^{n-2}\lambda p_1(\lambda p(x)^m p)^m p_1 \to_{\beta} \\
& \lambda x(\lambda s\lambda p(s)^m p)^{n-2}\lambda p_1(x)^{m^2} p_1 \to_{\beta_0} \\
& \vdots \\
& \lambda x(\lambda s\lambda p(s)^m p)\lambda p_1(x)^{m^{n-1}} p_1 \to_{\beta} \\
& \lambda x\lambda p_1(x)^{m^n} p_1 = \underline{m^n}
\end{aligned}$$

□

(3) la funzione divisione per 2

$$\texttt{half}(n) = \lfloor n/2 \rfloor$$

Soluzione. Ci ispiriamo alla costruzione fatta per il predecessore solo che questa volta iteriamo tramite $\underline{n}$ una funzione Ψ che ha il seguente comportamento: la funzione legge una coppia (i, j) e se la seconda componente j della coppia è diverso da zero restituisce una coppia che ha per elementi $(i-1, j-2)$ altrimenti se $j = 0$ non fa niente e restituisce direttamente $(i, 0)$.

Una siffatta funzione si definisce facilmente a partire da una funzione $\texttt{iszero}(n)$ che si valuta in `true` se n è uguale a zero e in `false` se n è diverso da zero:

$$\texttt{iszero} = \lambda n(((n)\texttt{false})\lambda x((x)\texttt{false})\texttt{true})\texttt{false}$$

allora

$$\Psi = \lambda p(((\texttt{iszero})(p)\texttt{false})p)(\Phi)p$$

dove $\Phi = \lambda p\lambda z((z)(\texttt{pred})(p)\texttt{true})(\texttt{pred})(\texttt{pred})(p)\texttt{true}$.

Allora per un dato numerale di Church abbiamo che

$$\begin{aligned}
(\underline{n})\Psi\lambda z(z)\underline{n}\,\underline{n} \to_\beta \quad & (\Psi)^n\lambda z(z)\underline{n}\,\underline{n} \to_\beta \\
& (\Psi)^{n-1}(\Psi)\lambda z(z)\underline{n}\,\underline{n} = \\
& (\Psi)^{n-1}((((\texttt{iszero})n)\lambda z(z)\underline{n}\,\underline{n})(\Phi)\lambda z(z)\underline{n}\,\underline{n} \to_\beta \\
& (\Psi)^{n-1}(((\texttt{false})\lambda z(z)\underline{n}\,\underline{n})(\Phi)\lambda z(z)\underline{n}\,\underline{n} \to_\beta \\
& (\Psi)^{n-1}(\Phi)\lambda z(z)\underline{n}\,\underline{n} \to_\beta \\
& (\Psi)^{n-1}\lambda z(z)\underline{n-1}\,\underline{n-2} \to_\beta \\
& \vdots \\
& (\Psi)^{n/2}\lambda z(z)\underline{n/2}\,\underline{0} = \\
& (\Psi)^{n/2-1}(\Psi)\lambda z(z)\underline{n/2}\,\underline{0} \to_\beta \\
& (\Psi)^{n/2-1}\lambda z(z)\underline{n/2}\,\underline{0} \to_\beta \\
& \vdots \\
& \lambda z(z)\underline{n/2}\,\underline{0} \to_\beta
\end{aligned}$$

per cui $\texttt{half} = \lambda n(((n)\Psi)\lambda z(z)nn)\texttt{true}$. □

(4) *la funzione che stabilisce se un intero è pari:*

$$\texttt{even}(n) = \begin{cases} \texttt{true} & \textit{se } n \textit{ è pari} \\ \texttt{false} & \textit{altrimenti} \end{cases}$$

(si ricorda la che la rappresentazione standard di `true` *è* $\lambda x\lambda yx$ *mentre quella di* `false` *è* $\lambda x\lambda yy$*).*

Soluzione. Consideriamo la funzione booleana `not` definita come $\lambda x((x)\texttt{false})\texttt{true}$. Allora la funzione richiesta si ottiene iterando proprio la funzione `not`: $\texttt{even} = \lambda b((b)\texttt{not})\texttt{true}$.

Allora se $n = 0$

$$\begin{aligned}
(\texttt{even})\underline{0} = \quad & (\lambda b((b)\texttt{not})\texttt{true})\underline{0} \to_\beta \\
& ((\lambda f\lambda xx)\texttt{not})\texttt{true}) \to_\beta \texttt{true}
\end{aligned}$$

altrimenti se $n \neq 0$

$$\begin{aligned}
(\texttt{even})\underline{n} = \quad & (\lambda b((b)\texttt{not})\texttt{true})\underline{n} \to_\beta \\
& ((\lambda f\lambda x(f)^n x)\texttt{not})\texttt{true}) \to_\beta \\
& (\texttt{not})^{n-1}(\texttt{not})\texttt{true} \to_\beta \\
& (\texttt{not})^{n-2}(\texttt{not})\texttt{false} \to_\beta \\
& (\texttt{not})^{n-3}(\texttt{not})\texttt{true} \to_\beta \\
& \vdots
\end{aligned}$$

□

(5) *utilizzando le funzioni successore* `succ`*,* `half` *e* `even` *definite sugli interi di Church, programmare il lambda termine che stabilisce se la rappresentazione binaria di un intero contiene un numero pari o dispari di uno.*

Soluzione. Ripeteremo una costruzione analoga a quella fatta per `half` in questo caso la funzione Ψ' prenderà una coppia (i, j) e se j è pari darà come risultato $(i, j/2)$ se invece eè dispari il risultato sarà $(i + 1, \lfloor j/2 \rfloor)$.

