

Metodi per il Trattamento delle Informazioni  
dispense a.a. 2005-2006

(versione definitiva)

Salvatore Caporaso

il dott. Gianluca Sforza ha collaborato ai capitoli 2 e 3

il dott. Nicola Corriero ha collaborato ai capitoli 4 e 5

23 dicembre 2005

# Indice

<b>1</b>	<b>Sintassi e Interpretazione</b>	<b>3</b>
1.1	Sistemi formali . . . . .	3
1.2	Linguaggi di programmazione e sistemi formali . . . . .	7
1.3	Stringhe e liste . . . . .	8
1.4	Notazioni . . . . .	11
<b>2</b>	<b>Computazioni</b>	<b>13</b>
2.1	Macchine di Turing ordinarie e loro varianti . . . . .	13
2.1.1	TM definite mediante triple . . . . .	15
2.1.2	TM con piú nastri . . . . .	16
2.1.3	TM con un solo stato interno . . . . .	16
2.1.4	Chiusura rispetto ai costrutti strutturati . . . . .	17
2.2	TM strutturate . . . . .	18
2.2.1	Una TM strutturata universale . . . . .	22
<b>3</b>	<b>Ricorsione</b>	<b>24</b>
3.1	List Processing . . . . .	24
3.1.1	Digressione: un programma C che si riproduce . . . . .	26
3.2	Il teorema della ricorsione . . . . .	28
3.2.1	Applicazione e commento: una LPM che si riproduce . . . . .	29
3.2.2	LPM universale . . . . .	29
3.2.3	Schemi di programma uniformi . . . . .	32
3.2.4	Teorema di Rogers . . . . .	33
3.2.5	Le classi totali non includono l'elemento universale . . . . .	35
3.2.6	Teorema di Rice . . . . .	35
3.3	Un interprete che impara definizioni ricorsive . . . . .	36
3.3.1	Definizioni ricorsive . . . . .	37
3.3.2	Interprete per programmi ricorsivi . . . . .	39

3.3.3	Funzionali . . . . .	40
<b>4</b>	<b>Complessità Computazionale</b>	<b>42</b>
4.1	Macchine di Turing non deterministiche . . . . .	42
4.2	Classi di complessità . . . . .	43
4.3	Quanto contano le costanti, gli alfabeti e il numero dei nastri? . . . .	45
4.4	3SAT e il problema $P=?NP$ . . . . .	47
4.5	$PSPACE=NPSPACE$ . . . . .	51
4.6	QBF e la separazione di $PSPACE$ dalle classi in esso contenute . . . .	51
<b>5</b>	<b>Calcolo dei predicati</b>	<b>55</b>
5.1	Il linguaggio . . . . .	55
5.2	Un apparato deduttivo . . . . .	56
5.3	Interpretazione . . . . .	58
5.4	Completezza . . . . .	60
5.5	Indecidibilità . . . . .	65

# Capitolo 1

## Sintassi e Interpretazione

Questo capitolo introduce alcune nozioni riconducibili ad idee di Fege e di Church. Esse sono *illustrate* in termini informali ed intuitivi perché una loro presentazione rigorosa supererebbe i limiti e gli obiettivi di questo libro.

### 1.1 Sistemi formali

#### Variabili, forme e Funzioni

**1.1 Classi numerabili** Per evitare certe difficoltà della teoria degli insiemi fisseremo la nostra attenzione su collezioni o *classi* di entità, che siano *numerabili*, nel senso che i loro *membri*  $E_1, \dots$  possano essere associati univocamente ad alcuni o a tutti i numeri naturali. I simboli  $\in, \notin, -, \cup, \subseteq$  si applicano alle classi nel modo consueto. È noto che il prodotto cartesiano  $C \times D$  di due classi numerabili è una classe numerabile.

**1.2 Funzioni parziali** (1) Date le classi  $C$  e  $D$ , una *funzione parziale* da  $C$  a  $D$  è una classe  $\varphi \subseteq C \times D$  tale che, per ogni  $x, y$  e  $z$  si ha

$$(x, y) \in \varphi \quad \text{nonché} \quad (x, z) \in \varphi \quad \text{implica} \quad y = z$$

**Notazione**  $\varphi : C \mapsto D$ . Il *dominio* e il *codominio* di  $\varphi$  sono definiti da

$$\begin{aligned} \text{dom}(\varphi) &= \{x : \text{si ha } (x, y) \in \varphi \text{ per qualche } y\} \\ \text{range}(\varphi) &= \{y : \text{si ha } (x, y) \in \varphi \text{ per qualche } x\} \end{aligned}$$

$\varphi : C \mapsto D$  è *totale* se si ha  $\text{dom}(\varphi) = C$ .

(2)  $\varphi$  è *definita in*  $x$  se  $x \in \text{dom}(\varphi)$ . In tal caso,  $\varphi x = y$  sta per  $(x, y) \in \varphi$ . Si scrive  $\varphi x \uparrow$  se  $x \notin \text{dom}(\varphi)$ . L'espressione  $\varphi x \simeq y$  significa che si ha  $\varphi x = y$  oppure  $x \notin \text{dom}(\varphi)$ .

$\varphi x \simeq \psi x$  significa che si ha  $\varphi x = \psi x$ , oppure  $\varphi x$  nonché  $\psi x$  non sono definite.

Si noti che  $\varphi x \simeq y$  è banalmente vero quando  $\varphi x \uparrow$ .

La *sostituzione* di  $k \geq 1$  funzioni parziali  $\psi_i$  in una funzione  $\chi$  produce una funzione  $\varphi x = \chi(\psi_1 x, \dots, \psi_k x)$ , che è definita in  $x$  sse<sup>1</sup> tutte le  $\psi_i$  lo sono, e sse  $\chi$ , a sua volta, è definita nei  $k$  valori  $\psi_i x$ .

(3) Due classi sono uguali se hanno gli stessi membri (*principio di estensionalità*). In particolare, date  $\varphi : C \mapsto D$  e  $\psi : C \mapsto D$ , si ha  $\varphi = \psi$  sse  $\varphi x \simeq \psi x$  per ogni  $x \in C$ .

**1.3 Senso e significato** Quindi, per  $C_1 = \{1, 3, 4\}$  e  $C_2 =$  *le prime quattro cifre di*  $\pi = 3.14156\dots$  si ha  $C_1 = C_2$ . Viene allora da chiedersi in cosa queste due classi differiscano. Questo punto può aiutare a cogliere la distinzione tra le funzioni e i programmi che le calcolano.

Un nome (proprio) è una combinazione di parole usate per individuare una persona o un oggetto, che spesso comincia con una maiuscola o con l'articolo determinativo. Uno stesso individuo può essere identificato in più modi (*Venere, Pianeta tra Mercurio e la Terra, Stella della sera, Stella del mattino*). Per descrivere questo fatto, si dice che un nome *denota un significato* ed *esprime un senso*. Un significato è un'entità indicata dai suoi nomi. Un senso è un concetto o un'idea. Un significato è visto dagli occhi o dall'immaginazione, mentre un senso viene capito. I sinonimi hanno sensi diversi ed uno stesso significato. Nel linguaggio naturale può accadere che un nome non denoti alcun significato, pur avendo un senso comprensibile: per concludere che l'espressione *Attuale re d'Italia* non ha un significato, dobbiamo prima coglierne il senso.

**1.4 Costanti** In matematica una *costante* è un nome provvisto di significato. Quindi  $5 + 7$ ,  $\frac{24}{2}$  e  $12$  sono costanti che esprimono diverse proprietà (sensi) del numer che esse denotano. 'Il multiplo di 5 compreso tra 6 e 9' non è una costante. Il significato di una costante è spesso chiamato *valore*. Si generalizza la nozione di costante ammettendo entità con più significati.

**1.5 Variabili** (1) Una *variabile*  $x$  è un nome che denota gli elementi di una classe  $C$ . Si dice che  $x$  è *definita sul dominio*  $C$ , e si scrive  $C = \text{dom}(x)$ .

(2) Si *assegna un valore* ad  $x$  scegliendo un elemento di  $\text{dom}(x)$ .

---

<sup>1</sup>= se e solo se

(3) La costante  $c$  e la variabile  $x$  sono *compatibili* se si ha  $c \in \text{dom}(x)$ .

(4) Le variabili  $x$  ed  $y$  sono compatibili se si ha  $\text{dom}(x) = \text{dom}(y)$ .

**Nota** Le variabili assomigliano piú ai *registri* di un calcolatore che agli oggetti che *variano* suggeriti dal loro uso in cinematica o in analisi. Assignare un valore ad una variabile, come caricare un registro, è semplicemente una scelta da un dato menu. Un'entità che varii sui naturali sarebbe pari e dispari, maggiore e minore di ogni  $n$ , soddisferebbe e falsificherebbe qualsiasi proprietà, ed un'entità così proteiforme sarebbe altrettanto incoerente che inutile.

**1.6 Schemi** In una costante o in un nome composti possono occorrere (figurare) delle costanti o nomi piú semplici (5 e 7 occorrono in  $5 + 7$ ). Un'espressione come

$$\Sigma(c_1, \dots, c_k)$$

denoterà una costante  $c$  in cui occorrono le costanti  $c_1, \dots, c_k$  (non necessariamente una volta, né in quest'ordine). Data una costante  $c$  in questa forma,

$$\Sigma(E_1, \dots, E_k)$$

denoterà il rimpiazzamento di ciascuna occorrenza di ogni  $c_i$  in  $c$  mediante  $E_i$ . Questo suggerisce l'idea di rimpiazzare mediante variabili delle costanti che occorrono in altre costanti. Una *forma* o *schema k-ario* è il risultato del rimpiazzamento di  $k \geq 1$  costanti  $c_i$  mediante  $k$  variabili compatibili  $x_i$  in una costante  $\Sigma(c_1, \dots, c_k)$ . Assegnando un sistema di valori  $a_i$  alle variabili  $x_i$  otteniamo

$$\Sigma(a_1, \dots, a_k).$$

Il suo (eventuale) valore è il *valore di*  $\Sigma(x_1, \dots, x_k)$  per  $a_1, \dots, a_k$ .

**Esempio** Se come  $c_1, c_2$  e  $\Sigma(c_1, c_2)$  si prendono 5, 7 e  $5 + 7$ , e se si rimpiazzano  $c_1$  e  $c_2$  con  $x_1$  e  $x_2$  si ottiene la forma o schema binario  $x_1 + x_2$ . Il suo valore per il sistema di valori  $a_1 = 0$  e  $a_2 = 0$  è 0.

**1.7** (1)  $\varphi = \lambda x. \Sigma(x)$  è la *funzione associata* alla forma unaria  $\Sigma(x)$  dall'*operatore di astrazione*  $\lambda$ . I membri di questa funzione sono dati da

$$\varphi a = b \quad \text{sse il valore di } \Sigma(x) \text{ per } a \text{ è } b$$

(2)  $\varphi = \lambda x_1 \dots x_k. \Sigma(x_1, \dots, x_k)$  è la funzione associata alla forma  $\Sigma(x_1, \dots, x_k)$ . I suoi membri sono dati da

$$\varphi a_1 \dots a_k = b \quad \text{sse il valore di } \Sigma(x_1, \dots, x_k) \text{ per } a_1, \dots, a_k \text{ è } b$$

Quindi  $\varphi_{a_1..a_k}$  non è definita se  $\Sigma(a_1, \dots, a_k)$  non ha un valore.

(3) Due forme sono *concorrenti* se le loro variabili sono a due a due compatibili e se le loro funzioni associate sono uguali.

**Esempio** Se  $x$  ed  $y$  sono entrambe definite sui naturali  $\mathcal{N}$  allora le forme  $\frac{1}{2}x(x+1)$  e  $\sum_{i=0}^y i$  sono concorrenti (si noti che possiamo ignorare la variabile *vincolata*  $i$ ).

**1.8 Nota** Una forma è un'entità linguistica, mentre una funzione (così come ogni classe) è un concetto. Il nome di  $\lambda$  evoca l'idea che esso elimina ogni attributo inessenziale: esso produce una funzione, mediante il processo mentale che trova il significato condiviso da forme diverse, se si ignorano le tecniche di calcolo (per esempio gli algoritmi) e i dettagli linguistici.

La distinzione tra forme e funzioni contribuisce a risolvere la questione apparentemente oziosa se  $\sin(x)^2$  e  $1 - \cos(y)^2$  siano la *stessa cosa* oppure no. In base a 1.2, l'uguaglianza può essere predicata delle funzioni, non delle forme. Quindi si ha  $\lambda x[\sin(x)^2] = \lambda y[1 - \cos(y)^2]$  perché queste due classi sono formate dagli stessi elementi (purché  $x$ ,  $y$ , *sinus* e *cosinus* siano opportunamente definite sui naturali). Le espressioni  $\sin(x)^2$  sono  $1 - \cos(y)^2$  forme diverse, benché concorrenti.

## Metateorie

In termini molto generali, qualsiasi teoria  $\mathcal{T}$  può essere considerata come una tripla  $(\mathcal{L}, \mathcal{U}, \mathcal{R})$ , tale che

- (1) il suo *universo*  $\mathcal{U}$ , è una totalità di individui, relazioni, o fatti, da studiare o *processare*;
- (2) il suo *linguaggio*  $\mathcal{L}$  è collegato ad  $\mathcal{U}$  in un modo che spesso precede la costituzione di  $\mathcal{T}$ ;
- (3) le sue *regole*  $\mathcal{R}$ , stabiliscono quali elementi di  $\mathcal{L}$  sono veri, interessanti o accettabili (dal punto di vista di  $\mathcal{T}$ ).

Per esempio è probabile che un ramo della matematica, fisica o biologia parli, nel suo particolare gergo, di questioni che concernono l'astrazione, la natura o la vita, dando ruolo e rilievo diversi a deduzione, induzione e sperimentazione.

**1.9** Una *metateoria*  $\mathcal{M}$  è una teoria il cui universo è un'altra teoria  $\mathcal{T}^o$ , che viene allora detta *teoria oggetto*.  $\mathcal{M}$  parla in un *metalinguaggio*, e procede mediante *metaregole*. In simboli

$$\mathcal{M} = (\mathcal{L}^m, (\mathcal{L}^o, \mathcal{U}^o, \mathcal{R}^o), \mathcal{R}^m)$$

Per esempio una metateoria può proporsi di dimostrare che la sua teoria oggetto è in/completa.

**1.10** Un *sistema formale*  $\mathcal{F}$  è una teoria  $(\mathcal{L}^f, \mathcal{R}^f)$  senza universo.  $\mathcal{L}^f$  è per lo più un *linguaggio formale* (si veda 1.16).

**1.11** Una *sintassi*  $\mathcal{S}$  è la metateoria di un sistema formale. Essa si serve di un *linguaggio sintattico* e di *regole sintattiche*. Ossia

$$\mathcal{S} = (\mathcal{L}^s, (\mathcal{L}^f, \mathcal{R}^f), \mathcal{R}^s)$$

Gli elementi di un linguaggio formale sono meri *segni*, non *simboli*, nel senso che essi non hanno alcun significato (ufficiale); ricevono tuttavia un qualche senso dalla rete di interazioni tra loro stabilite dalle regole formali. Per definire un sistema formale si usa un linguaggio sintattico.

**1.12 Example** Si può immaginare il gioco degli scacchi come un sistema formale. Il suo linguaggio è la classe delle posizioni sulla scacchiera. Le sue regole stabiliscono le mosse permesse. La sua (meta) teoria si esprime in un linguaggio sintattico, ottenuto aggiungendo al linguaggio naturale certi termini tecnici come *gambetto* o *matto*. Posizioni e regole sono spiegate in questo linguaggio, non nel linguaggio oggetto il quale, consistendo solo di posizioni, non ha significato alcuno. La teoria dimostra affermazioni come: matto in 3 mosse; re e alfiere da soli non vincono.

**1.13 Semantica** Si *interpreta* un sistema formale in un universo  $\mathcal{U}$  associando gli elementi del suo linguaggio con individui, relazioni, fatti e magari istruzioni relative alle entità di  $\mathcal{U}$ . Una *semantica*  $\mathcal{SEM}$  di un sistema formale  $\mathcal{F} = (\mathcal{L}^f, \mathcal{R}^f)$  è una metateoria di  $\mathcal{F}$  che, a differenza della sintassi, sceglie un universo  $\mathcal{U}^m$ , e dà un'interpretazione in  $\mathcal{U}$  di  $\mathcal{L}^f$ . Si ha allora

$$\mathcal{SEM} = (\mathcal{L}^m, (\mathcal{L}^f, \mathcal{U}^m, \mathcal{R}^f), \mathcal{R}^m)$$

**Nota** L'universo di una generica metateoria è una parte della teoria oggetto, mentre l'universo per un sistema formale è una parte della sua semantica e, in quanto tale, è una meta-entità (l'indice in alto di  $\mathcal{U}^m$  si propone di attirare l'attenzione su questo punto).

## 1.2 Linguaggi di programmazione e sistemi formali

**1.14** Un linguaggio di programmazione può essere visto come un sistema formale  $(\mathcal{D}, \mathcal{P})$ , in cui  $\mathcal{D}$  è una classe di *dati*  $x, \dots$  mentre  $\mathcal{P}$  è una classe di *programmi*  $P, Q, \dots$

1. La *sintassi* di  $\mathcal{P}$  è definita induttivamente, per *chiusura* di certi *elementi di base*, rispetto ad alcuni *schemi (di base) k-ary*  $\Sigma(\mathbf{E}_1, \dots, \mathbf{E}_k)$ . Ossia
  - (a) Gli elementi di base sono programmi appartenenti (per definizione) a  $\mathcal{P}$ ;
  - (b) se  $P_1, \dots, P_k$  appartengono a  $\mathcal{P}$ , allora anche  $\Sigma(P_1, \dots, P_k)$  appartiene (ancora per definizione) a  $\mathcal{P}$ .
2. Le *semantiche operazionali* per questi sistemi sono definite scegliendo una classe di funzioni da  $\mathcal{D}^n$  a  $\mathcal{D}$  in un modo induttivo che rispecchia la generazione per chiusura dei programmi, nel senso che that ( $\mathbf{x}$  è una  $n$ -pla di dati)
  - (a) una funzione  $\llbracket P \rrbracket : \mathcal{D}^n \mapsto \mathcal{D}$  *interpreta* ciascun programma di base  $P$ ;
  - (b) un funzionale  $S$ , che associa a ciascuna  $k$ -pla di funzioni  $n$ -arie  $f_1, \dots, f_k : \mathcal{D}^n \mapsto \mathcal{D}$  una funzione  $n$ -aria  $S(f_1, \dots, f_k) : \mathcal{D}^n \mapsto \mathcal{D}$ , *interpreta* ogni schema  $k$ -ario  $\Sigma$  in modo che

$$\llbracket \Sigma(P_1, \dots, P_k) \rrbracket(\mathbf{x}) = S(\llbracket P_1 \rrbracket, \dots, \llbracket P_k \rrbracket)(\mathbf{x}).$$

**Notazione**  $\llbracket P \rrbracket(\mathbf{x}) \simeq y$  se  $\llbracket P \rrbracket(\mathbf{x}) = z$  oppure  $\llbracket P \rrbracket(\mathbf{x}) \uparrow$ .

**Nota**  $\llbracket P \rrbracket(\mathbf{x}) \simeq y$  è banalmente vero per ogni  $y$  se  $\llbracket P \rrbracket(\mathbf{x})$  non è definita.

**1.15 Schemi derivati e schemi di base** Uno *schema di programma derivato*  $\Sigma(P_1, \dots, P_k, E_1, \dots)$  è uno strumento per la dimostrazione dell'esistenza di programmi. A tale scopo, si dimostra che, per ogni  $k$ -pla di programmi  $P_1, \dots, P_k$  e per ogni sistema di valori per le altre variabili  $E_1, \dots$  (se any), esiste un programma  $Q$ , che si comporta secondo l'*intended interpretation* del nuovo schema. Inoltrre e soprattutto: uno schema derivato consente l'assemblaggio di nuovi programmi se  $\mathcal{P}$  è chiuso *uniformemente* rispetto a  $\Sigma$ , ossia se  $Q$  è prodotto da un programma appartenente a  $\mathcal{P}$ . Si noti che invece gli schemi di base sono una parte *postulata* di  $\mathcal{P}$ , e non richiedono dunque alcuna dimostrazione.

## 1.3 Stringhe e liste

**1.16 (1)** Una *lettera* è un segno che, dal punto di vista di una data applicazione, può essere considerato come indivisibile. Un *alfabeto*  $\Gamma$  è una classe finita e non vuota, ai cui elementi sono associati biunivocamente dei segni detti *lettere*. Tra di esse, il segno  $_$  ha il ruolo di spazio o *bianco*.

(2) Una *stringa*  $w$  di *lunghezza*  $n$  su  $\Gamma$  è una sequenza di  $n \geq 0$  lettere di  $\Gamma$ . In genere la lunghezza di  $w$  viene denotata da  $|w|$ .  $\epsilon$  è la *stringa vuota*, ossia l'unica stringa di lunghezza 0.

(3)  $\Gamma^*$  è la classe di tutte le stringhe su  $\Gamma$ , e  $\Gamma^+$  è  $\Gamma^* - \{\epsilon\}$ .

(4) Un *linguaggio*  $\mathcal{L}$  su  $\Gamma$  è una qualsiasi sottoclasse di  $\Gamma^*$ .  $\mathcal{L}$  è un *linguaggio formale* se esiste un algoritmo che *enumera* tutti i suoi elementi (anche alla rinfusa e con ripetizioni).

(5) La *concatenazione*  $wu$  delle stringhe  $w$  e  $u$  è la stringa di lunghezza  $|w| + |u|$  che comincia con le lettere di  $w$  e termina con quelle di  $u$  (ovviamente nello stesso ordine).

