



NoiStudiamo

DISPENSA DI TEORIA



Fondamenti di Informatica

Linguaggi formali, automi e macchine di Turing, calcolo lambda, codici, macchine astratte, logica e semantica dei programmi

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

Gli argomenti della dispensa seguono il programma del corso, dai linguaggi formali alla semantica dei programmi.

1. Linguaggi formali	3
2. Grammatiche e gerarchia di Chomsky	7
3. Automi a stati finiti e linguaggi regolari	11
4. Cenni sui linguaggi non contestuali	17
5. Modelli computazionali e macchine di Turing	19
6. Programmazione funzionale e λ calcolo	23
7. Codici e rappresentazione dell'informazione numerica	28
8. Macchine astratte	32
9. Sistemi formali e logica proposizionale	36
10. Logica dei predicati e formalizzazione di teorie	41
11. Induzione e ricorsione	46
12. Semantica dei linguaggi di programmazione: il linguaggio WHILE	50

Linguaggi formali

Alfabeti, stringhe e linguaggi sono i mattoni di tutta la teoria: grammatiche, automi e macchine di Turing servono a descrivere o a riconoscere insiemi di stringhe.

Alfabeti, stringhe e linguaggi

Un linguaggio formale è un insieme di parole di lunghezza finita costruite a partire da un insieme finito di simboli. La teoria dei linguaggi formali interviene ovunque occorra stabilire se una sequenza di simboli abbia una certa forma: nella ricerca di una parola in un testo, nella descrizione di sequenze di DNA, nell'analisi sintattica svolta dai compilatori.

DEFINIZIONE

Alfabeto. Un alfabeto è un insieme finito e non vuoto Σ di simboli, detti caratteri. Esempio: $\Sigma = \{a, b\}$ oppure l'alfabeto binario $\{0, 1\}$.

DEFINIZIONE

Stringa o parola. Una stringa su Σ è una sequenza finita di simboli di Σ . La stringa vuota, indicata con ε (talvolta λ), non contiene alcun simbolo. L'insieme di tutte le stringhe su Σ si indica con Σ^* ed è definito per induzione: $\varepsilon \in \Sigma^*$; se $x \in \Sigma^*$ e $a \in \Sigma$, allora $xa \in \Sigma^*$; nient'altro appartiene a Σ^* .

Per convenzione si usano le lettere finali dell'alfabeto latino x, y, z per le stringhe e le prime lettere a, b, c per i simboli. Per mostrare che una stringa appartiene a Σ^* si ripercorre la definizione induttiva: con $\Sigma = \{a, b, c\}$ la stringa aab è in Σ^* perché $\varepsilon \in \Sigma^*$ e $a \in \Sigma$ danno $a \in \Sigma^*$, poi $a \in \Sigma^*$ e $a \in \Sigma$ danno $aa \in \Sigma^*$, infine $aa \in \Sigma^*$ e $b \in \Sigma$ danno $aab \in \Sigma^*$.

Il numero di stringhe su un alfabeto non vuoto è infinito, anche se ogni stringa ha lunghezza finita: è la stessa situazione dei numeri naturali, che sono infiniti pur avendo ciascuno un numero finito di cifre.

LUNGHEZZA, CONCATENAZIONE, POTENZA

La lunghezza $|w|$ di una stringa è definita per induzione: $|\varepsilon| = 0$ e $|xa| = |x| + 1$. La concatenazione di x e y , indicata con $x \circ y$ o semplicemente xy , giustappone le due stringhe.

PROPRIETÀ	ENUNCIATO	COMMENTO
Chiusura	$x, y \in \Sigma^*$ implica $xy \in \Sigma^*$	la concatenazione non esce da Σ^*
Associatività	$(xy)z = x(yz)$	le parentesi sono inessenziali
Elemento neutro	$x\varepsilon = \varepsilon x = x$	ε non altera nessuna stringa
Commutatività	in generale $xy \neq yx$	la concatenazione non è commutativa
Lunghezza	$ xy = x + y $	la lunghezza è additiva

Per queste proprietà la terna $(\Sigma^*, \circ, \varepsilon)$ è un monoide, detto monoide libero o monoide sintattico su Σ : rispetto a un gruppo manca l'inverso. La potenza di una stringa è $x^0 = \varepsilon$ e $x^n = x \circ x^{n-1}$ per $n > 0$.

STRINGA INVERSA

L'inversa (o riflessa) di una stringa x , indicata con x^R , si ottiene leggendola da destra a sinistra. Si definisce per induzione: l'inversa di ε è ε e l'inversa di ay , con $a \in \Sigma$ e $y \in \Sigma^*$, è l'inversa di y seguita da a . Una stringa che coincide con la propria inversa si dice palindroma.

DEFINIZIONE

Linguaggio. Dato un alfabeto Σ , un linguaggio formale su Σ è un qualsiasi sottoinsieme di Σ^* . Poiché $\Sigma \subseteq \Sigma^*$, ogni alfabeto è a sua volta un linguaggio. Il linguaggio vuoto, indicato con Λ , non contiene alcuna stringa.

DA RICORDARE

Il linguaggio vuoto Λ è l'insieme senza elementi: non va confuso con $\{\varepsilon\}$, che è il linguaggio formato dalla sola stringa vuota. Quindi $\Lambda \neq \{\varepsilon\}$, $|\Lambda| = 0$ e $|\{\varepsilon\}| = 1$.

Operazioni sui linguaggi

Poiché i linguaggi sono insiemi, si possono combinare con le operazioni insiemistiche. Si aggiungono due operazioni che sfruttano la struttura di monoide: la concatenazione e l'iterazione.

OPERAZIONE	DEFINIZIONE	ESEMPIO
Unione	$L_1 \cup L_2 = \{x \mid x \in L_1 \text{ oppure } x \in L_2\}$	$\{ab, a\} \cup \{a, b\} = \{ab, a, b\}$
Intersezione	$L_1 \cap L_2 = \{x \mid x \in L_1 \text{ e } x \in L_2\}$	$\{ab, a\} \cap \{a, b\} = \{a\}$
Complemento	complemento di $L = \Sigma^* - L$	le stringhe di Σ^* non in L
Concatenazione	$L_1 \circ L_2 = \{y_1 y_2 \mid y_1 \in L_1, y_2 \in L_2\}$	$\{ab, a\} \circ \{a, b\} = \{aba, abb, aa, ab\}$
Potenza	$L^0 = \{\varepsilon\}$, $L^h = L \circ L^{h-1}$	$L^2 = L \circ L$
Chiusura riflessiva	$L^* = \text{unione di } L^h \text{ per } h \geq 0$	$\{aa\}^* = \{a^{2n} \mid n \geq 0\}$
Chiusura positiva	$L^+ = \text{unione di } L^h \text{ per } h \geq 1$	$\{aa\}^+ = \{a^{2n} \mid n \geq 1\}$

PROPRIETÀ DA RICORDARE

- $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$: $\{\varepsilon\}$ è l'elemento neutro della concatenazione di linguaggi.
- $L \circ \Lambda = \Lambda \circ L = \Lambda$: il linguaggio vuoto annulla la concatenazione.
- $\Lambda^0 = \{\varepsilon\}$ per la convenzione sulla potenza zero, quindi $\Lambda^* = \{\varepsilon\}$.
- $\varepsilon \in L^*$ per ogni L , ed è sempre $L^* = L^+ \cup \{\varepsilon\}$.
- $\varepsilon \in L^+$ se e solo se $\varepsilon \in L$. Perciò $L^+ = L^*$ meno $\{\varepsilon\}$ vale solo quando $\varepsilon \notin L$.
- Se L è finito, ogni potenza L^n è finita; L^* è infinito non appena L contiene una stringa non vuota.
- La concatenazione di linguaggi non è commutativa e Σ^* è la chiusura riflessiva del linguaggio Σ .

L'asterisco si chiama chiusura riflessiva o stella di Kleene. Il nome "riflessiva" deriva dal fatto che ε appartiene sempre al risultato: due elementi qualsiasi di L^* concatenati danno ancora un elemento di L^* , quindi L^* è chiuso rispetto alla concatenazione.

Espressioni regolari

Le espressioni regolari sono un formalismo algebrico per descrivere linguaggi. Sono utili perché la classe di linguaggi che descrivono, i linguaggi regolari, coincide con quella riconosciuta dagli automi a stati finiti (capitolo 3).

DEFINIZIONE

Espressione regolare. Dato un alfabeto Σ e i simboli $\{+, *, (,), \cdot, \emptyset\}$, una stringa r è un'espressione regolare su Σ se vale una delle condizioni: $r = \emptyset$; $r \in \Sigma$; $r = (s + t)$, $r = (s \cdot t)$ oppure $r = s^*$, dove s e t sono espressioni regolari su Σ .

ESPRESSIONE	LINGUAGGIO $L(R)$ DENOTATO	NOTA
$\emptyset$	Λ	linguaggio vuoto
a , con $a \in \Sigma$	$\{a\}$	singolo carattere
$(s + t)$	$L(s) \cup L(t)$	unione
$(s \cdot t)$ o st	$L(s) \circ L(t)$	concatenazione
s^*	$(L(s))^*$	iterazione

Per omettere le parentesi si assume che la stella abbia precedenza sul prodotto e questo sul $+$. Si può scrivere $(r)^n$ per $rr\dots r$ (n volte) e r^+ per rr^* , ma questi arricchimenti aumentano solo la sinteticità. La stringa vuota non ha un simbolo proprio: il linguaggio $\{\epsilon\}$ è denotato da $\emptyset^*$, poiché $\Lambda^* = \{\epsilon\}$; per comodità si scrive talvolta ϵ .

ESEMPI

ESPRESSIONE	LINGUAGGIO DENOTATO
$(a + b)^*a$	stringhe su $\{a, b\}$ che terminano con a
$(a + b)^*bb(a + b)^*$	stringhe su $\{a, b\}$ che contengono bb come sottostringa
$b^*a(b^*ab^*ab^*)^*$	stringhe su $\{a, b\}$ con un numero dispari di a
$(b^*ab^*ab^*)^* + b^*$	stringhe su $\{a, b\}$ con un numero pari di a
$(a + b)^*b(a + b)$	stringhe su $\{a, b\}$ con penultimo carattere b

DA RICORDARE

Le espressioni regolari in cui compare l'iterazione sono le sole in grado di denotare linguaggi infiniti, ma l'iterazione non rende infinito ogni linguaggio: $\emptyset^*$ denota $\{\epsilon\}$, che è finito. Espressioni diverse possono denotare lo stesso linguaggio: per esempio $(a + b)^*$ e $(a^*b^*)^*$ sono equivalenti.

L'altezza di stella (star height) di un'espressione è il massimo numero di stelle annidate. Per ogni linguaggio regolare esiste un'espressione di altezza di stella minima, ma non sempre l'annidamento della stella si può eliminare.

Cardinalità dei linguaggi

Un problema fondamentale è decidere se una stringa appartenga a un linguaggio. Una semplice considerazione di cardinalità mostra che tale problema non può sempre essere risolto da un algoritmo.

DEFINIZIONE

Ordinamento lessicografico. Dato $\Sigma = \{a_1, \dots, a_n\}$ con $a_1 < a_2 < \dots < a_n$, si pone $x < y$ se vale una delle condizioni: $|x| < |y|$; oppure $|x| = |y|$ ed esiste una stringa z tale che $x = z a_i u$ e $y = z a_j v$, con $i < j$. Prima si confrontano le lunghezze, poi si guarda il primo simbolo in cui le stringhe differiscono.

Con $\Sigma = \{a, b\}$ e $a < b$, le stringhe di Σ^* si enumerano come $\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \dots$. L'ordinamento lessicografico stabilisce quindi una corrispondenza biunivoca tra Σ^* e i numeri naturali: Σ^* è numerabile.

PERCHÉ ESISTONO LINGUAGGI NON RICONOSCIBILI

- L'insieme di tutti i linguaggi su Σ è l'insieme delle parti $P(\Sigma^*)$. Per il teorema di Cantor l'insieme delle parti di un insieme numerabile non è numerabile: ha la cardinalità del continuo $2^{\aleph_0}$.
- Ogni programma, scritto in un qualunque linguaggio di programmazione, è una stringa finita su un alfabeto finito. Enumerando le stringhe in ordine lessicografico e conservando quelle sintatticamente corrette si ottiene un'enumerazione dei programmi: sono quindi al più numerabili.
- Esistono quindi più linguaggi che programmi. Per ogni formalismo con descrizioni finite, come programmi, grammatiche o macchine di Turing, esistono infiniti linguaggi privi di un algoritmo di riconoscimento e privi di una grammatica.

DA RICORDARE

Non esiste una corrispondenza biunivoca tra grammatiche e linguaggi: le grammatiche sono numerabili, i linguaggi no. Questo è il primo risultato limitativo del corso; sarà precisato dalla teoria della calcolabilità (capitolo 5).

Grammatiche e gerarchia di Chomsky

Le grammatiche sono lo strumento generativo per definire linguaggi: partendo da un simbolo iniziale si riscrivono le stringhe secondo regole di produzione. Limitando la forma delle regole si ottiene la gerarchia di Chomsky.

Grammatiche formali

Esistono due approcci complementari per definire un linguaggio. Quello generativo usa una grammatica, che costruisce le stringhe del linguaggio con un insieme prefissato di regole. Quello riconoscitivo usa un automa, che stabilisce se una stringa data appartiene al linguaggio. Le grammatiche presentate qui furono introdotte dal linguista Noam Chomsky per descrivere la sintassi delle lingue naturali; si rivelarono inadatte a quello scopo, ma sono fondamentali per definire la sintassi dei linguaggi di programmazione.

DEFINIZIONE

Grammatica formale. Una grammatica è una quadrupla $G = \langle V_T, V_N, P, S \rangle$ dove V_T è l'alfabeto terminale (insieme finito non vuoto di terminali), V_N è l'alfabeto non terminale (insieme finito non vuoto di variabili o categorie sintattiche), P è un insieme finito di regole di produzione, e $S \in V_N$ è l'assioma, il simbolo iniziale. Le regole sono coppie $\alpha \rightarrow \beta$ con $\alpha \in (V_T \cup V_N)^* V_N (V_T \cup V_N)^*$ e $\beta \in (V_T \cup V_N)^*$: la parte sinistra contiene almeno un non terminale.

Si pone $V = V_T \cup V_N$. Per convenzione le lettere maiuscole latine indicano i non terminali, le prime minuscole i terminali, le ultime minuscole le stringhe di terminali, le lettere greche le stringhe di V^* . Più produzioni con la stessa parte sinistra si abbreviano con una barra verticale: $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$. Una regola del tipo $\alpha \rightarrow \epsilon$ si chiama produzione ϵ .

DERIVAZIONI E LINGUAGGIO GENERATO

- **Derivazione diretta.** $\varphi \Rightarrow \psi$ se esistono $\gamma, \delta \in V^*$ e una regola $\alpha \rightarrow \beta$ di P tali che $\varphi = \gamma\alpha\delta$ e $\psi = \gamma\beta\delta$. Si riscrive una parte della stringa applicando una regola.
- **Derivazione.** Una sequenza $\varphi_1 \Rightarrow \varphi_2 \Rightarrow \dots \Rightarrow \varphi_n$. La relazione $\varphi \Rightarrow^* \psi$, chiusura riflessiva e transitiva di $\Rightarrow$, indica che esiste una derivazione (anche di zero passi) da φ a ψ .
- **Forma di frase.** Una stringa $\varphi \in V^*$ tale che $S \Rightarrow^* \varphi$.
- **Linguaggio generato.** $L(G) = \{x \in V_T^* \mid S \Rightarrow^* x\}$: le forme di frase composte solo da terminali.
- Due grammatiche si dicono **equivalenti** se generano lo stesso linguaggio.

ESEMPIO 1: UN LINGUAGGIO REGOLARE

Sia G con $V_T = \{a, b, c\}$, $V_N = \{S, B, C\}$ e produzioni $S \rightarrow aS \mid B, B \rightarrow bB \mid bC, C \rightarrow cC \mid c$. Il linguaggio generato è $L(G) = \{a^n b^m c^h \mid n \geq 0, m, h \geq 1\}$. Per la stringa $aabcccc$ si applica due volte $S \rightarrow aS$, poi $S \rightarrow B$, poi $B \rightarrow bC$, due volte $C \rightarrow cC$ e infine $C \rightarrow c$:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaB \Rightarrow aabC \Rightarrow aabcC \Rightarrow aabccC \Rightarrow aabcccc$$

Esempio 2: $a^n b^n$

La grammatica con produzioni $S \rightarrow aSb \mid ab$ genera $\{a^n b^n \mid n \geq 1\}$; la grammatica $S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \epsilon$ genera lo stesso linguaggio ed è quindi equivalente alla precedente. Esistono dunque infinite grammatiche per lo stesso linguaggio.

Esempio 3: $a^n b^n c^n$

Il linguaggio $\{a^n b^n c^n \mid n \geq 1\}$ si genera con la grammatica con regole $S \rightarrow aSBC, CB \rightarrow BC, SB \rightarrow bF, FB \rightarrow bF, FC \rightarrow cG, GC \rightarrow cG, G \rightarrow \epsilon$. Per aabbcc:

$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaSBBCC \Rightarrow aabFBCC \Rightarrow aabbFCC \Rightarrow aabbcGC \Rightarrow aabbccG \Rightarrow aabbcc$

Non ogni sequenza di derivazioni porta a una stringa del linguaggio: si può restare con forme di frase che contengono non terminali e a cui non si applica alcuna regola, e ci sono grammatiche che generano il linguaggio vuoto, come $S \rightarrow Ab, A \rightarrow Sa$.

La gerarchia di Chomsky

Imponendo restrizioni sulla forma delle produzioni si ottengono quattro tipi di grammatiche.

TIPO	NOME	FORMA DELLE PRODUZIONI	CONDIZIONI
0	non limitate	$\alpha \rightarrow \beta$	$\alpha \in V^* V_N V^*, \beta \in V^*$
1	contestuali (context sensitive)	$\alpha \rightarrow \beta$	$ \alpha \leq \beta , \beta \in V^+$
2	non contestuali (context free)	$A \rightarrow \beta$	$A \in V_N, \beta \in V^+$
3	regolari (lineari destre)	$A \rightarrow a$ oppure $A \rightarrow aB$	$A, B \in V_N, a \in V_T$

Il nome contestuale deriva dalla definizione storica di Chomsky: le regole $\beta_1 A \beta_2 \rightarrow \beta_1 \gamma \beta_2$ con $\gamma \in V^+$ sostituiscono A solo se si trova nel contesto (β_1, β_2) . Nel tipo 2 la riscrittura riguarda un singolo non terminale, in qualunque contesto compaia; nel tipo 1 si riscrive una stringa e la sostituzione dipende dal contesto. Le grammatiche di tipo 1 non hanno produzioni ϵ e non accorciano le forme di frase; le produzioni ϵ sono ammesse senza restrizioni nel tipo 0 (nei tipi superiori, per generare la stringa vuota, si ammette al più la regola $S \rightarrow \epsilon$). Per i regolari esiste anche la forma lineare sinistra $A \rightarrow a \mid Ba$, equivalente a quella destra.