Dunque come nei casi precedenti basterà iterare Ψ' per n volte a partire dalla coppia $(0, n)$.

Più precisamente consideriamo

$$\Psi' = \lambda p((\texttt{even})(p)\texttt{false})(\Phi_1)p)(\Phi_2)p$$

dove

$$\Phi_1 = \lambda p\lambda z((z)(\texttt{succ})(p)\texttt{true})(\texttt{half})(p)\texttt{false}$$

e

$$\Phi_2 = \lambda p\lambda z((z)(p)\texttt{true})(\texttt{half})(p)\texttt{false}.$$

La funzione cercata è allora

$$\texttt{numodd} = \lambda n((n)\Psi')\lambda z(z)\underline{0}\, n.$$

□

(programmare il lambda termine sia utilizzando l'operatore di punto fisso (Y) sia senza fare ricorso al costrutto della ricorsione o all'operatore di punto fisso).

ESERCIZIO 25. *(*) La definizione dei numerali di Church corrisponde (nel lambda calcolo) alla rappresentazione* unaria *dei numeri interi, in questo modo la complessità sintattica della rappresentazione dell'intero n è lineare, di ordine n.*

Dare una rappresentazione più efficace per gli interi in modo che da ottenere una complessità sintattica simile a quella della rappresentazione binaria (dell'ordine di $\log n$).

ESERCIZIO 26. *Effettuare le seguenti sostituzioni nell'insieme dei lambda termini quozientato rispetto alla α-equivalenza, ed eventualmente normalizzare i termini ottenuti:*

(1) $\lambda x(y)x[(m)n/y]$,
(2) $\lambda x(x)yx[\lambda x(x)x/y]$,
(3) $((x)y)(u)v[\lambda xx/u]$,
(4) $(\lambda x(y)x)[(c)b/y]$,
(5) $((x)\lambda y(x)y)[b/x]$,
(6) $((\lambda x(x)y)z)[(z)z/y]$,
(7) $((\lambda x\lambda y(z)y)z)[(x)z/z]$,
(8) $((\lambda m\lambda n(n)mm)\lambda x(x)x)[\lambda xx/n]$.

ESERCIZIO 27. *Programmare nel lambda calcolo gli operatori logici* `and`, `or`, `xor`, `not` *utilizzando la rappresentazione usuale dei booleani (vedi esercizio* ??.??*).*

Soluzione. Il lambda termine che corrisponde all'`and` è dato da $\lambda x\lambda y((x)y)(y)xy$, verifichiamo i quattro casi della tabella di verità per l'`and`, ricordando che $(\texttt{true})t \to_\beta \lambda yt$ per ogni termine t e che $(\texttt{false})t \to_\beta I$.

Allora

$$\begin{aligned}(\texttt{and})\texttt{true}\,\texttt{false} = \;& (\lambda x\lambda y((x)y)(y)xy)\texttt{true}\,\texttt{false} \to_\beta \\ & ((\texttt{true})\texttt{false})(\texttt{false})\texttt{truefalse} \to_\beta \\ & (\lambda y\texttt{false})(I)\texttt{false} \to_\beta \\ & (\lambda y\texttt{false})\texttt{false} \to_\beta \texttt{false}\end{aligned}$$

inoltre

$$\begin{aligned}(\texttt{and})\texttt{false}\,\texttt{false} = \;& (\lambda x\lambda y((x)y)(y)xy)\texttt{false}\,\texttt{false} \to_\beta \\ & ((\texttt{false})\texttt{false})(\texttt{false})\texttt{falsefalse} \to_\beta \\ & (I)(I)\texttt{false} \to_\beta \\ & (I)\texttt{false} \to_\beta \texttt{false}\end{aligned}$$

come pure

$$\begin{aligned}(\texttt{and})\texttt{false}\,\texttt{true} = \quad & (\lambda x\lambda y((x)y)(y)xy)\texttt{false}\,\texttt{true} \to_\beta \\ & ((\texttt{false})\texttt{true})(\texttt{false})\texttt{truefalse} \to_\beta \\ & (I)(I)\texttt{false} \to_\beta \\ & (I)\texttt{false} \to_\beta \texttt{false}\end{aligned}$$

e infine

$$\begin{aligned}(\texttt{and})\texttt{true}\,\texttt{true} = \quad & (\lambda x\lambda y((x)y)(y)xy)\texttt{true}\,\texttt{true} \to_\beta \\ & ((\texttt{true})\texttt{true})(\texttt{true})\texttt{truetrue} \to_\beta \\ & (\lambda y\texttt{true})(\lambda y\texttt{true})\texttt{true} \to_\beta \\ & (\lambda y\texttt{true})\texttt{true} \to_\beta \texttt{true}.\end{aligned}$$

In modo del tutto analogo si può verificare che il lambda termine $\lambda x\lambda y((x)x)y$ rappresenta la funzione booleana `or`.