Secondo il contesto,  $E^n$  può denotare sia il numero  $E$  alla  $n$  (potenza), sia la stringa  $E$  ripetuta per  $n$  volte (concatenazione).

(6)  $u^R$  e  $u$  *rovesciata* (cioè letta da destra a sinistra).

(7) Date  $u, x \in \Gamma^*$ , un'*occorrenza* di  $u$  in  $x$  è una coppia  $y$  e  $z$  tale che  $x = yuz$ .

**1.17 Liste** (1) Le *liste* sull'alfabeto  $\{0, \cdot\}$ , sono stringhe generate dalle regole: 0 è una lista; se  $x$  e  $y$  sono liste allora  $z = \cdot xy$  è una lista, spesso scritta nella forma

$$(x, y).$$

Si pone (*associatività a destra*)

$$(x, (y, z)) = \cdot x \cdot yz = (x, y, z).$$

Ma si ha invece  $((x, y), z) = \cdot \cdot xyz$ .

$y_i$  è la  $i$ -ma *componente* di  $z = (y_1, \dots, y_k)$ , e  $k = \#z$  è il numero delle sue componenti. Si pone  $\#0 = 1$ .

**Notazione**  $x = (y, \bar{z}^{(k)})$  sta per  $x = (y, z_1, \dots, z_k) = (y, z)$  se  $z = (z_1, \dots, z_k)$ . Spesso  $(k)$  viene omissso.

(2) Le liste sono chiuse rispetto alle operazioni *head* e *tail*<sup>2</sup> definite da

$$hd[\cdot xy] = x \qquad tl[\cdot xy] = y \qquad hd[0] = tl[0] = 0.$$

Per esempio, si ha  $hd[((u, w), \dots, y_k)] = (u, w)$ .

(3) Poniamo  $[x]_{i+1} = hd[tl^i[[x]]]$  ( $i \geq 0$  occorrenze di *tl*). Si ha

$$[(y_1, \dots, y_k)]_i = y_i \quad \text{se } i \leq k; \qquad [z]_i = 0 \quad \text{se } i > \#z$$

---

<sup>2</sup>Molti termini sono lasciati in inglese per non creare conflitti inutili con la terminologia internazionale (per non parlare di effetti spesso francamente grotteschi).

**Notazione**  $[x]_\omega$  per  $tl[x]$ . Inoltre,  $[x]_{ij}$  per  $[[x]_i]_j$ , e  $[x]_I$  per  $[x]_{i_1 \dots i_k}$ , se  $I$  è  $i_1 \dots i_k$  ( $i, j, i_h = 1, 2, \dots, \omega$ ).

**Esempio** Si ha  $[(x, y, z)]_{\omega 1} = [((x, y), z)]_{12} = [(x, y, z)]_2 = y$ .

Si noti che  $(hd[(x, y, z)], tl[(x, y, z)])$  è  $(x, (y, z))$ , non  $(x, y, z)$ .

(4) Il *numerale*  $\mathbf{k}$  di  $k \geq 0$  è la lista le cui  $k$  componenti sono identicamente 0. Ogni espressione, nella quale figurino un numerale, sta per il numerale del suo valore. Per esempio  $2\mathbf{k}$  è 4 per  $k = 2$ . Si noti che  $(2, 2)$  comincia con  $\cdot\cdot$  e quindi non è 4 che s'inizia con  $\cdot 0$ .

**Notazione** Per alcune liste (in particolare, numerali) saranno adottati dei nomi mnemonici, formati da lettere, cifre decimali, e punti ordinari scritti in fonte a **larghezza costante** (per esempio, `pop.1`).

L'altezza (*height*) dell'albero  $x$  è data da  $ht(a) = 0$  e  $ht(\cdot yz) = 1 + \max(ht(y), ht(z))$ . Per la lunghezza si ha  $|x| = 2n + 1$  dove  $n$  è il numero dei punti che figurano in  $x$ . Per esempio,  $|\bar{2}| = 5$  e  $ht(\bar{2}) = 2$ .

Dimostriamo che ogni lista può essere *parsed* (decomposta) univocamente.

**1.18 Lemma** (1) Se  $x = Y \cdot 0Z$  è una lista allora anche  $x^* = YZ$  è una lista.

(2) Se  $z = UW$  è una lista e se  $U$  è una stringa con  $l \geq 0$  punti e con  $l + 1$  zeri, allora  $W$  è vuota.

*Dimostrazione.* (1) Induzione su  $|x|$ . Base. Si ha  $x = \cdot 00$  e 0 è una lista.

Passo. Abbiamo  $x = \cdot y_1 y_2$ .

Caso 1.  $Y = \epsilon$ . Quindi  $y_1$  è 0, e possiamo porre  $x^* = y_2$ .

Caso 2.  $y_1 = Y \cdot 0Z_1$ . Per I.H. <sup>3</sup>  $y_1^* = YZ_1$  è una lista, e possiamo porre  $x^* = \cdot y_1^* y_2$ .

Caso 3.  $y_2 = Y_1 \cdot 0Z$ . Per I.H.  $y_2^* = Y_1 Z$  è una lista, e possiamo porre  $x^* = \cdot y_1 y_2^*$ .

(2) Induzione su  $|U|$ . Base.  $U = 0$ . Quindi  $z = U$  e  $W = \epsilon$ . Passo. Abbiamo

$$U = Y \cdot 0Z; \quad z = Y \cdot 0ZW.$$

Per la parte (1)  $v = YZW$  è una lista.  $YZ$  è formata da  $l - 1$  punti e  $l$  zeri. Quindi I.H. (con  $v$  come  $z$ , e con  $YZ$  come  $U$ ) dà  $W = \epsilon$ .

**1.19 Teorema** Per tutte le liste  $x, u, w, y, z$

$$x = \cdot uw = \cdot yz \quad \text{implica} \quad u = y.$$

---

<sup>3</sup> ipotesi induttiva.

*Dimostrazione.* Si assuma (ad abs.)  $u \neq y$ . Caso 1.  $|u| < |y|$ . Allora  $y = uX$  per qualche  $X \neq \epsilon$ . Contraddizione con Lemma 1.18(2).

Caso 2.  $|u| > |y|$ . Allora  $u = yX$  per qualche  $X \neq \epsilon$ . Stessa contraddizione.

**1.20 L** Un linguaggio  $L$  in forma polacca o *prefissa* è ottenuto per chiusura di una classe non vuota di simboli *0-ari* rispetto ad alcuni schemi *i-ari*. In simboli

$$\begin{aligned} L &\subset (\Gamma^{(0)} \cup \Gamma^{(1)} \cup \dots \cup \Gamma^{(k)})^+ \\ L &= \{a^{(i)}w_1 \dots w_i \mid a^{(i)} \in \Gamma^{(i)}; w_i \in L; i \geq 0\} \end{aligned}$$

**Esempio** Le formule proposizionali in forma polacca sono ottenute per chiusura di una classe di *literals*  $p, q, r, \dots$  rispetto al simbolo unario  $\neg$  ed ai simboli binari  $\wedge$  e  $\vee$ . Sicché in questo linguaggio si scrive per esempio  $\wedge \vee pq \vee rs$  invece di  $(p \vee q) \wedge (r \vee s)$ . I linguaggi prefissi e infissi sono piú facili da processare e (risp.) da leggere. Le liste sono un linguaggio prefisso, e la dimostrazione del teorema precedente si estende facilmente ad ogni linguaggio polacco.

## 1.4 Notazioni

1.  $u, \dots, z$  sono liste, mentre  $U, \dots, Z$  sono stringhe sull'alfabeto corrente.

Se il simbolo  $E$  è una variabile definita su una classe sintattica, allora la sua versione in grassetto  $\mathbf{E}$  denota una tupla di entità appartenenti a quella stessa classe. Una volta che la tupla  $\mathbf{E}$  è stata introdotta nel discorso che si sta facendo,  $E_i$  denota il suo  $i$ -mo elemento.

Se  $\mathbf{X}$  è una  $n$ -pla di stringhe allora  $U \mathbf{X}$  è un'abbreviazione per  $UX_1, \dots, X_n$ .

$\mathbf{M}, \dots, \mathbf{Q}$  sono (parti di) generici programmi del sistema in corso di discussione.

2. Quando la distinzione tra programma e funzione da esso calcolata è chiara,  $\mathbf{P}(\mathbf{x})$  sta per  $\llbracket \mathbf{P} \rrbracket(\mathbf{x})$ . Quando i dati sono liste, scriviamo  $\mathbf{P}[x]$ .
3. Un'espressione come  $E = F$  può significare due cose (almeno)
  - (a) che  $E$  ed  $F$  sono due differenti meta-nomi per lo stesso oggetto;
  - (b) che una qualche valutazione assegna loro la stessa entità.

Scriviamo  $E \equiv F$  nel primo caso, e invece  $E = F$  nel secondo. Inoltre scriveremo  $=$  invece di  $\simeq$  quando la questione della (non) definitezza è priva di interesse. In particolare, scrivendo (spesso in due linee separate)

$$\mathbf{P} \equiv E \qquad \mathbf{P}(\mathbf{x}) = y$$

intendiamo che il programma  $P$  è definito dall'espressione  $E$ , e che, per input  $\mathbf{x}$  esso dà  $y$ .

4. I nostri programmi avranno una forma *ufficiale* che ne permette una più facile elaborazione da parte di altri programmi (enumeratori, interpreti, etc.) ed una forma *leggibile*. Di regola la prima è prefissa, mentre la seconda (infix) è molte volte detta *esplicita*.
5. I tre schemi di programma seguenti saranno impiegati con la stessa interpretazione in diversi sistemi:

(a)  $P; Q$  è la composizione di  $\llbracket P \rrbracket$  (calcolata prima) con  $\llbracket Q \rrbracket$ .  $P^k$  è la composizione di  $k \geq 0$  copie di  $P$ .

(b) Come in Lisp, una sequenza di  $k$  *clausole*

$$[\tau_1 \rightarrow P_1; \tau_2 \rightarrow \dots; \tau_k \rightarrow P_k]$$

è uno *switch* al programma  $P_i$ , deciso dal *test*  $\tau_i$ . Un *break* (nel senso del C) è implicito prima di ogni punto-e-virgola e della quadra-chiusa.  $T$  è un test sempre vero. Scriviamo  $[\tau \rightarrow \Sigma(\tau)]$  se il programma dello switch da eseguire quando  $\tau$  è vero è individuato dallo schema  $\Sigma(\tau)$ .

In certi casi invece di  $[\tau \rightarrow P; T \rightarrow Q]$  converrà scrivere  $(\mathbf{sw}, \tau, P, Q)$ .

(c)  $\mathbf{wh} \tau \mathbf{do} M \mathbf{end}$  equivale a  $M^k$ , dove  $k$  è il minimo  $j \geq 0$  che soddisfa  $\tau$ .

# Capitolo 2

## Computazioni

### 2.1 Macchine di Turing ordinarie e loro varianti

**2.1** Il tipo di dati delle *macchine di Turing* (TM) consiste di un intero  $ob \geq 0$  e di un array  $tp$  di caratteri (lettere) appartenenti ad un alfabeto  $\Gamma$ . La forma delle assegnazioni  $A$  è

$$ob ++; \quad ob --; \quad tp[ob] = a; \quad (a \in \Gamma)$$

con il significato rispettivo di aggiungere o sottrarre 1 al valore corrente di  $ob$  e di assegnare il carattere  $a$  a  $tp[ob]$ .

Una TM  $M$  con  $q \geq 1$  linee o *stati* è una sequenza (non strutturata) di switch della forma (cf. 1.4.5, una clausola per ogni  $a \in \Gamma$ )

$$\begin{aligned} 1 & : [tp[ob] = a \rightarrow A_{1a} \text{ go to } s_{1a}] \\ \dots & \\ q & : [tp[ob] = a \rightarrow A_{qa} \text{ go to } s_{qa}]. \end{aligned}$$

dove  $s_{ia} \leq q + 1$  per ogni  $i \leq q$ .  $_$  è il valore di default per ogni  $tp[h]$ ; inoltre  $_$  è il valore fisso di  $tp[0]$ , nel senso che le assegnazioni  $ob --$  e  $tp[h] = a$  vengono ignorate se  $ob = 0$ .  $M$  si ferma se e quando esegue un  $go\ to\ q + 1$ .

Immaginando l'array  $tp$  come un *nastro* (potentialmente) infinito verso sinistra, è naturale dire che, ad ogni *passo*,  $M$  *si muove a sinistra/destra* oppure *scrive a*.  $ob = j$  significa che la  $j$ -ma *cella* a sinistra di quella iniziale è osservata; nella cella  $h$  è registrato  $tp[h]$ . La cella 0 è *read-only*. In termini più concisi:

1. L'espressione  $T = U \triangleright oW$  dice che nel nastro  $T$  è registrata la stringa  $UoW_$  con  $ob = |W| + 1$  (anche:  $U \triangleright W \circ U \triangleleft W$  se  $o$  è (resp.) la prima lettera di  $W$  o l'ultima di  $U$ ; ed anche:  $T = X$  per  $T = \triangleright X$ ).

Nastri che differiscano per bianchi non significativi (a sinistra) sono considerati uguali.

2. Le tre assegnazioni saranno abbreviate nella forma (risp.)  $l$ ,  $r$  and  $a$ .
3. Data una classe di literal composti nella forma

$$[i, a, j, \mathbf{A}] \quad (1 \leq i \leq q; 1 \leq j \leq q + 1; a \in \Gamma)$$

consideriamo  $\mathbf{M}$  come un'assegnazione di valori di verità che soddisfi tutti quelli nella forma

$$[i, a, s_{ia}, \mathbf{A}_{ia}]$$

nonché tutte le condizioni di *univocità*

$$(a) \quad [i, a, j, \mathbf{A}] \rightarrow \neg[i, a, h, \mathbf{B}] \quad h \neq j \text{ ovvero } \mathbf{A} \neq \mathbf{B}.$$

**2.2** Una *descrizione istantanea* (ID)  $D$  è una coppia nella forma  $(i, T)$  dove  $T$  è un nastro.  $D$  è *iniziale* per  $i = 1$  e *finale* per  $i = q + 1$ .

**Notazione**  $D \rightarrow_{\mathbf{M}} D^*$  se  $\mathbf{M}$  trasforma  $D$  in  $D^*$  in un passo.

Una *computazione*  $C(\mathbf{M}, T)$  di  $\mathbf{M}$  per  $T$  è una sequenza  $D_1, D_2, \dots$ , tale che  $D_t \rightarrow_{\mathbf{M}} D_{t+1}$  per ogni  $t$ . Se  $D_s$  è finale, poniamo  $D_t = D_s$  per ogni  $t > s$ , e diciamo che la computazione termina.  $\mathbf{M}$  per  $T$  non è definita se nessuna ID finale viene raggiunta.

**Notazione** (1)  $\mathbf{M}(T) = S$  e (risp.)  $\mathbf{M}(T) \uparrow$ .

(2) Un literal  $[\dots]$  in un punto del discorso in cui ci si aspetterebbe un enunciato vero/falso sta per  $[\dots]$  è soddisfatto dalla TM di cui si sta parlando. Inoltre

$$\mathbf{M} = l_1 \dots l_k$$

significa che  $\mathbf{M}$  soddisfa tutti e soli i literal  $l_1 \dots l_k$ .

**2.3 Esempio** Con

$$\mathbf{R}_1 = [1, -, 2, \mathbf{r}] \quad [1, 1, 2, \mathbf{r}] \quad [2, -, 3, -] \quad [2, 1, 2, \mathbf{r}]$$

si ha  $C(\mathbf{R}_1, X_{-11} \triangleright_{-} Y) =$

$$(1, X_{-11} \triangleright_{-} Y), (2, X_{-1} \triangleright 1_{-} Y), (2, X_{-} \triangleright 11_{-} Y), (3, X \triangleright_{-} 11_{-} Y)$$

quindi  $\mathbf{R}_1(X_{-11} \triangleright_{-} Y) = X \triangleright_{-} 11_{-} Y$ .



### 2.1.2 TM con piú nastri

Le TM ordinarie non sono un modello realistico di computazione perché con un solo nastro anche un'operazione semplice come copiare un numerale richiede un tempo quadratico.

**2.7** Una TM  $M$  con  $q$  stati e con  $c$  nastri su  $\Gamma$  è un'assegnazione di verità ad una classe di literal nella forma

$$[i, \mathbf{a}, j, \mathbf{A}] \quad \mathbf{A}_h = b, l, r; \quad a_h, b \in \Gamma; \quad 1 \leq h \leq c$$

che soddisfi le condizioni di univocità analoghe a quelle sub (a). Questi literal significano che se lo stato corrente è  $i$  e se la  $c$ -pla di lettere  $\mathbf{a}$  è osservata, allora  $M$  entra nel nuovo stato  $j$  ed effettua le  $c$  assegnazioni  $\mathbf{A}$ .

**2.8** ID e computazioni sono definite come sub §2.2, con tuple  $\mathbf{T}$  di nastri invece di un singolo nastro. Tempo e spazio come in §2.4; tuttavia: (a) in un passo, una TM con  $c$  nastri effettua  $c$  assegnazioni  $\mathbf{A}$ ; (b) la lunghezza di una ID è il max tra le lunghezze dei  $c$  nastri, non la loro somma.

**2.9 Teorema** Ogni TM  $M$  con  $c$  nastri è simulata da una TM  $M_1$  con un nastro.

*Dimostrazione.* Immaginiamo il nastro di  $M_1$  diviso in blocchi di  $2l$  celle consecutive ciascuno. Quando  $M_1$  comincia a simulare il passo  $t$ , se  $a$  è nella cella  $i$  del nastro  $j$ , allora c'è una  $a$  nella cella  $2j$  del blocco  $i$ , e il bit 1 (il bit 0) è in  $2j - 1$  se quella cella (non) è osservata.  $M_1$  raggiunge la prima cella del suo nastro, e ripete, finché  $M$  non si ferma, l'invariante seguente:

(a) scorre il suo nastro da sinistra a destra finché non ha trovato tutti i  $c$  simboli osservati da  $M$ ;

(b) alla fine della parta (a),  $M_1$  sa cosa  $M$  sta per fare, e lo simula, muovendosi, questa volta, da destra a sinistra.

Si noti che, se  $M$  su  $x$  si ferma in  $n$  passi in spazio  $m$ , allora  $M_1$  richiede meno di  $2cmn \leq 2cn^2$  passi e  $\leq 2ckm$  celle, dove  $k$  è la cardinalità di  $\Gamma$ .

### 2.1.3 TM con un solo stato interno

**2.10 Le TM  $\text{nxt}_M$**  Per ogni TM con  $q$  stati e  $c$  nastri su un alfabeto  $\Gamma$  di cardinalità  $K$ , denotiamo con  $\text{nxt}_M$  la TM

1. con 1 stato e con  $(c + 1)$  nastri sull'alfabeto  $\Gamma_q$ , ottenuto aggiungendo a  $\Gamma$  le cifre  $1, \dots, q$

2. che soddisfa il literal

$$[1, \mathbf{o}, i, 2, \mathbf{A}, j]$$

sse  $M$  soddisfa il literal

$$[i, \mathbf{o}, j, \mathbf{A}]$$

Essa simula un passo di  $M$  registrando in  $c + 1$  lo stato corrente di  $M$ , e fermandosi. Sicché

$$(i, \mathbf{U} \triangleright \mathbf{W}) \rightarrow_M (j, \mathbf{U}^* \triangleright \mathbf{W}^*) \quad \text{implica} \quad (1, \mathbf{U} \triangleright \mathbf{W} \triangleright i) \rightarrow_{\text{nxt}_M} (2, \mathbf{U}^* \triangleright \mathbf{W}^* \triangleright j).$$

**2.11 Teorema** Ogni TM  $M$  con  $q$  stati e con  $c$  nastri è simulata da una TM  $M^*$  con uno stato e con  $c + 1$  nastri su un alfabeto piú ampio. Si ha cioè che

$$M(\mathbf{X}) = \mathbf{Y} \quad \text{implica} \quad M^*(\mathbf{X} \ 1) = \mathbf{Y} \ q; \quad M(\mathbf{X}) \uparrow \quad \text{implica} \quad M^*(\mathbf{X} \ 1) \uparrow.$$

*Proof.* Si ottiene  $M^*$  rimpiazzando lo stato 2 con lo stato 1 in tutti i literal soddisfatti da  $\text{nxt}_M$ , salvo quelli nella forma  $[1, \mathbf{o}, i, 2, \mathbf{A}, q]$ .

Si vede subito che  $M^*$  ripete  $\text{nxt}_M$  finché  $q$  non è registrato in  $c + 1$ , ossia finché  $\text{nxt}_M$  non restituisce una ID finale.

## 2.1.4 Chiusura rispetto ai costrutti strutturati

**2.12 Lemma** Le TMs con  $c$  nastri sono chiuse rispetto agli schemi (cf. §1.4.5):

- (1) composizione  $\{M \ N\}$ ;
- (2) ripetizioni **while**  $a \ i \ M$  decise dal simbolo osservato nel nastro  $i$ ;
- (3) diramazioni  $[a, i \rightarrow M; T \rightarrow N]$  decise dallo stesso test.

*Proof.* Siano date le TMs  $M$  ed  $N$  con  $p$  e con  $q$  stati.

(1) La loro composizione (in sequenza) è la TM con  $p + q$  stati che soddisfa

(a) tutti i literal soddisfatti da  $M$ ;

(b) ogni  $[i + p, \mathbf{a}, j + p, \mathbf{A}]$  tale che  $[i, \mathbf{a}, j, \mathbf{A}]$  è soddisfatto da  $N$ .

(2) Per ottenere **wh**  $b \ i \ M$  si rimpiazza in  $M$  ogni  $[i, \mathbf{a}, q + 1, \mathbf{A}]$  con  $[i, \mathbf{a}, 1, \mathbf{a}]$  sse  $a_i$  è  $b$ , e si lascia tutto il resto invariato.