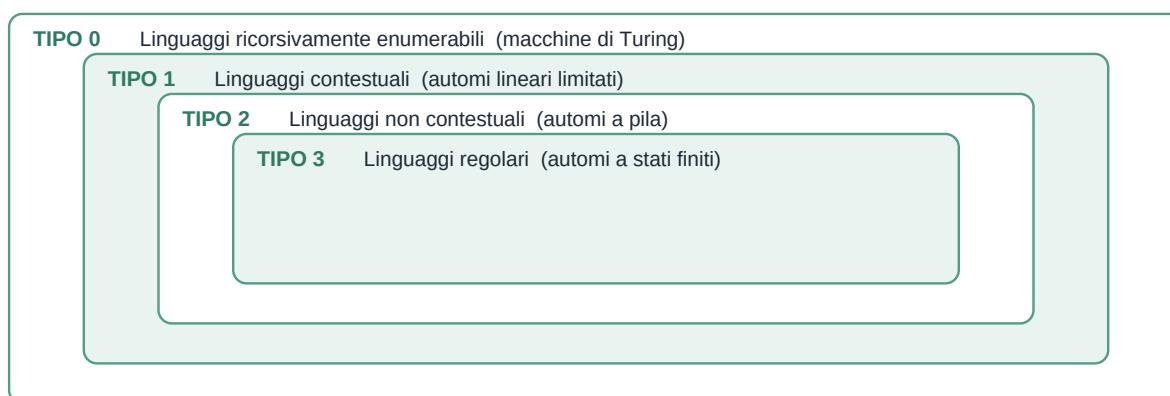


Figura 1. Gerarchia di Chomsky e dispositivi di riconoscimento associati.

Ogni grammatica di tipo $n + 1$ è anche di tipo n , quindi la classe dei linguaggi di tipo n contiene quella di tipo $n + 1$; le inclusioni sono strette. Un linguaggio è strettamente di tipo n se è generato da una grammatica di tipo n e da nessuna di tipo maggiore.

LINGUAGGIO	TIPO STRETTO	MOTIVAZIONE
$\{a^n b \mid n \geq 0\}$	3	grammatica $S \rightarrow aS \mid b$
$\{a^n b^n \mid n \geq 1\}$	2	$S \rightarrow aSb \mid ab$; non è regolare (Pumping Lemma)
$\{a^n b^n c^n \mid n \geq 1\}$	1	grammatica contestuale $S \rightarrow aBSc \mid abc, Ba \rightarrow aB, Bb \rightarrow bb$
$\{ww \mid w \in \{a, b\}^+\}$	1	grammatica contestuale con non terminali A e B che attraversano la stringa

DA RICORDARE

A parità di linguaggio, usare un tipo di grammatica più alto può richiedere meno produzioni: l'insieme delle stringhe formate da permutazioni di a, b, c terminate da # si scrive in modo compatto con una grammatica non contestuale e con molte più regole con una regolare.

Forma normale di Backus

Per definire la sintassi dei linguaggi di programmazione si usano grammatiche non contestuali scritte in una notazione compatta, la forma normale di Backus (BNF, Backus Naur Form), dotata dei seguenti accorgimenti:

1. i non terminali sono nomi di categorie sintattiche tra parentesi acute, per esempio (identificatore);
2. la freccia di produzione diventa $::=$, per non confonderla con simboli terminali dei linguaggi di programmazione;
3. le parentesi graffe $\{...\}$ indicano l'iterazione da zero a infinite volte; con un esponente n, al più n volte: $A ::= bA \mid a$ si riscrive $A ::= \{b\} a$;
4. le parentesi quadre $[...]$ indicano un elemento opzionale: $A ::= xy \mid y$ si riscrive $A ::= [x] y$;
5. le parentesi tonde $(...)$ indicano la fattorizzazione: $A ::= xu \mid xv \mid xy$ si riscrive $A ::= x (u \mid v \mid y)$.

L'identificatore di un linguaggio, fatto da una lettera seguita da lettere o cifre, si descrive così:

```
(identificatore) ::= (alfabetico) {(alfanumerico)}
(alfanumerico)  ::= (alfabetico) | (cifra)
(cifra)         ::= 0 | 1 | ... | 9
```

Un'altra notazione equivalente è quella dei diagrammi sintattici: grafi orientati con un ingresso e un'uscita, i cui archi sono etichettati con terminali (in un cerchio) o non terminali (in un rettangolo); ogni cammino definisce una forma di frase. Anche la sintassi di un linguaggio come WHILE (capitolo 12) è data in BNF.

Accettazione e riconoscimento di linguaggi

Le grammatiche non permettono di impostare in modo algoritmico il problema del riconoscimento: decidere, dati L e x, se $x \in L$. Ogni problema decisionale, per esempio stabilire se un numero è primo, si può formulare come problema di riconoscimento di un linguaggio, per esempio le stringhe binarie che rappresentano numeri primi.

AUTOMI E COMPUTAZIONI

Un automa è un dispositivo astratto che, data una stringa x in ingresso, esegue una computazione. È composto da un dispositivo interno con uno stato, da uno o più nastri (sui quali un nastro contiene inizialmente l'input) letti e scritti da testine, e da una funzione di transizione. Una configurazione descrive istantaneamente lo stato interno, il contenuto dei nastri e la posizione delle testine. A ogni input è associata una configurazione iniziale; alcune configurazioni sono dette di accettazione, le altre di rifiuto.

- La relazione di transizione $c_i \vdash c_j$ esprime che c_j segue da c_i per effetto di un passo; $\vdash^*$ ne è la chiusura riflessiva e transitiva.
- Una computazione è una sequenza di configurazioni $c_0, c_1, \dots$ tali che $c_i \vdash c_{i+1}$. Se è finita e massimale (dopo l'ultima configurazione c_n non esiste alcuna c con $c_n \vdash c$) termina; è di accettazione se termina in una configurazione di accettazione, di rifiuto altrimenti. Può anche non terminare.

DETERMINISMO E NON DETERMINISMO

Un automa deterministico associa a ogni stringa una sola computazione, per cui la funzione di transizione è una funzione sull'insieme delle configurazioni. Un automa non deterministico associa a una stringa un albero di computazioni, perché la funzione di transizione fornisce a una configurazione più successori; il grado di non determinismo è il massimo numero di successori. Una stringa è accettata se almeno una computazione porta a una configurazione di accettazione: c'è dunque un'asimmetria tra accettazione e rifiuto, poiché per rifiutare occorre che nessuna computazione accetti. Il non determinismo è un artificio matematico e non va confuso con il caso o con la probabilità.

LINGUAGGI DECIDIBILI E SEMIDECIDIBILI

DEFINIZIONE

Decidibile. Un linguaggio (o problema decisionale) è decidibile se esiste un algoritmo, in particolare un automa che termina sempre, che per ogni stringa risponde correttamente vero o falso. In questo caso si dice che l'automa riconosce il linguaggio.

DEFINIZIONE

Semidecidibile. Un linguaggio è semidecidibile se esiste un automa che risponde vero per tutte e sole le stringhe del linguaggio; sulle altre può non terminare. In questo caso si dice che l'automa accetta il linguaggio.

CLASSE DI LINGUAGGI	GRAMMATIC A	AUTOMA RICONOSCITORE	DECIDIBILITÀ
Tipo 3, regolari	lineare destra	automa a stati finiti	decidibili, con memoria costante e tempo lineare
Tipo 2, non contestuali	non contestuale	automa a pila non deterministico	decidibili
Tipo 1, contestuali	contestuale	macchina di Turing non deterministica con memoria lineare	decidibili
Tipo 0	non limitata	macchina di Turing	semidecidibili

Le classi di linguaggi dei capitoli successivi sono studiate in questo ordine: i linguaggi regolari e gli automi a stati finiti nel capitolo 3, i cenni sui linguaggi non contestuali nel capitolo 4, le macchine di Turing nel capitolo 5.

Automi a stati finiti e linguaggi regolari

I linguaggi di tipo 3 sono riconosciuti dai dispositivi più semplici, gli automi a stati finiti, e descritti dalle espressioni regolari. Sono la classe adatta agli elementi lessicali di un linguaggio di programmazione, come identificatori e costanti.

Automi a stati finiti deterministici

Un automa a stati finiti legge la stringa in ingresso un carattere alla volta con una testina che si sposta sempre verso destra su un nastro in sola lettura, e aggiorna uno stato interno scelto in un insieme finito. La memoria è quindi finita e limitata: l'automa non può contare quantità arbitrariamente grandi.

DEFINIZIONE

ASFD. Un automa a stati finiti deterministico è una quintupla $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ dove Σ è l'alfabeto di input, Q è un insieme finito e non vuoto di stati, $q_0 \in Q$ è lo stato iniziale, $F \subseteq Q$ è l'insieme degli stati finali e $\delta : Q \times \Sigma \rightarrow Q$ è la funzione di transizione, totale.

La funzione δ , avendo dominio finito, si rappresenta con una tabella di transizione (righe: stati, colonne: caratteri) oppure con un diagramma degli stati, un grafo orientato i cui nodi sono gli stati, con arco da q a q' etichettato a se $\delta(q, a) = q'$; gli stati finali hanno doppio cerchio e lo stato iniziale è marcato da una freccia entrante.

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Tabella 1. Un ASFD con $F = \{q_1\}$: riconosce il linguaggio $\{a^n b \mid n \geq 0\}$; q_2 è uno stato di errore.

CONFIGURAZIONI E LINGUAGGIO RICONOSCIUTO

Una configurazione di A è una coppia (q, x) con $q \in Q$ e $x \in \Sigma^*$, dove x è la parte di input ancora da leggere. È iniziale se $q = q_0$, finale se $x = \varepsilon$, accettante se $x = \varepsilon$ e $q \in F$. La relazione di transizione è definita da: $(q, x) \vdash (q', y)$ se e solo se esiste $a \in \Sigma$ tale che $x = ay$ e $\delta(q, a) = q'$. Con $\vdash^*$ si indica la chiusura riflessiva e transitiva.

$$L(A) = \{ x \in \Sigma^* \mid (q_0, x) \vdash^* (q, \varepsilon) \text{ con } q \in F \}$$

Per la stringa aab dell'automa della Tabella 1: $(q_0, aab) \vdash (q_0, ab) \vdash (q_0, b) \vdash (q_1, \varepsilon)$, con $q_1 \in F$, quindi aab è accettata.

FUNZIONE DI TRANSIZIONE ESTESA

La funzione estesa $\delta^* : Q \times \Sigma^* \rightarrow Q$ associa a uno stato e a una stringa lo stato raggiunto dopo averla letta:

$$\delta^*(q, \varepsilon) = q \quad \delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

$$L(A) = \{ x \in \Sigma^* \mid \delta^*(q_0, x) \in F \}$$

Applicata alla stringa aab , $\delta^*(q_0, aab) = \delta(\delta^*(q_0, aa), b) = \delta(\delta(\delta^*(q_0, a), a), b) = \delta(\delta(q_0, a), b) = \delta(q_0, b) = q_1$. Gli ASFD terminano sempre, perché a ogni passo consumano un carattere e la funzione è totale, quindi ogni ASFD

ricosce un linguaggio.

DA RICORDARE

Se la funzione di transizione fosse parziale, si ottiene un automa equivalente aggiungendo uno stato pozzo q_e non finale con $\delta(q_e, a) = q_e$ per ogni a , verso cui puntano tutte le transizioni mancanti. Per questo si può supporre senza perdita di generalità che la funzione sia totale.

ESEMPI

Il linguaggio delle stringhe su $\{a, b\}$ con un numero pari di a si riconosce con due stati che alternano la parità delle a ; lo stato iniziale q_0 è anche finale, perciò anche ϵ è accettata.



Figura 1. ASFD per le stringhe con un numero pari di a : q_0 è iniziale e finale.

Per le stringhe con un numero pari di a e un numero pari di b servono quattro stati, che ricordano la parità di entrambi i simboli, con $F = \{q_0\}$.

STATO	SIGNIFICATO	a	b
q_0	a pari, b pari	q_1	q_2
q_1	a dispari, b pari	q_0	q_3
q_2	a pari, b dispari	q_3	q_0
q_3	a dispari, b dispari	q_2	q_1

Se invece $F = \{q_0, q_1, q_2\}$ il linguaggio è formato dalle stringhe con un numero pari di a oppure un numero pari di b . Esistono infiniti automi che riconoscono lo stesso linguaggio.

Automi a stati finiti non deterministici

Un automa non deterministico può avere più successori per la stessa coppia stato e carattere. Le computazioni formano un albero, e la stringa è accettata se almeno una computazione termina in una configurazione di accettazione.

DEFINIZIONE

ASFND. Un automa a stati finiti non deterministico è una quintupla $A_N = \langle \Sigma, Q, \delta_N, q_0, F \rangle$ in cui $\delta_N : Q \times \Sigma \rightarrow P(Q)$ associa a ogni coppia un sottoinsieme di Q , eventualmente vuoto. La funzione è quindi parziale nel senso deterministico: $\delta_N(q, a) = \emptyset$ equivale a transizione non definita.

La funzione estesa è $\delta_N^*(q, \epsilon) = \{q\}$ e $\delta_N^*(q, xa) = \text{unione dei } \delta_N(p, a) \text{ per } p \in \delta_N^*(q, x)$. Una stringa x è accettata se $\delta_N^*(q_0, x) \cap F \neq \emptyset$. Equivalentemente, si può vedere l'automata eseguire una sola computazione che passa da insiemi di stati a insiemi di stati: $\{q_0\}, x \vdash^* (T, \epsilon)$ con $T \cap F \neq \emptyset$.

δ_N	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2, q_3\}$	$\{q_3\}$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$

Tabella 2. ASFND con $F = \{q_3\}$: riconosce le stringhe su $\{a, b\}$ che terminano con bb , ba o baa .

Per l'input bba l'insieme degli stati attuali evolve come $(\{q_0\}, bba) \vdash (\{q_0, q_1\}, ba) \vdash (\{q_0, q_1, q_3\}, a) \vdash (\{q_0, q_2, q_3\}, \epsilon)$: contiene $q_3 \in F$, quindi bba è accettata. Una variante, gli ASFND con ϵ transizioni, ammette passaggi di stato senza leggere caratteri; si dimostra con la ϵ chiusura che il modello resta equivalente. Un ASFND può sempre essere trasformato in uno equivalente con un solo stato finale.

Equivalenza tra determinismo e non determinismo

Nonostante il non determinismo accresca il potere di altri modelli, per gli automi a stati finiti non lo fa.

DEFINIZIONE

Teorema. Dato un ASFD che riconosce L esiste un ASFND che riconosce L ; viceversa, dato un ASFND che riconosce L esiste un ASFD che riconosce L .

La prima parte è banale: si pone $\delta_N(q, a) = \{\delta(q, a)\}$. Per la seconda si usa la costruzione per sottoinsiemi. Dato $A_N = \langle \Sigma, Q, \delta_N, q_0, F \rangle$ si definisce $A' = \langle \Sigma, Q', \delta', q_0', F' \rangle$ con:

- $Q' = P(Q)$: gli stati sono i sottoinsiemi di Q , scritti $[q_{i_1}, \dots, q_{i_k}]$, e $|Q'| = 2^{|Q|}$;
- $q_0' = [q_0]$;
- $F' =$ i sottoinsiemi che contengono almeno uno stato di F ;
- $\delta'([q_{i_1}, \dots, q_{i_k}], a) = [q_{j_1}, \dots, q_{j_h}]$ dove $\{q_{j_1}, \dots, q_{j_h}\} = \delta_N(q_{i_1}, a) \cup \dots \cup \delta_N(q_{i_k}, a)$, con $\delta'([], a) = []$.

La correttezza si prova per induzione sulla lunghezza di x , mostrando $\delta^*([q_0], x) = [q_{j_1}, \dots, q_{j_h}]$ se e solo se $\delta_N^*(q_0, x) = \{q_{j_1}, \dots, q_{j_h}\}$. Nel passo base entrambe le funzioni danno $\{q_0\}$; nel passo induttivo, con $x = x'a$, $\delta_N^*(q_0, x'a)$ è l'unione dei $\delta_N(p, a)$ per $p \in \delta_N^*(q_0, x')$, che per costruzione è proprio lo stato raggiunto da δ' .

ESEMPIO

L'ASFND con $Q = \{q_0, q_1\}$, $\delta_N(q_0, a) = \{q_0\}$, $\delta_N(q_0, b) = \{q_0, q_1\}$, $F = \{q_1\}$ riconosce le stringhe che terminano con b . L'ASFD equivalente ha, tra i quattro sottoinsiemi, solo due stati raggiungibili da $[q_0]$:

STATO	a	b
$[q_0]$	$[q_0]$	$[q_0, q_1]$
$[q_0, q_1]$	$[q_0]$	$[q_0, q_1]$

con $F' = \{[q_0, q_1]\}$. In pratica conviene costruire δ' a partire da $[q_0]$, generando solo gli stati raggiungibili.

DA RICORDARE

Il non determinismo rende gli automi più concisi: l'espressione $(a + b)^*b(a + b)^k$ è riconosciuta da un ASFND con $k + 2$ stati, ma ogni ASFD equivalente richiede un numero di stati dell'ordine di 2^k . Il costo dell'eliminazione del non determinismo può dunque essere esponenziale. Dall'equivalenza segue che i linguaggi regolari sono chiusi rispetto all'inversione: se L è riconosciuto da un ASF, esiste un ASF per le stringhe riflesse di L .

Linguaggi regolari, grammatiche ed espressioni regolari

La classe dei linguaggi riconosciuti da ASFD, o ASFND, si dice classe dei linguaggi regolari. Coincide con quella dei linguaggi generati dalle grammatiche di tipo 3 e con quella dei linguaggi denotati da espressioni regolari.

GRAMMATICHE REGOLARI E AUTOMI

Data una grammatica regolare $G = \langle V_T, V_N, P, S \rangle$, si costruisce un ASFND con $\Sigma = V_T$, un stato q_i per ogni $i \in V_N$ più uno stato finale q_F , stato iniziale q_S , e transizioni $\delta_N(q_B, a) = \{q_C \mid B \rightarrow aC \in P\} \cup \{q_F\}$ se $B \rightarrow a \in P$ (senza q_F altrimenti). L'insieme degli stati finali è $\{q_F\}$, più q_S se $S \rightarrow \varepsilon \in P$. Viceversa, da un ASFD $A = \langle \Sigma, Q, \delta, q_0, F \rangle$ si ottiene la grammatica con un non terminale A_i per ogni stato q_i , assioma A_0 , produzioni $A_i \rightarrow aA_j$ per ogni $\delta(q_i, a) = q_j$, e $A_i \rightarrow a$ se in più $q_i \in F$. Se $q_0 \in F$ si aggiunge un nuovo assioma A_0' con $A_0' \rightarrow \varepsilon$ e con le stesse produzioni di A_0 .