Per quanto riguarda il `not` abbiamo che $\lambda z(z)\texttt{false}\,\texttt{true}$ e infatti

$$\begin{aligned}(\texttt{not})\texttt{true} = \quad & (\lambda z(z)\texttt{false}\,\texttt{true})\texttt{true} \to_\beta \\ & ((\texttt{true})\texttt{false})\texttt{true} \to_\beta \\ & (\lambda y\texttt{false})\texttt{true} \to_\beta \texttt{false}\end{aligned}$$

mentre

$$\begin{aligned}(\texttt{not})\texttt{false} = \quad & (\lambda z(z)\texttt{false}\,\texttt{true})\texttt{false} \to_\beta \\ & ((\texttt{flase})\texttt{false})\texttt{true} \to_\beta \\ & (I)\texttt{true} \to_\beta \texttt{true}.\end{aligned}$$

Infine, la funzione `xor` si ottiene come combinazione delle precedenti, ovvero

$$(\texttt{xor})xy = ((\texttt{or})((\texttt{and})(\texttt{not})x)y)((\texttt{and})(\texttt{not})y)x.$$

□

Verificare i termini trovati sulle seguenti espressioni:

$$((\texttt{or})(\texttt{iszero})\underline{0})(\texttt{not})(\texttt{iszero})\underline{1}$$

$$(\texttt{not})(\texttt{iszero})\underline{2}$$

ESERCIZIO 28. *Calcolare la forma normale (se esiste) dei seguenti lambda termini*

$$((\lambda f\,\lambda x\,(f)(f)x)\lambda x\,(x)x)\lambda x\,x$$

$$(((\lambda f\,\lambda x\,\lambda z\,(f)x(f)(f)z)\lambda x\,x)y)\lambda w\,\lambda z\,(w)(w)z$$

ESERCIZIO 29. *Considerare la più piccola relazione di equivalenza sull'insieme dei lambda-termini $\simeq_{op}$ che passa al contesto e tale che*

$$((\lambda xu)v)w \simeq_{op} (\lambda x(u)w)v \qquad \text{se } x \notin FV(w) \tag{7}$$

e

$$(\lambda x\lambda yu)v \simeq_{op} \lambda y(\lambda xu)v \qquad \text{se } y \notin FV(v). \tag{8}$$

a) Trovare due termini u e v chiusi e normali non op-equivalenti.

Soluzione. Ad esempio due termini chiusi normali non *op*-equivalenti sono il termine λxx e $\lambda f\lambda x(f)x$; d'altronde due termini in forma normale non possono essere *op*-equivalenti visto che solo in presenza di un redesso una delle due equazioni (1) o (2) può essere applicata.

□

b) Mostrare che che presi comunque due termini u e v, se $u \simeq_{op} v$ allora $u \simeq_\beta v$.

Soluzione. Consideriamo i due termini dell'equazione (1) allora abbiamo

$$((\lambda x u)v)w \simeq_\beta (u[v/x])w$$

d'altra parte

$$(\lambda x(u)w)v \simeq_\beta ((u)w)[v/x]$$

dall'ipotesi che $x \notin FV(w)$ abbiamo

$$((u)w)[v/x] = (u[v/x])w$$

e dunque per transitività

$$((\lambda x u)v)w \simeq_\beta (u[v/x])w \simeq_\beta (\lambda x(u)w)v$$

Per quando riguarda la seconda equazione si procede in modo analogo

$$(\lambda x \lambda y u)v \to_\beta (\lambda y u)[v/x] = \lambda y u[v/x]$$

la seconda relazione si applica perché y non è libera in v e

$$\lambda y(\lambda x u)v \to_\beta \lambda y u[v/x].$$

Quindi, come prima,

$$(\lambda x \lambda y u)v \simeq_\beta \lambda y u[v/x] \simeq_\beta \lambda y(\lambda x u)v$$

.

D'altra parte la *op*-equivalenza è definita come la più piccola relazione che passa al contesto e contiene (1) e (2). Dunque poichè anche la β-equivalenza passa al contesto e contiene le coppie di lambda termini in (1) e (2) allora ogni coppia di termini *op*-equivalenti sono anche β-equivalenti. □

c) Mostrare che se $u_1 \simeq_{op} v_1$ e $u_2 \simeq_{op} v_2$ allora

$$u_1[u_2/x] \simeq_{op} v_1[v_2/x]$$

Soluzione. Per induzione strutturale su u_1:

- Se u_1 è una variabile allora $u_1 = x$ quindi $v_1 = x$ e allora $u_1[u_2/x] = u_2$ e $v_1[v_2/x] = v_2$ per cui dall'ipotesi $u_2 \simeq_{op} v_2$ abbiamo $u_1[u_2/x] \simeq_{op} v_1[v_2/x]$. Se $u_1 = y \neq x$ allora sempre abbiamo che $v_1 = y$ e $u_1[u_2/x] = y$ come pure $v_1[v_2/x] = y$, e per riflessività la tesi.
- Se $u_1 = (t_1)t_2$ allora

□

d) Mostrare che se $u \simeq_{op} v$ allora $FV(u) = FV(v)$.

Considerare la riduzione $\to_{op}$ di lambda-termini definita orientando le equazioni **??** *e* **??** *da sinistra verso destra :*

a) Mostrare che $\to_{op}$ termina per ogni lambda termine.
b) Mostrare che $\to_{op}$ non ha la proprietà di Church-Rosser.

ESERCIZIO 30. *Siano u e v due termini del lambda calcolo, e sia la seguente notazione $\langle u, v\rangle = \lambda z(z)uv$ con z una variabile che non occorre libera in u e v.*

Un lambda termine t si dice, per definizione, una coppia*, se esistono due lambda termini u e v per cui*

$$t \simeq_\beta \langle u, v\rangle$$

a) Mostrare che $\langle u, v\rangle \simeq_\beta \langle u', v'\rangle$ *se e solo se* $u \simeq_\beta u'$ *e* $v \simeq_\beta v'$.

Soluzione. L'implicazione da $u \simeq_\beta u'$ e $v \simeq_\beta v'$ discende $\langle u, v\rangle \simeq_\beta \langle u', v'\rangle$ si ottiene per il fatto che la β equivalenza passa al contesto.