(3) La TM  $[a, i \rightarrow M; T \rightarrow N]$  è costituita da

- (a) una prima linea che manda a 2 o a  $p + 1$  secondo il valore di  $o_i$ ;
- (b) tutte le linee di  $M$  e di  $N$ , ma con gli stati riassegnati come sub (1).

**2.13 Esempio** Definiamo (con un nastro)

$$\mathbf{sx} = [1, a, 2, l].$$

Il metodo della parte relativa alla *while* della dimostrazione dà

$$\mathbf{wh}(1)\mathbf{sx} = \begin{array}{l} [1, 1, 2, 1] \quad [1, -, 3, -] \\ [2, a, 1, qa] \end{array}$$

quello per la concatenazione dà

$$\{\mathbf{sx} \mathbf{wh}(1)\mathbf{sx}\} = \begin{array}{l} [1, a, 2, l] \\ [2, 1, 3, 1] \quad [2, -, 4, -] \\ [3, a, 2, a]. \end{array}$$

Confrontando con la definizione precedente di  $L_1$ , si vede che la strutturazione *uniforme* ha un costo. Onde la tesi di Knuth, secondo la quale i programmi vanno *concepiti* in modo strutturato fin dall'inizio. Sono noti esempi di *flow chart* non strutturati la cui lunghezza cresce in modo esponenziale quando vengono strutturati. È però importante notare che manca la loro interpretazione. Vedremo che, per ciascuna interpretazione delle operazioni indicate nelle loro box, c'è una realizzazione strutturata di lunghezza ragionevole. Essi pertanto non confutano la tesi della strutturabilità di tutti i programmi.

## 2.2 TM strutturate

Si chiama TM *push-down* una TM i cui nastri sono sempre bianchi alla sinistra delle celle osservate. Essa accede dunque all'informazione sui nastri solo in modo *last-in-first-out*. Ne studieremo una versione strutturata, con tre nastri sull'alfabeto  $\Gamma = \{0, \cdot, -\}$ . A tal fine cominciamo col considerare le TM

1.  $\text{pop}_i$  ( $i = 1, 2, 3$ ) le quali lasciano invariati i nastri  $j \neq i$ , e si spostano a destra sul nastro  $i$ , dopo aver cancellato (scrivendo il simbolo  $-$ ) il carattere precedentemente osservato;
2.  $\text{psh}_i\langle a \rangle$  ( $i = 1, 2, 3; a \in \Gamma$ ) le quali lasciano invariati i nastri  $j \neq i$ , si spostano a sinistra sul nastro  $i$ , e quindi scrivono il simbolo  $a$ .

**2.14** Le TM *strutturate* (STM) costituiscono una classe di TM ordinarie, definita per chiusura delle TM  $\text{pop}_i$  e  $\text{psh}_i\langle a \rangle$  rispetto ai costrutti strutturati del Lemma 2.12. Esse sono descritte in *forma implicita* da liste nella forma ( $M$  ed  $N$  sono STM,  $i = 1, 2, 3$ ,  $a \in \Gamma$ )

$$(\text{pop}.i, 0) \quad (\text{psh}.a.i, 0) \quad (M, N) \quad (\text{wh}.a.i, M) \quad (\text{sw}.a.i, M, N)$$

dove i simboli  $\text{pop}.i, \dots, \text{sw}.a.i$  denotano 30 numerali diversi che è inutile precisare. Le forme esplicite (leggibili) sono riportate nella tabella seguente

$\text{pop}_i$	$(\text{pop}.i)$
$\text{psh}_i\langle a \rangle$	$(\text{psh}.a.i)$
$\{M N\}$	$(M, N)$
$\text{wh}(a, i) M \text{ end}$	$(\text{wh}.a.i, M)$
$[(a, i) \rightarrow M T \rightarrow N]$	$(\text{sw}.a.i, M, N)$

Stabiliamo inoltre che  $i = 1$  per default, e che parentesi tonde e graffe saranno omesse quando il parsing è ovvio.

Siccome le STM sono casi particolari di TM si applicano loro le nozioni dei §2.1.1 e 2.8, nonché le nozioni di tempo e spazio. Ha quindi senso scrivere

$$M(\mathbf{X}) = \mathbf{Y}.$$

**2.15 Esempio** (1) L'*identità* è data da (cf. §1.4.3 per l'uso di  $\equiv$  e di  $=$ )

$$\begin{aligned} \text{id} &\equiv \text{psh}\langle - \rangle; \text{pop} \equiv (\text{psh}..1, \text{pop}.1) \\ \text{id}(\mathbf{X}) &= \mathbf{X}. \end{aligned}$$

(2) *move* (una stringa) di zeri da (nastro) 1, a 2 e 3

$$\begin{aligned} \text{mvz} &\equiv \text{wh } 0 \{ \text{pop}; \text{push}_2\langle 0 \rangle; \text{push}_3\langle 0 \rangle \} \\ \text{mvz}(0^k X, Y, Z) &= \_X, 0^k Y, Z \end{aligned}$$

**2.16 Schemi derivati** (1) Definiamo

$$\begin{aligned} \text{wh}(a, i) \vee (b, j)M &\equiv \text{wh}(a, i) M; \\ &\quad \text{wh}(b, j) \{M; \text{wh}(a, i) M\} \end{aligned}$$

Un attimo di riflessione basta per vedere che questo schema ripete  $M$  *while*  $a$  è osservato in  $i$  ovvero  $b$  lo è in  $j$ . Un informatico la verificherà osservando che l'espressione regolare  $a^*(ba^*)^*$  associata al flow chart di questo schema è equivalente a  $(a|b)^*$ . Altri può convincersi con qualche simulazione bruta.

Poiché non è necessario assumere  $i \neq j$ , possiamo in questo modo definire lo schema

$$\text{wh}(\neq b, i)M$$

che ripete  $M$  while  $o_i \neq b$ .

(2) Lo schema che segue applica  $M$  se il nastro  $i$  comincia con  $ab$

$$\begin{aligned} [ab, i \rightarrow M; T \rightarrow N] &\equiv [a, i \rightarrow \text{pop}_i; [b, i \rightarrow \text{push}_i\langle a \rangle; M; \\ &\quad T \rightarrow \text{push}_i\langle a \rangle; N]; \\ &\quad T \rightarrow N]. \end{aligned}$$

Generalizzandolo si ottengono switch decisi da stringhe di lunghezza qualsiasi.

(3) Altri *switch* decisi da espressioni booleane nei caratteri osservati possono essere definiti facilmente. Per esempio

$$[(a, i) \wedge (b, j) \rightarrow M T \rightarrow N] \equiv [(a, i) \rightarrow [(b, j) \rightarrow M T \rightarrow N] T \rightarrow N]$$

**2.17 Elaborazione di stringhe** Chiamiamo *connessa* una stringa in cui non figurino bianchi.

(1) Per ogni stringa  $U = a_1 \dots a_m$  definiamo (cf. 1.4(1))

$$\begin{aligned} \text{psh}_i\langle U \rangle &\equiv \text{psh}_i\langle a_m \rangle \dots \text{psh}_i\langle a_1 \rangle \\ \text{psh}\langle U \rangle(\mathbf{X}) &= U \mathbf{X}. \end{aligned}$$

(2) *reverse-move* e *copy* di una stringa connessa in  $i$

$$\begin{aligned} \text{rvstr}_{ij} &\equiv \text{wh}(\neq -, i) [a, i \rightarrow \text{pop}_i \text{psh}_j\langle a \rangle] \\ \text{rvstr}_{ijk} &\equiv \text{wh}(\neq -, i) [a, i \rightarrow \text{pop}_i \text{psh}_j\langle a \rangle \text{psh}_k\langle a \rangle] \\ \text{cstr}_{ik} &\equiv \text{psh}_j\langle - \rangle \text{rvstr}_{ij} \text{rvstr}_{jik} \text{pop}_j \\ \text{rvstr}_{12}(U \_ X, Y, Z) &= \_ X, U^R Y, Z \\ \text{rvstr}_{123}(U \_ X, Y, Z) &= \_ X, U^R Y, U^R Z \\ \text{cstr}_{12}(U \_ X, Y, Z) &= U \_ X, U Y, Z. \end{aligned}$$

(3) Uguaglianza tra stringhe connesse  $U$  e  $W$

$$\begin{aligned} \text{eq}_{ij} &\equiv \text{psh}_3\langle - \rangle \\ &\quad \text{wh}(\_, 3)\{ \\ &\quad \quad [(0, 1) \wedge (0, 2) \rightarrow \{\text{pop pop}_2\} \\ &\quad \quad (1, 1) \wedge (1, 2) \rightarrow \{\text{pop pop}_2\} \\ &\quad \quad (\_, 1) \wedge (\_, 2) \rightarrow \text{psh}_3\langle 0 \rangle \\ &\quad \quad T \rightarrow \text{psh}_3\langle 1 \rangle] \} \\ &\quad \text{erstr}_1 \text{erstr}_2 \text{pop}_2 \\ &\quad [(a, 3) \rightarrow \text{pop}_3 \text{psh}\langle a \rangle \text{pop}_3] \\ \text{eq}_{12} &= (U \_ X, W \_ Y, Z) = b \_ X, Y, Z \quad b = 0/1 \text{ se } U \text{ uguale/diversa } W \end{aligned}$$

**2.18 Elaborazione di liste** (1) *reverse and move* la lista  $u$  da  $i$  a  $j$  (usando  $k$  come stack del numero di zeri attesi)

$$\begin{aligned} \text{rvli}_{ij} &\equiv [\cdot, i \rightarrow \text{psh}_k\langle 00\_ \rangle \text{pop}_i \text{psh}_j\langle \cdot \rangle \\ &\quad \text{wh}(0, k) \\ &\quad [\cdot, i \rightarrow \text{psh}_j\langle \cdot \rangle \text{psh}_k\langle 0 \rangle \\ &\quad \quad T \rightarrow \text{psh}_j\langle 0 \rangle \text{pop}_k] \\ &\quad \text{pop}_i \text{ end} \\ &\quad \text{pop}_k \\ \text{rvli}_{12}(uX, Y, Z) &= X, u^R Y, Z. \end{aligned}$$

(2) *erasing e checking*. Per la prima STM si elimini in  $\text{rvli}$  ogni  $\text{psh}_j$ . Per la seconda restituisce  $\_$  sse  $u$  non è una lista.

(3) *move e copy*

$$\begin{aligned} \text{cli}_{ij} &\equiv \text{psh}_k\langle \_ \rangle \text{rvli}_{ik} \text{rvstr}_{kji} \text{pop}_k \\ \text{mli}_{ij} &\equiv \text{psh}_k\langle \_ \rangle \text{rvli}_{ik} \text{rvstr}_{kj} \text{pop}_k \\ \text{cli}_{13}(uX, Y, Z) &= u X, Y, u Z. \end{aligned}$$

Il tempo per queste STM è ovviamente lineare perché ciascun carattere dell'input è spostato per un numero di volte maggiorato da una costante (minore di dieci, tanto per fare i signori).

(4) *head, tail e quote*

$$\begin{aligned} \text{tl} &\equiv \text{pop erli} & \text{tl}[\cdot xy] &= y \\ \text{hd} &\equiv \text{pop mli}_{12} \text{erli mli}_{21} & \text{hd}[\cdot xy] &= x \\ \text{qt}\langle u \rangle &\equiv \text{erli psh}\langle u \rangle & \text{qt}\langle u \rangle[x] &= u. \end{aligned}$$

**Notazione**  $I$  per  $M$  se  $M$  è una composizione delle STM  $\text{hd}$  e  $\text{tl}$ , tale che (cf. §1.17(2))

$$M[x] = [x]_I.$$

Inoltre “ $u$ ” per  $\text{qt}\langle u \rangle$ .

**2.19** (1) Una WBTM è una STM  $M$  che si comporta bene (*well behaves*), nel senso che essa modifica solo la lista sul top del nastro 1, ed usa gli altri nastri solo come scratch, lasciandoli immutati. Ossia

$$M(u X, Y, Z) = w X, Y, Z$$

(2) Schema per unire in una sola lista gli output delle WBTM  $M$  ed  $N$  per lo stesso input

$$\begin{aligned} \text{jo}(M, N) &\equiv \text{cli}_{13} M \text{mli}_{12} \text{mli}_{31} N \text{mli}_{21} \text{psh}\langle \cdot \rangle \\ \text{jo}(M, N)[z] &= (x, y) \quad \text{se} \\ M[z] &= x \\ N[z] &= y. \end{aligned}$$

**Notazione**

$$\begin{aligned} \langle M, N \rangle &\equiv \text{jo}(M, N) \\ \langle M_0, \dots, M_k \rangle &\equiv \text{jo}(M_0, \langle M_1, \dots, M_k \rangle) \end{aligned}$$

(3) Schemi in WBTM che decidono i loro test

$$\begin{aligned} [J = L \rightarrow M \ T \rightarrow N] &\equiv \text{cli}_{13} J \text{psh}_2\langle - \rangle \text{mli}_{12} \\ &\quad \text{psh}_- \text{cli}_{31} L \text{eq}_{12} \\ &\quad [0 \rightarrow \text{pop} \text{pop} \text{mli}_{31} M \\ &\quad T \rightarrow \text{pop} \text{pop} \text{mli}_{31} N] \\ [J = L \rightarrow M \ T \rightarrow N][z] &= [J[z] = L[z] \rightarrow M[z] \\ &\quad T \rightarrow N[z]] \end{aligned}$$

$$\begin{aligned} \text{wh}(J = L) M &\equiv [J = L \rightarrow \{ \text{psh}\langle 0 \rangle \\ &\quad \text{wh } 0 \{ \\ &\quad \quad \{ \text{pop } M \\ &\quad \quad [J = L \rightarrow \text{push}\langle 0 \rangle \\ &\quad \quad T \rightarrow \text{push}\langle \cdot \rangle] \} \} \\ &\quad T \rightarrow \text{id}] \\ \text{wh}(J = L) M[z] &= \text{wh}(J[z] = L[z]) M[z]. \end{aligned}$$

### 2.2.1 Una TM strutturata universale

**2.20 Teorema** Esiste una STM *universale*  $U$  tale che, per ogni STM  $x$  ed ogni  $\mathbf{Z}$

$$U[x \ \mathbf{Z}] \simeq x[\mathbf{Z}]$$

*Dimostrazione.* La STM  $U$  riportata piú avanti si comporta come segue  
 $\ell 1$  ripete finché  $x$  non si riduce a 0 l'invariante delle righe successive  
 $\ell 2$  se  $x$  non è la composizione di due STM aggiunge uno 0 come coda fittizia;  
 $\ell 3$  casi di **code** in  $x = ((\text{code}, y), u)$

$\ell 4,5$  se  $\text{code} = \text{pop.i}/\text{psh.a.i}$  si ha  $y = 0$ ; esegue e prosegue con  $u$ ; nell'esecuzione, per  $i = 1$  deve prima salvare  $x$  sul nastro 2, e poi recuperarla (la stessa cosa vale anche per gli altri casi di  $\text{code}$ );

$\ell 6$  se  $x = ((\text{wh.a.i}, \text{M}), u)$  e il test risulta vero continua con  $((\text{M}, (\text{wh.a.i}, \text{M})), u)$ ;

$\ell 7$  se  $a$  non è osservato in  $i$ , continua con  $u$ ;

$\ell 8, 9$  se  $a$  è osservato in  $i$ , continua con  $(y_1, u)$ ; altrimenti con  $(y_2, u)$ ;

$\ell 10$  se la forma di  $x$  non è  $((\text{code}, y), u)$ , si ha  $x = ((\text{M}, \text{N}), u)$  e si continua con  $(\text{M}, \text{N}, u)$ ; si osservi che in questo caso si ha sempre  $u \neq 0$ ;

$\ell 11$  eliminazione della coda fittizia nei casi  $\ell 6, 8$  e  $9$ .

$$\begin{array}{l} \text{wh}(\cdot) \{ \\ \quad [1 \leq \text{"30"} \rightarrow \langle \text{id}, \text{"0"} \rangle T \rightarrow \text{id}] \\ \quad [11 = \\ \quad \text{pop.i} \quad \rightarrow \text{prel}_i \text{ pop}_i \text{ postl}_i \omega \\ \quad \text{psh.a.i} \quad \rightarrow \text{prel}_i \text{ psh}_i \langle a \rangle \text{ postl}_i \omega \\ \quad \text{wh.a.i} \quad \rightarrow \text{prel}_i [a, i \rightarrow \text{postl}_i \langle \langle 12, 1 \rangle, \omega \rangle \\ \quad \quad \quad T \rightarrow \text{postl}_i \omega] \\ \quad \text{sw.a.i} \quad \rightarrow \text{prel}_i [a, i \rightarrow \text{postl}_i \langle 12, \omega \rangle \\ \quad \quad \quad T \rightarrow \text{postl}_i \langle 13, \omega \rangle] \\ \quad T \quad \quad \rightarrow \langle 11, 12, \omega \rangle] \\ \quad [0 \rightarrow \text{id} \quad 2 = \text{"0"} \rightarrow 1 T \rightarrow \text{id}] \quad \} \\ \text{pop} \end{array}$$

dove

$$\text{prel}_1 \equiv \text{mli}_{12} \quad \text{postl}_1 \equiv \text{mli}_{21}$$

mentre per  $i = 2, 3$ , non occorrendo proteggere la macchina oggetto, possiamo porre

$$\text{prel}_i \equiv \text{postl}_i \equiv \text{id}$$

e dove ovviamente

$$[1 \leq \text{"h"} \rightarrow \text{M} T \rightarrow \text{N}] \equiv [1 = \text{"1"} \rightarrow \text{M} \quad 1 = \text{"2"} \rightarrow \text{M} \dots 1 = \text{"h"} \rightarrow \text{M} \quad T \rightarrow \text{N}]$$

# Capitolo 3

## Ricorsione

### 3.1 List Processing

Il livello del linguaggio che ora introduciamo è abbastanza alto da consentire costruzioni interessanti, ma non al punto di impedire ai suoi elementi di essere, a loro volta, oggetto di computazioni astratte.

**3.1 SINTASSI** Le *list processing machines* (LPM) sono liste definite per chiusura delle LPM di base (una  $(qt, u)$  per ciascuna lista  $u$ ;  $hd, \dots, jo$  sono 7 numerali distinti che non occorre precisare, cf. §1.17(4))

(a)  $hd \quad t1 \quad (qt, u)$

rispetto agli schemi

(b)  $(M, N) \quad (hcmp, M, N) \quad (wh, J, L, M) \quad (sw, J, L, M, N) \quad (jo, M, N)$

Per migliore leggibilità si scrive “ $u$ ” per  $(qt, u)$ , mentre gli schemi divengono

$\{M N\} \quad M_{;h} N \quad wh (J = L) M \quad [J = L \rightarrow M \ T \rightarrow N] \quad \langle M, N \rangle$

e le graffe sono omesse quando il parsing è ovvio o facilitato dalle indentazioni, etc. Ogni lista  $x$  non conforme a questa sintassi è una LPM *impropria*, spesso denotata da  $id$ . L’input/output di queste macchine è una lista  $z$ . Scriviamo  $x[z] \simeq y$  se, nella semantica che ora definiamo, la LPM  $x$  per input  $z$  dà in uscita  $y$  ovvero diverge.

**SEMANTICA**  $hd, t1, qt, \langle M, N \rangle$  e  $\{M N\}$  come nel §2.18.  $\llbracket id \rrbracket[z] = z$ . Per la *head composition* si ha

$$\llbracket (hcmp, M, N) \rrbracket[(y, \vec{z})] = \llbracket N \rrbracket[(\llbracket M \rrbracket[y], \vec{z})].$$

Switch e ripetizioni while sono decise dal test  $J[x] = L[x]$  ossia

$$\begin{aligned} \llbracket (\mathbf{sw}, J, L, M, N) \rrbracket [z] &\uparrow && \text{qualora} && \llbracket J \rrbracket [z] \uparrow && \text{ovvero} && \llbracket L \rrbracket [z] \uparrow \\ \llbracket (\mathbf{sw}, J, L, M, N) \rrbracket [z] &\simeq && \llbracket M \rrbracket [z] && \text{qualora} && \llbracket J \rrbracket [z] = && \llbracket L \rrbracket [z] \\ \llbracket (\mathbf{sw}, J, L, M, N) \rrbracket [z] &\simeq && \llbracket N \rrbracket [z] && \text{qualora} && \llbracket J \rrbracket [z] \neq && \llbracket L \rrbracket [z] \\ \llbracket (\mathbf{wh}, J, L, M) \rrbracket [z] &\simeq && \llbracket [J = L \rightarrow \{M \text{ wh } (J = L) M\} \ T \rightarrow \text{id}] \rrbracket [z]. \end{aligned}$$

**Notazione** Come in §2.18:

- (a)  $I$  per composizioni delle LPM  $\mathbf{hd}$  e  $\mathbf{tl}$ , tali che  $M[x] = [x]_I$ .
- (b)  $\langle L, M, N \rangle \equiv \langle L, \langle M, N \rangle \rangle$ .

**3.2 Esempio** Si ha  $(\mathbf{qt}, \mathbf{qt})[x] = \mathbf{qt}$  nonché  $\mathbf{id}[x] = x$ . Pertanto

(A)  $\langle (\mathbf{qt}, \mathbf{qt}), \mathbf{id} \rangle [x] = (\mathbf{qt}, x) = "x"$ .

Questa LPM trasforma l'input  $x$  nella LPM che dà in uscita  $x$ .

Per ciascuna costante  $u$ , la LPM seguente unisce  $u$  al suo input

(B)  $\langle \mathbf{jo}, (\mathbf{qt}, u), \mathbf{id} \rangle [x] \equiv \langle "u", \mathbf{id} \rangle [x] = (u, x)$ .