ESPRESSIONI REGOLARI E AUTOMI

- Ogni linguaggio denotato da un'espressione regolare è regolare: $\emptyset$ e $\{a\}$ hanno automi banali, e la classe è chiusa per unione, concatenazione e iterazione (si veda la sezione seguente).
- Ogni linguaggio regolare è denotato da un'espressione regolare: per ogni ASFD si costruisce un'espressione eliminando via via gli stati e trasformando le etichette degli archi in espressioni.
- Il linguaggio riconosciuto da un automa si può quindi descrivere in tre modi equivalenti: automa, grammatica di tipo 3, espressione regolare.

I linguaggi regolari sono decidibili e per essi sono decidibili anche problemi come stabilire se $L(A)$ è vuoto, se è infinito, se due automi riconoscono lo stesso linguaggio. Esistono infiniti automi che riconoscono lo stesso linguaggio; tra essi ce n'è uno con il minimo numero di stati.

Proprietà di chiusura

La classe dei linguaggi regolari è chiusa rispetto alle operazioni elencate: a partire dagli automi che riconoscono i linguaggi dati si costruisce un automa per il linguaggio risultante.

OPERAZIONE	COSTRUZIONE DELL'AUTOMA
Unione $L_1 \cup L_2$	nuovo stato iniziale che eredita le transizioni uscenti da entrambi gli stati iniziali; $F = F_1 \cup F_2$, con il nuovo iniziale finale se uno dei due automi accetta ε
Complemento	si completa l'ASFD (stato pozzo) e si scambiano stati finali e non finali: $F' = Q \text{ meno } F$
Intersezione	per De Morgan, $L_1 \cap L_2$ è il complemento dell'unione dei complementi; oppure con l'automa prodotto, con stati (q_1, q_2)
Concatenazione $L_1 \circ L_2$	gli stati finali di A_1 ereditano le transizioni dello stato iniziale di A_2 ; finali: F_2 (più F_1 se $\varepsilon \in L_2$)
Iterazione L^*	gli stati finali ereditano le transizioni dello stato iniziale; si aggiunge uno stato iniziale e finale per ε
Inversione	si invertono gli archi e si scambiano i ruoli di stato iniziale e stati finali, ottenendo un ASFND, poi eventualmente determinizzato

Il Pumping Lemma

I linguaggi regolari hanno una struttura ripetitiva: una stringa abbastanza lunga contiene un tratto che può essere ripetuto quanto si vuole restando nel linguaggio. Il Pumping Lemma esprime questa proprietà e si usa per dimostrare che certi linguaggi non sono regolari.

DEFINIZIONE

Pumping Lemma. Per ogni linguaggio regolare L esiste una costante n tale che, se $z \in L$ e $|z| \geq n$, allora z si può scrivere $z = uvw$ con $|uv| \leq n$, $|v| \geq 1$ e $uv^i w \in L$ per ogni $i \geq 0$.

DIMOSTRAZIONE

Sia $A = (\Sigma, Q, \delta, q_0, F)$ un ASFD per L e sia $n = |Q|$. Sia $z \in L$ con $|z| = k \geq n$; allora $\delta^*(q_0, z) \in F$. Sia $q_{i_0}, q_{i_1}, \dots, q_{i_k}$ la sequenza degli stati attraversati, con $q_{i_h} = \delta^*(q_0, z_h)$ dove z_h è il prefisso di z di lunghezza h . I primi $n + 1$ elementi sono più degli n stati di A : per il principio dei cassetti esistono due indici $r < s \leq n$ con $q_{i_r} = q_{i_s}$. Posto $u = z_r$, $uv = z_s$, $uvw = z$, si ha $|uv| = s \leq n$ e $|v| = s - r \geq 1$. Per induzione su i si prova che $uv^i w \in L$:

- $i = 0$: $\delta^*(q_0, uw) = \delta^*(\delta^*(q_0, z_r), w) = \delta^*(\delta^*(q_0, z_s), w) = \delta^*(q_0, z) \in F$;
- $i > 0$: $\delta^*(q_0, uv^i w) = \delta^*(\delta^*(q_0, uv), v^{i-1}w) = \delta^*(\delta^*(q_0, u), v^{i-1}w) = \delta^*(q_0, uv^{i-1}w) \in F$ per ipotesi induttiva.

Il Pumping Lemma mostra un limite degli automi finiti: non sanno contare. Fornisce una condizione necessaria e non sufficiente: se L è regolare, soddisfa il lemma; se non lo soddisfa, non è regolare. Ci sono linguaggi non regolari che lo soddisfano, perciò non si può usare il lemma per provare che un linguaggio è regolare.

SCHEMA PER DIMOSTRARE CHE UN LINGUAGGIO NON È REGOLARE

Il lemma, negato, afferma che L non è regolare se:

per ogni n esiste $z \in L$ con $|z| \geq n$ tale che, per ogni scomposizione $z = uvw$ con $|uv| \leq n$ e $v \neq \epsilon$, esiste $i \geq 0$ per cui $uv^i w \notin L$

Esempio: $a^n b^n$ non è regolare

Per ogni n si sceglie $z = a^n b^n$, di lunghezza $2n \geq n$. Ogni scomposizione $z = uvw$ con $|uv| \leq n$ ha v formata di sole a , cioè $v = a^h$ con $h \geq 1$. Allora $uv^i w = a^{n+(i-1)h} b^n$, che appartiene al linguaggio se e solo se $(i-1)h = 0$, cioè $i = 1$. L'insieme degli i per cui $uv^i w \in L$ è dunque $\{0\} \cup \{i \mid i \geq 2\}$. Già con $i = 0$ si toglie v e si ottiene meno a che b : la stringa esce dal linguaggio, e L non è regolare.

LINGUAGGI REGOLARI E NON REGOLARI: ESEMPI

LINGUAGGIO	REGOLARE?	MOTIVO
$\{a^n b^m \mid n, m \geq 0\}, \{a^h b^k \mid h, k \geq 1\}$	sì	espressioni a^*b^* , aa^*bb^* ; non c'è relazione tra le lunghezze
$\{a^n b^n \mid 0 \leq n \leq 11\}$	sì	è finito
Ogni linguaggio finito	sì	unione finita di linguaggi $\{x\}$, ciascuno regolare
$\{a^n b^n \mid n \geq 0\}$	no	Pumping Lemma: l'automa dovrebbe contare le a
$\{w w^R \mid w \in \{a, b\}^*\}$	no	Pumping Lemma: serve ricordare w per confrontarla con la riflessa
$\{abab, baba\}$	sì	è finito
Concatenazione e unione di due linguaggi regolari	sì	chiusura della classe, per esempio $\{a^m b^n \mid m, n \geq 1\}$ ◦ $\{abab, baba\}$

DA RICORDARE

Essere finito è una condizione sufficiente per la regolarità, non necessaria: a^*b^* è infinito e regolare. D'altra parte non ogni linguaggio infinito è regolare, e non ogni linguaggio che soddisfa il Pumping Lemma lo è: il lemma serve solo a escludere la regolarità.

Cenni sui linguaggi non contestuali

Le grammatiche di tipo 2 descrivono strutture annidate, come le parentesi bilanciate o le espressioni aritmetiche. Sono lo strumento standard per la sintassi dei linguaggi di programmazione, ma presentano il problema dell'ambiguità.

Grammatiche e alberi di derivazione

Una grammatica non contestuale (context free) ha produzioni $A \rightarrow \beta$ con $A \in V_N$ e $\beta \in V^+$: ogni non terminale si riscrive indipendentemente dal contesto. Il linguaggio $\{a^n b^n \mid n \geq 1\}$, non regolare, è non contestuale: lo genera $S \rightarrow aSb \mid ab$. Anche il linguaggio delle parentesi ben bilanciate lo è, con $S \rightarrow () \mid SS \mid (S)$, e così la sintassi delle espressioni aritmetiche in una variabile i , con $E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow i \mid (E)$.

DEFINIZIONE

Albero di derivazione. Data una grammatica non contestuale G , l'albero di derivazione di una stringa $x \in L(G)$ ha come radice l'assioma S , come nodi interni non terminali, come figli di un nodo A i simboli della parte destra β di una produzione $A \rightarrow \beta$ usata nella derivazione, e come foglie, lette da sinistra a destra, i terminali di x .

Un albero astrae dall'ordine in cui si applicano le regole: più derivazioni diverse possono avere lo stesso albero. Per renderle confrontabili si usa una derivazione particolare.

DEFINIZIONE

Derivazione sinistra. Una derivazione $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n$ in una grammatica non contestuale è sinistra (leftmost) se a ogni passo si riscrive il non terminale più a sinistra, cioè se ogni passo è del tipo $\alpha_i = xA\beta \Rightarrow x\gamma\beta = \alpha_{i+1}$, con $A \rightarrow \gamma \in P$ e $x \in V_T^*$.

A ogni albero di derivazione è associata una e una sola derivazione sinistra; quindi, se si ignora l'ordine di applicazione delle regole, due derivazioni sono equivalenti quando hanno lo stesso albero. Per esempio, con $S \rightarrow aSb \mid ab$ la stringa $aabb$ ha la derivazione sinistra $S \Rightarrow aSb \Rightarrow aabb$, il cui albero ha radice S con figli a , S , b , e l' S interna con figli a , b .

Ambiguità

DEFINIZIONE

Grammatica ambigua. Una grammatica non contestuale G è ambigua se esiste una stringa $x \in L(G)$ con due diversi alberi di derivazione, o equivalentemente con due diverse derivazioni sinistre.

Si consideri G con $E \rightarrow E + E \mid E * E \mid a$. La stringa $a + a * a$ ha due alberi, che corrispondono a due interpretazioni dell'espressione:

DERIVAZIONE SINISTRA	STRUTTURA DELL'ALBERO
$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$	$a + (a * a)$
$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$	$(a + a) * a$

Cambia la precedenza degli operatori: la prima lettura dà il risultato di $3 + (2 * 5)$, la seconda di $(3 + 2) * 5$. Una grammatica ambigua non va bene per i compilatori, perché il significato di un programma dipende dall'albero.

ELIMINARE L'AMBIGUITÀ

- **Parentesi.** Si impone la parentesizzazione completa, $E \rightarrow (E + E) \mid (E * E) \mid a$: le due stringhe diventano $(a + (a * a))$ e $((a + a) * a)$. È però gravoso per chi scrive.
- **Precedenza.** Si codifica la precedenza nella grammatica, come in $E \rightarrow E + T \mid T$, $T \rightarrow T * F \mid F$, $F \rightarrow (E) \mid a$, che è non ambigua: il $*$ lega più strettamente del $+$ perché sta più in basso nell'albero.
- Non esiste un algoritmo che, data una grammatica non contestuale, stabilisca se è ambigua: il problema è indecidibile (si riduce al problema delle corrispondenze di Post).

AMBIGUITÀ INERENTE

Un linguaggio non contestuale è inerentemente ambiguo se tutte le grammatiche che lo generano sono ambigue. Anche questa proprietà è indecidibile. Un esempio è $L = \{a^n b^n c^m \mid n, m \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}$: con $S \rightarrow AB \mid CD$, $A \rightarrow aAb \mid ab$, $B \rightarrow cB \mid c$, $C \rightarrow aC \mid a$, $D \rightarrow bDc \mid bc$ la stringa abc ha due alberi, uno che passa per AB e uno per CD , e qualunque altra grammatica deve generare in due modi le stringhe $a^n b^n c^n$.

DA RICORDARE

Ambiguità di una grammatica e ambiguità inerente di un linguaggio sono nozioni diverse. Una grammatica ambigua può spesso essere sostituita da una equivalente non ambigua, come per le espressioni aritmetiche; i linguaggi inerentemente ambigui non ammettono alcuna grammatica non ambigua.

Forme normali

Ogni grammatica non contestuale (senza ϵ produzioni) può essere trasformata in una equivalente di forma particolare. Le due forme normali più usate riducono la varietà delle regole e semplificano l'analisi sintattica.

FORMA NORMALE	PRODUZIONI	CONDIZIONI
Chomsky (CNF)	$A \rightarrow XY$ oppure $A \rightarrow a$	$A, X, Y \in V_N$, $a \in V_T$
Greibach (GNF)	$A \rightarrow a\beta$	$A \in V_N$, $a \in V_T$, $\beta \in V_N^*$

Tutte le grammatiche non contestuali, senza ϵ produzioni, si possono rappresentare in forma normale di Chomsky. Verificare se una stringa appartiene a un linguaggio riconosciuto da un automa a stati finiti è semplice: basta seguire gli stati. Per un linguaggio non contestuale la verifica è più semplice se la grammatica è in forma normale di Chomsky, perché ogni derivazione di una stringa di lunghezza n ha esattamente $2n$ meno 1 passi. Nella forma di Greibach ogni passo produce un terminale, quindi una stringa di lunghezza n richiede esattamente n passi: la stessa idea degli automi a pila, che riconoscono i linguaggi non contestuali con una memoria a pila e non determinismo.

DEFINIZIONE

Confronto tra le classi. I linguaggi regolari sono un sottoinsieme proprio dei non contestuali: $a^n b^n$ è non contestuale e non regolare. I non contestuali sono decidibili: l'appartenenza di una stringa si stabilisce con un algoritmo; il problema dell'ambiguità e quello dell'ambiguità inerente non lo sono.

Modelli computazionali e macchine di Turing

Computare significa trasformare un'informazione da una forma implicita a una forma esplicita. Un modello computazionale rende preciso questo concetto: la macchina di Turing è il modello di riferimento per la calcolabilità.

Che cosa significa computare

Quando si trasforma l'espressione $(3 + 4) * 2$ in 14 si esegue una computazione: entrambe le espressioni denotano lo stesso numero, ma il valore è più esplicito in 14. La definizione è però troppo vaga. Un modello computazionale è una formalizzazione matematica di computazione e delle nozioni collegate: dati, passo elementare di computazione, e modo di organizzare i passi in un procedimento effettivo, cioè un algoritmo.

Sono stati proposti molti modelli, anche molto diversi: le macchine di Turing, le macchine a registri illimitati (URM), i sistemi di Post, il λ calcolo (capitolo 6), le funzioni parziali ricorsive, la programmazione logica. I linguaggi di programmazione sono ispirati a uno o più modelli: i linguaggi imperativi alle macchine di Turing e alle URM, quelli funzionali al λ calcolo, quelli logici alla ricerca di deduzioni in un frammento della logica del primo ordine.

Tutti i modelli hanno caratteristiche comuni: un numero finito di costrutti, computazione discreta passo dopo passo, nessun limite a priori su tempo e memoria ma uso finito di entrambi in ogni computazione finita, e ogni passo che coinvolge una quantità finita di dati.

L'analisi di Turing

Nel 1936 Turing analizzò una persona che calcola con un procedimento meccanico e ne trasse le componenti di una macchina: un nastro suddiviso in celle, ciascuna contenente un simbolo di un alfabeto finito; una testina che legge e scrive; un controllo finito, che memorizza uno stato interno, l'analogo dello stato mentale. L'azione compiuta dipende solo dallo stato corrente e dal simbolo letto, e consiste nel cambiare il simbolo sotto la testina, nel cambiare stato e nello spostare la testina di una cella. Turing formulò la sua analisi prima dell'invenzione dei calcolatori moderni.

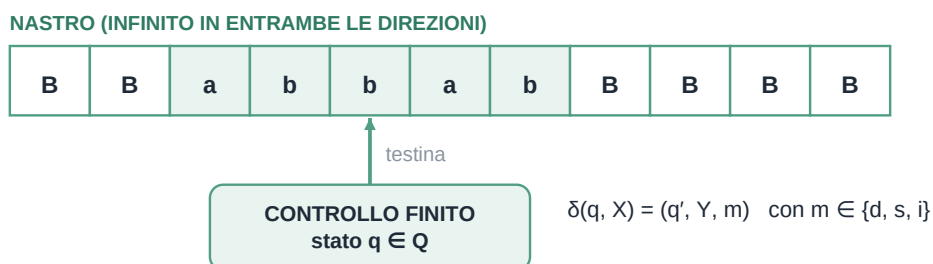


Figura 1. Macchina di Turing: nastro, testina e controllo finito con stato interno.

Nella formulazione per quintuple una regola (q_1, a, b, M, q_2) si legge: nello stato q_1 , leggendo a , passa allo stato q_2 , scrivi b e sposta la testina a sinistra, a destra o resta ferma a seconda che M sia S, D o I. Il modello contiene in forma essenziale i tratti della programmazione imperativa: celle come locazioni, scrittura come assegnamento, nastro come memoria sequenziale.

Macchine di Turing deterministiche

DEFINIZIONE

MTD. Una macchina di Turing deterministica è una sestupla $M = (\Gamma, B, Q, q_0, F, \delta)$ dove Γ è l'alfabeto dei simboli di nastro, $B \notin \Gamma$ è il carattere speciale blank, Q è un insieme finito e non vuoto di stati, $q_0 \in Q$ è lo stato iniziale, $F \subseteq Q$ è l'insieme degli stati finali e δ è la funzione parziale di transizione: $\delta : (Q \text{ meno } F) \times (\Gamma \cup \{B\}) \rightarrow Q \times (\Gamma \cup \{B\}) \times \{d, s, i\}$, dove d, s, i indicano spostamento a destra, a sinistra, nessuno spostamento.

Si pone $\Gamma_B = \Gamma \cup \{B\}$. L'alfabeto di input Σ è un sottoinsieme di Γ . Una MTD può funzionare da riconoscitore, per stabilire se una stringa appartiene a un linguaggio, oppure da trasduttore, per calcolare una funzione. La funzione di transizione si rappresenta con una matrice (righe: stati, colonne: simboli letti, elementi: triple nuovo stato, simbolo scritto, spostamento) o con un grafo i cui archi sono etichettati con la tripla simbolo letto, simbolo scritto, spostamento.

CONFIGURAZIONI

Una configurazione istantanea è l'insieme delle informazioni costituito dal contenuto del nastro, dalla posizione della testina e dallo stato. Poiché il nastro ha una parte finita non blank, la configurazione si rappresenta con una stringa xqy :

DEFINIZIONE

Configurazione. Una configurazione è una stringa $c = xqy$ con $x \in \Gamma_B^* \cup \{\epsilon\}$, $q \in Q$ e $y \in \Gamma_B^* \cup \{B\}$. Il contenuto non blank del nastro è xy , lo stato è q e la testina è sul primo carattere di y . Quindi x non inizia con un blank, y non termina con un blank, a meno che sia esattamente B . Se $x = \epsilon$, a sinistra della testina ci sono solo blank; se $y = B$, la cella corrente e quelle a destra sono blank.

- Configurazione **iniziale**: $x = \epsilon$, $q = q_0$, $y \in \Gamma^+ \cup \{B\}$: con input w si ha $q_0 w$ (o $q_0 B$ se $w = \epsilon$).
- Configurazione **finale**: $q \in F$. Da una configurazione finale non si possono fare ulteriori passi, perché δ non è definita sugli stati finali.

PASSO DI COMPUTAZIONE

Se $\delta(q, a) = (q', a', m)$ la configurazione successiva c' di c , indicata con $c \vdash c'$, si ottiene così:

MOSSA	CONFIGURAZIONE C	CONFIGURAZIONE C'
Destra	$x q a y$	$x a' q' y$
Destra, a fine nastro	$x q a$	$x a' q' B$
Sinistra	$x a q b y$	$x q' a b' y$
Sinistra, a inizio nastro	$q b y$	$q' B b' y$
Ferma	$x q a y$	$x q' a' y$

Le regole sono regole di riscrittura del nastro. Per esempio, $\delta(q_1, a) = (q_2, b, s)$ applicata alla configurazione $abbBb$ $q_1 abb$ porta a $abbB q_2 b bbb$. Con $\vdash^*$ si indica la chiusura riflessiva e transitiva, cioè zero o più passi, e con $\vdash^+$ uno o più passi.