Per quanto riguarda l'altra implicazione basta osservare che

$$(\langle u, v\rangle)\texttt{true} \to_\beta u$$

mentre

$$(\langle u, v\rangle)\texttt{false} \to_\beta v$$

Quindi in modo analogo abbiamo che

$$(\langle u', v'\rangle)\texttt{true} \to_\beta u'$$

allora dal fatto che la beta equivalenza passa al contesto abbiamo che

$$(\langle u, v\rangle)\texttt{true} \simeq_\beta (\langle u', v'\rangle)\texttt{true}$$

e per transitività

$$u \simeq_\beta (\langle u, v\rangle)\texttt{true} \simeq_\beta (\langle u', v'\rangle)\texttt{true} \simeq_\beta u'$$

quindi la tesi $u \simeq_\beta u'$.

Analogamente per l'equivalenza $v \simeq_\beta v'$. □

b) Dare un esempio di lambda termine che non è una coppia.

Soluzione. λxx. Infatti ogni lambda-termine in forma normale non contiene nessun redesso su cui applicare la *op*-riduzione quindi ogni termine in forma normale forma una classe di equivalenza con se stesso. □

Siano F *e* G *due lambda termini in cui la variabile* x *non ha occorrenze libere e sia*

$$\Phi = \lambda x\langle((F)(x)\texttt{true})(x)\texttt{false}, ((G)(x)\texttt{true})(x)\texttt{false}\rangle$$

dove `true` *è* $\lambda x\lambda yx$ *e* `false` *è* $\lambda x\lambda yy$.

(1) Fissati u *e* v *per cui* $u \simeq_\beta (F)uv$ *e* $v \simeq_\beta (G)uv$, *mostrare che* $\langle u, v\rangle$ *è un punto fisso di* Φ *(si ricorda che un punto fisso è un termine* t_0 *per cui* $(\Phi)t_0 \simeq_\beta t_0$*).*

Soluzione. Ricordiamo che $(\langle u, v\rangle)\texttt{true} \to_\beta u$, infatti

$$\begin{array}{rl} (\langle u, v\rangle)\texttt{true} = & (\lambda z(z)uv)\texttt{true} \\ \to_\beta & (\texttt{true})uv = (\lambda x\lambda yx)uv \\ \to_\beta & u \end{array}$$

Calcoliamo allora $(\Phi)\langle u, v\rangle$

$$\begin{array}{rl} (\Phi)\langle u, v\rangle = & (\lambda x\langle((F)(x)\texttt{true})(x)\texttt{false}, ((G)(x)\texttt{true})(x)\texttt{false}\rangle)\langle u, v\rangle \to_\beta \\ \to_\beta & \langle((F)(\langle u, v\rangle)\texttt{true})(\langle u, v\rangle)\texttt{false}, ((G)(\langle u, v\rangle)\texttt{true})(\langle u, v\rangle)\texttt{false}\rangle \\ \simeq_\beta & \langle((F)u)v, ((G)u)v\rangle \\ \simeq_\beta & \langle u, v\rangle \end{array}$$

□

(2) Se t_0 *è punto fisso di* Φ *allora* t_0 *è una coppia. Inoltre, se* $u \simeq_\beta (t_0)\texttt{true}$ *e* $v \simeq_\beta (t_0)\texttt{false}$ *allora* $u \simeq_\beta (F)uv$ *e* $v \simeq_\beta (G)uv$.

Soluzione. La prima affermazione è molto semplice da far vedere, infatti se t_0 è un punto fisso di Φ allora sappiamo che è beta-equivalente a $(\Phi)t_0$ ovvero, per riduzione del redesso di testa,

$$t_0 \simeq_\beta \langle((F)(t_0)\texttt{true})(t_0)\texttt{false}, ((G)(t_0)\texttt{true})(t_0)\texttt{false}\rangle$$

Per quanto riguarda la seconda affermazione dal fatto che la β-equivalenza passa al contesto abbiamo che

$$(t_0)\texttt{true} \simeq (\langle((F)(t_0)\texttt{true})(t_0)\texttt{false}, ((G)(t_0)\texttt{true})(t_0)\texttt{false}\rangle)\texttt{true} \simeq_\beta (F)(t_0)\texttt{true})(t_0)\texttt{false}$$

e per ipotesi $((F)(t_0)\texttt{true})(t_0)\texttt{false} \simeq_\beta ((F)u)v$ e $(t_0)\texttt{true} \simeq_\beta u$, dunque

$$(t_0)\texttt{true} \simeq (F)uv$$

in modo analogo si ricava l'altra equivalenza. □

ESERCIZIO 31. *Si definisca l'insieme*

$$X = \bigcup_{k\in\mathbb{N}} X_k$$

dove

$$X_0 = \mathbb{N}$$
$$X_{k+1} = X_k \cup \{(\sigma_1, \ldots, \sigma_m) | m \in \mathbb{N}^*, \sigma_i \in X_k\}$$

Per ogni variabile x del lambda calcolo e per ogni $\sigma \in X$ definiamo i seguenti lambda-termini:

$$\varphi_\sigma(x) = \begin{cases} \lambda y_1 \ldots \lambda y_\sigma(x) y_\sigma \ldots y_1 & \sigma \in \mathbb{N} \\ \lambda y_1 \ldots \lambda y_n(x)\varphi_{\sigma_n}(y_n)\ldots\varphi_{\sigma_1}(y_1) & \sigma = (\sigma_1, \ldots, \sigma_n) \quad \sigma_i \in X \end{cases}$$

a) Calcolare il termine $\varphi_\sigma(x)$ dove $\sigma = ((2), ((1), 3))$.