Così come la LPM (A) anche la prossima LPM dà in uscita una LPM

(C)  $\langle \langle \mathbf{jo}, \langle \mathbf{qt}, \mathbf{id} \rangle, \mathbf{id} \rangle [w] = (\mathbf{jo}, "w", \mathbf{id})$

Può aiutare a capire la sintassi osservare che, invece,  $\langle \mathbf{qt}, \mathbf{qt} \rangle = (\mathbf{jo}, \mathbf{qt}, \mathbf{qt})$  è una LPM impropria (perché la sintassi detta  $(\mathbf{jo}, M, N)$ , ma invece  $\mathbf{qt}$  da sola non è una LPM propria). Quindi

$$\langle \langle \mathbf{qt}, \mathbf{qt} \rangle, \mathbf{id} \rangle [x] = x.$$

**3.3 Nota** In uno *pseudo-C* (ossia senza dichiarazioni ed altre precisazioni sintattiche), analoghi delle LPM 3.2(A) e (B) sono dati più o meno da

```
fgets (list);
printf("printf(\"%s,list\n\n\");\n");
```

e da (ma con output  $u, w$  invece di  $(u, x)$ )

```
strcpy(first, " u");
fgets (second);
strcat (first, ",");
strcat (first, second);
```

### 3.1.1 Digressione: un programma C che si riproduce

```
char s[ ]={
    '\t',
    '0',
    '\n',
    '}',
    ';',
    '\n',
    '\n',
    '/',
    '*',
    '\n',

    (200 righe ca. omesse)

    0,
};

/*
 * la stringa s rappresenta tutto
 * il programma da 0 in poi
 */
main( )
{
    int i;
    printf("char\t s[ ]={\n");
    for(i=0;s[i];i++)
        printf("\t%d,\n",s[i]);
    printf("%s",s);
}
```

Chiamiamo P questo programma e Q la sua uscita. Abbiamo  $Q = Q_1Q_2$ , dove  $Q_1$  è l'output delle righe di P fino allo scope della `for` incluso e  $Q_2$  è prodotto dall'ultima `printf`. Si verifica per simulazione che  $Q_1 =$

```
char s[ ]={
```

```
9,  
48,  
10,  
125,  
73,  
10,  
10,  
47,  
42,  
10,
```

(200 righe ca. omesse)

a questo punto il test della `for` diviene falso (perché si ha `s[i] = 0` — si osservi la differenza tra `'0' = 48` e `0`) e si passa all'ultima `printf`, ottenendo  $Q_2 =$

```
0  
};
```

```
/*  
 * la stringa s rappresenta tutto  
 * il programma da 0 in poi  
 */  
main( )  
{  
    int i;  
    printf("char\t s[ ]={\n");  
    for(i=0;s[i];i++)  
        printf("\t%d,\n",s[i]);  
    printf("%s",s);  
}
```

$Q$  differisce da  $P$  solo perché l'enumerazione delle componenti dell'array di caratteri  $s$  è fatta con i numeri ASCII, invece che con caratteri espliciti. Siccome la seconda `printf` usa `%d`, l'output di  $Q$  è  $Q$  stesso.

**3.4 Im/morale** Così come  $Q$  scrive perfino il suo commento  $C$ , si può scrivere un qualunque programma che, non solo si riproduce, ma porta con sé, invece di un innocente commento, tutto il piú o meno esplosivo bagaglio  $B$  che vuole.

**3.5 Cattivo consiglio** Chiamiamo  $P(B)$  il programma ottenuto sostituendo  $C$  in  $P$  con  $B$ . Si è scritto  $P$  in modo che sia facile scrivere un programma  $R$  che per input il solo  $B$  dia in uscita  $P(B)$ , evitando la parte piú noiosa e soprattutto piú suscettibile di errori del lavoro di redazione di  $P(B)$ .

## 3.2 Il teorema della ricorsione

Il prossimo teorema fornisce un metodo generale per scrivere programmi come quello appena visto e per scrivere alcuni programmi ricorsivi.

**3.6 Teorema di Kleene** Per ogni  $x$  esiste un *punto fisso*  $u$  tale che

$$u[z] \simeq x[(u, z)].$$

*Costruzione.* Definiamo una prima LPM  $s$

- (a)  $s \equiv \langle \langle \text{“jo”}, \langle \text{“qt”}, \text{id} \rangle, \text{“id”} \rangle, \text{id} \rangle$   
 (b)  $s[w] = ((\text{jo}, \text{“w”}, \text{id}), w)$  Esempio 3.2(C).

Data  $x$ , definiamo ora un'altra LPM  $x^*$

- (c)  $x^* \equiv (\text{hcmp}, s, x) \equiv s;_h x$   
 (d)  $x^*[(y, z)] \simeq x[(s[y], z)]$

con “ $\simeq$ ” perché se  $x$  diverge, diverge pure  $x^*$ . Possiamo ora porre

- (e)  $u = s[x^*]$

*Dimostrazione.* Abbiamo

$$\begin{aligned} u[z] &\equiv ((\text{jo}, \text{“x”}, \text{id}), x^*)[z] && \text{(b) con } x^* \text{ come } w \\ &\equiv \{ \langle \text{“x”}, \text{id} \rangle x^* \}[z] && \text{ossia composizione di } \langle \text{“x”}, \text{id} \rangle \text{ con } x^* \\ &\simeq x^*[(x^*, z)] && \text{Esempio 3.2(C)} \\ &\equiv s;_h x[(x^*, z)] && \text{(c)} \\ &\simeq x[(s[x^*], z)] && \text{(d)} \\ &\simeq x[(u, z)] && \text{definizione (e) di } u \end{aligned}$$

### 3.2.1 Applicazione e commento: una LPM che si riproduce

**3.7** La procedura implicita nella dimostrazione, applicata al caso  $x = \text{hd}$ , dà (qui con “=” invece di “ $\simeq$ ” perché  $\text{hd}$  è ovviamente totale)

$$\begin{aligned}
 x^* &\equiv (\text{hcmp}, \mathbf{s}, \text{hd}) && (c) \\
 u &= \mathbf{s}[x^*] && (e) \\
 u &\equiv ((\text{jo}, \langle \langle \text{“hcmp}, \mathbf{s}, \text{hd} \text{”} \rangle, \text{id} \rangle, (\text{hcmp}, \mathbf{s}, \text{hd})) && (b) \\
 &\equiv \{ \langle \langle \text{“hcmp}, \mathbf{s}, \text{hd} \text{”} \rangle, \text{id} \rangle (\text{hcmp}, \mathbf{s}, \text{hd}) \} && \text{composizione} \\
 u[z] &= (\text{hcmp}, \mathbf{s}, \text{hd})[\{ \langle \langle \text{“hcmp}, \mathbf{s}, \text{hd} \text{”} \rangle, \text{id} \rangle [z] \}] && \text{semantica della composizione} \\
 &\equiv (\text{hcmp}, \mathbf{s}, \text{hd})[(\text{hcmp}, \mathbf{s}, \text{hd}), z] && \text{Esempio 3.2(esB)} \\
 &\equiv \mathbf{s}_{;h} \text{hd}[(\text{hcmp}, \mathbf{s}, \text{hd}), z] && \text{definizione di hcmp} \\
 &= \text{hd}[\mathbf{s}[(\text{hcmp}, \mathbf{s}, \text{hd}), z]] && \text{semantica di hcmp} \\
 &\equiv \text{hd}[\mathbf{s}[x^*], z] && \ell 1 \\
 &= \text{hd}[(u), z] && \ell 2 \\
 &= u && \text{semantica di hd.}
 \end{aligned}$$

**3.8 Commento** I programmi  $u$  forniti dal teorema contengono:

1. una copia del programma  $x$  originale;
2. le istruzioni per il proprio *boot*; esse usano la copia di  $x$  registrata nella loro memoria finita interna per ricostruire  $u$ , e scriverla.

Questo consentirà le auto-chiamate ricorsive.

### 3.2.2 LPM universale

**3.9 Teorema** Esiste una LPM  $U$  tale che

$$U[(x, z)] \simeq x[z]$$

*Costruzione.* Si cicla cercando di ridurre  $x$  e si termina se e quando  $x$  è ridotta ad una LPM di base o ad una **wh** il cui test risulta falso. Il problema principale è che in questo linguaggio occorre una chiamata ricorsiva per decidere i test della LPM oggetto. Infatti in un caso come

$$U[(\mathbf{sw}, \mathbf{J}, \mathbf{L}, \mathbf{M}, \mathbf{N}), \mathbf{K}], z]$$

per eseguire  $\mathbf{M}$  oppure  $\mathbf{N}$  bisogna prima aver eseguito sia  $\mathbf{J}$  che  $\mathbf{L}$ . Occorrerebbe qualcosa come

$$\begin{aligned}
 U[(\mathbf{sw}, \mathbf{J}, \mathbf{L}, \mathbf{M}, \mathbf{N}), \mathbf{K}], z] &= \begin{array}{l} U[(\mathbf{J}, z)] = U[(\mathbf{L}, z)] \\ T \end{array} \begin{array}{l} \rightarrow U[(\mathbf{M}, \mathbf{K}), z] \\ \rightarrow U[(\mathbf{N}, \mathbf{K}), z] \end{array} ]
 \end{aligned}$$

ma il linguaggio permette di ripetere l'invariante delle while non le while intere. Definiamo prima una LPM  $U^*$  il cui intended input è costituito da *tre* componenti:  $y$  (programma generico da chiamare),  $x$  (la macchina oggetto) e  $z$  (l'input oggetto). Ricorriamo poi al teorema di ricorsione per far coincidere il comportamento della LPM in corso di costruzione con quello di  $y$ .

Per trovare sempre allo stesso *indirizzo* 21 il numerale che ci dice la prossima azione da simulare, ricorriamo al trucco di aggiungere alla macchina oggetto  $M$  una *coda fittizia* 0 ogni volta che la sua forma non sia  $M = JK$ .

$$\begin{array}{l}
 U^*[(y, x, z)] \equiv \\
 \text{wh}(2 \neq "0") \\
 \quad \text{fict.tl} \\
 \quad [21 = \text{"hd"} \quad \rightarrow \text{ex.hd} \quad 21 = \text{"t1"} \quad \rightarrow \text{ex.tl} \\
 \quad 211 = \text{"qt"} \quad \rightarrow \text{ex.qt} \\
 \quad 211 = \text{"sw"} \quad \rightarrow \text{ex.sw} \quad 211 = \text{"wh"} \quad \rightarrow \text{ex.wh} \\
 \quad 211 = \text{"hcmp"} \quad \rightarrow \text{ex.hcmp} \quad 211 = \text{"jo"} \quad \rightarrow \text{ex.jo} \\
 \quad T \quad \rightarrow \text{ex.cmp} ] \\
 \quad \text{er.fct.tl} \\
 \text{tl tl}
 \end{array}$$

dove si veda §3.12 per il test su *diverso* nella *wh* in  $\ell 1$  e:

(1) si aggiunge una coda fittizia se la forma di  $x$  non è  $MN$  ponendo

$$\begin{array}{l}
 \text{fict.tl} \equiv [(2 = \text{"hd"} \vee 2 = \text{"t1"} \vee 21 = \text{"qt"} \vee 21 = \text{"sw"} \\
 \quad \vee 21 = \text{"wh"} \vee 21 = \text{"hcmp"} \vee 21 = \text{"jo"}) \quad \rightarrow \langle 1, \langle 2, "0" \rangle, 3 \rangle \\
 T \quad \rightarrow \text{id}]
 \end{array}$$

(2) Le seguenti LPM eseguono le LPM di base e continuano con la coda di  $x$

$$\begin{array}{l}
 \text{ex.hd} \equiv \langle 1, 2\omega, 31 \rangle \\
 \text{ex.tl} \equiv \langle 1, 2\omega, 3\omega \rangle \\
 \text{ex.qt} \equiv \langle 1, 2\omega, 21\omega \rangle
 \end{array}$$

(3) Il caso  $x = ((\text{sw}, J, L, M, N), w)$  viene gestito

(a) decidendo il test mediante la LPM  $y$  chiamata con input  $(J, z)$  e risp.  $(L, z)$ ;

(b) continuando con  $(y, M/N, z)$  secondo l'esito del test:

$$\begin{array}{l}
 \text{ex.sw} \equiv [\{\langle 212, 3 \rangle 1\} = \{\langle 213, 3 \rangle 1\}] \rightarrow \langle 1, \langle 214, 2\omega \rangle, 3 \rangle \\
 T \quad \rightarrow \langle 1, \langle 215, 2\omega \rangle, 3 \rangle
 \end{array}$$

Si veda la Nota 3.10 per una spiegazione del numero degli argomenti di  $y$ .

(4) Il caso  $x = ((\mathbf{wh}, \mathbf{J}, \mathbf{L}, \mathbf{M}), w)$  viene gestito in modo simile da

$$\text{ex.wh} \equiv \begin{array}{l} \{ \langle 212, 3 \rangle 1 \} = \{ \langle 213, 3 \rangle 1 \} \rightarrow \langle 1, \langle 214, 2 \rangle, 3 \rangle \\ T \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \rightarrow \langle 1, 2\omega, 3 \rangle \end{array}$$

perché in questo caso  $\langle 214, 2 \rangle$  dà  $(\mathbf{M}, (\mathbf{wh}, \mathbf{J}, \mathbf{L}, \mathbf{M}), w)$ .

(5) Il caso  $x = ((\mathbf{hcmp}, \mathbf{M}, \mathbf{N}), w)$  viene gestito da

$$\text{ex.hcmp} \equiv \langle \mathbf{a}1, \langle 213, 2\omega \rangle, \langle \mathbf{b} \{ \langle 212, 31 \rangle 1 \}, 3\omega \rangle \mathbf{b} \rangle \mathbf{a}$$

che, se e quando  $\{ \langle 212, 31 \rangle 1 \} = y[(\mathbf{M}, z_1)]$  restituisce la lista  $\mathbf{M}[z_1]$ , continua con

$$(y, (\mathbf{N}, w), (\mathbf{M}[z_1], z_\omega))$$

(6) Il caso  $x = ((\mathbf{jo}, \mathbf{M}, \mathbf{N}), w)$  viene gestito da

$$\text{ex.jo} \equiv \langle \mathbf{a}1, 2\omega, \langle \mathbf{b} \{ \langle \mathbf{c}212, 3 \rangle \mathbf{c} 1 \}, \{ \langle \mathbf{c}21\omega, 3 \rangle \mathbf{c} 1 \} \rangle \mathbf{b} \rangle \mathbf{a}$$

che, se e quando le due chiamate  $y[(\mathbf{M}, z)]$  e  $y[(\mathbf{N}, z)]$  restituiscono le liste  $s$  e  $t$ , esce dal ciclo con  $(y, w, (s, t))$ .

(7) Nel caso  $x = ((\mathbf{M}, \mathbf{N}), w)$  si esce dal ciclo con  $(y, (\mathbf{M}, \mathbf{N}, w), z)$  mediante la LPM

$$\text{ex.cmp} \equiv \langle 1, \langle 211, 21\omega, 2\omega \rangle, 3 \rangle$$

(8) Infine, se la LPM oggetto non è svanita, la coda fittizia viene eliminata da

$$\text{er.fct.tl} \equiv \begin{array}{l} [2 = \text{"0"} \rightarrow \text{id} \\ 2\omega = \text{"0"} \rightarrow \langle 1, 21, 3 \rangle \\ T \rightarrow \text{id}] \end{array}$$

**Definizione**  $\mathbf{U}$  è la LPM associata dal teorema di ricorsione ad  $\mathbf{U}^*$ .

**3.10 Nota** (1) Può sembrare strano che  $y$  (e quindi  $\mathbf{U}$ ) tratti con quel  $3$  una terza componente, ma venga chiamata passandole due valori soli. La spiegazione è che tutte le LPM sono applicate ad una sola lista, il cui numero di componenti si evolve durante la computazione. Nel nostro caso,  $\mathbf{U}$  ha un intended input  $(x, z)$  con due componenti, mentre  $\mathbf{U}^*$  ne ha uno con tre. La prima cosa che  $\mathbf{U}$  fa è un booting di sé stessa che aggiunge una terza componente.

(2) In **ex.sw** e altrove, alla destra di  $\rightarrow$  non si poteva scrivere  $\{ \langle \langle 214, 2\omega \rangle, 3 \rangle 1 \}$  invece di  $\langle 1, \langle 214, 2\omega \rangle, 3 \rangle$  per due ragioni.

(a) Si sarebbe sostituite dei ritorni all'iterazione principale con una o due ricorsioni in più, che è sempre cattiva pratica.

(b) Si sarebbe dovuto aggiungere una testa fittizia con funzione di **break** per impedire che l'uscita dalla chiamata ricorsiva fosse ulteriormente e indebitamente processata dal ciclo principale.



### 3.2.4 Teorema di Rogers

Il prossimo teorema dice che ogni schema di programma ammette una soluzione ricorsiva (altri commenti in Nota 3.16). Esso consente inoltre la determinazione di limiti di principio alla risolvibilità mediante algoritmi di molti problemi generali.

**3.14 Notazione**  $x[y][z] \simeq u$  sta per  $x[y] \simeq w$  implica  $w[z] \simeq u$ .  
Diciamo che  $x$  è *totale* se  $\llbracket x \rrbracket$  lo è.

**3.15 Teorema** Per ogni  $x$  totale esiste  $u$  tale che  $\llbracket x[u] \rrbracket = \llbracket u \rrbracket$ , ossia

$$x[u][z] \simeq u[z] \quad \text{per ogni } z.$$

*Dimostrazione.* Sia data  $x$ . Definendo

- (a)  $x^* \equiv x;_{\mathbf{h}}\mathbf{U}$
- (b)  $u \equiv$  punto fisso associato a  $x^*$  dal teorema 3.6

otteniamo

$$\begin{aligned} u[z] &\simeq x^*[(u, z)] && \text{teorema 3.6} \\ &\equiv x;_{\mathbf{h}}\mathbf{U}[u, z] && \text{(a)} \\ &\equiv \mathbf{U}[(x[u], z)] && \text{semantica di hcmp} \\ &\simeq x[u][z] && \text{costruzione di U.} \end{aligned}$$

**3.16 Nota** Un programma ricorsivo  $P$  può essere visto come soluzione di un'equazione della forma

$$P = \Sigma(P)$$

a partire da uno schema  $\Sigma(Q)$  dato. Il teorema dice che ogni equazione come questa ammette una soluzione. Si ottiene un programma  $P$  che chiama se stesso, scrivendo uno schema in cui si descrive il suo comportamento *al di fuori* delle auto-chiamate, e usando una variabile per le chiamate a  $P$ . Si noti ancora che:

- (1) La procedura implicita nelle dimo dei teoremi 3.6 e 3.15 è perfettamente programmabile; possiamo quindi contarci per avere la forma finale di  $P$ ,
- (2) *ma* a meno di una codifica dei programmi nei dati, se le due classi sono distinte (cosa che non è nel nostro caso);
- (3) *purché* il linguaggio ammetta un elemento universale (visto che l'abbiamo usato nella seconda dimo); il §3.19 dimostra che questa condizione non è banale.

**3.17 Esempio: somma ricorsiva** (1) Poniamo

$$\begin{aligned}
 \mathbf{rs}^* &\equiv \langle_a \text{“sw”}, \text{“1”}, \text{“0”}, \text{“2”}, \\
 &\quad \langle_b \langle 1\omega, 2 \rangle, \text{id}, \text{“p1”} \rangle_b \rangle_a \\
 \mathbf{rs}^*[y] &= [1 = \text{“0”} \rightarrow 2 \quad T \rightarrow \{\langle 1\omega, 2 \rangle \ y \ \mathbf{p1}\}] \\
 \mathbf{rs} &\equiv \text{punto fisso di } \mathbf{rs}^* \text{ rispetto a } y \\
 \mathbf{rs}[h, k] &= \mathbf{rs}^*[\mathbf{rs}][h, k] && \text{teorema 3.15} \\
 &= [h = \text{“0”} \rightarrow k \quad T \rightarrow \{\langle h - 1, k \rangle \ \mathbf{rs} \ \mathbf{p1}\}]
 \end{aligned}$$

(2) Simuliamo  $\mathbf{rs}$  per capire come precisamente funziona. A tal fine osserviamo preliminarmente che, dalla parte (7) della costruzione del §3.9 si vede che si ha

(a)  $U((M, N), z) \simeq U(N, U((M, z)))$

Ricordiamo inoltre che dalle dimostrazioni dei due teoremi precedenti si ha

(b)  $s[w][z] = w[(w, z)]$   
(c)  $\mathbf{rs} = \mathbf{s}[\mathbf{s}; \mathbf{rs}^*; \mathbf{U}]$  (omettendo la  $h$  in “ $;$ ”)

Per  $z = (2, 7)$  abbiamo  $\mathbf{rs}[(2, 7)]$

$$\begin{aligned}
 &= \mathbf{s}[\mathbf{s}; \mathbf{rs}^*; \mathbf{U}][(2, 7)] && \text{(c)} \\
 &= \mathbf{s}; \mathbf{rs}^*; \mathbf{U}[(\mathbf{s}; \mathbf{rs}^*; \mathbf{U}, 2, 7)] && \text{(a) con } \mathbf{rs} \text{ come } w \\
 &= \mathbf{rs}^*; \mathbf{U}[(\mathbf{rs}, 2, 7)] && \text{come in } \ell 1, \text{ sem. di hcmp} \\
 &= \mathbf{U}[(\mathbf{rs}^*[\mathbf{rs}], 2, 7)] && \text{sem. di hcmp} \\
 &= \mathbf{U}[(1 = \text{“0”} \rightarrow 2 \quad T \rightarrow \{\langle 1\omega, 2 \rangle \ \mathbf{rs} \ \mathbf{p1}\}], 2, 7)] && \text{def. (1) di } \mathbf{rs}^* \text{ con } \mathbf{rs} = y \\
 &= \mathbf{U}[(\langle 1\omega, 2 \rangle \ \mathbf{rs} \ \mathbf{p1}), 2, 7)] && \text{def. di } \mathbf{U} \text{ perché } 1[z] = 2 \neq 0 \\
 &= \mathbf{U}[(\langle \mathbf{rs} \ \mathbf{p1} \rangle, 1, 7)] && \text{def. di } \mathbf{U} \\
 &= \mathbf{U}[(\mathbf{p1}, \mathbf{rs}[(1, 7)])] && \text{(a)} \\
 &= \mathbf{U}[(\mathbf{p1}, \mathbf{U}[(\mathbf{p1}, \mathbf{rs}[(0, 7)])])] && \text{ripetendo con } h = 1 \\
 &= \mathbf{U}[(\mathbf{p1}, \mathbf{U}[(\mathbf{p1}, 7)])] = 9 && \ell 1-4, \text{ poi test } 1 = \text{“0”} \text{ vero}
 \end{aligned}$$

STUDENTI 2005/6 motivazioni di questa simulazione ancora da correggere  
Si noti la costruzione di uno stack di programmi  $\mathbf{p1}$  in attesa di applicazione, così come ci è ben noto dallo studio della differenza tra iterazione e ricorsione.