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

La macchina con $\Gamma = \{0, 1, x, y\}$, stati $q_0, \dots, q_4$, $F = \{q_4\}$ e la seguente funzione di transizione riconosce il linguaggio. Il principio è marcare con x uno zero, con y l'uno corrispondente, tornare indietro e ripetere.

STATO	0	1	x	y	B
q_0	(q_1, x, d)			(q_3, y, d)	
q_1	$(q_1, 0, d)$	(q_2, y, s)		(q_1, y, d)	
q_2	$(q_2, 0, s)$		(q_0, x, d)	(q_2, y, s)	
q_3				(q_3, y, d)	(q_4, B, d)

Tabella 1. Le celle vuote indicano δ non definita: la macchina si ferma in uno stato non finale e rifiuta.

Per l'input 0011 la computazione, scritta come sequenza di configurazioni, è:

$$q_0 0011 \vdash xq_1 011 \vdash x0q_1 11 \vdash xq_2 0y1 \vdash q_2 x0y1 \vdash xq_0 0y1 \vdash xxq_1 y1 \vdash xxyq_1 1 \vdash xxq_2 yy \vdash xq_2 xyy \\ \vdash xxq_0 yy \vdash xxyq_3 y \vdash xyyyq_3 B \vdash xxyyBq_4 B$$

Si arriva a uno stato finale: 0011 è accettata. Con 001, 0101 o 011 la macchina si ferma in uno stato non finale perché δ non è definita, e rifiuta; la stringa vuota è rifiutata dallo stato q_0 , che sul blank non ha transizione.

Computazioni, accettazione e decidibilità

Sia M una MTD e $\Sigma \subseteq \Gamma$ l'alfabeto di input. Una stringa $x \in \Sigma^*$ è **accettata** (rifiutata) da M se esiste una computazione di accettazione (di rifiuto) con configurazione iniziale $q_0 x$, cioè una computazione massimale che termina in uno stato finale (non finale). A differenza degli automi finiti, per una macchina di Turing una computazione può non terminare mai.

DEFINIZIONE

Riconoscere e accettare. M riconosce (decide) $L \subseteq \Sigma^*$ se per ogni $x \in \Sigma^*$ esiste una computazione massimale $q_0 x \vdash^* wqz$ che termina in uno stato q tale che $q \in F$ se e solo se $x \in L$. M accetta L se $L = \{x \in \Sigma^* \mid q_0 x \vdash^* wqz \text{ con } q \in F\}$: sulle stringhe fuori da L la macchina può rifiutare o non terminare.

Una MTD decide un linguaggio solo se si ferma su ogni input. Il caso di non terminazione non è osservabile in tempo finito: non si può mai sapere, dopo un numero finito di passi, se la computazione terminerà. Il problema della terminazione, cioè stabilire date M e x se M termina su x , non è risolubile: non esiste alcuna macchina di Turing che lo decida.

DEFINIZIONE

Decidibile secondo Turing. Un linguaggio è T decidibile se esiste una macchina di Turing che lo riconosce; è T semidecidibile se esiste una macchina di Turing che lo accetta. Ogni linguaggio T decidibile è T semidecidibile; il viceversa è falso.

La classe dei linguaggi riconosciuti dalle MT si chiama classe dei linguaggi **ricorsivi**; quella dei linguaggi accettati si chiama classe dei linguaggi **ricorsivamente enumerabili** e coincide con quella generata dalle grammatiche di tipo 0. I ricorsivi sono contenuti propriamente nei ricorsivamente enumerabili. Tutti i linguaggi regolari, non contestuali e contestuali sono decidibili.

CALCOLO DI FUNZIONI

Un trasduttore M calcola una funzione $f : \Sigma^* \rightarrow \Sigma^*$ se per ogni $x \in \Gamma^*$: se $x \in \Sigma^*$ e $f(x) = y$, allora $q_0 x \vdash^* x B q y$ con $q \in F$; altrimenti, se $x \notin \Sigma^*$ o $f(x)$ non è definita, non esiste una computazione massimale oppure essa termina in uno stato non finale. Una funzione è T calcolabile se esiste una macchina che la calcola; non è necessariamente totale. Funzioni su domini qualunque si riconducono a funzioni su stringhe scegliendo una codifica. La definizione di T calcolabilità si riferisce a una macchina deterministica.

Varianti della macchina di Turing

- **Più nastri.** Una macchina a k nastri ha una testina per nastro e funzione $\delta^{(k)} : (Q \text{ meno } F) \times \Gamma_{B1} \times \dots \times \Gamma_{Bk} \rightarrow Q \times \Gamma_{B1} \times \dots \times \Gamma_{Bk} \times \{d, s, i\}^k$. Non aggiunge potere: ogni macchina multinastro è simulabile da una a un nastro.
- **Non deterministica.** Ha $\delta_N : (Q \text{ meno } F) \times \Gamma_B \rightarrow P(Q \times \Gamma_B \times \{s, d, i\})$; ma MTD e MTND sono equivalenti dal punto di vista di ciò che calcolano.

Macchina universale e tesi di Church e Turing

Una macchina di Turing si può descrivere con una stringa: basta scrivere in sequenza le quintuple della funzione di transizione. Turing mostrò che esiste una **macchina di Turing universale**, che riceve in ingresso la descrizione di una macchina M e un input x e ne simula la computazione. Prima, per ogni problema occorre una macchina diversa, programmata dentro il suo controllo finito; la macchina universale è programmabile. Nasce così il dualismo tra hardware e software: ciò che è realizzato fisicamente e ciò che è descrizione formale delle operazioni; ciò che si può fare in software si può realizzare in hardware e viceversa. La macchina universale è alla base dell'architettura di Von Neumann, con la CPU che interpreta un programma memorizzato nella RAM.

DA RICORDARE

Tesi di Church e Turing. Ogni funzione intuitivamente calcolabile è calcolabile con una macchina di Turing. Non si può dimostrare, perché identifica una nozione intuitiva con una matematica; la sostiene l'equivalenza di tutti i modelli proposti (macchine di Turing, funzioni ricorsive, λ calcolo).

Codifica di stringhe e numeri

Una funzione di codifica per un insieme X su un alfabeto A è una funzione iniettiva $\text{cod} : X \rightarrow A^+$, con la relativa funzione di decodifica $\text{decod} : A^+ \rightarrow X \cup \{\text{errore}\}$ tale che $\text{decod}(\text{cod}(y)) = y$ per ogni $y \in X$.

La differenza tra macchine di Turing e funzioni parziali ricorsive (dette μ ricorsive) è che le prime manipolano stringhe, le seconde numeri naturali. Dato che i naturali si codificano con stringhe su $\{0, 1\}$, le macchine di Turing lavorano anche sui numeri; viceversa, esiste un isomorfismo tra A^+ e $\mathbb{N}$, quindi la maggiore ampiezza delle macchine è solo apparente. Basta anche una funzione iniettiva da A^+ a $\mathbb{N}$, costruita con l'isomorfismo di Cantor tra $\mathbb{N} \times \mathbb{N}$ e $\mathbb{N}$: si riempie la tabella delle coppie per diagonal successive e a (n, m) si associa il numero nella casella di riga n e colonna m , cioè $(n + m)(n + m + 1)/2 + n$.

Si numerano le diagonal: 0; poi 1, 2; poi 3, 4, 5; poi 6, 7, 8, 9. In formule la casella di riga n e colonna m contiene $(n + m)(n + m + 1)/2 + n$.

Per esempio (3, 1) corrisponde a 13 e (1, 5) a 22. Dopo aver rappresentato i caratteri con numeri naturali, per esempio a, b, g, d con 0, 1, 2, 3, si pone, indicando con $(\cdot, \cdot)'$ l'isomorfismo di Cantor:

$$F(c_1 c_2 \dots c_n) = ((\dots ((c_1, c_2)', c_3)' \dots, c_n)', n)'$$

La stringa agaa, con codici 0, 2, 0, 0, dà $(0, 2)' = 3$, poi $(3, 0)' = 9$, poi $(9, 0)' = 54$ e infine $(54, 4)' = 1765$. La lunghezza n in coda rende F invertibile: senza di essa a e aa, con $a = 0$, avrebbero la stessa immagine.

Programmazione funzionale e λ calcolo

Il λ calcolo è il modello computazionale su cui si fondano i linguaggi funzionali. Con sole tre nozioni, variabile, astrazione e applicazione, e un solo passo di calcolo, la β riduzione, esprime ogni funzione calcolabile.

Computare in un linguaggio funzionale

I linguaggi di programmazione si classificano in base al modello computazionale a cui si ispirano. I linguaggi imperativi, come C e Pascal, si basano sulle macchine di Turing e sulle URM: le nozioni fondamentali sono memoria, variabile intesa come cella che si legge e modifica, assegnamento, istruzione, iterazione. I linguaggi funzionali, come Haskell, Scheme, ML, OCaml ed Erlang, fanno riferimento al λ calcolo e sostituiscono questi concetti con espressione, variabile in senso matematico (entità sconosciuta, astrazione di un valore), valutazione, che non modifica nulla, e ricorsione.

In un linguaggio funzionale programmare significa definire funzioni che descrivono implicitamente anche un metodo per calcolarle. Per esempio, la funzione che inverte una stringa si definisce, in Haskell, esattamente come nella definizione matematica per induzione:

```
inv [] = []
inv (a:y) = (inv y) ++ [a]
```

Qui `[]` è la lista vuota, `a:y` è la lista con testa `a` e coda `y`, `++` concatena. In Haskell le stringhe sono liste di caratteri.

FUNZIONI ANONIME E ORDINE SUPERIORE

Definire una funzione consiste nel costruire un'espressione a partire da variabili, costanti, funzioni predefinite e nomi di funzioni già definite, con l'applicazione funzionale; poi nel costruire una funzione anonima rendendo un'espressione, il corpo, funzione di un parametro formale con l'astrazione funzionale (in Haskell si usa la barra rovesciata); infine nell'associare un nome all'espressione con `=`. La definizione `square x = x * x` è zucchero sintattico per `square = \x. x * x`. Una funzione anonima è anch'essa un'espressione e può quindi essere passata come argomento o restituita come risultato: si parla di funzioni di ordine superiore.

```
square x = x * x
atzero f = f 0
compose f g x = f (g x)
```

La funzione `atzero` prende una funzione e ne restituisce il valore in 0; `compose` compone due funzioni, cosicché `compose square inc 3` vale 16 se `inc n = n + 1`.

CURRYFICAZIONE

Curryficare significa trasformare una funzione $f : A_1 \times \dots \times A_n \rightarrow B$ in una funzione di ordine superiore $f_C : A_1 \rightarrow (A_2 \rightarrow \dots \rightarrow (A_n \rightarrow B) \dots)$ tale che $f(a_1, \dots, a_n) = f_C(a_1)(a_2) \dots (a_n)$. In Haskell le funzioni sono implicitamente curryficate. Per questo nel λ calcolo le funzioni hanno un solo argomento: le funzioni a più argomenti si ottengono per curryficazione senza perdere nulla.

PASSI DI COMPUTAZIONE E STRATEGIE DI VALUTAZIONE

Il calcolo di un'espressione consiste in una sequenza di passi: A, la sostituzione di un nome con l'espressione associata; B, la sostituzione del parametro attuale al posto del parametro formale nel corpo di una funzione anonima; C, il calcolo di una funzione predefinita su valori espliciti. Per esempio:

$$\text{atzero square} \Rightarrow (\lambda f. f\ 0)\ \text{square} \Rightarrow \text{square}\ 0 \Rightarrow (\lambda n. n * n)\ 0 \Rightarrow 0 * 0 \Rightarrow 0$$

Quando un'espressione contiene più sottoespressioni riducibili, una strategia di valutazione sceglie quale ridurre. Haskell usa una strategia lazy, appartenente alla classe call by name, che riduce solo ciò che è indispensabile per il risultato; Scheme usa una strategia call by value. Le strategie possono dare risultati diversi sulla terminazione: con `alwaysseven n = 7` e `inf n = inf n`, valutare prima l'argomento di `alwaysseven (inf 3)` non termina, mentre applicare subito `alwaysseven` dà 7.

DA RICORDARE

Nei linguaggi funzionali puri vale la trasparenza referenziale: il valore di una sottoespressione è lo stesso ovunque e quante volte la si valuti, perché non ci sono effetti collaterali. In un programma imperativo il valore di una procedura può dipendere da una variabile globale modificata dalla procedura stessa, e due chiamate identiche possono dare risultati diversi. La trasparenza referenziale permette di dimostrare proprietà dei programmi con l'induzione (capitolo 11) e di valutare in parallelo le sottoespressioni.

Sintassi del λ calcolo

Costanti, funzioni base, definizioni ricorsive con nomi e i passi A e C non sono indispensabili: si possono definire usando solo variabili, applicazione, astrazione funzionale e il passo B. I termini del λ calcolo rappresentano allo stesso tempo programmi e dati.

DEFINIZIONE

λ termini. Sia V un insieme infinito numerabile di variabili $x, y, z, \dots$. L'insieme Λ dei λ termini è definito dalla grammatica $M, N ::= x \mid (M\ N) \mid (\lambda x.M)$. Un termine è dunque una variabile, un'applicazione di M a N oppure un'astrazione di variabile x su M , cioè la funzione anonima che, dato x , restituisce il corpo M .

CONVENZIONI

- Si omettono le parentesi più esterne: $M\ N$ invece di $(M\ N)$.
- L'applicazione associa a sinistra: $M\ N\ P$ significa $(M\ N)\ P$ e $f\ x\ y\ z$ significa $((f\ x)\ y)\ z$.
- Il corpo di un'astrazione si estende il più possibile a destra: $\lambda x.M\ N$ significa $\lambda x.(M\ N)$ e non $(\lambda x.M)\ N$.
- Più astrazioni si possono contrarre: $\lambda xyz.M$ abbrevia $\lambda x.\lambda y.\lambda z.M$.

Per esempio $(\lambda x.(x\ (\lambda y.(y\ x))))$ si scrive $\lambda x.x(\lambda y.yx)$. Il termine $\lambda x.x$ è l'identità, $\lambda f.\lambda x.f(f\ x)$ applica due volte una funzione.

Variabili libere e legate, α equivalenza

In un termine $\lambda x.N$ l'occorrenza di x è legata, il λx è il legame e N è il suo ambito; un'occorrenza non legata è libera. Nel termine $(\lambda x.xy)(\lambda y.yz)$ la x è legata, la z è libera, la y ha sia un'occorrenza libera sia una legata. L'insieme $FV(M)$ delle variabili libere si definisce per induzione sul termine:

$$FV(x) = \{x\} \quad FV(M\ N) = FV(M) \cup FV(N) \quad FV(\lambda x.M) = FV(M) \text{ meno } \{x\}$$

Le variabili legate, BV , si definiscono in modo analogo: $BV(x) = \emptyset$, $BV(M\ N) = BV(M) \cup BV(N)$, $BV(\lambda x.M) = BV(M) \cup \{x\}$. Un termine senza variabili libere è chiuso.

Due termini che differiscono solo per i nomi delle variabili legate rappresentano la stessa funzione: $\lambda x.x$ e $\lambda y.y$ sono α equivalenti, in simboli $\lambda x.x =_{\alpha} \lambda y.y$. L'equivalenza α è la più piccola relazione di congruenza che soddisfa, per ogni y che non occorre in M , $\lambda x.M =_{\alpha} \lambda y.(M\{y/x\})$, dove $M\{y/x\}$ indica la ridenominazione di x in y . Il sistema formale per l'equivalenza α ha le seguenti regole:

REGOLA	ENUNCIATO
(refl)	$M = M$
(symm)	da $M = N$ segue $N = M$
(trans)	da $M = N$ e $N = P$ segue $M = P$
(cong)	da $M = M'$ e $N = N'$ segue $M N = M' N'$
(ξ)	da $M = M'$ segue $\lambda x.M = \lambda x.M'$
(α)	se y non occorre in M , allora $\lambda x.M = \lambda y.(M\{y/x\})$

Le regole (refl) e (α) sono gli assiomi, le altre sono regole di inferenza vere e proprie. Il sistema è corretto e completo rispetto alla nozione di validità "M si ottiene da N rinominando variabili legate", ed è consistente: $x =_{\alpha} y$ non è derivabile. Da ora in poi i termini si identificano modulo α equivalenza (convenzione di Barendregt: si assume che le variabili legate abbiano nomi distinti tra loro e dalle libere).

Sostituzione

$M[N/x]$ indica il termine ottenuto da M sostituendo N alle occorrenze libere di x . Due problemi richiedono attenzione: si sostituiscono solo occorrenze libere, e non si deve catturare una variabile libera di N , che non può diventare legata.

DEFINIZIONE

Sostituzione senza cattura. $x[N/x] = N$; $y[N/x] = y$ se $y \neq x$; $(P Q)[N/x] = (P[N/x])(Q[N/x])$; $(\lambda x.P)[N/x] = \lambda x.P$; $(\lambda y.P)[N/x] = \lambda y.(P[N/x])$ se $y \neq x$ e $y \notin FV(N)$; $(\lambda y.P)[N/x] = \lambda y'.(P\{y'/y\}[N/x])$ se $y \neq x$, $y \in FV(N)$ e y' è una variabile nuova.

Esempio di cattura: sia $M = \lambda x.y x$ e $N = \lambda z.x z$, in cui x è libera in N e legata in M . Sostituire N a y direttamente dà $\lambda x.(\lambda z.x z) x$, dove la x di N risulta legata; occorre prima rinominare la variabile legata di M : $\lambda x'.y x'$ diventa $\lambda x'.(\lambda z.x z) x'$.

β riduzione

Valutare un termine significa sostituire a ogni applicazione di una funzione a un argomento il corpo con il parametro attuale al posto del formale.

DEFINIZIONE

β riduzione. Un termine della forma $(\lambda x.M)N$ si chiama redex e si riduce a $M[N/x]$, il suo contratto. Un passo di riduzione $M \rightarrow_{\beta} M'$ si ha quando M' si ottiene sostituendo in M un sottoterminale redex col suo contratto; $M \twoheadrightarrow_{\beta} M'$ significa zero o più passi (chiusura riflessiva e transitiva); $M =_{\beta} M'$ è la chiusura riflessiva, simmetrica e transitiva, che ammette anche passi inversi (espansioni).