Soluzione.

$$\varphi_\sigma(x) = \varphi_{((2),((1),3))}(x) = \lambda y_1 \lambda y_2(x)\varphi_{(2)}(y_2)\varphi_{((1),3)}(y_1)$$

Ora abbiamo che

$$\varphi_{(2)}(y_2) = \lambda y_1(y_2)\varphi_2(y_1) = \lambda y_1(y_2)\lambda z_1 \lambda z_2(y_1) z_2 z_1$$

□

b) Mostrare che per ogni coppia di elementi distinti $\sigma_1, \sigma_2 \in X$ abbiamo che

$$\varphi_{\sigma_1}(x) \neq \varphi_{\sigma_2}(x)$$

c) Definita per un lambda termine generico t

$$\varphi_\sigma(t) = \begin{cases} \varphi_\sigma(x) & \text{Se } t = x \text{ è una variabile} \\ (\varphi_\sigma(u))\varphi_\sigma(v) & \text{Se } t = (u)v \\ \lambda x'(\lambda x \varphi_\sigma(u))\varphi_\sigma(x') & \text{Se } t = \lambda x u \text{ con } x' \notin FV(u). \end{cases}$$

i) Dimostrare che $FV(t) = FV(\varphi_\sigma(t))$ per ogni $\sigma \in X$.

ii) Dimostrare che il numero di beta-redessi aggiunti in $\varphi_\sigma(t)$ rispetto a quelli presenti in t è linearmente limitato dalla lunghezza di t.

ESERCIZIO 32. *Calcolare le variabili libere e le variabili legate per ognuno dei seguenti lambda termini:*

(1) $\lambda x(y)x$,
(2) $(\lambda x(y)x)\lambda y(y)x$,
(3) $\lambda x(y)(\lambda z z)x$,
(4) $\lambda z(z)z$,
(5) $(x)\lambda c(x)(x)c$,
(6) $\lambda p \lambda q(r)q$.

6. Semantica del Lambda Calcolo

ESERCIZIO 33. *Un insieme (parzialmente) ordinato $(D, \leq)$ si dice* σ-completo *se ogni successione crescente $(d_n)_{n\in\mathbb{N}}$ di elementi in D ammette estremo superiore $d = \sup_{n\in\mathbb{N}} d_n$.*

Una funzione $f : D \to D$ si dice σ-continua *se per ogni successione $(d_n)_{n\in\mathbb{N}}$ crescente si ha*

$$f(\sup_{n\in\mathbb{N}} d_n) = \sup_{n\in\mathbb{N}} f(d_n).$$

Una funzione $f : D \to D$ si dice crescente *se per ogni $d_1, d_2 \in D$ se $d_1 \leq d_2$ allora $f(d_1) \leq f(d_2)$.*

Diremo che una funzione è σ-cc *se essa è σ-continua e crescente.*

a) Mostrare che preso $D = \{[a, b] | a \leq b \in \mathbb{R}\}$ con la relazione d'ordine $[a, b] \leq_D [a', b']$ se $[a', b'] \subseteq [a, b]$, questo costituisce un insieme σ-completo.

Soluzione. Infatti, per ogni successione di intervalli $I_n = [a_n, b_n]$ che risulti crescente abbiamo che $I_{n+1} \subseteq I_n$. E allora si ha che la successione degli a_n è una successione non decrescente limitata superiormente (ad esempio da b_0) quindi ammette limite, diciamo a. Analogamente la successione dei b_n è una successione non crescente limitata inferiormente (ad esempio da a_0) e quindi ammette estremo inferiore b.

Quindi $[a, b]$ rappresenta l'estremo inferiore della successione degli I_n. □

b) Dare un esempio di funzione $f : D \to D$ che non sia σ-cc.

Soluzione. Consideriamo la funzione

$$f([a, b]) = [a^2, b^2]$$

e i due intervalli $I_1 = [1/2, 1]$ e $I_2 = [-2/3, 2]$ allora $I_1 \subseteq I_2$ mentre $f(I_1) = [1/4, 1] = [0.25, 1]$ e $I_2 = [4/9, 4] \approx [0.44, 4]$.

Quindi $f(I_1) \not\subseteq f(I_2)$ e $f(I_2) \not\subseteq f(I_1)$. □

c) Sia ora $f_{n,k} : D \to D$ indotta dalla funzione $\phi_{n,k}(x) = \dfrac{x}{n} + k$ con $n, k \in \mathbb{N}$ ed $n \neq 0$, ponendo $f_{n,k}(I) = \{\phi_{n,k}(x) \,|\, x \in I\}$.

Dimostrare che per ogni $n, k \in \mathbb{N}$ con $n \neq 0$, la funzione $f_{n,k}$ è σ-cc.

Soluzione. Per dimostrare che la funzione è σ-cc bisogna fare vedere che preserva l'ordine su D e che

$$f(\sup_{n\in\mathbb{N}} d_n) = \sup_{n\in\mathbb{N}} f(d_n).$$

- per ogni intervallo $[a, b]$ si ha che

$$f_{n,k}([a, b]) = [a/n + k, b/n + k]$$

dunque se $[a, b] \subseteq [a', b']$ abbiamo che

$$f_{n,k}([a, b]) = [a/n + k, b/n + k]$$

e

$$f_{n,k}([a', b']) = [a'/n + k, b'/n + k]$$

ora abbiamo che $a \leq a'$ (risp. $b' \leq b$) allora basta osservare che la funzione $x/n + k$ è crescente (la sua derivata è sempre strettamente positiva pari a $1/n$) per cui $a/n + k \leq a'/n + k$ (risp. $b'/n + k \leq b/n + k$).