**3.18 Esempio: divergenza** Poniamo

$$\begin{aligned}
 x &\equiv (\text{jo}, \text{id}, \text{“p1”}) \\
 x[y] &= \{y \ \mathbf{p1}\}.
 \end{aligned}$$

Per il punto fisso  $u$  otteniamo

$$u[z] \simeq x[u][z] \simeq \{u \text{ p1}\}[z] \simeq (0, u[z]).$$

Questo esempio dimostra che la cautela di usare “ $\simeq$ ” è essenziale, per non dire sciocchezze come che esista un programma  $P$  tale che  $P[z] = (0, P[z])$ . Simulando come nell’esempio precedente si vede che  $u$  diverge per ogni input  $z$  perché, in mancanza di una condizione di uscita, continua a chiamare se stessa, costruendo uno stack illimitato di  $\text{p1}$ .

### 3.2.5 Le classi totali non includono l’elemento universale

**3.19 Teorema** Se tutti i programmi di un linguaggio  $\mathcal{L}$  sono totali, allora  $\mathcal{L}$  non ammette un elemento universale.

*Dimostrazione.* Siano dati una classe  $\mathcal{L}$  di programmi totali e una *codifica*  $\lceil \cdot \rceil$  dei programmi  $P \in \mathcal{L}$  nei dati  $x$ . (Nel caso delle LPM la codifica sarà l’identità.) Supponiamo (ad abs.) che si abbia

$$U(\lceil P \rceil, x) = P(x)$$

Con piccoli cambiamenti (un programma di copia e uno per il successore) si ottiene un nuovo programma  $U? \in \mathcal{L}$  tale che

$$U?(x) = U(x, x) + 1$$

Ma questo comporta

$$U?(\lceil U? \rceil) = U(\lceil U? \rceil, \lceil U? \rceil) + 1 = U?(\lceil U? \rceil) + 1.$$

### 3.2.6 Teorema di Rice

FC denoti la classe delle funzioni calcolabili e FCT quella delle classi calcolabili totali. Sarebbe bello avere un programma  $\text{tot}[x]$  che *decide* se una generica  $x$  è totale, nel senso che

$$\text{tot}[x] = 0 \quad \text{se } x \in \text{FCT}; \quad \text{tot}[x] = 1 \quad \text{se } x \notin \text{FCT}.$$

Purtroppo  $\text{tot}$  non esiste.

**3.20 Definizione** La classe  $\mathcal{C} \subset \text{FC}$  è decidibile se esiste un programma  $d$  tale che

$$(a) \quad d[x] = 0 \quad \text{se } \llbracket x \rrbracket \in \mathcal{C}; \quad d[x] = 1 \quad \text{se } \llbracket x \rrbracket \notin \mathcal{C}.$$

Per esempio la classe vuota  $\emptyset$  e la classe FC sono decise risp. dai programmi “1” e “0”.

**3.21 Teorema** Le sole classi decidibili sono  $\emptyset$  e FC.

*Dimostrazione.* La classe  $\mathcal{C}$  sia decisa (ad abs.) da  $d$ , sicché valga la (a). Esistono per ipotesi due funzioni

$$(b) \quad \llbracket w \rrbracket \in \mathcal{C}; \quad \llbracket z \rrbracket \notin \mathcal{C}.$$

Definendo

$$d? = (d, \text{“0”}, \text{“z”}, \text{“w”}).$$

otteniamo

$$(c) \quad d?[x] = z \quad \text{se } \llbracket x \rrbracket \in \mathcal{C}; \quad d?[x] = w \quad \text{se } \llbracket x \rrbracket \notin \mathcal{C}.$$

Per il teorema di Rogers esiste  $u$  tale che

$$(d) \quad \llbracket u \rrbracket = \llbracket d?[u] \rrbracket$$

Questo dà la contraddizione seguente

$$\begin{array}{ll} \llbracket u \rrbracket \in \mathcal{C} \Rightarrow d?[u] = z & \text{per la (c) con } u \text{ come } x \\ \llbracket d?[u] \rrbracket \notin \mathcal{C} & \text{per la (b)} \\ \llbracket d?[u] \rrbracket \in \mathcal{C} & \llbracket u \rrbracket \text{ e } \llbracket d?[u] \rrbracket \text{ sono la stessa funzione per la (d)} \end{array}$$

mentre l'ipotesi  $\llbracket u \rrbracket \notin \mathcal{C}$  comporta la contraddizione simmetrica.

### 3.3 Un interprete che impara definizioni ricorsive

In questa sezione proponiamo di arricchire il linguaggio delle LPM allo scopo di consentire

1. schemi di definizione ricorsiva molto generali;
2. un interprete capace di *imparare* nuove funzioni definite ricorsivamente.

### 3.3.1 Definizioni ricorsive

Talora si dice che le singole TM sono hardware astratti. Da questo punto di vista è coerente dire che la STM  $U$  del §2.20 e l'analogo LPM  $U$  del §3.9 sono *macchine* universali, con statuto concettuale di equivalenti astratti di un elaboratore. Ogni TM, nel momento in cui viene codificata e data in entrata ad  $U$ , diviene *software* per  $U$ . Ci accingiamo ora a rendere piú complesso il nostro software, permettendo *programmi* che includono definizioni ricorsive esplicite (ossia senza passare per punti fissi forniti dal teorema di ricorsione). Uno di questi programmi che abbia la capacità di simulare gli altri potrà essere visto come un *interprete*.

(Una ragione ulteriore per vedere questi nuovi enti come programmi invece che come macchine è che la realizzazione materiale, non solo astratta, delle TM ordinarie è facilmente immaginabile. Questo vale anche per le STM e le LPM che sono solo notazioni per TM ordinarie. Non è agevole invece immaginare il modo di costruire una macchina che si comporti in modo ricorsivo.)

**3.22** 1. Un *funtore*  $f$  è un'espressione nella forma  $(\mathbf{8}, \mathbf{h})$ .

2. La sua *dichiarazione* è una lista  $(\mathbf{def}, f, x)$ , in cui sia  $f$  che altri funtori possono occorrere. La sua forma leggibile è  $f =_{\mathbf{df}} x$ .

3. Uno *statement*  $S$  è definito

- (a) per chiusura rispetto agli stessi schemi usati per le LPM
- (b) degli *statement di base*  $\mathbf{hd}$ ,  $\mathbf{tl}$  e  $\mathbf{qt}$ , e dei funtori.

4. La forma di ogni *programma ricorsivo*  $P$  è

$$(\mathbf{S}_1, \dots, \mathbf{S}_h, \mathbf{d}_1, \dots, \mathbf{d}_k, 0) \quad h \geq 1, k \geq 0.$$

Gli  $\mathbf{S}_i$  formano il *corpo*  $B$  e le  $\mathbf{d}_i$  formano la *libreria* del programma. I programmi saranno dispiegati nella forma (spesso con a capo separatori)

$$\mathbf{d}_1 \quad \dots \quad \mathbf{d}_k \quad ; \quad \mathbf{S}_1 \quad \dots \quad \mathbf{S}_h$$

**3.23 Esempio**  $\mathbf{sum}$  è sia un funtore dichiarato nella libreria che l'unico statement del programma  $M_+$  seguente (si veda il §3.12 per  $\mathbf{p1}$ )

$$\begin{aligned} \mathbf{prd} &=_{\mathbf{df}} 2\omega \\ \mathbf{sum} &=_{\mathbf{df}} [2 = \text{"0"} \rightarrow 1 \ T \rightarrow \langle 1, \mathbf{prd} \rangle \ \mathbf{sum} \ \mathbf{p1}]; \\ \mathbf{sum} & \end{aligned}$$

**3.24 Notazione** L'esempio precedente sarebbe forse piú chiaro cosí

$$\begin{aligned} \text{prd}[(0, z)] &=_{\text{df}} z \\ \text{sum}[x, y] &=_{\text{df}} [y = 0 \rightarrow x \quad T \rightarrow \text{p1}[\text{sum}[x, \text{prd}[y]]]] \end{aligned}$$

In questo spirito, stabiliamo che:

1. quando inseriamo, nella costruzione di un programma o dichiarazione, la struttura dell'*intended input*: (a) non siamo interessati al comportamento del programma per dati in altra forma; (b) ogni occorrenza
  - di dati; o di loro parti; o di liste ottenute ri-assemblando (parti di) dati
  - in punti in cui la sintassi imporrebbe programmi
  - sta per il programma (composto da `hd`, `tl` e `jo`) che li produce.
2. una composizione come  $\{ \langle M_1, \dots, M_k \rangle N \}$  sarà scritta nella forma  $N[M_1, \dots, M_k]$ ;
3. il corpo  $C$  viene omissa, se si riduce all'ultimo funtore dichiarato.
4. Nelle chiamate ricorsive si omette il passaggio dei parametri se ovvio.
5. Quando  $M$  calcola una funzione numerica, le espressioni

$$M + 1 \quad M \dot{-} 1$$

stanno risp. per  $\text{p1}[M]$  e per  $[0 \rightarrow \text{id} \quad T \rightarrow \text{tl}[M]]$

### Esempio

$$\begin{aligned} \text{mlt} &\equiv \text{sum}[h, k] =_{\text{df}} [h = 0 \rightarrow k \quad T \rightarrow \text{sum}[h \dot{-} 1, k] + 1] \\ \text{mlt}[h, k] &=_{\text{df}} [h = 0 \rightarrow 0 \quad T \rightarrow \text{sum}[k, \text{mlt}[h \dot{-} 1, k]]]; \end{aligned}$$

### 3.25 Semantica degli identificatori

$[[f[\dots]]][\dots] = [[x[\dots]]][\dots]$  se  $f =_{\text{df}} x$  è la prima dichiarazione di  $f$  nella libreria (ossia la piú lontana dallo 0 finale) nella forma 3.22.4.

### 3.26 Ricerca in un programma della dichiarazione di un funtore

$$\begin{aligned} \text{findlib}[P] &=_{\text{df}} \text{whnt}(P = \text{"0"} \vee P_{11} = \text{"def"}) \text{tl}[P] \\ \text{search}[f, P] &=_{\text{df}} \text{findlib} \\ &\quad [P = \text{"0"} \rightarrow \text{"0"} \\ &\quad P_{12} = f \rightarrow P_{13} \\ &\quad T \rightarrow \text{search}[f, \text{tl}[P]]] \end{aligned}$$

### 3.3.2 Interprete per programmi ricorsivi

Il Lettore può verificare che la presenza di dichiarazioni e funtori non interferisce con le dimostrazioni della sez.3.2. Possiamo dunque fare ricorso al teorema di Rogers per definire un interprete che arricchisce la sua libreria con quella del programma oggetto.

$$\begin{aligned}
 \text{srch} &=_{\text{df}} \text{ come sopra} \\
 \text{ex.h/t/q} &=_{\text{df}} y[w, z_1/z_\omega/v] \\
 \text{ex.sw} &=_{\text{df}} [y[\text{J}, z] = y[\text{L}, z] \rightarrow y[\langle \text{M}, w \rangle, z] \\
 &\quad T \qquad \qquad \qquad \rightarrow y[\langle \text{N}, w \rangle, z]] \\
 \text{ex.wh} &=_{\text{df}} [y[\text{J}, z] = y[\text{L}, z] \rightarrow y[\langle \text{M}, x \rangle, z] \\
 &\quad T \qquad \qquad \qquad \rightarrow y[w, z]] \\
 \text{ex.hcmp} &=_{\text{df}} y[\langle \text{N}, w \rangle, \langle y[\text{M}, z_1], z_\omega \rangle] \\
 \text{ex.cmp} &=_{\text{df}} y[\langle \text{M}, \text{N}, w \rangle, z] \\
 \text{ex.fct} &=_{\text{df}} \text{ si veda piú avanti}
 \end{aligned}$$

$$\begin{aligned}
 \text{LI}[y, x, z] &=_{\text{df}} \\
 [x = \text{"0"} \vee x_{11} = \text{"def"}] &\rightarrow \text{tl tl} \\
 x = (\text{hd/tl}, w) &\rightarrow \text{ex.h/t} \\
 x = (\text{"v"}, w) &\rightarrow \text{ex.q} \\
 x = ((\text{sw}, \text{J}, \text{L}, \text{M}, \text{N}), w) &\rightarrow \text{ex.sw} \\
 x = ((\text{wh}, \text{J}, \text{L}, \text{M}), w) &\rightarrow \text{ex.wh} \\
 x = ((\text{hcmp}, \text{M}, \text{N}), w) &\rightarrow \text{ex.hcmp} \\
 x_1 = ((\text{f}, \text{f}), w) &\rightarrow \text{ex.fct} \\
 x = ((\text{M}, \text{N}), w) &\rightarrow \text{ex.cmp}
 \end{aligned}$$

dove, per inserire  $d$  come prima dichiarazione del programma,  $P$  si pone

$$\begin{aligned}
 \text{ndcl}[d, P] &\equiv [P = \text{"0"} \vee P_{11} = \text{"def"} \rightarrow \langle d, P \rangle \\
 &\quad T \rightarrow \langle P_1, \text{ndcl}[d, P_\omega] \rangle]
 \end{aligned}$$

e quindi

$$\begin{aligned}
 \text{ex.fct} &=_{\text{df}} \\
 [\text{srch}[\text{f}, x] = u \wedge u \neq \text{"0"}] &\rightarrow \\
 \text{ndcl}[(\text{def}, \text{f}, u), y][\langle (u, v), w \rangle, z] & \\
 \text{srch}[\text{f}, y] = u &\rightarrow y[\langle u, w \rangle, z]
 \end{aligned}$$

in questo modo se la dichiarazione cercata si trova nella libreria oggetto, viene applicato  $y$ , ma aumentato con la nuova dichiarazione; altrimenti si applica la dichiarazione che *dovrebbe* trovarsi nella libreria di  $y$ .

### 3.3.3 Funzionali

I matematici chiamano *funzionale* una funzione che trasforma una o più funzioni (più eventualmente numeri o altri enti) in una funzione. I funzionali che ora vengono introdotti permettono di ridurre schemi ricorsivi a dichiarazioni di funzionali.

- 3.27** 1. SINTASSI Un *funzionale*  $F$  è un'espressione nella forma  $\langle 9, h \rangle$ .
2. La sua *dichiarazione* è nella forma  $(DEF, F, x)$ , anche scritta nella forma  $F =_{df} x$  (con ambiguità nei confronti di  $(def, f, x)$  eliminata adottando sempre nomi in maiuscolo per i funzionali).
3. Da ora un *programma ricorsivo* è definito permettendo nella definizione 3.22
- (a) tra gli elementi della base i funzionali, oltre ai funtori;
  - (b) che anche le dichiarazioni di funzionali figurino nelle librerie.
4. SEMANTICA Se sono le prime del programma, le dichiarazioni

$$F =_{df} x \quad f =_{df} F[y] \quad \text{equivalgono alla dichiarazione} \quad f =_{df} x[(f, y)]$$

**3.28 Ricorsione primitiva** Per ragioni storiche si dice che una funzione numerica a  $k + 1$  posti  $f(n, \vec{m})$  è definita per ricorsione primitiva (PR) nella funzione  $k$ -aria  $g(\vec{m})$  e nella funzione  $(k + 2)$ -aria  $h(n, \vec{m}, l)$  se si ha

$$\begin{cases} f(0, \vec{m}) & = g(\vec{m}) \\ f(n + 1, \vec{m}) & = h(n + 1, \vec{m}, f(n, \vec{m})) \end{cases}$$

**Esempio** La somma è PR nella funzione unaria identità e nella funzione a 3 posti ottenuta per composizione di una funzione selettiva del terzo elemento di una tripla col successore. (In questo caso dunque la funzione-passo  $h$  dipende dai suoi primi due argomenti solo formalmente, per ragioni sintattiche.)

**3.29 Ricorsione primitiva come funzionale** Definiamo (con le notazioni del §3.24) uno schema uniforme di programma nelle variabili  $f, g, h$

$$pr[(f, g, h)] \equiv [1 = "0" \rightarrow g[\omega] \quad T \rightarrow h[1, \omega, f[1, \omega]]]$$

Definiamo un funzionale per la PR e un programma per l'addizione, ponendo

$$\begin{aligned} PR &=_{df} pr \\ sum &=_{df} PR[(id, \langle "0", 3 \rangle)] \end{aligned}$$

Dando in entrata a `pr` la lista  $(\text{sum}, \text{id}, \langle \text{"0"}, 3 \rangle)$  si vede che, per la semantica ora definita questo è equivalente a

$$\begin{aligned} \text{sum} &=_{\text{df}} [1 = \text{"0"} \rightarrow 2 \quad T \rightarrow \langle \text{"0"}, 3 \rangle [1, \omega, \text{sum}[1, \omega]] \\ \text{sum}[h, k] &=_{\text{df}} [h = 0 \rightarrow k \quad T \rightarrow \langle \text{"0"}, \text{sum}[h - 1, k] \rangle \end{aligned}$$

Per la moltiplicazione possiamo porre

$$\begin{aligned} \text{PR} &=_{\text{df}} \text{pr} \\ \text{sum} &=_{\text{df}} \text{PR}[(\text{id}, \langle \text{"0"}, 3 \rangle)] \\ \text{mult} &=_{\text{df}} \text{PR}[(\text{"0"}, \text{sum}[2, 3])] \end{aligned}$$

Il programma seguente calcola  $k^h$  se viene dato ad un interprete che impara anche le definizioni dei funzionali, e che ha già applicato le dichiarazioni di `PR` e `mult`

$$\text{exp} =_{\text{df}} \text{PR}[(\text{"1"}, \text{mult}[\text{"2"}, 3])]$$

# Capitolo 4

## Complessità Computazionale

### 4.1 Macchine di Turing non deterministiche

**4.1** Per decidere se esiste in un dato albero un cammino che soddisfi una data condizione può essere comodo separare la visita dell'intero albero dall'analisi dei suoi singoli cammini. La complessità del cammino migliore può essere cruciale fino al punto di rendere gli altri cammini inessenziali. Si immagina allora una macchina astratta che, nel corso di una computazione, *sceglie* tra diversi comportamenti. Per esempio un programma NOTPRIME potrebbe accettare l'input  $n$ , se non primo, scegliendo (o, più suggestivamente, *guessing*) una coppia  $(h, k)$ , e controllando se  $hk = n$ . Una piccola quantità di tempo è sufficiente per ciascun guess, mentre il modo ovvio di accettare il linguaggio dei numeri non primi richiede un tempo esponenziale (nella lunghezza dell'input). Si osservi che quest'astrazione non è, in linea di principio idonea ad accettare i numeri primi, perché in questo secondo caso una singola risposta ad un guess fortunato non basterebbe<sup>1</sup>.

**4.2** Una TM *nondeterministica* (NTM)  $M$  è un'assegnazione di valori di verità così come definita in §2.1 che non soddisfa però la condizione di univocità (cf. (a))

$$(Q - 1, a, Q - 2) \leftrightarrow \neg(Q - 1, a, Q - 3).$$

In altri termini, l'ultimo stato (non finale)  $Q - 1$  è nondeterministico, nel senso che, dopo di esso  $M$  entra in  $Q - 2$  oppure in  $Q - 3$ . L'omologo strutturato è definito per

---

<sup>1</sup> In realtà esiste un metodo aritmetico raffinato che riduce al caso nondeterministico il problema dei numeri primi. Tuttavia il metodo non è generale, e non si sa se valga per altri casi, come le formule non soddisfacibili della logica degli enunciati, ossia per il complemento del linguaggio SAT che sarà definito nel seguito.

chiusura delle STM rispetto allo schema postulato

choose  $M_1$  or  $M_2$  end

Una computazione deterministica produce un albero i cui nodi sono tutti unari. Ora invece si ha un albero i cui nodi sono ancora unari, a meno che non si stia nello stato nondeterministico. Due *subcomputazioni* differenti cominciano con questi nodi. Ogni volta che non ci occupiamo di risorse di calcolo molto piccole useremo macchine con un solo nastro.

**4.3 Accettori** La NTM  $M$  accetta il linguaggio  $L \subseteq \Gamma^*$  se, per ogni input  $x \in \Gamma^*$

1. c'è almeno una computazione accettante sse  $x \in L$  (le computazioni che rifiutano o non terminano non contano);
2. non c'è alcuna computazione accettante sse  $x \notin L$  (in particolare  $x$  non è accettato se *tutte* le computazioni non si fermano).