Il passo è formalizzato dalle regole (β) $(\lambda x.M)N \rightarrow_{\beta} M[N/x]$, (cong_1) da $M \rightarrow_{\beta} M'$ segue $M N \rightarrow_{\beta} M' N$, (cong_2) da $N \rightarrow_{\beta} N'$ segue $M N \rightarrow_{\beta} M N'$, e (ξ) da $M \rightarrow_{\beta} M'$ segue $\lambda x.M \rightarrow_{\beta} \lambda x.M'$. Per esempio:

$$(\lambda x.y)((\lambda z.zz)(\lambda w.w)) \rightarrow_{\beta} (\lambda x.y)((\lambda w.w)(\lambda w.w)) \rightarrow_{\beta} (\lambda x.y)(\lambda w.w) \rightarrow_{\beta} y$$

Si può ridurre in un altro ordine, ottenendo subito y : ridurre un redex può crearne di nuovi o cancellarne altri, e il numero di passi dipende dall'ordine.

Forme normali, strategie e confluenza

Un termine privo di redex è in forma normale: $\lambda xy.y$ e xyz lo sono. Un termine M ha forma normale, o è debolmente normalizzabile, se esiste N in forma normale con $M \rightarrow_{\beta} N$; è fortemente normalizzabile se non esiste alcuna sequenza infinita di riduzioni a partire da M , cioè ogni strategia termina. Il termine $(\lambda x.y)((\lambda z.z)t)$ ha forma normale y ed è fortemente normalizzabile, perché le sue due sole sequenze di riduzione terminano in y . $\Omega = (\lambda x.xx)(\lambda x.xx)$ non ha forma normale, perché si riduce a se stesso. Il termine $(\lambda x.xx)(\lambda y.yyy)$ non ha forma normale e a ogni passo la sua dimensione cresce: la dimensione di un termine non diminuisce necessariamente riducendo. Il termine $(\lambda y.a)\Omega$ ha forma normale a , ma ammette anche una riduzione infinita: è normalizzabile e non fortemente normalizzabile.

TERMINE	PROPRIETÀ
$(\lambda x.x)(\lambda x.xx)$	ha forma normale $\lambda x.xx$, è fortemente normalizzabile
$(\lambda x.x)(\lambda x.x)$	fortemente normalizzabile, forma normale $\lambda x.x$
$(\lambda x.xx)(\lambda x.x)$	fortemente normalizzabile: si riduce a $(\lambda x.x)(\lambda x.x)$ e poi a $\lambda x.x$
$\Omega = (\lambda x.xx)(\lambda x.xx)$	senza forma normale
$(\lambda y.a)\Omega$	ha forma normale a , non è fortemente normalizzabile

STRATEGIE

- **Call by value:** non si riduce un redex finché l'argomento non è un valore, qui un termine in forma normale. In Scheme anche una funzione $\lambda x.M$ è un valore.
- **Call by name:** si riduce senza controllare l'argomento. Una sottoclasse è la strategia lazy (call by need), che riduce solo se strettamente necessario.
- **Ordine normale:** si riduce il redex più a sinistra e più esterno (leftmost outermost). È una strategia call by name.

DEFINIZIONE

Teorema di standardizzazione. Se un termine ha forma normale, questa si raggiunge applicando la strategia a ordine normale. Con call by value, $(\lambda y.a)\Omega$ non termina mai; con ordine normale dà subito a .

DEFINIZIONE

Teorema di Church Rosser (confluenza). Se $M \rightarrow_{\beta} P$ e $M \rightarrow_{\beta} Q$, esiste un termine R tale che $P \rightarrow_{\beta} R$ e $Q \rightarrow_{\beta} R$. Equivalentemente: se $M =_{\beta} N$ allora esiste R con $M \rightarrow_{\beta} R$ e $N \rightarrow_{\beta} R$.

CONSEGUENZE

- **Unicità della forma normale.** Se M ha due forme normali N_1 e N_2 , per confluenza esiste R con $N_1 \rightarrow_{\beta} R$ e $N_2 \rightarrow_{\beta} R$; poiché N_1 e N_2 non hanno redex, R coincide con entrambe: $N_1 = N_2$.
- **Consistenza della β equivalenza.** I termini x e $\lambda y.y$ non sono β equivalenti, altrimenti avrebbero un riduttore comune, che non esiste. La teoria $=_{\beta}$ è perciò consistente: non tutti i termini sono equivalenti.
- Due programmi che realizzano lo stesso algoritmo in linguaggi con strategie diverse, come Scheme e Haskell, danno lo stesso risultato, se esiste.

Dati e funzioni nel λ calcolo

Nel λ calcolo non servono tipi di dato primitivi: booleani e numeri si codificano con termini.

BOOLEANI

Si pone $T = \lambda xy.x$ e $F = \lambda xy.y$. Allora $\text{and} = \lambda ab.a \ b \ F$, $\text{or} = \lambda ab.a \ T \ b$, $\text{not} = \lambda a.a \ F \ T$ e $\text{if_then_else} = \lambda x.x$. Verifica: $\text{and } T \ T \Rightarrow T$, $\text{and } T \ F \Rightarrow F$, $\text{and } F \ T \Rightarrow F$, $\text{and } F \ F \Rightarrow F$; $\text{if_then_else } T \ M \ N \Rightarrow M$ e $\text{if_then_else } F \ M \ N \Rightarrow N$.

NUMERI DI CHURCH

Se f e x sono termini, $f^n x$ indica $f(f(\dots(f \ x)\dots))$ con n occorrenze di f . L' n esimo numerale di Church, indicato con c_n , è $c_n = \lambda fx.f^n x$:

$$c_0 = \lambda fx.x \quad c_1 = \lambda fx.f \ x \quad c_2 = \lambda fx.f(f \ x) \quad c_3 = \lambda fx.f(f(f \ x))$$

Si osservi che c_0 coincide con F . Un termine M rappresenta (numeralwise) una funzione f di k argomenti sui naturali se, applicato ai numerali degli argomenti $n_1, \dots, n_k$, si riduce al numerale c_m con $m = f(n_1, \dots, n_k)$.

FUNZIONE	TERMINE	VERIFICA
Successore	$\text{succ} = \lambda nfx.f(n \ f \ x)$	$\text{succ } c_n \Rightarrow \lambda fx.f(f^n x) = c_{n+1}$
Somma	$\text{add} = \lambda nmfx.n \ f \ (m \ f \ x)$	$\text{add } c_n \ c_m \Rightarrow \lambda fx.f^n(f^m x) = c_{n+m}$
Prodotto	$\text{mult} = \lambda nmf.n(m \ f)$	$\text{mult } c_n \ c_m \Rightarrow \lambda fx.c_n(c_m \ f) = c_{nm}$
Potenza	$\text{exp} = \lambda nm.m \ n$	$\text{exp } c_n \ c_m$ rappresenta n^m
Test di zero	$\text{iszero} = \lambda nxy.n(\lambda z.y)x$	$\text{iszero } c_0 \Rightarrow T$ e $\text{iszero } c_n \Rightarrow F$ per $n > 0$
Predecessore	$\text{pred} = \lambda nfx.n(\lambda gh.h(g \ f))(\lambda u.x)(\lambda u.u)$	$\text{pred } c_{n+1} \Rightarrow c_n$ e $\text{pred } c_0 \Rightarrow c_0$

Per esempio $\text{add } c_2 \ c_3 \Rightarrow \lambda fx.c_2 \ f \ (c_3 \ f \ x) \Rightarrow \lambda fx.f(f(f(f(f \ x)))) = c_5$. La funzione pred non è semplice: Church credette a lungo che fosse impossibile definirla. La funzione $f(n) = (n + 3)^2$ si rappresenta con $\lambda n.\text{mult} \ (\text{add } n \ c_3) \ (\text{add } n \ c_3)$; il predicato di parità con $\lambda n.n \ \text{not } T$, perché not applicato n volte a T dà T se n è pari. Le coppie $\langle M, N \rangle = \lambda z.z \ M \ N$ hanno le proiezioni $\text{left} = \lambda p.p(\lambda xy.x)$ e $\text{right} = \lambda p.p(\lambda xy.y)$; analogamente si codificano n tuple, liste e alberi.

Ricorsione e punti fissi

Nel λ calcolo non si possono dare nomi ai termini, ma si definiscono comunque funzioni ricorsive, grazie ai punti fissi. Una funzione come il fattoriale, $\text{fact } n = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \text{fact}(n - 1)$, contiene fact nel proprio corpo: sostituire fact con il suo corpo non termina mai.

DEFINIZIONE

Punto fisso. N è un punto fisso di F se $F \ N =_\beta N$. Nel λ calcolo ogni termine ha un punto fisso: posto $A = \lambda xy.y(xxy)$ e $\Theta = AA$ (combinatore di Turing), per ogni F il termine ΘF è un punto fisso di F . Infatti $\Theta F = AAF \Rightarrow F(AAF) = F(\Theta F)$.

Il fattoriale soddisfa $\text{fact} = F \ \text{fact}$, con $F = \lambda f.\lambda n.\text{if_then_else} \ (\text{iszero } n) \ c_1 \ (\text{mult } n \ (f \ (\text{pred } n)))$: si risolve con $\text{fact} = \Theta F$, un termine chiuso che non contiene fact . Vale $\Theta F \Rightarrow F(\Theta F)$, quindi $\text{fact } c_2 \Rightarrow \text{mult } c_2 \ (\text{fact } c_1) \Rightarrow \text{mult } c_2 \ (\text{mult } c_1 \ (\text{fact } c_0)) \Rightarrow \text{mult } c_2 \ (\text{mult } c_1 \ c_1) \Rightarrow c_2$.

Il combinatore $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ di Curry è un altro operatore di punto fisso: per ogni F , $Y \ F =_\beta F(Y \ F)$, ma non vale $Y \ F \Rightarrow F(Y \ F)$ come per Θ ; esistono infiniti combinatori di punto fisso.

Le funzioni λ definibili coincidono con le funzioni parziali ricorsive e con quelle calcolabili da macchine di Turing (tesi di Church).

Codici e rappresentazione dell'informazione numerica

Per elaborare informazioni un sistema di calcolo deve rappresentarle con simboli che sa manipolare. Un codice stabilisce la corrispondenza: la scelta del codice determina quanto siano semplici ed efficienti gli algoritmi che operano sulle rappresentazioni.

Codici

Sia X un insieme qualsiasi, finito o infinito, di oggetti da rappresentare, e sia A un alfabeto finito di simboli (le dieci cifre, le lettere, tre colori, due bit).

DEFINIZIONE

Codice. Un codice per X su A è formato da una funzione iniettiva di codifica $\text{cod} : X \rightarrow A^*$ e da una funzione di decodifica $\text{decod} : A^* \rightarrow X \cup \{\text{errore}\}$ tali che $\text{decod}(\text{cod}(y)) = y$ per ogni $y \in X$. La sequenza $\text{cod}(y)$ è la codifica, o rappresentazione, di y .

In generale non tutte le sequenze di A^* sono codifiche di elementi di X : se b non è una codifica, $\text{decod}(b) = \text{errore}$. Un codice a lunghezza fissa predetermina il numero di simboli di una codifica corretta, e le altre sequenze non vengono considerate. La scelta del codice dipende dall'uso: per scrivere un numero su un foglio conta la compattezza, per eseguire una somma conta la semplicità dell'algoritmo, per un uso generale serve un compromesso.

Rappresentazione dei numeri naturali

NUMERI ROMANI E NUMERI POSIZIONALI

La rappresentazione romana usa sette simboli, I V X L C D M, per i numeri uno, cinque, dieci, cinquanta, cento, cinquecento e mille; gli altri numeri si ottengono con un algoritmo che minimizza le occorrenze (per questo quattro è IV e non IIII). Difetti: la lunghezza cresce linearmente con il valore, non c'è uniformità per numeri da quattromila in poi e il calcolo è complesso. Pregio: i numeri arrotondati si scrivono in modo breve.

Nella rappresentazione posizionale in base B , con alfabeto di B cifre $\{0, \dots, B \text{ meno } 1\}$, la stringa $c_{p-1} \dots c_1 c_0$ rappresenta il numero:

$$n = c_{p-1} B^{p-1} + \dots + c_1 B + c_0 = \sum_{i=0}^{p-1} c_i B^i$$

Il numero di cifre cresce in modo logaritmico con il valore, e gli algoritmi di somma, prodotto e divisione operano direttamente sulle cifre, con complessità logaritmica nel valore: da qui il successo del sistema decimale. Il vantaggio non deriva dalle dieci cifre, ma dal modo di interpretarle: lo stesso schema vale in qualsiasi base.

BASE	NOME	CIFRE	USO
2	binaria	0, 1	dispositivi elettronici con due livelli, più semplici ed economici
8	ottale	0 a 7	abbreviazione del binario: una cifra ogni tre bit
16	esadecimale	0 a 9, A a F	abbreviazione del binario: una cifra ogni quattro bit, due per byte

CONVERSIONE PER DIVISIONI SUCCESSIVE

Per ottenere le cifre in base B di un naturale n, si divide n per B: il resto è la cifra meno significativa c_0 ; si ripete la divisione sul quoziente, il cui resto è c_1 , e così via fino a quoziente zero. L'algoritmo funziona perché $n = B (\sum_{i \geq 1} c_i B^{i-1}) + c_0$; c_0 è il resto della divisione per B e il quoziente è $\sum_{i \geq 1} c_i B^{i-1}$, che a sua volta si scrive $B (\sum_{i \geq 2} c_i B^{i-2}) + c_1$. Per esempio 45: $45 = 22 \cdot 2 + 1$; $22 = 11 \cdot 2 + 0$; $11 = 5 \cdot 2 + 1$; $5 = 2 \cdot 2 + 1$; $2 = 1 \cdot 2 + 0$; $1 = 0 \cdot 2 + 1$; leggendo i resti dal basso si ottiene 101101.

CONVERSIONE TRA BASI UNA POTENZA DELL'ALTRA

Se $B' = B^k$, si passa dalla base B alla base B' raggruppando le cifre a blocchi di k, a partire da destra, e sostituendo ogni blocco con la cifra corrispondente: perché per $n = \sum c_i B^i$ il raggruppamento dà $n = \sum e_j (B^k)^j$, dove e_j è il valore del j-esimo blocco. Viceversa si espande ogni cifra nei k bit corrispondenti.

CONVERSIONE	PROCEDIMENTO	RISULTATO
Da binario a esadecimale	00101101 si divide in 0010 e 1101, che valgono 2 e 13	0x2D
Da esadecimale a binario	0xB3: B vale 11, cioè 1011; 3 vale 0011	10110011
Da esadecimale a binario	0xA7B: 1010, 0111, 1011	101001111011
Da ottale a binario	755: 7 è 111, 5 è 101, 5 è 101	111101101

Per questo il contenuto di un byte si scrive con due cifre esadecimali precedute dal prefisso 0x: il byte 00101101, che vale 45, è 0x2D.

Lunghezza fissa e lunghezza variabile

Su carta si adotta la regola di non scrivere zeri a sinistra: la rappresentazione è unica, perché altrimenti 57, 057, 0057 rappresenterebbero lo stesso numero, ed è a lunghezza variabile. Un algoritmo di somma, però, richiede di allineare gli addendi a destra e, in un sistema di calcolo, di riservare risorse in anticipo.

- **Lunghezza fissa.** Si fissa il massimo valore rappresentabile e si usa sempre lo stesso numero p di cifre. Con valori fino a 9999 e quattro cifre decimali, 57 diventa 0057. L'intervallo dei naturali rappresentabili è $[0, B^p \text{ meno } 1]$.
- **Lunghezza variabile.** Per leggere un numero a lunghezza variabile occorre riconoscerne la fine. Senza un simbolo in più, un numero non può avere un numero arbitrario di cifre: serve un $(N + 1)$ -esimo simbolo di delimitazione, in carta lo spazio bianco. Per esempio con alfabeto di dieci cifre più # si scrivono #0, #1, #2, #57, con # come cifra più a sinistra e con 0 che non può seguire # tranne se è l'ultima cifra. Analogamente un testo ASCII a 128 simboli riserva un simbolo, il carattere nul, per segnare la fine, e ne restano 127 per il testo.

Codici a espansione

Un codice a espansione ottiene la lunghezza variabile da un codice a lunghezza fissa senza cambiare l'alfabeto: si prevede un insieme di formati con numero crescente di bit e si fa capire dai primi bit quale formato è in uso. I numeri piccoli usano pochi bit e quelli grandi molti; l'idea è usata per i codici operativi delle istruzioni di un linguaggio macchina.

FORMATI MULTIPLI PREDEFINITI

Quattro formati da 4, 7, 10 e 16 bit, distinti da un prefisso di due bit che ne indica il numero d'ordine:

PREFISSO	BIT TOTALI	BIT UTILI	NUMERI RAPPRESENTATI
00	4	2	da 0 a 3
01	7	5	da 4 a 35
10	10	8	da 36 a 291
11	16	14	da 292 a 16675

CIFRA DI ESPANSIONE

Lo stesso schema con prefissi di lunghezza diversa, per avere più numeri piccoli: formati da 4, 7, 10 e 16 bit con prefissi 0, 10, 110, 111.

PREFISSO	BIT TOTALI	BIT UTILI	NUMERI RAPPRESENTATI
0	4	3	da 0 a 7
10	7	5	da 8 a 39
110	10	7	da 40 a 167
111	16	13	da 168 a 8359

CONFIGURAZIONE DI ESPANSIONE

Si riserva una sola configurazione per segnalare che il formato non è quello corrente: 1111 nel formato da 4 bit, 1111111 in quello da 7, 1111111111 in quello da 10. Aumenta ancora la capacità dei formati piccoli e riduce quella dei grandi.

FORMATO	CONFIGURAZIONI UTILI	CONFIGURAZIONE DI ESPANSIONE	NUMERI RAPPRESENTATI
4 bit	15	1111	da 0 a 14
7 bit	7	1111111	da 15 a 21
10 bit	7	1111111111	da 22 a 28
16 bit	64	nessuna	da 29 a 92

Numeri interi in complemento a due

Si cerca un codice a lunghezza fissa per gli interi. Quello più diffuso è il complemento alla base, detto complemento a due in base 2, con vantaggi: rappresentazione unica dello zero, riconoscimento facile del segno, algoritmi semplici per somma e moltiplicazione. Si indica con B la base e con p il numero di cifre; cod_N è la codifica posizionale dei naturali con p cifre.

DEFINIZIONE

Complemento alla base. $\text{cod}_Z(r) = \text{cod}_N(r)$ se $r \geq 0$; $\text{cod}_Z(r) = \text{cod}_N(B^p \text{ meno } |r|)$ se $r < 0$. Per $B = 2$ l'intervallo di rappresentabilità è $[-2^{p-1}, 2^{p-1} - 1]$.