- osserviamo inoltre che

$$\sup I_i = \sup[a_i, b_i] = [\sup a_i, \inf b_i]$$

e che dalla continuità della funzione $\phi_{n,k}$ abbiamo che

$$\phi_{n,k}(\sup_i a_i) = \sup_i \phi_{n,k}(a_i)$$

e

$$\phi_{n,k}(\inf_i a_i) = \inf_i \phi_{n,k}(a_i)$$

quindi

$$\sup f(I_i) = [\sup f(a_i), \inf f(b_i)] = [f(\sup a_i), f(\inf b_i)] = f([\sup a_i, \inf b_i]$$

da cui abbiamo la σ-continuità.

Notiamo che abbiamo utilizzato delle funzioni $\phi_{n,k}$ il fatto che sono non decrescenti e continue dunque la stessa prova si può ripetere pre qualziasi funzione non decrescenti e continue (vedi punto seguente). □

d) Dimostrare più generalmente che l'insieme delle funzioni da D in D indotte da funzioni $\phi : \mathbb{R} \to \mathbb{R}$ non decrescenti e continue è σ-cc.

Soluzione. vedi sopra. □

e) (facoltativo) Mostrare che non è sufficente supporre la non decrescenza della ϕ per avere una funzione indotta σ-cc.

Soluzione. É sufficente considerare una funzione discontinua a sinistra in un punto, diciamo a, in modo che sia

$$\lim_{x \to a^-} f(x) = l \neq f(a)$$

Allora considerando una successione $I_n = [a_n, b_n]$ di intervalli in D strettamente crescenti, per cui

$$\sup_{n \in \mathbb{N}} I_n = [a, b]$$

avremo che

$$f([a, b]) \neq [\sup_{n \in \mathbb{N}} a_n, \sup_{n \in \mathbb{N}} b_n].$$

□

7. Programmazione Funzionale

ESERCIZIO 34. *Fornire il tipo delle seguenti funzioni specificate in ocaml:*

a)

```
 let rec nt lst =
match lst with
[] -> lst
|
(x::y::z::ls)-> [z;y;z]::(nt ls);;
```

b)

```
 let rec f lst =
match lst with
[] -> []
|
(x::ls) -> (f ls) @ [x];;
```

Valutare la complessità dei due programmi.

ESERCIZIO 35. *Considerare la seguente funzione definita in ocaml*

```
let rec twist n x y z w =
if n = 0 then x
else if n mod 2 = 0
then twist (n-1) y x w z
else twist (n-1) x z y w
```

Descrivere la sequenza delle chiamate fatte alla funzione `twist` *quando viene valutata l'espressione* `twist 6 "A" "B" "C" "D"`.

ESERCIZIO 36. *A partire seguente definizione ocaml*

```
let rec what n = n = 0 || not (what (n-1))
```

descrivere la funzione matematica che viene calcolata.

ESERCIZIO 37. *Per ognuna delle seguenti espressioni rispondere in uno dei modi seguenti:*

- *se l'espressione è ben tipata in ocaml e la sua valutazione termina dare il risultato della valutazione,*
- *se l'espressione non è accettata dall'interprete perchè non ammette un tipo, rispondere 'èrrore",*
- *se la valutazione dell'espressione genera un ciclo rispondere "non terminante".*

(a) `2 + 3 * ( 4 + 5)`
Soluzione. 29 □

(b) `let rec f ( x : int ) = if x mod 10 = 0 then 5 else 1 + (f (x+1)) in f 8`
Soluzione. 7 □

(c) `let x = 2 in let x = x*x in let x = x*x in x*x`
Soluzione. 256 □

(d) `let x=3 in let x = 4 in x*x`
Soluzione. 16 □

(e) `let x = 3 in ( let x = 4 in x * x ) * x`
Soluzione. 24 □

(f) `let two x = x + x in two ( two ( two 3 ) )`
Soluzione. 21 □

(g) `let f x = let g y = x + y in g ( x + 1) in f 10`
Soluzione. true □

(h) `let rec f g = (g true) || ( f g ) ;; let g x = not x ;; f g`
Soluzione. non termina □

(i) `let rec f g = (g true) || (f g) in let c x = not x in f c`
Soluzione. non termina □

(j) `let f x = let g y = x + y in g ( x + 2 ) in f (g 5)`
Soluzione. errore: la `g` é definita da `g y = x + y` nella definizione `let g y = x + y in g ( x + 2 )` del corpo di della `f`. Quindi non pu 'o essere usata nel in `f( g 5)`, perchè il suo valore è sconosciuto fuori dalla definizione di `g`. □

(k) `let rec f x = match x with h ::[] -> h + 5 | h::t -> h + f t | _ -> 7 in h [1;2;3]`

Soluzione. errore: `h` sconosciuta fuori dai due casi che ne fanno uso □

(l) `let rec f x = match x with h ::[] -> h + 5 | h::t -> h + f t | _ -> 7 in f [1;2;3]`

Soluzione. 11 □

(m) `let rec f x = match x with h ::[] -> h + 5 | h::t -> h + f t | _ -> 7.2 in f [1;2;3]`

Soluzione. errore: uso di un `float` (7.2) in un `int` □

(n) `[1]::([]::[2])`

Soluzione. errore: si tenta di accodare ad una lista di interi `[2]` una lista `[]` □