Le funzioni calcolate nondeterministicamente sono studiate di rado.

## 4.2 Classi di complessità

1. Il *tempo* di una computazione terminante è il numero delle sue ID (tenendo conto che in uno stesso passo la macchina si può muovere su tutti i nastri).
2. La *lunghezza* di una ID è il max tra le lunghezze dei suoi nastri.
3. Lo *spazio* di una computazione è il max tra le lunghezze delle sue ID diminuito della lunghezza dell'input.
4. Il tempo richiesto da una TM nondeterministica per accettare  $x$  è il min tra i tempi delle sue computazioni accettanti.
5. Lo spazio richiesto da una TM nondeterministica per accettare  $x$  è il min tra gli spazi delle sue computazioni accettanti.

**4.4 Runtime e working space** per una TM  $M$  (deterministica o no) sono dati da

$$\begin{aligned}\tau_M(n) &= \max l (M(x) = y \text{ in tempo } l \text{ per qualche } x, y \text{ con } |x| \leq n) \\ \sigma_M(n) &= \max l (M(x) = y \text{ in spazio } l \text{ per qualche } x, y \text{ con } |x| \leq n).\end{aligned}$$

Ma osservando che nel caso non deterministico prima si guarda il miglior comportamento per ciascun input  $x$ ; poi si guarda il peggior  $x$  di lunghezza  $\leq n$ .

**4.5 Notazione** Per il resto di questa sezione,  $f$  e  $g$  sono funzioni numeriche. Definiamo

$$f \in O(g) \text{ sse per ogni } n \text{ e per una costante } c \text{ si ha } f(n) \leq cg(n).$$

Definiamo inoltre  $Lg(n) = \min h(2^h > n)$ .

**4.6 Classi di complessità** Data  $f$ , definiamo le classi di funzioni

$$\begin{aligned} \text{DTIME}(f) &= \{ \llbracket \mathbb{M} \rrbracket \mid \tau_{\mathbb{M}} \in O(f) \text{ per una TM deterministica multi-nastro } \mathbb{M} \}; \\ \text{DSPACE}(f) &= \{ \llbracket \mathbb{M} \rrbracket \mid \sigma_{\mathbb{M}} \in O(f) \text{ per una TM deterministica multi-nastro} \}. \end{aligned}$$

Le restrizioni di queste classi a linguaggi (e quindi ad accettori) sono denotate da

$$\text{DTIME}(f) \quad \text{DSPACE}(f).$$

Definiamo inoltre le classi di linguaggi

$$\begin{aligned} \text{NTIME}(f) &= \{ L \mid L \text{ è accettato da una NTM } \mathbb{M}, \text{ con } \tau_{\mathbb{M}} \in O(f) \}; \\ \text{NSPACE}(f) &= \{ L \mid L \text{ è accettato da una NTM } \mathbb{M}, \text{ con } \sigma_{\mathbb{M}} \in O(f) \}. \end{aligned}$$

**4.7 Gerarchie** Queste classi formano una gerarchia propria, nel senso che si dimostra che

$$\begin{aligned} f(n)Lg(n) \prec g(n) \text{ implica } & \text{DTIME}(f) \subset \text{DTIME}(g) \\ & \text{nonché } \text{NTIME}(f) \subset \text{NTIME}(g) \\ f \prec g \text{ implica } & \text{DSPACE}(f) \subset \text{DSPACE}(g) \\ & \text{nonché } \text{NSPACE}(f) \subset \text{NSPACE}(g) \\ \text{dove } & f \prec g \stackrel{\text{def}}{=} \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0. \end{aligned}$$

**4.8 Relazioni tra classi di complessità** Si ha

- (1)  $\text{DTIME}(f) \subseteq \text{NTIME}(f)$  (ovvio)
- (2)  $\text{NTIME}(f) \subseteq \text{DSPACE}(f)$  (ovvio)
- (3)  $\text{DSPACE}(f) \subseteq \text{DTIME}(2^{O(f)})$  (per il Teor. 4.12)
- (4)  $\text{NTIME}(f) \subseteq \text{DTIME}(2^{O(f)})$  (similmente).

**4.9 Le classi centrali** Le classi piú studiate sono (la prima colonna riporta il loro nome piú comunemente usato)

$$\begin{aligned}
 L &= \text{DSPACE}(\text{Lg}(n)) \\
 \text{NL} &= \text{NSPACE}(\text{Lg}(n)) \\
 \text{P} &= \bigcup_c \text{DTIME}(n^c) \\
 \text{PSPACE} &= \bigcup_c \text{DSPACE}(n^c) \\
 \text{NP} &= \bigcup_c \text{NTIME}(n^c) \\
 \text{NSPACE} &= \bigcup_c \text{DSPACE}(n^c) \\
 \text{Polytime} &= \bigcup_c \text{DTIMEF}(n^c) \\
 \text{Lintime} &= \text{DTIMEF}(n) \\
 \text{Polyspace} &= \bigcup_c \text{DSPACEF}(n^c) \\
 \text{Linspace} &= \text{DSPACEF}(n).
 \end{aligned}$$

Si dice spesso che *Polytime* è la contro-parte *feasible* (in pratica) della classe delle funzioni *computable* (in linea di principio), anche se non è chiaro quanto sia fattibile in pratica un calcolo che richieda un tempo  $n^{9!}$ .

**4.10 Problemi aperti** Sappiamo da §4.8 che

$$L \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE};$$

mentre da §4.7 sappiamo che

$$L \subset \text{PSPACE}$$

è un'inclusione propria. Cosa possiamo dire delle altre inclusioni?

Si sa che le classi di linguaggi accettate in tempo polinomiale e risp. in spazio lineare devono essere diverse, ma non se una di esse è contenuta nell'altra o se sono non confrontabili.

## 4.3 Quanto contano le costanti, gli alfabeti e il numero dei nastri?

**4.11 Lemma** Ci sono meno di  $2^{O(n)}$  ID diverse di lunghezza  $n$  per ogni TM.

*Dimostrazione.* Sia data una TM  $M$  con  $Q > 2$  stati,  $l$  nastri in un alfabeto  $\Gamma$  con  $2^K$  lettere. La cardinalità della classe di tutte le stringhe in  $\Gamma$  di lunghezza  $ln$  è  $2^{Kln}$ .

Considerando lo stato corrente e l'*indirizzo* del carattere osservato (che richiede  $\text{Lg}(n)$  bit) si ottiene

$$Q2^{Kln}\text{Lg}(n) < 2^{QKln}$$

L'asserto segue perché  $K$ ,  $l$  e  $Q$  sono costanti.

**4.12 Teorema** Ogni linguaggio accettato in spazio deterministico  $S(n)$  è accettato in tempo  $2^{O(n)}$ .

Lo stesso per lo spazio non deterministico.

*Dimostrazione.*  $M$  accetti  $L$  in spazio  $S(n)$ . Per il Lemma 4.11 per ogni  $x \in L$  c'è una computazione accettante di lunghezza  $2^{cn}$  per una  $c$  indipendente da  $M$  e da  $x$  (perché una stessa ID può occorrere due volte solo nelle computazioni infinite).

**4.13 Lemma** Ogni  $L$  è accettato da una TM  $M$  con  $l > 1$  nastri in tempo  $T(n)$ , è anche accettato da una TM con  $l$  nastri in tempo  $T(n)/2$  (purché  $\lim_{n \rightarrow \infty} \frac{T(n)}{n} = \infty$ ).

*Dimostrazione.* Assumiamo  $\Gamma = \mathbf{B}$  ed  $l = 2$ ;  $O_i$  sia il carattere osservato sul nastro  $i$ . L'input  $x$  per  $M$  sia nelle prime  $n = |x|$  celle del nastro 1, alla sinistra di  $O_1$ . Immaginiamo che i nastri siano divisi in *blocchi* di 16 celle consecutive ciascuno, e chiamiamo *blocco osservato* quello che include  $O_i$  ( $i = 1, 2$ ). Chiamiamo *core* i blocchi osservati e i quattro alla loro sinistra e destra. Sfruttiamo tre idee:

- (1) prima di lasciare un core,  $M$  trasforma il suo contenuto  $X_1, X_2$  in due stringhe  $Y_1, Y_2$  appartenenti ad una classe finita (cf. Lemma 4.11);
- (2) la classe di tutte queste  $(X_1, X_2, Y_1, Y_2)$  è un'informazione finita che può essere registrata nel controllo finito di una TM e recuperato (in *zero* passi!) mentre sta facendo qualche altra cosa;
- (3) poiché in un passo si può muovere su entrambi i nastri, per poter lasciare un core,  $M$  ha bisogno di più di 16 passi.

Si può definire una TM  $M^*$  in un alfabeto di cardinalità  $2^{16}$ , che simula  $M$  per blocchi (ossia, una cella di  $M^*$  per blocco di  $M$ ).

$M^*$  prima legge  $x$ , e *comprime* i suoi blocchi in  $n/16$  celle; a tal fine usa il suo secondo nastro per metterci solo  $n$  passi; con pochi calcoli elementari si vede che l'ipotesi su  $\frac{T(n)}{n}$  rende trascurabile questo tempo.

Poi  $M^*$  ripete, finché  $M$  si ferma, il ciclo seguente:

- (a) in 4 passi, visita le celle a destra e sinistra degli  $O_i$  e registra nel suo controllo finito il loro contenuto  $X_i$ ;
- (b) in altri 4 passi, scrive  $Y_i$  in  $O_i$  e nelle due celle adiacenti; e va alle celle che corrispondono ai nuovi blocchi osservati.

Per (3)  $M^*$  impiega al massimo 8 passi per ogni sequenza di 16 da parte di  $M$ .

**4.14 Teorema (Linear speed-up, linear space-compression)** Se  $L$  è accettato dalla TM  $M$  con  $l$  nastri in tempo  $T(n)$  e spazio  $S(n)$ , allora, per ogni costante  $c$ , esso è accettato:

- (1) da un'altra TM  $M^*$  con  $l$  nastri in tempo  $\leq cT(n)$ .
- (2) da un'altra TM  $M^+$  con  $l$  nastri in tempo  $\leq cS(n)$ .

Lo stesso per il caso nondeterministico.

*Dimostrazione.* Sia  $c < 1$  (se no il teorema è scemo).

- (1) Mediante  $m \geq \text{Lg}(1/c)$  applicazioni del Lemma 4.13.
- (2) Similmente per l'analogo per lo spazio dello stesso lemma, la cui dimo è simile e piú facile.

**4.15 Teorema** Se  $L$  è accettato dalla TM  $M$  (dalla NTM  $N$ ) con  $l$  nastri in tempo  $T(n)$ , allora è accettato da una TM  $M^*$  (da una NTM  $N^*$ ) con due nastri, in tempo  $T(n)\text{Lg}(T(n))$ .

Inoltre esso è accettato in tempo  $T(n)^2$  da una TM (da una NTM) con un nastro.

*Dimostrazione.* In preparazione. (Non in programma per l'a.a. 2005/6.)

## 4.4 3SAT e il problema $P=?NP$

**4.16 I linguaggi SAT e 3SAT** Le variabili  $p, q, r, \dots$  siano definite su una classe  $\mathcal{A}$  di *literal*  $l, l_1, \dots, \bar{l}, \bar{l}_1, \dots$

1. Una  $k$ -*clausola* è una disgiunzione di  $i \leq k$  literal.
2. Una  $k$ -*formula* è una classe di  $n \geq 1$   $k$ -clausole interpretate congiuntivamente.
3.  $L_{k\text{SAT}}$  è la classe delle  $k$ -formule, e  $L_{\text{SAT}}$  è l'unione delle  $L_{k\text{SAT}}$ .
4. Un'assegnazione di verità è una funzione  $v : \mathcal{A} \mapsto \{0, 1\}$  tale che  $v(p) \neq v(\bar{p})$
5.  $v^+$  è l'estensione di  $v$  a  $L_{\text{SAT}}$  secondo le tavole di verità.
6.  $k\text{SAT} \subset L_{k\text{SAT}}$  è il linguaggio delle  $k$ -formule soddisfatte da qualche  $v$ , e  $\text{SAT} = \cup_k k\text{SAT}$  è il linguaggio delle formule *soddisfacibili*.

Per lavorare su questi linguaggi adottiamo le notazioni seguenti

$$\begin{array}{llll} \neg p = \bar{p} & & \bar{\bar{p}} = p & p \rightarrow q = \neg p \vee q \\ p \rightarrow q \vee r = \neg p \vee q \vee r & p \rightarrow q \wedge r = p \rightarrow q, p \rightarrow r & & (2 \text{ 2-clausole}) \end{array}$$

**4.17 Rappresentazione di array** Siano dati dei literal quadrupli della forma  $(a, b, c, d)$ . La seguente formula in  $L_{2SAT}$  è soddisfatta sse i literal veri formano un array ad indici  $a$  e  $c$  (ossia se ce n'è uno solo vero per ogni valore degli indici)

$$(a, b, c, d)!_{bd} =_{df} \bigcup_{b^* \neq b \text{ or } d^* \neq d; a; b} (a, b, c, d) \rightarrow \neg(a, b^*, c, d^*)$$

Il metodo è facilmente generalizzabile a matrici a  $m$  dimensioni i cui elementi siano literal  $(m + n)$ -pli.

**4.18 Lemma** Si ha  $SAT \in DTIME(2^n)$  nonché  $SAT \in NP$

*Dimostrazione.* Nel primo caso si procede per esplorazione brutta di tutte le assegnazioni di verità. Nel secondo si sceglie nondeterministicamente una  $v$  e si vede se essa soddisfa l'input o no.

**4.19 Teorema di Cook**  $P = NP$  sse  $3SAT \in P$ .

*Dimostrazione.*  $3SAT \notin P$  implica  $P \neq NP$  per il lemma precedente.

Dimostriamo l'altra metà del teorema facendo vedere che per ogni  $L \in NP$  esiste un programma  $\mathbf{red}_L$  che *riduce* in tempo polinomiale  $L$  a  $3SAT$  nel senso che si ha

$$x \in L \text{ sse } \mathbf{red}_L(x) \in 3SAT$$

Se ci fosse un programma  $\mathbf{3sat}$  che decide  $3SAT$  in tempo polinomiale  $L$  sarebbe deciso<sup>2</sup> in tempo polinomiale dal programma  $\mathbf{red}_L; \mathbf{3sat}$ .

*Costruzione del programma  $\mathbf{red}_L$ .* Sia dato  $L \in NP$  ed una NTM  $M$  con  $Q$  stati,  $K$  caratteri di nastro, che accetta  $L$  in tempo  $n^c$ ; dato poi un input  $x$  per  $M$ , poniamo  $\tau = |x|^c$ . Definiamo l'alfabeto

$$\mathbf{A} = \{1, \dots, Q + K + 2\}$$

una *cifra* per ciascuna lettera, associata nel modo piú ovvio agli stati, alle lettere del nastro e agli spostamenti a sinistra e a destra. Identifichiamo  $Q + 1$  col *blank*  $\dots$ . Ci accingiamo a costruire un programma  $\mathbf{red}_L$  tale che

$$\mathbf{red}_L(x) = A_{Mx} \quad A_{Mx} \in 3SAT \text{ sse } x \in L$$

**I literal costituenti** La formula  $A_{Mx}$  è costituita da alcuni literal tripli  $(i, j, I)$  che descrivono  $M$  e da alcuni literal quintupli  $(t, i, j, s, h)$  che descrivono le ID nel modo

---

<sup>2</sup>Dico *deciso* e non solo *accettato* perché le classi deterministiche sono ovviamente chiuse rispetto al complemento.

che sarà spiegato poco piú avanti. I domini delle *metavariabili* che compongono questi literal sono definiti (per tutta questa dimostrazione) nel modo seguente

$$\begin{array}{ll} t \leq \tau & -\tau \leq s \leq \tau \\ 0 \leq i, i^* \leq (i \neq i^*) & 1 \leq j, j^*, h \leq K \quad (j \neq j^*) \\ 1 \leq I, I^* \leq Q + K + 2 & \end{array}$$

**La descrizione della macchina** è un'assegnazione di verità ai literal

$$(i, j, I)$$

che soddisfa la formula (come in §4.17 con  $m = 2$  e  $n = 1$ )

$$(i, j, I)!_I^{nd} =_{\text{df}} (i, j, I)!_I - \{(Q - 1, j, Q - 2) \leftrightarrow \neg(Q - 1, j, Q - 3)\}$$

ottenuta togliendo dalla formula  $(i, a, I)!_I$  le due 2-clausole che imporrebbero il determinismo.

**La descrizione delle ID** è un'assegnazione di verità ai literal

$$(t, i, j, \pm s, h)$$

che vogliamo siano veri se prima del passo  $t$  lo stato è  $i$ ,  $j$  è osservato nella cella 0 e  $h$  è nella  $s$ -ma cella a destra/sinistra del carattere osservato a seconda che  $s$  sia positivo/negativo.

**La rappresentazione della ID iniziale** è ottenuta soddisfacendo la 1-formula

$$init_x =_{\text{df}} \cup_{sb}(1, 1, -, s, b) \text{ dove } b = \begin{cases} x_s & \text{per } 1 \leq s \leq |x| \text{ e per } x = x_1 \dots x_{|x|} \\ - & \text{altrimenti} \end{cases}$$

la quale dice che si comincia alla sinistra di  $x$ , col nastro bianco altrove.

**La rappresentazione di un passo** è data dall'unione/congiunzione delle formule seguenti che rappresentano le diverse azioni da intraprendere, ossia:

cambiamento di stato ( $\models$  sta per “*soddisfa*”)

$$cst_t = \bigcup_{ii^*jsh; \mathbb{M}=(i, j, i^*)} (t, i, j, s, h) \rightarrow (t + 1, i^*, j, s, h)$$

cambiamento del carattere osservato

$$cobs_t = \bigcup_{ijj^*sh; \mathbb{M}=(i, j, j^*); s \neq 0} (t, i, j, s, h) \rightarrow (t + 1, i, j^*, s, h) \wedge (t + 1, i, j^*, 0, j^*)$$

alt M termina *dinamicamente*, ciclando senza cambiare lo stato il nastro

$$alt_t = \bigcup_{jhs} \{(t, Q, j, s, h) \rightarrow (t + 1, Q, j, s, h)\}$$

spostamento a sinistra

$$sx_t = \bigcup_{ijj^*sh; \mathbb{M}=(i,j,Q+K+1)} (t, i, j, s, h) \wedge (t, i, j, -1, j^*) \rightarrow (t + 1, i, j^*, s + 1, h)$$

spostamento a destra

$$sx_t = \bigcup_{ijj^*sh; \mathbb{M}=(i,j,Q+K+2)} (t, i, j, s, h) \wedge (t, i, j, 1, j^*) \rightarrow (t + 1, i, j^*, s - 1, h)$$

passo non deterministico

$$chs_t = \bigcup_{jsh} (t, Q - 1, j, s, h) \rightarrow (t + 1, Q - 2, j, s, h) \vee (t + 1, Q - 3, j, s, h)$$

una sola ID per passo

$$unID_t = (t, i, j, s, h)!_{iab}$$

ID accettante M accetta cancellando tutto il nastro

$$acc = \bigcup_s (\tau, Q, 0, s, 0)$$

possiamo ora porre

$$nxt_t = cst_t \wedge cobs_t \wedge dx_t \wedge sx_t \wedge chs_t \wedge alt_t \wedge unID_t$$

La costruzione termina ponendo (M è l'insieme dei literal soddisfatti da M)

$$A_{Mx} = init_x \wedge \bigcup_t nxt_t \wedge acc \wedge \mathbb{M}$$

Si ha

$$A_{Mx} \in 3SAT \text{ sse NTM M accetta } x$$

*Conclusione della dimostrazione* Poniamo

```

red_M = for t:=1 to tau do
        for s:=-tau to tau do
            write.appropriate.clause;;

```

Rimane da verificare che il runtime per questo programma è polinomiale in  $n = |x|$ . L'invariante richiede un numero costante (e quindi tracciabile) di passi per decidere qual'è la clausola appropriata da scrivere, più un numero di passi per scriverla. Questo è dell'ordine della lunghezza della clausola più lunga, e quindi dell'ordine della lunghezza dei literal. La lunghezza di ogni literal è  $O(n)$  perché vi figurano due variabili  $\leq n^c$ , più altre  $\leq Q + K + 2$  (si tenga presente che si ha  $|x|^c = cn$ ). L'invariante è ripetuto dalla **for** più interna per  $n^c$  volte; e questa, a sua volta, è ripetuta dalla **for** esterna per  $n^c$  volte. Moltiplicando i tre importi si vede che il tempo richiesto dal nostro programma è  $O(n^{2c+1})$ .

## 4.5 PSPACE=NPSPACE

**4.20 Teorema di Savitch**  $L \in \text{NSPACE}(f(n))$  implica  $L \in \text{DSPACE}(f(n)^2)$  purché  $f(n) \geq \text{Lg}(n)$ .

*Dimostrazione.* Per il teorema 4.12 esiste una NTM che accetta  $L$  in tempo  $2^{cf(n)}$  per una costante  $c$ . Ogni  $x \in L$  è allora accettato dal programma seguente ( $I, I_1, \dots$  sono ID;  $I_0^x$  e  $I_f$  sono la ID iniziale per  $x$  e la ID accettante finale;  $\text{nxt}(I_1, I_2)$  decide la relazione:  $I_2$  è la ID raggiunta in un passo a partire da  $I_2$  oppure  $I_1 = I_2$ )

```
read  $x$ ;  $n := |x|$ ;  $S := S(n)$ ;  $m := cS$ ;  
if TEST( $I_0^x, I_f, m$ ) then YES else NO
```

```
  procedure TEST( $I_1, I_2, i$ )  
    if  $i = 0$  and  $\text{nxt}(I_1, I_2)$  then return true;  
    if  $i > 0$  then for all  $I$  such that  $|I| \leq S$  do;  
      if TEST( $I_1, I, i - 1$ ) and TEST( $I, I_2, i - 1$ ) then return true  
    return false
```

L'altezza dello stack associato è  $\leq S$ . Ogni nuovo record richiede  $S$  lettere per la parte delle ID relativa ai nastri di lavoro. L'input  $x$  è in un nastro read-only, che può essere rappresentato nelle ID dai  $\text{Lg}(n)$  bit dell'indirizzo del suo carattere osservato (perché è la sola cosa che cambia da una ID all'altra). Se dunque si ha  $S(n) \geq \text{Lg}(n)$ , lo spazio per le ID è  $O(S(n))$  e lo spazio totale è  $O(S(n)^2)$ .