I numeri si possono immaginare disposti su una circonferenza con 2^p configurazioni. Con $p = 4$:

CONFIGURAZIONE	INTERO	CONFIGURAZIONE	INTERO
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

PROPRIETÀ IN BASE 2

- **Segno.** Il numero rappresentato è negativo se e solo se il bit più a sinistra è 1.
- **Opposto.** L'opposto di r si ottiene complementando tutte le cifre della rappresentazione di r (0 diventa 1 e viceversa) e sommando 1. Infatti, detta $c_{p-1} \dots c_0$ la rappresentazione di r e c'_i il complemento di c_i , $\sum c_i 2^i + \sum c'_i 2^i = 2^p - 1$ (p cifre pari a 1), quindi $\sum c'_i 2^i + 1 = 2^p - \sum c_i 2^i$: per $r > 0$ questa è la rappresentazione di $-r$ per definizione; per $r < 0$, dove $\sum c_i 2^i = 2^p - |r|$, vale $|r|$. Esempio: 5 è 0101, il complemento è 1010, più 1 dà 1011, cioè -5. L'unica eccezione è -2^{p-1} : il suo opposto non è rappresentabile.
- **Somma.** $\text{cod}_Z(r_1 + r_2) = \text{cod}_Z(r_1) \oplus \text{cod}_Z(r_2)$, dove $\oplus$ è la normale somma cifra per cifra con riporto, troncata a p cifre, senza guardare il segno degli addendi. Poiché $\text{cod}_Z(-r)$ si ottiene con l'opposto, la sottrazione si riduce a una somma. Il risultato è corretto solo se $r_1 + r_2$ appartiene all'intervallo di rappresentabilità.
- **Supero di capacità.** Si ha overflow se la somma supera il limite superiore, underflow se scende sotto il limite inferiore. In base 2 si riconosce confrontando i riporti entrati e uscenti nella posizione del segno: se sono diversi c'è stato supero. Esempio con $p = 4$: $0111 + 0001 = 1000$, cioè $7 + 1$ dà -8 , riporto in ingresso 1 e in uscita 0.
- **Estensione.** Per passare da p a $p + k$ cifre si copia k volte a sinistra la cifra più a sinistra: 1011 (-5) diventa 11111011 con otto cifre. In base B si estende con 0 se il numero è positivo e con B meno 1 se è negativo.
- **Prodotto.** $\text{cod}_Z(r_1 \cdot r_2) = \text{cod}_Z(r_1) \otimes \text{cod}_Z(r_2)$ con gli algoritmi del prodotto sui naturali, ancora in assenza di supero.

DA RICORDARE

La somma è corretta sulle rappresentazioni perché, per $X, Y \geq 0$ con $Y > X$: $\text{cod}_N(X) \oplus \text{cod}_N(B^p - Y) = \text{cod}_N(B^p - (Y - X)) = \text{cod}_Z(X - Y)$; per $Y \leq X$: $\text{cod}_N(B^p + (X - Y)) = \text{cod}_N(X - Y)$, perché con p cifre $\text{cod}_N(B^p + n) = \text{cod}_N(n)$. Gli altri casi sono analoghi.

Macchine astratte

Dalla macchina di Turing universale si ricava il concetto di macchina astratta, che permette di descrivere in modo uniforme sia i calcolatori sia i linguaggi di programmazione e di organizzare un sistema di calcolo in livelli.

Automi a stati finiti con uscita

Gli automi a stati finiti studiati nei capitoli precedenti restituiscono come risultato un valore booleano, o non terminano. Un'importante variante restituisce stringhe su un insieme Z di caratteri di uscita: si elimina la nozione di stato finale e si prende come funzione di transizione $\delta : \Sigma \times K \rightarrow K \times Z$, con K insieme degli stati, che associa al carattere letto e allo stato interno lo stato successivo e un carattere di uscita. Nel diagramma degli stati si indicano su ogni arco il carattere letto e il carattere prodotto. Queste macchine elaborano anche stringhe infinite, restituendo passo passo una stringa di uscita, e descrivono il comportamento dei circuiti sequenziali; da una specifica espressa con un automa si ricavano, con procedure automatizzabili, i circuiti che la realizzano.

Il concetto di macchina astratta

La macchina di Turing universale contiene già le componenti da cui si astrae uno schema generale: una memoria che contiene dati e programmi, un modo di recuperare gli argomenti delle operazioni, un modo per stabilire la prossima istruzione da eseguire, e qualcosa che coordini il tutto ed esegua i programmi. Non esiste una definizione di macchina astratta valida per ogni modello: alla base c'è sempre un modello computazionale. Le macchine basate sul modello di Turing si dicono imperative, perché fondate sul concetto di comando. Partendo da modelli diversi si ottengono altri tipi di macchine astratte: funzionali, a oggetti, logiche, parallele.

DEFINIZIONE

Macchina astratta (imperativa). Un insieme di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi. Le sue componenti sono: un **interprete**; una **memoria**, per il programma da eseguire e per i dati; un insieme di **operazioni primitive** sui dati primitivi; operazioni e strutture dati per il **controllo del flusso** (l'ordine di esecuzione); operazioni e strutture dati per il **controllo del trasferimento dei dati** (recupero degli operandi, memorizzazione dei risultati); operazioni e strutture dati per la **gestione della memoria**.

Macchine astratte diverse differiscono per la struttura della memoria, i tipi primitivi, le operazioni elementari e le componenti di controllo. Moduli per l'input e l'output non sono inclusi per generalità, perché ne esistono troppe varietà.

IL CICLO DELL'INTERPRETE

Il cuore è l'interprete, che esegue un ciclo di fetch ed execute finché non incontra una particolare istruzione primitiva di arresto, HALT. In ogni iterazione acquisisce la prossima istruzione (FETCH), la decodifica e acquisisce gli operandi, esegue l'operazione primitiva e memorizza l'eventuale risultato.

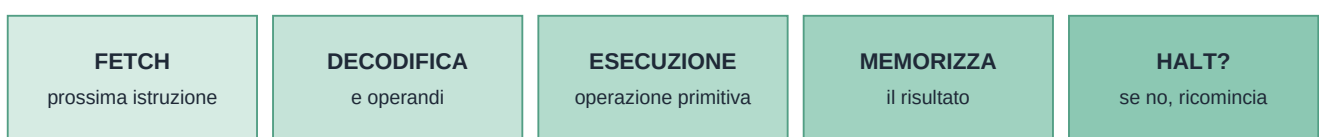


Figura 1. Ciclo dell'interprete di una macchina astratta.

ESEMPI

- **Macchina fisica tradizionale.** Le operazioni primitive sono quelle aritmetico logiche, la manipolazione di stringhe di bit, la lettura e la scrittura di celle e registri. Il controllo di sequenza usa il contatore di programma PC e le strutture per i ritorni dai sottoprogrammi, con salti, condizionali, chiamate e ritorni. Il controllo del trasferimento dei dati riguarda gli operandi, con registri indice e indirizzamento indiretto.
- **Macchina a pila.** La struttura fondamentale è lo stack, da cui si prelevano gli operandi e su cui si memorizzano i risultati; la gestione della memoria coincide con quella dello stack.
- **Il ristorante.** Anche un ristorante è una macchina astratta: i programmi sono sequenze di ordinazioni, le operazioni primitive sono i cuochi, la memoria è il taccuino del cameriere e la dispensa, l'interprete è il cameriere che decodifica ciascuna ordinazione.

Linguaggi di programmazione e macchine astratte

Il **linguaggio macchina** L_M di una macchina astratta M è il linguaggio in cui si esprimono i programmi interpretabili dall'interprete di M ; definisce le strutture dati che rappresentano internamente i programmi, che sono dati primitivi su cui opera l'interprete. La versione come stringhe di caratteri si chiama **linguaggio macchina esterno**, $L_{M\text{EST}}$; la conversione da $L_{M\text{EST}}$ a L_M è automatizzabile ed è svolta dal loader. Per esempio i livelli di tensione nei bit di una memoria sono un L_M , le stringhe di 0 e 1 il $L_{M\text{EST}}$.

Data una macchina astratta, il suo linguaggio macchina è un linguaggio di programmazione; viceversa, dato un linguaggio, resta definita una macchina astratta che ne supporta le caratteristiche e ha il linguaggio come linguaggio macchina. I linguaggi ad alto livello definiscono macchine tanto più complesse quanto più sono espressivi: un linguaggio con le liste come tipo primitivo ha una macchina con istruzioni elementari per head, tail e isNull e con una gestione automatica dell'allocazione dinamica. Perciò una realizzazione in hardware è conveniente solo per macchine relativamente semplici.

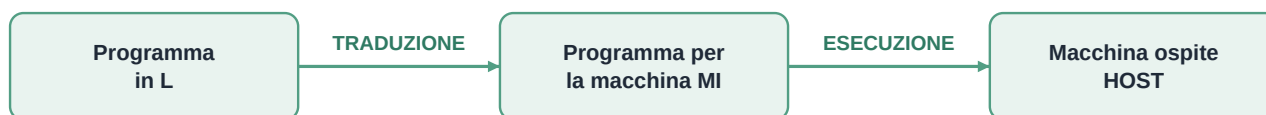
Realizzazione di una macchina astratta

Per eseguire programmi in un linguaggio L bisogna realizzare la macchina astratta M_L corrispondente, in una di tre maniere.

TECNICA	IDEA	VANTAGGI E LIMITI
Hardware	tutte le componenti sono realizzate fisicamente	sempre possibile in linea di principio; poco flessibile e costosa per funzionalità complesse
Interpretazione	le componenti sono realizzate con programmi e strutture dati di una macchina ospite (host): per emulazione via firmware (macchina ospite microprogrammata, velocità vicina all'hardware) o via software	flessibile; più lenta; l'interprete della macchina realizzata è un programma eseguito dall'interprete dell'ospite
Compilazione	un compilatore traduce una volta per tutte l'intero programma di L in un programma equivalente per l'ospite	nessuna componente della macchina astratta è visibile, ma se ne vedono gli effetti

MACCHINA INTERMEDIA

La differenza di potenza espressiva tra la macchina da realizzare e la macchina ospite si chiama **semantic gap**. Quando è molto grande non si procede per pura interpretazione, a causa della bassa efficienza, né per pura compilazione, a causa delle dimensioni e della lentezza del codice prodotto e delle difficoltà di scrivere traduttori tra linguaggi molto diversi: si introduce una macchina intermedia MI , realizzata sull'ospite. I programmi in L si traducono nel linguaggio di MI e quindi si eseguono per interpretazione di MI sulla macchina ospite.



semantic gap tra L e HOST colmato dalla macchina intermedia MI e dal suo run time system

Figura 2. Realizzazione di L con una macchina intermedia MI sulla macchina ospite.

Gli obiettivi nel progetto di MI sono la velocità di simulazione su HOST e la compattezza del codice prodotto. Tipicamente MI è un'estensione di HOST, ne condivide l'interprete e lo potenzia con un insieme di routine, il run time system. La pura interpretazione e la pura compilazione sono i casi estremi in cui MI coincide con la macchina del linguaggio o con la macchina ospite. L'esempio tipico è Java: il compilatore traduce in bytecode, il linguaggio della Java Virtual Machine (JVM), che è la macchina intermedia, realizzata per interpretazione; il run time system è il Java Runtime Environment. La JVM non è la macchina astratta del linguaggio Java, bensì la sua macchina intermedia.

Struttura a livelli dei sistemi di calcolo

Generalizzando, un calcolatore è una gerarchia di macchine astratte, ciascuna realizzata sulla precedente per traduzione o interpretazione: il livello più basso è il computer reale, ogni livello superiore è una macchina i cui programmi sono tradotti o interpretati dai livelli inferiori. Gli scopi sono gestire la complessità della progettazione e aumentare la flessibilità. Un livello non copre necessariamente tutto il livello sottostante: alcune sue componenti possono coincidere con quelle dell'ospite, altre essere nuove.

LIVELLO 5	Linguaggi ad alto livello: C, Java, Haskell, Prolog
LIVELLO 4	Linguaggio assembly
LIVELLO 3	Sistema operativo: file, memoria virtuale, processi
LIVELLO 2	Livello ISA: istruzioni del linguaggio macchina
LIVELLO 1	Datapath, registri, unità aritmetico logica
LIVELLO 0	Logica digitale: porte logiche, registri, memorie

Figura 3. Una possibile gerarchia di macchine astratte di un calcolatore.

LIVELLO	CONTENUTO
0, logica digitale	porte logiche (not, and, or, ...) formalizzate dall'algebra di Boole; da esse registri e memorie. È il ponte tra il mondo analogico e quello digitale.
1	registri e circuiti aritmetico logici; datapath e sua unità di controllo; qui compare il flusso di informazione
2, ISA	il linguaggio macchina in senso tecnico (ADD, MOVE, SUB, ...); realizzato in hardware nelle architetture RISC, per interpretazione firmware nelle CISC
3, sistema operativo	ripropone la maggior parte delle istruzioni ISA e aggiunge file e file system gerarchico, memoria virtuale e processi con multitasking
4, assembly	primo livello con caratteristiche dei linguaggi moderni (etichette, mnemonici, macro, procedure); realizzato per traduzione
5, linguaggi ad alto livello	C, Java, C++, Haskell, Prolog: livello più vario, anche per lo stesso linguaggio (versioni interpretate e compilate)

I livelli 0 a 3 sono tipicamente programmati dai sistemisti, perché servono a gestire la complessità delle macchine superiori. Con Java si inserisce un livello tra l'alto livello e l'assembly: la JVM, sempre più usata come macchina standard. Sotto la logica digitale ci sono i livelli di elettronica circuitale (dispositivi come transistor, diodi, MOS, da cui le porte logiche) e di fisica dello stato solido: sono livelli di astrazione di natura diversa, che non descrivono macchine astratte.

DA RICORDARE

Ogni programma scritto in un linguaggio L aggiunge un livello alla gerarchia, perché estende la macchina astratta associata a L definendo nuove operazioni e nuovi tipi di dato. Programmare significa dunque definire e realizzare macchine astratte, e la potenza di un linguaggio dipende dai meccanismi di astrazione che offre: la procedura per le operazioni, i tipi di dato astratti nei linguaggi a oggetti. Si programma facendo riferimento alla specifica di una macchina astratta e non alla sua realizzazione, per mantenere la portabilità, a meno di vincoli particolari sulle prestazioni.

Sistemi formali e logica proposizionale

La logica formalizza il ragionamento: un sistema formale definisce in modo puramente sintattico che cosa si può dedurre, e una semantica associa alle formule un significato. Correttezza e completezza collegano i due livelli.

Sistemi formali

Il processo deduttivo consiste nel passare da affermazioni vere ad altre affermazioni vere secondo meccanismi di pura trasformazione sintattica. Si vuole che a questo gioco di simboli corrisponda un significato, la semantica, fatta di strutture matematiche: se le trasformazioni portano sempre a risultati validi il sistema è corretto; se tutto ciò che è valido è derivabile è completo.

DEFINIZIONE

Sistema formale. Un sistema formale D è dato da un insieme numerabile S di simboli (alfabeto); un insieme decidibile $W \subseteq S^*$ di formule ben formate (fbf); un insieme $Ax \subseteq W$ di assiomi; un insieme finito R di regole finitarie, ciascuna una relazione su W^n per un opportuno n . La coppia $\langle S, W \rangle$ è il linguaggio formale. Una regola si scrive con le premesse sopra e la conclusione sotto una linea; un assioma è una regola senza premesse. Se Ax è decidibile il sistema è ricorsivamente assiomatizzato.

DEFINIZIONE

Derivazione. Dato un insieme M di fbf, una D derivazione a partire da M è una successione finita di fbf $\alpha_1, \dots, \alpha_n$ tale che ogni α_i è un assioma, oppure un elemento di M (ipotesi), oppure è la conclusione di una regola le cui premesse la precedono nella successione. Si scrive $M \vdash_D \alpha$ se esiste una derivazione la cui ultima fbf è α ; se $M = \emptyset$ si scrive $\vdash_D \alpha$ e α è un teorema.

Nei sistemi alla Hilbert le regole sono relazioni tra fbf e le derivazioni sono sequenze di fbf. Esistono sistemi, come la deduzione naturale, le cui regole non sono relazioni tra giudizi perché scaricano ipotesi: qui le derivazioni sono alberi, con le regole nei nodi e le ipotesi non scaricate nelle foglie.

PROPRIETÀ DELLE DERIVAZIONI

- Se $M \vdash \alpha$ allora esiste $N \subseteq M$ finito con $N \vdash \alpha$, perché una derivazione è finita: le dimostrazioni sono esponibili in tempo finito.
- Se $M \vdash \alpha_1, \dots, M \vdash \alpha_n$ e $\{\alpha_1, \dots, \alpha_n\} \vdash \beta$, allora $M \vdash \beta$: in teoria i lemmi non sono necessari (regola di taglio).

REGOLE DERIVABILI E AMMISSIBILI

Una regola R non appartenente a D è **derivabile** se per ogni istanza di R , dalle premesse segue la conclusione in D ; è **ammissibile** se, aggiungendo R a D , non si ottengono nuovi teoremi. Ogni regola derivabile è ammissibile, perché ogni applicazione di R si sostituisce con la sua derivazione; il viceversa non vale in generale.

CONSISTENZA E TEORIE

NOZIONE	DEFINIZIONE
Sistema consistente	esiste una fbf α tale che $\not\models \alpha$; altrimenti è inconsistente
Insieme Γ consistente	$\text{Con}(\Gamma) \neq W$, cioè esiste α con $\Gamma \not\models \alpha$; altrimenti è contraddittorio
Teoria	insieme Γ chiuso rispetto a $\vdash$: $\text{Con}(\Gamma) = \Gamma$
Teoria pura	$\text{Con}(\emptyset) = \text{Con}(Ax)$, cioè l'insieme di tutti i teoremi

Un sistema inconsistente dimostra tutto e non serve a nulla; se è corretto e completo, per provare che è consistente basta esibire un giudizio non valido.

DA RICORDARE

La correttezza richiede che gli assiomi siano validi e che le regole conservino la validità; ha senso così com'è solo per derivazioni senza ipotesi. Una definizione generale che tenga conto delle ipotesi non esiste: si dà caso per caso, come per la logica proposizionale e per quella dei predicati. La completezza richiede che tutti i giudizi validi siano derivabili. La logica proposizionale è corretta e completa; l'aritmetica non è completa.

Esempi di sistemi formali

IL SISTEMA CL

Il sistema CL formalizza i combinatori. I simboli sono k, s , le parentesi e $=$; le fbf sono $P = Q$ con P, Q termini, definiti da: k, s sono termini, e (PQ) è un termine se lo sono P e Q . Gli assiomi sono gli schemi $((kP)Q) = P$ (Ax_k), $((sP)Q)R = ((PR)(QR))$ (Ax_s) e $P = P$ (riflessività). Le regole sono la simmetria, la transitività e due regole di congruenza (CONGR1 e CONGR2), che consentono di sostituire termini uguali all'interno di un'applicazione. Con la definizione esplicita $I = ((sk)k)$ si deriva (IM) = M per ogni M : $((sk)k)M = ((kM)(kM))$ per Ax_s , e $((kM)(kM)) = M$ per Ax_k con transitività. CL è un esempio di sistema formale alla Hilbert per il quale non è specificata una semantica.

A EQUIVALENZA E B RIDUZIONE

L'equivalenza α (capitolo 6) è un sistema formale i cui giudizi sono $M =_{\alpha} N$ e la cui validità significa "M si ottiene da N rinominando variabili legate". Gli assiomi sono (refl) e (α) . Il sistema è consistente perché $x =_{\alpha} y$ non è valido. Allo stesso modo la β riduzione è un sistema formale i cui giudizi $M \rightarrow_{\beta} N$ valgono se in M c'è un sottoterminale redex sostituito dal contratto.