(o) `[1]::([]::[])`

Soluzione. il risultato é una lista di liste di interi che contiene due elementi una lista che contiene il solo elemento 1 e una lista vuota: `[[1];[]]` □

(p) `([1]::[])::[]`

Soluzione. il risultato é una lista di liste di liste di interi: `[[[1]]]` □

(q) `let x = 3.0 in if true then true else true`

Soluzione. true □

(r) `2.3 + 5`

Soluzione. errore □

(s) `let x = 5 in let y = x + 2 in let x = false in if x>3 then "A" else "B"`

Soluzione. error: al momento di valutare `x > 3` la `x` è un booleano (`false`) ma viene chiamato in un confronto tra interi. □

(t) `let f x = x +.  2.0`

Soluzione. `float -> float` □

(u) `let swap (x,y) = (y,x)`

Soluzione. 'a * 'b -> 'b * a' □

ESERCIZIO 38. *Sia* `map` *la funzione polimorfa che prende come argomenti una funzione* `f` *e una lista* `l = [a1;a2;a3; ... ;an]` *e che si valuta nella lista i cui elementi sono dati dalla funzione applicata agli elementi di* `l` *ovvero* `map f l = [f(a1);f(a2);f(a3); ... ; f(an)]`.

Programmare le funzioni ausiliarie `head` *e* `tail` *che appaiono nella seguente definizione di* `map`*:*

```
let rec map f l =
if l = [] then []
else f (head l ) ::  map f (tail l)
```

Soluzione. La funzione ausiliaria `head` si può definire mediante *pattern matching* sulle liste non vuote (attenzione: questo comporterà un'eccezione nel tipo della funzione che sarà indefinita qualora sia applicata alla lista vuota):

```
 let head l = match l with
x::  l1 -> x;;
```

Analogamente (ma questa volta senza alcuna eccezione sul tipo) abbiamo:

```
 let tail l = match l with
[] -> []
|
x::  l1 -> l1;;
```

□

ESERCIZIO 39. *Considerare la seguente funzione:*

```
let rec scanl l =
match l with
[] -> []
| [x] -> [x]
| h1::h2::t -> if h1 < h2 then h1 :: scanl ( h2 :: t)
else h2 :: scanl ( h1 :: t)
```

(a) *calcolare il valore di* `scanl [8;8;7;3;2;1]`
Soluzione. `scanl [8;8;7;3;2;1] = [8;7;3;2;1;8]` □

(b) *calcolare il valore di* `scanl (scanl [8;8;7;3;2;1])`
Soluzione. `scanl (scanl [8;8;7;3;2;1]) = scanl [8;7;3;2;1;8] = [7;3;2;1;8;8]` □

(c) *calcolare il valore di* `scanl (scanl (scanl [8;8;7;3;2;1]))`
Soluzione. `scanl (scanl (scanl [8;8;7;3;2;1])) = scanl (scanl [8;7;3;2;1;8]) = scanl [7;3;2;1;8;8] = [3;2;1;7;8;8]` □

(d) *sia ora* `sorted : 'a list -> bool` *un predicato (funzione a valori booleani) che ritorna* `true` *se la lista in argomento é ordinata; completare la seguente definizione di una funzione di ordinamento che utilizzi* `scanl`*: q*

```
let rec sort l =
      if sorted l then l
      else ??????
```

(e) *programmare la funzione* `sorted`*,*

(f) *supponendo* `sorted` *una funzione a complessità lineare valutare la complessità della funzione* `sort`*.*

ESERCIZIO 40. *Dire se è possibile determinare due lambda termini u e v in modo che la seguente sequenza di equivalenze sia soddisfatta:*

$$(\underline{0})\, u\, v \simeq_\beta (\underline{1})\, u\, v \simeq_\beta \cdots \simeq_\beta (\underline{n})\, u\, v \simeq_\beta \ldots$$

(in caso affermativo fornire un esempio).
Soluzione. Prendere $u = I$ e v qualsiasi. □

ESERCIZIO 41. *Calcolare il termine dopo la sostituzione (in questo casi si tratta della sostituzione semplice, ovvero la sostituzione che non tiene conto del fenomeno della cattura delle varibili):*

(1) $((\lambda x(x)x)\lambda x(x)x)\langle t/x\rangle$,
(2) $((\lambda x(x)x)(x)x)\langle t/x\rangle$,
(3) $((\lambda y(y)x)\lambda x(x)x)\langle y/x\rangle$.

ESERCIZIO 42. *Verificare a partire dalla definizione (induzione strutturale sui lambda termini):*

(1) $\lambda x\lambda y y \simeq_\alpha \lambda x\lambda x x$,
(2) $(\lambda y\lambda x(x)y)(x)z \simeq_\alpha (\lambda y\lambda x(x')y)(x)z$,
(3) $(\lambda x\lambda x(x)x)x \simeq_\alpha (\lambda x\lambda y(y)y)x \simeq_\alpha (\lambda z\lambda y(y)y)x \simeq_\alpha (\lambda y\lambda x(x)x)x$.