## 4.6 QBF e la separazione di PSPACE dalle classi in esso contenute

**4.21** Il linguaggio  $L_{QBF}$  è la chiusura

1. delle *costanti* 0 (=vero) e 1; e delle *variabili booleane*  $x, y, x_1, \dots$
2. rispetto ai connettivi  $\neg, \wedge, \vee$
3. e alle quantificazioni  $\forall x, \exists x$

Il linguaggio QBF  $\subset L_{\text{QBF}}$  delle *quantified boolean formule* è la classe delle formule  $\in L_{\text{QBF}}$  *chiuse e soddisfacibili*.

**Esempio**  $\forall x(1 \wedge x \vee \exists y(x \vee y)) \notin \text{QBF}$  perché  $\wedge$  lega più di  $\vee$ , e perché  $1 \notin \text{QBF}$ .  
 $\forall x(0 \vee x \vee (x \vee y)) \notin \text{QBF}$  perché non è chiusa.

**4.22 Rappresentazione di un passo di TM nel linguaggio QBF** Il problema  $A \in \text{SAT}$  è un caso ristretto di QBF in cui non ci sono costanti e tutte le quantificazioni sono esistenziali prenesse (ossia, all'inizio della formula). Per esempio abbiamo

$$p \vee q \wedge \neg r \in \text{SAT} \quad \text{sse} \quad \exists p \exists q \exists r (p \vee q \wedge \neg r) \in \text{QBF}$$

Possiamo rappresentare una ID di lunghezza  $\sigma$  mediante  $2\sigma$  variabili booleane quaduple  $(i, j, \pm s, h)$  che dicono che in una certa ID si è nello stato  $i$ , si osserva  $j$  e  $h$  è in  $\pm s$ . Denoteremo con  $I, \dots, K$  delle collezioni di variabili booleane che rappresentano una ID.  $A(I, J, I_1, \dots)$  dice che nella formula  $A \in L_{\text{QBF}}$  occorrono le collezioni di variabili booleane  $I, J, I_1, \dots$ .  $\exists I(A(I))$  denota la formula ottenuta quantificando in modo esistenziale all'inizio di  $A$  le variabili booleane che compongono  $I$ .

Immaginiamo di prendere la formula  $next_t$  della dimostrazione 4.19 e:

1. riscrivere in rosso i literal in cui figura  $t$ , e togliere  $t$ ;
2. riscrivere in verde i literal in cui figura  $t + 1$ , e togliere  $t + 1$ ;
3. considerare le espressioni scritte in verde e in rosso come variabili booleane;
4. denotare con  $I$  e con  $J$  le collezioni risp. delle variabili rosse e verdi;
5. premettere  $\exists I$  ed  $\exists J$ .

Si ottiene un'espressione della forma

$$\overbrace{\exists(i_1, j_1, s_1, h_1) \dots \exists(i_{2\sigma}, j_{2\sigma}, s_{2\sigma}, h_{2\sigma})}^{\text{in rosso}} \overbrace{\exists(i_1, j_1, s_1, h_1) \dots \exists(i_{2\sigma}, j_{2\sigma}, s_{2\sigma}, h_{2\sigma})}^{\text{in verde}} \quad ($$

$$\underbrace{(i_1, j_1, s_1, h_1)}_{\text{in rosso}} \rightarrow \underbrace{(i_1, j_1, s_1, h_1)}_{\text{in verde}} \wedge \dots \wedge \underbrace{(i_{2\sigma}, j_{2\sigma}, s_{2\sigma}, h_{2\sigma})}_{\text{in rosso}} \rightarrow \underbrace{(i_{2\sigma}, j_{2\sigma}, s_{2\sigma}, h_{2\sigma})}_{\text{in verde}} \quad )$$

che denoteremo con

$$\exists I \exists J (\text{nxt}(I, J))$$

e che dice che si va dalla ID rappresentata da  $I$  a quella rappresentata da  $J$  in un passo. In generale, se  $A$  esprime una proprietà in QBF allora  $\exists I(A(I)) \in \text{QBF}$  equivale a dire che esiste una ID  $I$  per la quale  $A$  è vera.

#### 4.23 Lemma $\text{QBF} \in \text{DSPACE}(n^2)$

*Dimostrazione.*  $A_b^x$  denoti ( $b = 0, 1$ ) la sostituzione delle *occorrenze libere* (nel senso della logica matematica) di  $x$  in  $A$  con  $b$ . QBF è deciso dal programma

$$\begin{array}{lll} \text{ev}(A) \equiv [\text{cst}(A) & \rightarrow & A \\ A \equiv B \wedge / \vee C & \rightarrow & \text{ev}(B) \text{ and/or } \text{ev}(C) \\ A \equiv \forall / \exists x B(x) & \rightarrow & \text{ev}(B_0^x) \text{ and/or } \text{ev}(B_1^x) \\ A \equiv \neg B & \rightarrow & \text{not } \text{ev}(B) ] \end{array}$$

L'asserto segue perché le chiamate ricorsive del programma  $\text{ev}$  formano uno stack di altezza  $\leq |A|$  formato da record di lunghezza  $\leq |A|$ .

#### 4.24 Teorema di Stockmeyer $\text{P} = \text{PSPACE}$ sse $3\text{SAT} \in \text{PSPACE}$ .

*Dimostrazione.*  $\text{QBF} \notin \text{P}$  implica  $\text{P} \neq \text{PSPACE}$  per il lemma precedente. Dimostriamo l'altra metà del teorema facendo vedere che per ogni  $L \in \text{PSPACE}$  esiste un programma  $\text{rd}_L$  che *riduce* in tempo polinomiale  $L$  a QBF nel senso che si ha

$$x \in L \text{ sse } \text{rd}_L(x) \in \text{QBF}$$

Se ci fosse un programma  $\text{qbf}$  che decide QBF in tempo polinomiale,  $L$  sarebbe deciso in tempo polinomiale dal programma  $\text{rd}_L; \text{qbf}$ .

*Costruzione del programma  $\text{rd}_L$ .* Sia dato  $L \in \text{PSPACE}$  ed una TM  $M$  con  $Q$  stati,  $K$  caratteri di nastro, che accetta  $L$  in spazio  $n^c$ , e quindi in tempo  $2^{dn^c}$  per una costante  $d$ . Dato un input  $x$  poniamo  $m = 2^d |x|^c$ .

Dopo la dimostrazione del teorema di Savitch uno sarebbe tentato di porre

$$\begin{array}{ll} F_1(I, J) & = \text{nxt}(I, J) \\ F_{c+1}(I, J) & = \exists K_c (F_c(I, K_c) \wedge F_c(K_c, J)) \end{array}$$

di scrivere gli analoghi booleani delle  $I_0^x$  e  $I_f$  di quel teorema, e di concludere che

$$x \in L \text{ sse } F_m(I_0^x, I_f).$$

La conclusione sarebbe perfettamente legittima, ma inidonea a dimostrare il teorema. Le formule  $F_{c+1}$  sono infatti di lunghezza quasi doppia delle  $F_c$ , e dunque la lunghezza di  $F_m$  è esponenziale in  $|x|$ .

Il *trucco* consiste nell'usare i parametri di una stessa procedura per chiamarla due volte con ruoli diversi. Considerando infatti che

$$(a \rightarrow c) \wedge (b \rightarrow c) \text{ è equivalente a } (a \vee b \rightarrow c)$$

possiamo definire  $F_{c+1}$  scrivendo  $F_c$  una volta sola, nel modo che segue

$$\begin{aligned} F_1(I_1, I_2) &\equiv \text{next}(I_1, I_2) \\ F_{c+1}(I_1, I_2) &\equiv \exists I_c \forall J_c \forall K_c \left( \right. \\ &\quad \left. (J_c = I_1 \wedge K_c = I_2) \vee (I_c = J_c \wedge K_c = I_2) \rightarrow F_c(J_c, K_c) \right) \end{aligned}$$

# Capitolo 5

## Calcolo dei predicati

Un calcolo dei predicati (*puro* o del *primo ordine*) **PC** è un sistema formale che produce dimostrazioni formali di teoremi di logica pura, ossia leggi generali che valgono in in una classe molto vasta di domini, indipendentemente dalla loro natura.

### 5.1 Il linguaggio

**5.1**  $a, b, c, a_1, \dots$  sono variabili definite su una classe di segni formali, chiamati *variabili formali*, che non saranno mai esibiti esplicitamente <sup>1</sup>.

$P^{(n)}, Q^{(n)}, R^{(n)}, P_1^{(n)}, \dots$  sono variabili informali, definite su una classe di segni, chiamati *lettere (predicative) n-arie*, anch'essi mai esibiti esplicitamente.

Un *atomo* è un'espressione nella forma

$$(\neg)P^{(n)}x_1 \dots x_n$$

dove  $(\neg)$  significa che il segno  $\neg$  può esser assente.

Una *formula*  $A, \dots, D, A_1, \dots$  è un elemento della chiusura degli atomi rispetto ai *connettivi*  $\wedge$  e  $\vee$ ; e rispetto alle *quantificazioni*  $\forall a$  e  $\exists a$ .

**5.2 Notazione** (1) Omettiamo tutte le parentesi ridondanti rispetto alla priorità di  $\wedge$  su  $\vee$  e delle quantificazioni sui connettivi. Omettiamo inoltre le arità delle lettere quando note o generiche.

---

<sup>1</sup> Così come in matematica si dice per brevità che “ $h$  è un numero” invece che una “variabile definita sui numeri”, in metamatematica, quando si dice che  $a$  è una variabile formale, che  $A$  è una formula, etc., si intende che esse sono variabili definite risp. sulle variabili formali e sulle formule.

(2) In logica  $\Gamma, \Delta, \Gamma_1, \dots$  sono classi finite di formule, non alfabeti.  $\Gamma, \Delta$  è  $\Gamma \cup \Delta$  e  $\Gamma, A$  sta per  $\Gamma, \{A\}$ . Identificheremo ogni disgiunzione  $A_1 \vee \dots \vee A_k$  non nel raggio di un  $\wedge, \forall, \exists$  con  $\Gamma = A_1, \dots, A_k$ . Sicché  $\Gamma = A, B$  può denotare  $A \vee B, B \vee A, A \vee B \vee A \vee A$  e così via.

(3)  $at(\Gamma) \subseteq \Gamma$  è la classe di tutti gli atomi che occorrono in  $\Gamma$ . L'altezza  $ht(A)$  di  $A$  è 0 se  $A$  è un atomo, ed è  $1 + \max_i(ht(A_i))$  se  $A$  è costruita dalle  $A_i$  ( $1 \leq i \leq n$ ) mediante un connettivo ( $n = 2$ ) o un quantificatore ( $n = 1$ ).

(4)  $Fv(\Gamma)$  è la classe di tutte le variabili libere in  $\Gamma$  (ossia, non nel raggio di un  $\forall a$  o di un  $\exists a$ ).  $\Gamma$  è chiuso se si ha  $Fv(\Gamma) = \emptyset$ . La chiusura universale di  $A$  è la formula

$$\bar{\forall}A \equiv_{df} \forall a_1 \dots \forall a_n A \quad (Fv(A) = \{a_1, \dots, a_n\}).$$

**5.3 Sostituzione** La notazione composta  $A(a)$  dice che la variabile indicata può essere in  $Fv(A)$  (non necessariamente da sola).

Una volta assegnato un sistema di valori alle metavariable  $A, a, e b$ , l'espressione  $A(b)$  denota la sostituzione <sup>2</sup> di  $a$  con  $b$  in  $A$

**5.4 Altri connettivi** Definiamo  $\neg\neg A \equiv_{df} A$ , nonché (induttivamente)

$$\begin{array}{ll} \neg(B \vee C) \equiv_{df} \neg B \wedge \neg C & \neg(B \wedge C) \equiv_{df} \neg B \vee \neg C \\ \neg\forall a A \equiv_{df} \neg\exists a \neg A & \neg\exists a A \equiv_{df} \neg\forall a \neg A \end{array}$$

Poniamo infine

$$A \rightarrow B \equiv_{df} \neg A \vee B; \quad A \leftrightarrow B \equiv_{df} (A \rightarrow B) \wedge (B \rightarrow A).$$

## 5.2 Un apparato deduttivo

Si dice che ogni studioso di logica matematica prima o poi si costruisca per conto suo un apparato logico deduttivo: la *compilation* di regole e convenzioni esistenti più confacente alle sue personalissime idiosincrasie. Non credo che esista un sistema meno complicato di quello che ora vedremo.

**5.5 Derivazioni** una derivazione  $d$  di  $\Gamma$  in **PC** (notazione  $d \vdash_{\mathbf{PC}} \Gamma$ , o semplicemente  $\vdash \Gamma$ ) è un'assegnazione di formule ad un albero binario in modo che

(1) un *assioma* della forma

$$\Theta, A, \neg A \quad (A \text{ is an atom}).$$

<sup>2</sup> *bona fide*, ossia senza divertirsi a provocare conflitti tra variabili.

regola	nome	abbreviazione	restrizione
$\frac{\Gamma}{\Gamma, \Delta}$	$\forall$ -Introduzione	$I_{\forall}$	
$\frac{\Gamma, B_0 \quad \Gamma, B_1}{\Gamma \vee B_0 \wedge B_1}$	$\wedge$ -Introduzione	$I_{\wedge}$	
$\frac{\Gamma, B(b)}{\Gamma, \forall a B(a)}$	$\forall$ -Introduzione	$I_{\forall}$	$b \notin \text{Fv}(\Gamma)$
$\frac{\Gamma, B(b)}{\Gamma, \exists a B(a)}$	$\exists$ -Introduzione	$I_{\exists}$	

Figure 5.1: Regole di derivazione di un calcolo dei predicati

sia assegnato a ciascuna foglia, e  $\Gamma$  sia assegnato alla radice;

(2)  $\Delta$  è assegnato al nodo  $k$ -ario ( $k = 1, 2$ ) se ai  $k$  nodi sopra di lui sono assegnate le premesse  $\Delta_1, \dots, \Delta_k$  di una delle regole (di derivazione)  $k$ -arie mostrate in Fig. 5.1.

**5.6 Nota** Siccome  $A, \Gamma$ , etc. sono metavariable definite su domini numerabili questi assiomi e regole sono in realtà *schemi* di assioma e di regola. Pertanto il sistema ha infiniti assiomi ed infinite regole. Si può rendere finito il loro numero

- (1) rimpiazzando le metavariable con una scelta finita di entità formali;
- (2) aggiungendo una regola di sostituzione postulata. Ma uso e formulazione di una tale regola danno buone possibilità di errore (tanto agli autori quanto ai lettori).

**5.7 Lemma**  $\vdash \Gamma, A, \neg A$  per ogni  $\Gamma$  e  $A$ .

*Proof.* Induzione sulla costruzione di  $A$ . Base.  $A, \neg A$  è un assioma.

Passo. Caso 1.  $A$  è  $B \wedge C$ , e quindi  $\neg A$  è  $\neg B \vee \neg C$ . I.H. dà

$$d_0 \vdash \Gamma, B, \neg B; \quad d_1 \vdash \Gamma, C, \neg C.$$

L'asserto segue perché si ha

$$d \left\{ \frac{d_0 \left\{ \frac{\dots}{\Gamma, B, \neg B} \right. \quad d_1 \left\{ \frac{\dots}{\Gamma, C, \neg C} \right.}{\Gamma, \neg B, \neg C, B \wedge C} \right.$$

con ultima inferenza per  $I_{\wedge}$  con formule *lateral*  $\Gamma, \neg B$  e  $\Gamma, \neg C$ .

Case 2.  $A$  è  $\forall aB$ . I.H. dà per ogni  $c$

(a)  $\vdash \Gamma, B(c), \neg B(c)$ .

Scegliamo un valore  $b$  per  $c$  in modo che  $b \notin \text{Fv}(\Gamma)$ . Si ha allora

$$\frac{\frac{\Gamma, B(b), \neg B(b)}{\Gamma, B(b), \exists a \neg B(a)}}{\Gamma, \forall a B(a), \exists a \neg B(a)}$$

dove l'ultima inferenza è corretta perché si ha  $b \notin \text{Fv}(\Gamma)$ .

Case 3.  $A \equiv B \vee C$  oppure  $A \equiv \exists aB$ . Simmetricamente.

## 5.3 Interpretazione

**5.8 Metacostanti logiche** Per tenere distinta la logica che adoperiamo nel nostro studio dalla logica formale che ne è oggetto, aggiungiamo al linguaggio naturale le seguenti *estensioni significanti convenzionali* (in parole povere: abbreviazioni)

*non et vel seq om ex*

**5.9 Interpretation** Data una classe finita o numerabile  $D$ , una *interpretazione* di **PC** in  $D$  è una rappresentazione  $\mathfrak{S}_l^D$  delle lettere  $n$ -arie nelle funzioni da  $D^n$  ai valori di verità 0/1 (*vero/falso*). In simboli

$$\mathfrak{S}_l^D P^{(n)} : D^n \mapsto \{0, 1\}.$$

Ogni  $\mathfrak{S}^D$  definisce una interpretazione di tutte le formule mediante le regole ( $x, x_1, \dots$  sono variabili definite su  $D$ )

$$\begin{aligned} \mathfrak{S}^D P a_1, \dots, a_n &= \mathfrak{S}_l^D P(x_1, \dots, x_n) \\ \mathfrak{S}^D \neg A &= \text{non } \mathfrak{S}_l^D A && \text{per ogni atomo } A \\ \mathfrak{S}^D B \wedge C &= \mathfrak{S}^D B \text{ et } \mathfrak{S}^D C \\ \mathfrak{S}^D B \vee C &= \mathfrak{S}^D B \text{ vel } \mathfrak{S}^D C \\ \mathfrak{S}^D \forall a B &= (\text{om } x)(\mathfrak{S}^D B) \\ \mathfrak{S}^D \exists a B &= (\text{ex } x)(\mathfrak{S}^D B) \end{aligned}$$

**5.10 Validità** Siano date una formula  $A$  e un'interpretazione  $\mathfrak{S}^D$ .

1. Se  $A$  è chiusa  $\mathfrak{S}^D A$  è un valore di verità.

$\mathfrak{S}^D$  è un *modello/contro-esempio* in  $D$  per  $A$  se  $\mathfrak{S}^D A = 0/1$ .

2. Se  $l$  variabili sono libere in  $A$  allora  $\mathfrak{S}^D A$  è una funzione  $F(x_1, \dots, x_l)$  con valori 0/1.  
 $\mathfrak{S}^D$  è un contro-esempio in  $D$  per  $A$  se c'è una  $l$ -pla  $h_1, \dots, h_l \in D$  di valori per le variabili di  $F$  tale che  $F(h_1, \dots, h_l)$  è falsa.
3.  $A$  è *valida* in  $D$  se ogni  $\mathfrak{S}^D$  è un modello per  $\bar{\forall}A$  (notazione:  $\Vdash^D A$ ).  
 $A$  è valida se lo è in ogni  $D$  (notazione:  $\Vdash A$ ).

**5.11 Teorema**  $\vdash \Gamma$  implica  $\Vdash \Gamma$ .

*Dimostrazione.* (Schema.) Induzione sull'altezza (lunghezza del cammino piú lungo) della derivazione  $d \vdash \Gamma$  in ipotesi.

Base. Immediata per il principio (metamatematico) del terzo escluso.

Passo. Casi secondo l'ultima inferenza  $J$  di  $d$ .

Caso 1.  $J \neq I_{\forall}$ . Si vede subito che la verità in qualunque  $\mathfrak{S}^D$  è conservata da ciascuna  $J$ .

Caso 2.  $J = I_{\forall}$ . La penultima formula di  $d$  è  $\Delta, B(b)$ , e si ha

$$\Gamma = \Delta, \forall a B \quad (b \notin \text{Fv}(\Delta))$$

I.H. dà allora

$$\mathfrak{S}^D \Delta \text{ vel } \mathfrak{S}^D B(y) \quad \text{per ogni } \mathfrak{S}^D \text{ e per ogni } y \in D$$

L'asserto segue con l'argomento di uso comune in matematica:

- (a) abbiamo visto che le ipotesi *non*  $\mathfrak{S}^D \Delta$  implicano la tesi  $\mathfrak{S}^D B(y)$ ;
- (b) ma le ipotesi non dipendono da  $y$ ;
- (c) dunque la tesi vale per ogni  $y$ .

**5.12 Note** Immaginiamo ammessa l'inferenza

$$\frac{P(b), \neg P(b)}{\forall a P(a), \neg P(b)}$$

La regola  $I_{\forall}$  darebbe allora (correttamente)  $\forall a P(a) \vee \forall a \neg P(a)$  — una sciocchezza, perché non puoi dedurre, dal fatto che ogni numero è pari o dispari, che tutti i numeri sono pari o tutti i numeri sono dispari.

## 5.4 Completezza

**5.13 Teorema** Esiste un algoritmo `cmp1` che per ogni  $\Gamma$  chiuso dà un albero  $Tr(\Gamma)$  che

1. se  $\Vdash \Gamma$  è una derivazione  $d \vdash \Gamma$ ;
2. se non  $\Vdash \Gamma$  può avere
  - (a) una foglia che dà un controesempio;
  - (b) oppure un cammino infinito che definisce un contro esempio.