Il sistema P0

DEFINIZIONE

Sistema formale P0 (calcolo proposizionale). Alfabeto: variabili proposizionali $p, q, r, \dots$, i connettivi $\rightarrow$ e $\neg$ e le parentesi. Fbf: ogni variabile; se α e β sono fbf, anche $(\alpha \rightarrow \beta)$ e $(\neg \alpha)$. Definizioni esplicite: $\alpha \vee \beta = (\neg \alpha) \rightarrow \beta$; $\alpha \wedge \beta = \neg((\neg \alpha) \vee (\neg \beta))$; $\alpha \leftrightarrow \beta = (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$. Precedenza: $\neg$, poi $\wedge$ e $\vee$, poi $\rightarrow$. Assiomi (schemi): $(A_k) \alpha \rightarrow (\beta \rightarrow \alpha)$; $(A_g) (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$; $(A_{\neg}) (\neg \alpha \rightarrow \neg \beta) \rightarrow ((\neg \alpha \rightarrow \beta) \rightarrow \alpha)$. Regola: modus ponens, da $\alpha \rightarrow \beta$ e α segue β .

Ogni teorema di P0 è una tautologia, e viceversa (correttezza e completezza, più sotto). Esempio: $\vdash \alpha \rightarrow \alpha$ si deriva con A_g, A_k e due modus ponens:

PASSO	FBF	GIUSTIFICAZIONE
1	$(\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha))$	A_S
2	$\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha)$	A_k
3	$(\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha)$	MP(1, 2)
4	$\alpha \rightarrow (\beta \rightarrow \alpha)$	A_k
5	$\alpha \rightarrow \alpha$	MP(3, 4)

TEOREMA DI DEDUZIONE (HERBRAND)

DEFINIZIONE

Teorema di deduzione. Per ogni insieme Γ di fbf di P0, $\Gamma \vdash \alpha \rightarrow \beta$ se e solo se $\Gamma, \alpha \vdash \beta$.

La direzione ($\Rightarrow$) è immediata: dalla derivazione di $\alpha \rightarrow \beta$, si aggiunge l'ipotesi α e si applica il modus ponens. Per ($\Leftarrow$), data una derivazione $\gamma_1, \dots, \gamma_n = \beta$ di β da $\Gamma \cup \{\alpha\}$, si costruisce per induzione completa sulla lunghezza n una derivazione di $\alpha \rightarrow \gamma_i$: se γ_i è un assioma o un'ipotesi in Γ , si usa γ_i insieme ad A_k e modus ponens; se $\gamma_i = \alpha$, si usa $\vdash \alpha \rightarrow \alpha$; se γ_i è ottenuta per modus ponens da γ_j e $\gamma_j \rightarrow \gamma_i$, da $\alpha \rightarrow \gamma_j$ e $\alpha \rightarrow (\gamma_j \rightarrow \gamma_i)$ e da A_S si ottiene $\alpha \rightarrow \gamma_i$. Il teorema permette di accorciare le derivazioni: per dimostrare $\alpha \rightarrow \beta$ basta assumere α .

RISULTATO	ENUNCIATO
Prop. 3.2	$\vdash \neg\alpha \rightarrow (\alpha \rightarrow \beta)$
Prop. 3.3 (transitività)	$\alpha \rightarrow \beta, \beta \rightarrow \gamma \vdash \alpha \rightarrow \gamma$
Prop. 3.6	$\neg\neg\alpha \vdash \alpha$
Prop. 3.10 (contrapposizione)	$\alpha \rightarrow \beta \vdash \neg\beta \rightarrow \neg\alpha$

CONSISTENZA

Γ è contraddittorio se e solo se esiste α con $\Gamma \vdash \alpha$ e $\Gamma \vdash \neg\alpha$ (Teorema 3.2). Se Γ è consistente e $\Gamma \not\vdash \alpha$, allora $\Gamma \cup \{\neg\alpha\}$ è consistente (Teorema 3.3).

Semantica e completezza di P0

DEFINIZIONE

Valutazione. Un assegnamento proposizionale B associa a ogni variabile un valore 0 o 1; si estende a una valutazione B^* con $B^*(\alpha \rightarrow \beta) = 0$ se e solo se $B^*(\alpha) = 1$ e $B^*(\beta) = 0$, e $B^*(\neg\alpha) = 1 - B^*(\alpha)$. Una fbf è una **tautologia** se $B^*(\alpha) = 1$ per ogni B ; è **soddisfacibile** se esiste B con $B^*(\alpha) = 1$; è **contraddittoria** se non è soddisfacibile. $\Gamma \models \alpha$ (α è conseguenza tautologica di Γ) se ogni B che soddisfa tutte le fbf di Γ soddisfa α .

Una fbf α è una tautologia se e solo se $\neg\alpha$ è contraddittoria; essere una tautologia è decidibile con la tabella di verità, perché la valutazione dipende solo dalle variabili presenti.

α	β	$\alpha \rightarrow \beta$	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$
0	0	1	1	0	0
0	1	1	1	0	1
1	0	0	0	0	1
1	1	1	0	1	1

CORRETTEZZA

Se $\Gamma \vdash \alpha$ allora $\Gamma \models \alpha$. Si procede per induzione sulla lunghezza della derivazione: gli assiomi sono tautologie (si verifica con tabelle di verità), un'ipotesi in Γ è conseguenza di Γ , e se α è ottenuta per modus ponens da $\beta \rightarrow \alpha$ e β , con $\Gamma \models \beta \rightarrow \alpha$ e $\Gamma \models \beta$, ogni assegnamento che soddisfa Γ dà valore 1 a β e a $\beta \rightarrow \alpha$, perciò anche ad α .

Corollario: P0 è consistente, perché α e $\neg\alpha$ non possono essere entrambe tautologie.

COMPLETEZZA

Se $\Gamma \models \alpha$ allora $\Gamma \vdash \alpha$. Si dimostra per contrapposizione, in tre passi.

- Insiemi consistenti massimali.** Un insieme è consistente massimale se è consistente e nessuna estensione propria lo è. Vale il lemma di Lindenbaum: ogni insieme consistente Γ è contenuto in un insieme consistente massimale Γ^* , calcolabile effettivamente. Si enumerano tutte le fbf $\alpha_1, \alpha_2, \dots$, si pone $\Gamma_0 = \Gamma$ e $\Gamma_{n+1} = \Gamma_n \cup \{\alpha_n\}$ se questo è consistente, altrimenti $\Gamma_{n+1} = \Gamma_n$; Γ^* è l'unione.
- Proprietà.** Un insieme consistente massimale Δ è una teoria; per ogni α si ha $\alpha \in \Delta$ se e solo se $\neg\alpha \notin \Delta$; e $\alpha \rightarrow \beta \in \Delta$ se e solo se da $\alpha \in \Delta$ segue $\beta \in \Delta$.
- Lemma di soddisfacibilità.** Ogni insieme consistente è soddisfacibile: si estende a Δ massimale e si pone $B(p) = 1$ se e solo se $p \in \Delta$; per induzione sulla struttura, $B^*(\alpha) = 1$ se e solo se $\alpha \in \Delta$.

Se $\Gamma \not\models \alpha$, allora $\Gamma \cup \{\neg\alpha\}$ è consistente (Teorema 3.3), quindi soddisfacibile: esiste B che soddisfa Γ e dà a α valore 0, e $\Gamma \not\models \alpha$. Dunque $\Gamma \models \alpha$ se e solo se $\Gamma \vdash \alpha$; in particolare i teoremi di P0 sono tutte e sole le tautologie e il problema di decidere se una fbf è un teorema (Entscheidungsproblem) è decidibile.

Deduzione naturale

Le prove si possono rappresentare in altro modo che con una sequenza di fbf: la deduzione naturale usa alberi, non ha assiomi, e usa regole di introduzione ed eliminazione per ogni connettivo, più il falso $\perp$, che rappresenta l'assurdo. Una regola può scaricare ipotesi, scritte tra parentesi quadre con un indice.

CONNETTIVO	INTRODUZIONE	ELIMINAZIONE
$\wedge$	($\wedge I$) da P e Q segue $P \wedge Q$	($\wedge E$) da $P \wedge Q$ segue P , e da $P \wedge Q$ segue Q
$\vee$	($\vee I$) da P segue $P \vee Q$, e da Q segue $P \vee Q$	($\vee E$) da $P \vee Q$, da una derivazione di R con ipotesi P e da una di R con ipotesi Q segue R , scaricando P e Q
$\rightarrow$	($\rightarrow I$) se da P (scaricata) si deriva Q , segue $P \rightarrow Q$	($\rightarrow E$) modus ponens: da P e $P \rightarrow Q$ segue Q
$\neg$	($\neg I$) se da P (scaricata) si deriva $\perp$, segue $\neg P$	($\neg E$) da P e $\neg P$ segue $\perp$
$\perp$	($\perp I$) da $\perp$ segue qualunque P	($\perp E$) se da $\neg P$ (scaricata) si deriva $\perp$, segue P

Un albero di derivazione di P da Γ è definito per induzione: una ipotesi P è un albero di P da $\{P\}$; applicando una regola agli alberi delle premesse si ottiene un albero della conclusione; nelle regole con scarico, tutte le

occorrenze dell'ipotesi indicata, nel sottoalbero, si dichiarano scaricate. Si scrive $P_1, \dots, P_n \vdash Q$ se esiste un albero di conclusione Q con ipotesi non scaricate tra $P_1, \dots, P_n$, e $\vdash B$ se tutte sono scaricate: B è un teorema, o legge logica. Se $\Gamma \vdash Q$ allora $\Gamma \models Q$ (teorema di validità), per cui è inutile cercare un albero quando la conseguenza non vale.

COME SI COSTRUISCE UNA DERIVAZIONE

Si parte dalla conclusione e si procede dal basso verso l'alto: se è composta si prova la regola di introduzione del suo connettivo principale e si riconduce il problema a un problema più semplice con eventuali ipotesi in più; se è atomica o la regola non è applicabile si usano le regole di eliminazione sulle premesse. Per una disgiunzione tra le premesse si ragiona per casi con $\vee E$; per una conclusione che è una disgiunzione, in genere non si usa $\vee I$. La regola RAA non è né di introduzione né di eliminazione e non segue questa strategia.

ESEMPIO	COME SI OTTIENE
$\vdash A \rightarrow A$	da A si applica $\rightarrow I$ scaricando A
$A \rightarrow B, A \rightarrow C \vdash A \rightarrow (B \wedge C)$	$\rightarrow I$ con ipotesi A ; da A e $A \rightarrow B$ si ottiene B , da A e $A \rightarrow C$ si ottiene C ; $\wedge I$ e $\rightarrow I$ scaricando A
$A \rightarrow C, B \rightarrow C \vdash (A \vee B) \rightarrow C$	$\rightarrow I$ con ipotesi $A \vee B$; $\vee E$ con i due casi A e B e $\rightarrow E$ sulle premesse; scarico di $A, B, A \vee B$
$\vdash (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$	ipotesi $A \rightarrow B, \neg B, A$: $\rightarrow E$ dà B , $\neg E$ dà $\perp$, $\neg I$ scarica A , $\rightarrow I$ scarica $\neg B$ e $A \rightarrow B$
$\neg\neg A \vdash \neg A$	ipotesi A : con l'ipotesi $\neg A$, $\neg E$ dà $\perp$ e $\neg I$ scarica $\neg A$ dando $\neg\neg A$; da $\neg\neg A$ e $\neg\neg A$, $\neg E$ dà $\perp$; $\neg I$ scarica A dando $\neg A$
$\vdash (\neg\neg A) \rightarrow A$	RAA scaricando $\neg A$: da $\neg A$ e $\neg\neg A$ segue $\perp$; $\rightarrow I$
$\vdash A \vee \neg A$ (terzo escluso)	si assume $\neg(A \vee \neg A)$: con A si ottiene $A \vee \neg A$, quindi $\perp$, e $\neg I$ dà $\neg A$; con $\neg A$ si ottiene $\perp$ e $\neg I$ dà $\neg\neg A$; $\neg E$ su $\neg A$ e $\neg\neg A$ dà $\perp$; RAA scarica l'ipotesi e dà $A \vee \neg A$

Corrispondenza tra deduzioni e programmi

Si consideri il frammento di deduzione naturale con solo $\rightarrow I$ e $\rightarrow E$. A ogni sottoderivazione si associa un λ termine, scritto $M : \sigma$ per indicare che M è associato a una derivazione della fbf σ .

ELEMENTO DELLA DEDUZIONE	TERMINE ASSOCIATO
Ipotesi	una variabile; la stessa variabile non può essere associata a ipotesi diverse
$(\rightarrow E)$ da $M : \sigma \rightarrow \tau$ e $N : \sigma$	$(M N) : \tau$
$(\rightarrow I)$ scaricando $x : \sigma$ da $M : \tau$	$\lambda x.M : \sigma \rightarrow \tau$

Per esempio la deduzione di $(\sigma \rightarrow (\tau \rightarrow \gamma)) \rightarrow ((\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \gamma))$ è associata a $\lambda x.\lambda y.\lambda z.xz(yz)$; le ipotesi non scaricate corrispondono a variabili libere.

Un $\rightarrow I$ immediatamente seguito da $\rightarrow E$ è un giro inutile: si evita mettendo la sottoderivazione D di σ al posto dell'ipotesi. Il termine corrispondente è $(\lambda x.M)N$, un redex β , e la semplificazione corrisponde alla sostituzione $M[N/x]$. Computare, cioè β ridurre, equivale dunque a semplificare una dimostrazione. Lo stesso vale per il frammento implicativo di PO e CL: A_k corrisponde a k , A_S a s e il modus ponens all'applicazione; $\alpha \rightarrow \alpha$ corrisponde a $((sk)k)$, il programma dell'identità, perché $skkP$ si riduce a P .

Logica dei predicati e formalizzazione di teorie

La logica proposizionale non esprime relazioni tra oggetti, né esistenza e universalità. Il calcolo dei predicati aggiunge termini, predicati e quantificatori, e permette di formalizzare teorie matematiche come l'aritmetica e la teoria dei gruppi.

Il linguaggio dei predicati

DEFINIZIONE

Segnatura. Una segnatura Σ è una coppia formata da un insieme di simboli di funzione e un insieme di simboli di predicato, ciascuno con la propria arità; si scrive f^n , P^n . Un simbolo di costante è un simbolo di funzione 0 aria. Esempi: la segnatura dell'ordinamento parziale ha il solo predicato $<$; quella dell'aritmetica ha 0, il successore s , la somma e il prodotto e l'uguaglianza.

DEFINIZIONE

Termini e formule. I termini su Σ sono: le variabili individuali $x, y, \dots$; $f^n(t_1, \dots, t_n)$ con $t_1, \dots, t_n$ termini. Le fbf sono: $P^n(t_1, \dots, t_n)$ (atomiche); $t_1 = t_2$ se il linguaggio è con uguaglianza; $\neg\alpha$ e $\alpha \rightarrow \beta$; $\forall x\alpha$. L'esistenziale è una definizione esplicita: $\exists x\alpha = \neg\forall x\neg\alpha$. La portata di $\forall x$ è la fbf α .

Per esempio $\neg(<(x, x))$ e $\forall x\forall y\forall z(x < y \wedge y < z \rightarrow x < z)$ sono fbf nella segnatura dell'ordinamento, e $\forall x \text{pop}(\text{push}_0(x)) = x$ nella segnatura delle pile. Una fbf può non avere significato intuitivo, come $\text{pop}(\text{empty}) = \text{empty}$: ogni fbf riceve comunque una denotazione, perché non si hanno predicati parziali.

Variabili libere e sostituzione

Per una fbf α , $FV(\alpha)$ è l'insieme delle variabili libere e $BV(\alpha)$ quello delle legate. $FV(P(t_1, \dots, t_n)) = \cup V(t_i)$, $FV(\neg\alpha) = FV(\alpha)$, $FV(\alpha \rightarrow \beta) = FV(\alpha) \cup FV(\beta)$, $FV(\forall x\alpha) = FV(\alpha)$ meno $\{x\}$; $BV(\forall x\alpha) = BV(\alpha) \cup \{x\}$. Una fbf con $FV(\alpha) = \emptyset$ è chiusa. Le fbf $\forall xP(x)$ e $\forall zP(z)$ sono sintatticamente diverse ma equivalenti: il nome di una variabile legata è irrilevante.

Un termine t è **libero per x in α** se $x \notin FV(\alpha)$ oppure nessuna occorrenza libera di x in α sta nella portata di un quantificatore $\forall y$ con $y \in V(t)$. La sostituzione $[t/x]\alpha$ si definisce per induzione; per i quantificatori:

CASO	RISULTATO DI $[t/x](\forall y\alpha)$
$x \notin FV(\forall y\alpha)$	$\forall y\alpha$
$x \in FV(\forall y\alpha)$ e $y \notin FV(t)$	$\forall y([t/x]\alpha)$
$x \in FV(\forall y\alpha)$ e $y \in FV(t)$	$[t/x](\forall z([z/y]\alpha))$, con z la prima variabile che non compare in t , libera per y in α e non libera in α

L'ultimo caso risolve il problema della cattura con una ridenominazione preliminare. Se y non è presente in α , $[x/y][y/x]\alpha = \alpha$. La sostituzione si usa nelle regole sui quantificatori, come in λ calcolo (capitolo 6).

Semantica

DEFINIZIONE

Struttura e ambiente. Una struttura A per Σ è una tripla $\langle |A|, F, P \rangle$: un insieme non vuoto $|A|$, detto supporto, una funzione $f^A : |A|^n \rightarrow |A|$ per ogni simbolo f^n e una relazione $P^A : |A|^n \rightarrow \{0, 1\}$ per ogni P^n . Un ambiente è una funzione ρ che associa a ogni variabile un elemento di $|A|$.

DEFINIZIONE

Interpretazione. $[[x]]_\rho = \rho(x)$ e $[[f(t_1, \dots, t_n)]]_\rho = f^A([[t_1]]_\rho, \dots, [[t_n]]_\rho)$. Per le fbf: $[[P(t_1, \dots, t_n)]]_\rho = P^A([[t_1]]_\rho, \dots, [[t_n]]_\rho)$; $[[\alpha \rightarrow \beta]]_\rho = 1$ se e solo se da $[[\alpha]]_\rho = 1$ segue $[[\beta]]_\rho = 1$; $[[\neg\alpha]]_\rho = 1$ se e solo se $[[\alpha]]_\rho = 0$; $[[\forall x\alpha]]_\rho = 1$ se e solo se per ogni $a \in |A|$ vale $[[\alpha]]_{\rho[a/x]} = 1$, dove $\rho[a/x]$ coincide con ρ salvo che in x , dove vale a ; $[[t' = t'']]_\rho = 1$ se e solo se $[[t']]_\rho$ e $[[t'']]_\rho$ sono lo stesso elemento.