ESERCIZIO 43. *Calcolare le forme normali (se esistono) dei seguenti lambda termini:*

(1) $\lambda x\lambda y(z)\lambda s(x)syz$,
Soluzione. Questo lambda termine è già in forma normale. □

(2) $(\lambda x \lambda y y)x$,

Soluzione. $\lambda y y$. □

(3) $(\lambda x(x)x)\lambda x(x)yx$,

Soluzione. $(\lambda x(x)x)\lambda x(x)yx \to_{\beta_0}$
$((x)x)[\lambda x(x)yx/x] = (\lambda x(x)yx)\lambda x(x)yx \to_{\beta_0}$
$((x)yx)[\lambda x(x)yx/x] = ((\lambda x(x)yx)y)\lambda x(x)yx \to_{\beta_0}$
$(((x)yx)[y/x])\lambda x(x)yx = ((y)yy)\lambda x(x)yx$. □

(4) $\lambda z(z)(\lambda y \lambda x y)x$,

Soluzione. $\lambda z(z)\lambda x' x$. □

(5) $(\lambda z(z)y)\lambda x x$.

Soluzione. $(\lambda z(z)y)\lambda x x \to_{\beta_0}$
$((z)y)[\lambda x x/z] = (\lambda x x)y \to_{\beta_0}$
$(x)[y/x] = y$. □

8. Programmazione Object-Oriented

ESERCIZIO 44. *Data la classe*

```
public abstract class item {

    item next = null;

    item item() {
        this.next = null;
        return this ;
    }

}
```

Fornire una classe derivata che implementa le liste di interi. Estendere ulteriormente la classe alle liste ordinate fornendo un metodo di `insertion_sort` *(per l'inserimento di elementi in una lista ordinata).*

Soluzione.

□ *Per estendere la classe basta definire una classe che chiamiamo* `list_int` *che aggiunge un campo* `value` *di tipo intero alla classe item*

```
public class list_int extends item {

    int  value  ;
    list_int list_int() {
        this.next = null;
        this.value = 0 ;
                return this ;
    }


}
```

Questa classe può a sua volta essere estesa con il metodo di `insertion_sort`*, definiamo allora la classe* `sorted_list_int` *che estende* `list_int`*:*

```
public class sorted_list_int extends list_int {
```

```
    int  value  ;
    sorted_list_int sorted_list_int() {
        this.next = null;
        this.value = 0 ;
                return this ;
    }

    sorted_list_int sorted_list_int(int x) {
        this.next = null;
        this.value = x ;
                return this ;
    }

    sorted_list_int insert( int x )
{
    if (x < this.value) then
        this.next.insert(x)
    else
        {
          list_in
        }


}
```

Esercizio 45. *Definire in ocaml una classe per i polimoni a coefficenti interi:*

$$p(x) = a_0 + a_1 x + a_2 x + \cdots + a_n x^n \qquad a_i \in \mathbb{Z}.$$

La classe dovrà possedere dei metodi per le seguenti operazioni sui polinomi: addizione, sottrazione, moltiplicazione, divisione, resto, derivazione.

Attenzione: la classe dovrà evidentemente contenere metodi binari (a due argomenti).

Esercizio 46. *Considerata la definizione della seguente classe in ocaml:*

```
 class c2 =
object (self)
val mutable y = 0
val mutable static = false
method private mystery x = y <- (
if (not static)
then (static <- true ; x*x )
else y
) ; y <- y + x ; y - x
method main = print_int (self#mystery 1) ; print_string "
";print_int (self#mystery (-2));print_string "/n"
end
```

Descrivere l'output corrispondente alla sequenza di valutazioni:

```
 let c = new c2;;
c#main;;
c#main;;
```

ESERCIZIO 47. *Considerata la seguente dichiarazione di classe in ocaml:*

```
 class zp ( p :  int) =
object
val mutable value :  int = 0
val ordine :  int = p
method print_zp = print_int value ; print_string "\n"
method set x = value <- (x mod p)
method get = value
method add_zp (x :  zp ) (y :  zp) = (x#get + y#get) mod p
end ;;
```

Descrivere l'output della seguente serie di valutazioni:

```
 let a = new zp 4 ;;
let b = new zp 4 ;;
```

Soluzione. Queste due valutazioni corrispondono alla creazione di due istanze della classe funzionale `zp`.

Il parametro intero `p` da cui dipende la classe viene assegnato uguale a 4. Notare che questo parametro non è `mutable`, quindi resta fissato una volta per tutte per gli oggetti `a` e `b`. □

```
 a#get ;;
a#add_zp a a;;
a#print_zp;;
a#set 5;;
a#print_zp;;
```

Soluzione.

- restituisce il valore intero che inizializza il campo `value` dell'oggetto `a`: dunque, `0 : int`,
- somma modulo $p = 4$ il campo `value` della classe a: dunque $0+0 \mod p = 0$, risposta `0:int`.
- stampa il valore di `value` ovvero `0 : unit`,
- invoca il metodo `set` della classe `a` e quindi modifica il valore del campo `value`, che d'ora in poi varrà $5 \mod p$ ovvero 1.
- stampa il valore di `value` ovvero `1 : unit`.

□

```
 b#set 7;;
b#print_zp;;
b#set(a#add_zp a b);;
a#print_zp;;
b#print_zp;;
```

Soluzione.

- invoca il metodo `set` della classe `b` e quindi modifica il valore del campo `value`, che d'ora in poi varrà $7 \mod p$ ovvero 3.

- stampa il valore di `value` per l'oggetto `b` ovvero `3 : unit`.
- invoca il metodo set per l'oggetto `b` sul risultato della invocazione del metodo `add_zp` dell'oggetto `a` con argomenti gli oggetti `a` e `b`. Il risultato è l'oggetto `b` con il campo `value` modificato, pari a `0 : int`;
- stampa il valore di `value` per l'oggetto `a` ovvero `1 : unit`.
- stampa il valore di `value` per l'oggetto `b` ovvero `0 : unit`.

□