**5.14 Corollario**  $\vdash \Gamma$  sse  $\Vdash \Gamma$  per ogni  $\Gamma$  chiuso.

*Costruzione.* `cmp1` sarà descritto in un linguaggio nonderministico *ad hoc*. Ciascuna delle sue computazioni definisce un cammino  $\mathbf{C}$  di  $Tr(\Gamma)$ . Ogni  $\mathbf{C}$  può terminare a una foglia  $\Delta$  che

- (a) può essere un assioma;
- (b) se non lo è, i suoi atomi ammettono un controesempio che falsifica anche  $\Gamma$ ;
- (c) infine  $\mathbf{C}$  può divergere; in questo caso gli infiniti atomi che occorrono in  $\mathbf{C}$  ammettono un controesempio sull'insieme  $\mathcal{N}$  dei naturali, che è un controesempio anche per  $\Gamma$ .

Si ottengono questi atomi riducendo (o cercando di ridurre) ad ogni nuovo nodo la complessità logica della formula associata al nodo precedente. Supponiamo che alla fine della *fase i* `cmp1` lavori su una foglia

$$\Gamma_i = A_1, \dots, A_{n(i)}$$

in cui  $k(i) < n(i)$  tra le  $A_j$  non sono atomi. La fase  $i + 1$  è allora divisa in  $k(i)$  *sottofasi* non/deterministiche

1. se la sottofase è deterministica la nuova formula è falsificabile sse lo è quella precedente
2. altrimenti la nuova formula di una o dell'altra computazione che comincia è falsificabile sse lo è quella precedente
3. in ogni caso le nuove foglie sono meno complicate della precedente

La forma di ogni cammino è

$$\Gamma = \Gamma_0, \dots, \Gamma_i = \Delta_{i0}, \dots, \Delta_{i, k(i)} = \Gamma_{i+1}.$$

Un controesempio per  $\Gamma_{i+1}$  lo è per tutte le formule del cammino.  
In ogni sottofase si applica una di queste regole

$$\frac{\frac{\frac{\Theta, B(b_{q+1})}{\Theta, \forall a B(a)} \quad E_{\forall}}{\Theta, B_i} \quad E_{\wedge}}{\frac{\Theta, \exists a B_a, B(b_1), \dots, B(b_{\max(1,q)})}{\Theta, \exists a B(a)} \quad E_{\exists}}$$

dove

1.  $q \geq 0$  è il numero di  $E_{\forall}$  nel cammino da  $\Theta, \forall a B(a)$  a  $\Gamma$  (questo è un punto su cui concentrare l'attenzione per cogliere il senso dell'algoritmo);
2.  $b_1, b_2 \dots$  sono variabili non in  $\Gamma$
3. in  $E_{\wedge}$  si sceglie  $B_i$  nondeterministicamente (ossia, due distinte computazioni mettono sopra la foglia  $\Theta, B_0 \wedge B_1$  le nuove foglie  $\Theta, B_0$  e  $\Theta, B_1$ );
4. il secondo punto nondeterministico di `cmp1` è la scelta della formula da processare nella sottofase corrente.

```

cmpl =    $s := true; t := true; q := 1; \Theta := \Gamma;$ 
           while  $s$  and  $t$  do
              $\Delta := at(\Gamma); \Gamma := \Gamma - at(\Gamma)$ 
             while  $\Gamma \neq \emptyset$  do
               riduci;
             if  $\Delta = \Lambda, P(a), \neg P(a)$ 
               then  $s := false$ ;
             if  $\Delta \subseteq \Theta$ 
               then  $t := false$    else    $\Theta := \Theta, \Delta; \Gamma := \Delta$ 
             end;
           if  $s = false$  then “axiom”   else   “no”

riduci = choose  $A \in \Gamma;$ 
            $\Gamma := \Gamma - A;$ 
           if  $A = B \wedge C$ 
             then choose  $\Delta := \Delta, B$  or  $\Delta := \Delta, C;$ 
           if  $A = \forall a B(a)$ 
             then  $\Delta := \Delta, B(b_q); q := q + 1;$ 
           if  $A = \exists a B(a)$ 
             then  $\Delta := \Delta, A, B(b_1), \dots, B(b_q)$ 

```

Si osservi che **cmpl** si ferma se e quando trova un  $\Gamma_i$  che è un assioma ( $s = false$ ) oppure che non dice nulla di nuovo, nel senso che tutte le sue formule sono già nella formula  $\Theta$  nella quale sono *accumulate* tutte le formule che già si trovano da qualche parte lungo il cammino corrente ( $t = false$ ).

**5.15 Lemma** non  $\Vdash \Gamma$  se una computazione di **cmpl** dà un *no* oppure è infinita.

*Dimostrazione.* Un cammino **C** di **cmpl** sia infinito, o restituisca un *no* alla fine della fase  $i$ . Definiamo

$$D = \mathcal{N} \text{ se } \mathbf{C} \text{ è infinito; } D = \{1, \dots, q\} \text{ altrimenti}$$

Dimostriamo che è un controesempio la  $\mathfrak{S}^D$  definita mediante i seguenti sistemi di valori  $h_i \in D$  per le funzioni  $\mathfrak{S}_i^D P(\cdot)$

$$(a) \quad \begin{aligned} \mathfrak{S}_i^D(P^{(n)})(h_1, \dots, h_n) &= 1 \text{ sse } P(b_{h_1}, \dots, b_{h_n}) \in \Theta \\ \mathfrak{S}_i^D(P)(h_1, \dots, h_n) &= 0 \text{ altrimenti.} \end{aligned}$$

(Dunque associamo il numero  $i$  alla variabile  $b_i$  e poniamo per esempio  $\mathfrak{S}_i^D P(2, 3) = 1$  se  $P(b_2, b_3)$  figura in **C**.)

Assumiamo (ad abs.) che non sia vuota la classe  $\Theta^+ \subseteq \Theta$  di tutte le formule vere (secondo  $\mathfrak{S}^D$ ) appartenenti a  $\Theta$ . Sia  $B$  una formula di altezza minima tra le formule di  $\Theta^+$ . Sicché per ogni  $C \in \Theta^+$  si ha

$$(b) \quad ht(B) \leq ht(C)$$

Osserviamo che, siccome il programma non modifica gli atomi (negati o no) tutti gli atomi in  $\Theta$  sono falsi e dunque  $ht(B) > 0$ .

Casi secondo la forma di  $B$ .

Caso 1.  $B \equiv C_0 \wedge C_1$ . L'una o l'altra delle  $C_j$  è in  $\Theta$  in seguito ad una  $E_\wedge$ . Essa però non è in  $\Theta^+$  per la (b). Pertanto  $C_j$  è falsa e così anche  $B$ .

Caso 2.  $B \equiv \forall aC(a)$ . Per qualche  $j \in D$  si ha  $C(b_j) \in \Theta$ . Essa però non è in  $\Theta^+$  per la (b). Pertanto  $C(b_j)$  è falsa e così anche  $B$ .

Caso 3.  $B \equiv \exists aD(a)$ . Per ogni  $j \in D$  si ha  $C(b_j) \in \Theta$ , benché  $C(b_j) \notin \Theta^+$  per la (b). Pertanto tutte le  $C(b_j)$  sono false e così anche  $B$ .

**5.16 Lemma** Se tutte le computazioni restituiscono *assioma*, allora  $\vdash \Gamma$ .

*Dimostrazione.* Facciamo vedere che tutti i nodi  $\Delta$  di  $Tr(\Gamma)$  sono derivabili. Induzione sull'altezza del sottoalbero  $Tr(\Delta)$  di  $Tr(\Gamma)$  che ha  $\Delta$  come sua radice. Base.  $\Delta$  è un assioma.

Passo. Casi secondo la regola (di eliminazione)  $J$  piú in basso in  $Tr(\Delta)$ .

Caso 1.  $\Delta \equiv \Theta, B_0 \wedge B_1$  e

$$\frac{\Theta, B_i}{\Theta, B_0 \wedge B_1} J = E_\wedge$$

l'asserto segue immediatamente per I.H. (due volte) e  $I_\wedge$ .

Caso 2. Si ha

$$\frac{\Theta, B(b_{q+1})}{\Theta, \forall aB(a)} J = E_\forall$$

Per I.H. si ha  $d \vdash \Theta, B(b_{q+1})$ , e l'asserto segue per  $I_\forall$ , (correttamente in quanto  $b_{q+1}$  non può occorrere in  $\Theta$ , perché non era in  $\Gamma$ , e perché tutte le variabili in  $\Theta - \Gamma$  sono nella forma  $b_h$  con  $h \leq q$ ).

Caso 3. Si ha

$$\frac{\Theta, \exists aB(a), B(b_1), \dots, B(b_{\max(1,q)})}{\Theta, \exists aB(a)} E_\exists$$

e l'asserto segue per I.H. con al piú  $q$  inferenze per  $I_\exists$  (una per ogni  $B(b_i) \notin \Theta$ ).

**5.17 Definizione** L'*ordine* di un nodo di un albero è il numero degli archi che ne escono. L'ordine dell'albero è il max tra gli ordini dei suoi nodi, e può essere infinito.

**5.18 Fan Principle** In ogni albero d'ordine finito con infiniti nodi c'è un cammino infinito.

Questo principio è un teorema della teoria degli insiemi (con assioma di scelta) noto come Lemma di König. Esso dice che c'è un cammino  $N_0, \dots, N_i, \dots$  che è infinito perché ogni nodo ha un successore. Per *illustrarlo* facciamo vedere che c'è un cammino i cui nodi sono tutti radici di un sottoalbero con infiniti nodi. Prendiamo la radice come  $N_0$ . Se  $N_i$  è unario, prendiamo il padre come  $N_{i+1}$ . Se è binario l'uno o l'altro dei suoi genitori è radice di un albero infinito, e lo possiamo *prendere* come  $N_{i+1}$ .

**5.19 Nota** Nell'albero-ordinale noto come  $\omega$

1. C'è una radice dalla quale esce un arco per ogni  $i$ ;
2. dal  $i$ -mo nodo adiacente la radice parte un cammino fatto di  $i$  nodi unari.

Esso non contraddice il lemma di König perché c'è un nodo infinitario (la radice).

*Dimostrazione del teorema.* Se tutte le computazioni di `cmp1` si arrestano abbiamo o una derivazione (per 5.16) oppure un controesempio (per 5.15). Se qualche cammino non si arresta,  $Tr(\Gamma)$  ha infiniti nodi e pertanto (per 5.18) un cammino infinito. Ancora per 5.15 c'è allora un controesempio.

**Una dimostrazione non costruttiva** Il Fan Principle dice che c'è un cammino infinito, non quale sia (nell'illustrazione che ne abbiamo fatto non abbiamo detto come si faccia a sapere qual'è il nodo da *prendere*.) Quindi tutte le derivazioni sono realmente generate da un programma, ma la dimostrazione che il programma funziona non è costruttiva (come del resto tutte le dimostrazioni note di questo teorema).

**Corollario** (Teorema di Löwenheim-Skolem) Ogni formula, che ammette un modello, ammette un modello finito o numerabile.

**Nota** Si osservi che nel nostro sistema non ci sono regole di tipo *modus ponens*, *cut*, etc. Le *divagazioni* e gli insiemi non numerabili non sono dunque necessari per dimostrare le leggi della logica pura.

## 5.5 Indecidibilità

L'idea centrale nella dimostrazione del teorema di Cook è che una formula della logica degli enunciati può *rappresentare* il comportamento di una NTM  $M$  se  $\tau_M$  è nota. Si ottiene una tale rappresentazione

1. assegnando un meta-significato convenzionale ad una classe di variabili proposizionali;
2. scrivendo alcune formule che rispecchiano il comportamento di  $M$  per input  $x$ ;
3. dimostrando che una certa formula può assumere il valore vero sse  $M$  accetta  $x$ .

Il lavoro di questa sezione è analogo. Riduciamo il problema della terminazione di una generica TM ad una formula di **CP** che è valida sse  $M$  converge. Siccome  $M$  può divergere non disponiamo di una funzione  $\tau_M$  e questo rende necessario trovare un modo di esprimere ricorsioni in **CP**.

**5.20 Stringhe nell'alfabeto unario** Siano date le lettere ad un posto  $P_1, P_2$  e  $P_3$ , la lettera a due posti  $Q$  e quella a tre  $R$ . Conveniamo di scrivere

$$E \equiv_{\text{ab}} F$$

per dire che  $E$  è un'abbreviazione (sperabilmente più leggibile) per  $F$ , e poniamo

$$\begin{aligned} a = b &\equiv_{\text{ab}} Q(a, b) \\ a = 1 &\equiv_{\text{ab}} P_1(a) \\ \mathbf{str}(a) &\equiv_{\text{ab}} P_2(a) \\ a = \epsilon &\equiv_{\text{ab}} P_3(a) \\ a = bc &\equiv_{\text{ab}} R(a, b, c) \end{aligned}$$

Consideriamo ora alcune formule

$$\begin{aligned} \text{EQL} &=_{\text{df}} a = b \wedge a = c \rightarrow b = c \\ \text{EMPTbase} &=_{\text{df}} \epsilon a = a \\ \text{EMPTstep} &=_{\text{df}} a = \epsilon \rightarrow \forall b \forall c (b = c \rightarrow ab = ac \wedge ba = ca) \\ \text{STRING} &=_{\text{df}} \mathbf{str}(a) \leftrightarrow a = \epsilon \vee \exists b \exists c (b = 1 \wedge \mathbf{str}(c) \wedge a = bc) \\ \text{CAT} &=_{\text{df}} a = bc \leftrightarrow a = b = c = \epsilon \vee \\ &\quad \exists d \exists e \exists f (d = 1 \wedge a = de \wedge b = df \wedge e = fc) \end{aligned}$$

Chiamiamo *postulati del semigruppone unario* queste formule, perché ogni loro modello in

$$D = \{x \mid x = 1^n \text{ per } n \geq 0\}$$

conferisce a  $D$  la struttura di semigruppone rispetto alla concatenazione. In particolare, EQL rispecchia la definizione di uguaglianza di Euclide: due cose uguali ad una terza sono uguali tra loro<sup>3</sup>.

**5.21 Stringhe generiche** Si ottiene il semigruppone delle stringhe nel generico alfabeto  $\mathbf{A} = \{1, \dots, K\}$ <sup>4</sup>, aggiungendo una lettera unaria per ogni nuova lettera, scrivendo

$$\begin{aligned} a = b &\equiv_{\text{ab}} Q(a, b) \\ a = i &\equiv_{\text{ab}} P_i(a) & 1 \leq i \leq K \\ \mathbf{str}(a) &\equiv_{\text{ab}} P_{K+1}(a) \\ a = \epsilon &\equiv_{\text{ab}} P_{K+2}(a) \\ a = bc &\equiv_{\text{ab}} R(a, b, c) \\ \mathbf{lett}(a) &\equiv_{\text{ab}} \bigvee_{1 \leq i \leq K} P_i \end{aligned}$$

aggiungendo un postulato, e modificando leggermente un altro

$$\begin{aligned} \text{EQL} &=_{\text{df}} a = b \wedge a = c \rightarrow b = c \\ \text{EMPTbase} &=_{\text{df}} \epsilon a = a \\ \text{EMPTstep} &=_{\text{df}} a = \epsilon \rightarrow \forall b \forall c (b = c \rightarrow ab = ac \wedge ba = ca) \\ \text{STRING} &=_{\text{df}} \mathbf{str}(a) \leftrightarrow a = \epsilon \vee \exists b \exists c (\mathbf{lett}(b) \wedge \mathbf{str}(c) \wedge a = bc) \\ \text{CAT} &=_{\text{df}} a = bc \leftrightarrow a = b = c = \epsilon \vee \\ &\quad \exists d \exists e \exists f (d = 1 \wedge a = de \wedge b = df \wedge e = fc) \\ \text{UNIVLETT} &=_{\text{df}} \bigwedge_{1 \leq i, j \leq K; i \neq j} a = i \rightarrow \neg a = j \end{aligned}$$

**Rappresentazione di una stringa particolare.** La formula

$$a = \text{“1431”} \equiv_{\text{ab}} \exists d \exists e \exists w \exists c (a = 1d \wedge d = 4e \wedge e = 3w \wedge w = 1)$$

ha un modello in  $\mathbf{A}$  solo se ad  $a$  viene associata la stringa 1431. Il metodo è ovviamente generalizzabile a stringhe qualsiasi.

<sup>3</sup> Si osservi che EQL insieme con EMPTbase implica le consuete proprietà dell’uguaglianza. Per esempio, sostituendo simultaneamente  $\epsilon a$  ad  $a$ , e  $a$  sia a  $b$  che a  $c$  si ottiene  $\epsilon a = a \wedge \epsilon a = a \rightarrow a = a$  da cui, con due *cut* la simmetria.

<sup>4</sup> Immaginiamo di avere una lettera o *cifra* diversa  $i$  per ogni  $1 \leq i \leq K$

**5.22 Lemma** Per ogni TM  $M$  c'è una formula  $A(a)$  tale che

$$\Vdash \forall a (a = \text{"T"} \rightarrow A(a)) \text{ sse } M \text{ accetta } T$$

*Dimostrazione.* Sia data una TM  $M$  con  $Q$  stati su un alfabeto di cardinalità  $K$ . Essa manipola stringhe secondo certe sue regole. Per rappresentare il suo comportamento, aggiungiamo al lavoro per le stringhe su  $\{0 = \#, 1, \dots, Q+K\}$  altre lettere e postulati che descrivono computazioni ( $\#$  è una lettera in piú che useremo come separatore tra due ID).

(1) Formule che dicono che  $x$  è risp. uno stato, un simbolo di nastro, e un nastro ( $i, j, L, R$  sono variabili formali a cui abbiamo assegnato un meta-nome mnemonico)

$$\begin{aligned} \text{state}(i) &\equiv_{\text{ab}} i = 1 \vee \dots \vee i = Q \\ \text{tsymb}(j) &\equiv_{\text{ab}} j = Q + 1 \vee \dots \vee j = Q + K \\ \text{tape}(a) &\equiv_{\text{ab}} \text{come string ma con tsymb invece di lett} \\ a = \# &\equiv_{\text{ab}} \text{lett}_0(a) \end{aligned}$$

(2) Formula che dice che  $a$  è una ID

$$\text{id}(a) \equiv_{\text{ab}} \exists L \exists R \exists i \exists j (a = Li jR \wedge \text{tape}(L) \wedge \text{tape}(R) \wedge \text{state}(i) \wedge \text{tsymb}(j))$$

(3) Postulati che descrivono  $M$ . Per ogni

$$1 \leq i, i^* \leq Q (i \neq i^*); Q + 1 \leq j, j^* \leq Q + K (j \neq j^*); 1 \leq I \leq Q + K + 2$$

prendiamo una lettera predicativa  $P_{ijI}$ ;  $M$  è descritta dalla formula

$$M \equiv_{\text{ab}} \forall a \bigwedge_{M \models [i, j, I]} (P_{ijI}(a))$$

(4) Un passo di  $M$  è descritto dalla formula

$$\begin{aligned} \text{nxt}(a, b) &\equiv_{\text{ab}} \exists L \exists R \exists i \exists j ( \text{\textsubscript{1}} a = Li jR \wedge i = Q \wedge b = a \vee \\ &\quad \vee_{ij i^*} (P_{ij i^*} \wedge b = Li^* jR) \vee \\ &\quad \vee_{ij, Q+j^*} (P_{ij, Q+j^*} \wedge b = Li j^* R) \vee \\ &\quad \vee_{ij} (P_{ij, Q+K+1} \wedge \exists L \exists h (L = L^* h \wedge b = L^* i h j R)) \vee \\ &\quad \vee_{ij} (P_{ij, Q+K+2} \wedge b = L j i R) ) \text{\textsubscript{1}} \end{aligned}$$

(5) Computazione accettante  $a$  (con la convenzione che  $M$  cominci in stato 1, osservando un 1 alla destra dell'input, e accetti osservando un 1 in stato  $Q$ )

$$\begin{aligned} \text{accmp}(a, b) &= \exists d ( \text{\textsubscript{1}} \exists L \exists R (a = d \# L Q 1 R \wedge \text{nxt}(d, L Q 0 R)) \vee \\ &\quad \exists e \exists w (a = d \# e \# w \wedge \text{nxt}(d, e) \wedge \text{accmp}(e \# w)) \text{\textsubscript{1}} \\ &\quad \exists c (b = a 1 1 \# c) \end{aligned}$$

Il lemma segue perché  $M$  accetta  $x$  sse è valida la formula (PSTL è la congiunzione dei postulati del §5.21)

$$\forall a(a = "x" \rightarrow (\text{PSTL} \wedge M \rightarrow \exists b(\text{accmp}(a, b))))$$

**5.23 Teorema di Church-Turing** L'insieme delle formule valide è indecidibile.

*Dimostrazione.* Ad abs facendo vedere che l'insieme

$$K = \{x \mid x \text{ accetta } x\}$$

è Turing-riducibile all'insieme  $\text{VAL} =_{\text{df}} \{A \mid \Vdash A\}$ . La decidibilità di VAL sarebbe allora in contraddizione col teorema di Rice Sia  $U_2$  una variante della LPM universale che duplica il suo input  $x$  e lo accetta se e solo se  $x$  accetta  $x$  sicché

$$U_2 \text{ accetta } a \text{ sse } a \text{ accetta } a$$

per la formula  $A(a)$  associata dal lemma ad  $U_2$  si ha

$$\begin{aligned} \Vdash \forall a(a = "x" \rightarrow A(a)) & \text{ sse } U_2 \text{ accetta } x \\ & \text{ sse } x \text{ accetta } x \end{aligned}$$