Una fbf α è **soddisfacibile in A** se esiste ρ con $[[\alpha]]_\rho = 1$; è **vera** (valida) in A , e si scrive $A \models \alpha$, se $[[\alpha]]_\rho = 1$ per ogni ρ ; è **soddisfacibile** se lo è in qualche struttura; è **valida** se è vera in ogni struttura; è **contraddittoria** se non è soddisfacibile. Una struttura è un modello di un insieme Γ se ogni fbf di Γ è vera in essa. Si ha $A \models \exists x\alpha$ se e solo se per ogni ρ esiste $a \in |A|$ con $[[\alpha]]_{\rho[a/x]} = 1$ (a è il testimone). Due lemmi tecnici: il lemma di coincidenza afferma che il valore di una fbf dipende solo dalle sue variabili libere (quindi una formula chiusa soddisfacibile in A è vera in A); il lemma di sostituzione afferma che $[[[t/x]\alpha]]_\rho = [[\alpha]]_{\rho[t/x]}$.

DEFINIZIONE

Conseguenza logica. $\Gamma \models \alpha$ se per ogni struttura A e ogni ambiente ρ , da $[[\gamma]]_\rho = 1$ per ogni $\gamma \in \Gamma$ segue $[[\alpha]]_\rho = 1$. Si dice che α è valida in Γ se per ogni struttura A , dal fatto che ogni $\gamma \in \Gamma$ è vera in A segue che α è vera in A .

Se $\Gamma \models \alpha$, α è valida in Γ , ma non vale il viceversa: sia $\Gamma = \{x = 0\}$ e $\alpha = \forall x(x = 0)$. L'insieme Γ è vero in A solo se $|A|$ ha il solo elemento 0^A , quindi α è valida in Γ ; ma $\Gamma \not\models \alpha$, perché in $\mathbb{N}$ con $\rho(x) = 0$ la prima è vera e la seconda falsa.

Il sistema formale PL

DEFINIZIONE

PL. Gli assiomi sono gli schemi di P0 ($A_k, A_S, A_{\neg}$), più: (AV) $\alpha \rightarrow \forall x\alpha$ se $x \notin FV(\alpha)$; (SPEC) $\forall x\alpha \rightarrow [t/x]\alpha$ se t è libero per x in α ; ($\forall \rightarrow$) $(\forall x(\alpha \rightarrow \beta)) \rightarrow ((\forall x\alpha) \rightarrow (\forall x\beta))$; se il linguaggio è con uguaglianza (E0) $t = t$, (E1) $t_1 = t_1' \rightarrow (\dots \rightarrow f(t_1, \dots, t_n) = f(t_1', \dots, t_n'))$, (E2) lo stesso per ogni predicato P ; e tutte le generalizzazioni dei precedenti assiomi, $\forall x_1 \dots \forall x_k \alpha$ (GEN). La sola regola è il modus ponens.

PL è la teoria che formalizza la conseguenza logica. Valgono il teorema di deduzione (se $\Gamma, \alpha \vdash \beta$ allora $\Gamma \vdash \alpha \rightarrow \beta$) e la **regola di generalizzazione**: se $\Gamma \vdash \alpha$ e $x \notin FV(\Gamma)$ oppure $x \notin FV(\alpha)$, allora $\Gamma \vdash \forall x\alpha$. Vale inoltre la legge di ridenominazione delle variabili legate: se $y \notin FV(\forall x\alpha)$, $\vdash \forall x\alpha \rightarrow \forall y[y/x]\alpha$. L'uguaglianza è riflessiva, simmetrica e transitiva. La traduzione che cancella i quantificatori e i termini, ponendo $(P(t_1, \dots, t_n))^* = P$ e $(\forall x\alpha)^* = \alpha^*$, manda i teoremi di PL in teoremi di P0, perciò PL è consistente ed è un'estensione conservativa di P0: se α è nel linguaggio di P0 e $\vdash_{PL} \alpha$, allora $\vdash_{P0} \alpha$.

Teoremi fondamentali

TEOREMA	ENUNCIATO	IDEA DELLA DIMOSTRAZIONE
Correttezza	se $\Gamma \vdash \alpha$ allora $\Gamma \models \alpha$	induzione sulla derivazione; per SPEC si usa il lemma di sostituzione, per AV il lemma di coincidenza
Completezza	se $\Gamma \models \alpha$ allora $\Gamma \vdash \alpha$	per contrapposizione: se $\Gamma \not\models \alpha$, $\Gamma \cup \{\neg\alpha\}$ è consistente, quindi soddisfacibile
Compattezza	Γ è soddisfacibile se e solo se ogni sottoinsieme finito di Γ è soddisfacibile	se Γ è insoddisfacibile è inconsistente: una contraddizione usa un numero finito di ipotesi
Enumerabilità	l'insieme delle fbf valide è ricorsivamente enumerabile	si enumerano i teoremi applicando il modus ponens agli assiomi

La parte difficile della completezza è il **lemma di soddisfacibilità**: ogni insieme consistente è soddisfacibile. Si procede in tre passi: (lemma di Henkin) si aggiungono nuovi simboli di costante c_n e per ogni formula esistenziale $\exists x \alpha_n$ la fbf $\exists x \alpha_n \rightarrow [c_n/x]\alpha_n$, ottenendo un insieme consistente con testimoni; (lemma di Lindenbaum) lo si estende a un insieme T consistente massimale che ha ancora testimoni; (modello dei termini) si costruisce la struttura il cui supporto è l'insieme dei termini chiusi quozientato dalla relazione $t \approx s$ se e solo se $t = s \in T$, con $f^A([t_1], \dots, [t_n]) = [f(t_1, \dots, t_n)]$ e P^A vero se e solo se $P(t_1, \dots, t_n) \in T$; per induzione su α si prova $[[\alpha]] = 1$ se e solo se $\alpha \in T$. Il caso $\forall x$ usa i testimoni di Henkin.

Per la compattezza esiste un modello dell'insieme $\Gamma = \{x > 0\} \cup \{x \leq 1/n \mid n \geq 1\}$: ogni sottoinsieme finito è soddisfacibile nei reali, ma il modello garantito dal teorema non è $\mathbb{R}$, ed è alla base dell'analisi non standard di Robinson. Si dimostra che l'insieme delle formule valide non è ricorsivo, come stabilisce il teorema di Church: l'indcidibilità del calcolo dei predicati contrasta con la decidibilità di P0.

Deduzione naturale per i quantificatori

Alle regole proposizionali (capitolo 9), con la RAA, si aggiungono quelle sui quantificatori; A, B sono fbf del primo ordine.

REGOLA	ENUNCIATO	CONDIZIONE
($\forall$ I)	da A segue $\forall xA$	x non occorre libera nelle ipotesi non scaricate della sottoderivazione di A
($\forall$ E)	da $\forall xA$ segue $A[t/x]$	t libero per x in A
($\exists$ I)	da $A[t/x]$ segue $\exists xA$	t libero per x in A
($\exists$ E)	da $\exists xA$ e da una derivazione di B con ipotesi A (scaricata) segue B	x non libera in B e non libera in alcuna ipotesi non scaricata diversa da A

Per l'uguaglianza: (RIF) $x = x$; e, per ogni termine t e ogni fbf A, da $x_1 = y_1, \dots, x_n = y_n$ si deduce $t[x/z] = t[y/z]$ e $A[x/z] \rightarrow A[y/z]$. Da queste si derivano simmetria (da $x = y$ segue $y = x$) e transitività (da $x = y$ e $y = z$ segue $x = z$). Un albero è un albero di derivazione, e P derivabile da Γ si scrive $\Gamma \vdash P$. Poiché un albero è finito, vale il teorema di finitezza: se $\Gamma \vdash P$ esiste $\Gamma_0 \subseteq \Gamma$ finito con $\Gamma_0 \vdash P$.

DERIVAZIONI ERRATE TIPICHE

- **$\forall I$ con ipotesi libera.** Da $[x = 0]$ si ottiene $\forall x(x = 0)$ con $\forall I$, e poi con $\rightarrow I$ la fbf $x = 0 \rightarrow \forall x(x = 0)$: è errato, perché al momento di $\forall I$ la variabile x è libera nell'ipotesi non ancora scaricata $x = 0$.
- **$\exists E$ con variabile libera nella conclusione.** Da $\exists x(x = 0)$ e dall'ipotesi $x = 0$ si ottiene $x + 0 = 0$; applicare $\exists E$ per concludere $x + 0 = 0$ e poi $\forall I$ per ottenere $\forall x(x + 0 = 0)$ è errato, perché x è libera nella conclusione di $\exists E$.
- **$\forall I$ dopo $\exists E$.** Da $\exists x\varphi$ e dalle ipotesi φ e $\neg\varphi$ si ottiene $\perp$ e, con $\exists E$, $\perp$; con RAA si ottiene φ , ma non si può passare a $\forall x\varphi$ con $\forall I$, perché x è libera nell'ipotesi non scaricata $\neg\varphi$.

LEGGI LOGICHE

Per ogni α e β valgono, derivabili in deduzione naturale:

EQUIVALENZA	CONDIZIONE
$\neg\forall x\alpha \leftrightarrow \exists x\neg\alpha$	nessuna
$\neg\exists x\alpha \leftrightarrow \forall x\neg\alpha$	nessuna
$\forall x\forall y\alpha \leftrightarrow \forall y\forall x\alpha$ e $\exists x\exists y\alpha \leftrightarrow \exists y\exists x\alpha$	nessuna
$\forall x(\alpha \wedge \beta) \leftrightarrow \forall x\alpha \wedge \forall x\beta$	nessuna
$\exists x(\alpha \vee \beta) \leftrightarrow \exists x\alpha \vee \exists x\beta$	nessuna
$\forall x(\alpha \vee \beta) \leftrightarrow \forall x\alpha \vee \beta$	$x \notin FV(\beta)$
$\exists x(\alpha \wedge \beta) \leftrightarrow \exists x\alpha \wedge \beta$	$x \notin FV(\beta)$
$\forall x(\alpha \rightarrow \beta) \leftrightarrow (\exists x\alpha) \rightarrow \beta$	$x \notin FV(\beta)$
$\exists x(\alpha \rightarrow \beta) \leftrightarrow (\forall x\alpha) \rightarrow \beta$	$x \notin FV(\beta)$
$(\beta \rightarrow \forall x\alpha) \leftrightarrow \forall x(\beta \rightarrow \alpha)$	$x \notin FV(\beta)$
$(\beta \rightarrow \exists x\alpha) \leftrightarrow \exists x(\beta \rightarrow \alpha)$	$x \notin FV(\beta)$

Esempi di leggi con più quantificatori: $\vdash \exists x(A(x) \rightarrow \forall yA(y))$, $\exists x\forall yA \vdash \forall y\exists xA$, $\forall x(A(x) \rightarrow B(x))$, $\exists x\neg B(x) \vdash \exists x\neg A(x)$. La formula $\exists x\forall y(R(x, y) \leftrightarrow \neg R(y, y))$ è contraddittoria: $\vdash \neg\exists x\forall y(R(x, y) \leftrightarrow \neg R(y, y))$.

Formalizzare teorie

Una teoria si formalizza scegliendo una segnatura e un insieme di formule, gli assiomi; i suoi modelli sono le strutture in cui tutti gli assiomi sono veri.

ORDINAMENTI

Il predicato $\leq$ permette di esprimere: l'esistenza dell'elemento minimo $\exists x\forall y(x \leq y)$; l'ordine totale $\forall xy(x \leq y \vee y \leq x)$; con $x < y$ come abbreviazione di $x \leq y \wedge \neg(x = y)$, la densità $\forall xy(x < y \rightarrow \exists z(x < z \wedge z < y))$.

GRUPPI

Segnatura: costante e , funzione unaria inversa $()^{-1}$, funzione binaria composizione $\cdot$, uguaglianza. Un gruppo è un modello degli assiomi:

ASSIOMA	FORMULA
Associatività	$\forall xyz((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Elemento neutro	$\forall x(x \cdot e = x \wedge e \cdot x = x)$
Inverso	$\forall x(x \cdot x^{-1} = e \wedge x^{-1} \cdot x = e)$
Gruppo abeliano, in più	$\forall xy(x \cdot y = y \cdot x)$

L'unicità dell'elemento neutro si esprime con $\forall x(\forall y(y \cdot x = y \wedge x \cdot y = y) \rightarrow x = e)$. In generale l'esistenza e unicità di x con la proprietà P , indicata con $\exists!xP$, si formalizza con $\exists x(P(x) \wedge \forall z(P(z) \rightarrow x = z))$.

ARITMETICA DI PEANO

La segnatura ha la costante 0, il successore s , la somma $+$ e il prodotto $*$, con l'uguaglianza. Con PA si indica l'insieme di assiomi:

ASSIOMA	FORMULA
1	$\forall x \neg(0 = s(x))$
2	$\forall xy(s(x) = s(y) \rightarrow x = y)$
3	$\forall x(x + 0 = x)$
4	$\forall xy(x + s(y) = s(x + y))$
5	$\forall x(x * 0 = 0)$
6	$\forall xy(x * s(y) = x + (x * y))$
7, induzione	$(P(0) \wedge \forall x(P(x) \rightarrow P(s(x)))) \rightarrow \forall xP(x)$

L'assioma 7 è uno schema: $P(x)$ è un qualunque predicato definibile nel linguaggio con la sola variabile libera x . L'interpretazione intesa è il modello standard, con supporto $\mathbb{N}$, 0, successore, somma e prodotto.

DA RICORDARE

Per dimostrare un'affermazione Q sull'aritmetica occorre mostrare $PA \vdash Q$, cioè Q conseguenza logica di PA, vera in tutti i modelli di PA. Non basta $\vdash s(0) + s(s(0)) = s(s(s(0)))$: per la correttezza questa formula dovrebbe essere valida, ma nella struttura con supporto $\{T, F\}$, 0 interpretato in F, s l'identità e $+$ la funzione costantemente T, essa è falsa. Nulla nella logica obbliga i simboli a comportarsi come intendiamo.

Il modello standard non è l'unico modello di PA: per compattezza, nessuna aggiunta di formule del primo ordine a PA lo rende l'unico (esistono modelli non standard). Inoltre i teoremi di incompletezza di Gödel affermano, in sintesi, che ogni teoria consistente, ricorsivamente assiomatizzata e abbastanza ricca da contenere l'aritmetica è incompleta: esiste una fbf chiusa α tale che né α né $\neg\alpha$ è derivabile; e che la consistenza di una tale teoria non è dimostrabile al suo interno. Per questo PA non è completa, a differenza della logica proposizionale.

INSIEMI E PARADOSSO DI RUSSELL

Per formalizzare la nozione di insieme con un predicato binario $\in$, lo schema di comprensione $\exists z \forall x(x \in z \leftrightarrow P(x))$, con P qualunque predicato, appare naturale. Ma per $P(x) = \neg(x \in x)$ la formula è insoddisfacibile: l'insieme $W = \{x \mid x \notin x\}$ soddisferebbe $W \in W \leftrightarrow W \notin W$, una contraddizione (paradosso di Russell, 1902). Ogni assiomatizzazione della teoria degli insiemi deve quindi evitare lo schema nella sua forma generale.

Induzione e ricorsione

Il principio di induzione permette di dimostrare proprietà di infiniti elementi riducendole a casi più semplici. Alla stessa idea corrisponde, sul piano del calcolo, la definizione di funzioni per ricorsione: è per questo che l'induzione serve a verificare i programmi.

Induzione matematica

Una buona definizione ricorsiva ha casi base, in cui la funzione è definita direttamente, e casi ricorsivi, in cui il calcolo si riconduce a quello su argomenti più semplici. La nozione di essere più semplice induce un ordinamento sul dominio, e permette di dimostrare proprietà della funzione. Su $\mathbb{N}$ ciò si formalizza così.

DEFINIZIONE

Principio di induzione. Sia φ una proprietà sui naturali e $n_0 \in \mathbb{N}$. Se $\varphi(n_0)$ vale e per ogni $n \geq n_0$, $\varphi(n)$ implica $\varphi(n+1)$, allora $\varphi(n)$ vale per ogni $n \geq n_0$. In simboli: se $\varphi(n_0)$ e $\forall n \geq n_0 (\varphi(n) \rightarrow \varphi(n+1))$, allora $\forall n \geq n_0, \varphi(n)$.

La dimostrazione si divide in **caso base**, che si verifica direttamente, e **caso induttivo**, in cui si fissa un n generico, si assume $\varphi(n)$ (ipotesi induttiva) e si prova $\varphi(n+1)$.

ESEMPLI

- **Somma dei primi n dispari.** $\varphi(n) = "1 + 3 + \dots + (2n-1) = n^2"$. Caso base: $\varphi(0)$ vale, poiché la somma vuota è $0 = 0^2$. Passo: $1 + \dots + (2n-1) + (2n+1) = n^2 + 2n + 1 = (n+1)^2$, usando l'ipotesi induttiva.
- **Crescita esponenziale.** $\varphi(n) = "2^n \geq n + 1"$. Base: $2^0 = 1 \geq 1$. Passo: $2^{n+1} = 2 \cdot 2^n \geq 2n + 2 \geq n + 2$.

ERRORI TIPICI

Una dimostrazione per induzione è valida solo se il caso base e il passo sono corretti per un n generico, con $n \geq n_0$.

ARGOMENTO SBAGLIATO	ERRORE
Tutte le persone hanno lo stesso colore di occhi: base $n = 1$, passo da n a $n+1$ togliendo una persona alla volta	nel passo, per concludere che s_1 e s_2 hanno lo stesso colore, S meno $\{s_1\}$ e S meno $\{s_2\}$ devono avere un elemento in comune, quindi $n \geq 2$; il passo vale solo da $n \geq 2$ e il caso base significativo è $n = 2$, che è falso
Per ogni $n \geq 1$ vale $a^{n-1} = 1$, con base $a^0 = 1$ e passo $a^n = a^{n-1} \cdot a^{n-1} / a^{n-2}$	il passo usa $a^{n-2} = 1$, cioè l'ipotesi sul caso $n-1$, che esiste solo se $n \geq 2$: il passo da 1 a 2 non è coperto e il caso $n = 2$ è falso se $a \neq 1$, perché $a^1 = a$

Induzione completa

In alcuni casi l'ipotesi su n non basta a provare $n+1$: per esempio la fattorizzazione di $n+1$ in primi non ha relazione con quella di n . Si usa allora un'ipotesi induttiva più forte.

DEFINIZIONE

Induzione completa. Sia φ una proprietà sui naturali. Se $\varphi(n_0)$ e per ogni $n > n_0$ vale $(\forall m, n_0 \leq m < n. \varphi(m)) \rightarrow \varphi(n)$, allora $\varphi(n)$ vale per ogni $n \geq n_0$. L'ipotesi induttiva è dunque: φ vale per tutti i numeri minori di n .

I due principi sono equivalenti (dimostrabile uno dall'altro), ma l'induzione completa è spesso più comoda.

Esempio: ogni $n \geq 2$ è un prodotto di numeri primi. Base: 2 è primo. Passo: se n è primo, vale; altrimenti $n = a \cdot b$ con $2 \leq a, b < n$, e per ipotesi induttiva a e b sono prodotto di primi, quindi lo è n .

Analogamente, una definizione ricorsiva che usa i valori su numeri minori di n , non soltanto su $n - 1$, corrisponde all'induzione completa.

Insiemi con precedenza e induzione ben fondata

I principi precedenti si generalizzano ad altri insiemi, come liste e alberi, e ad altre relazioni di precedenza, non solo l'ordinamento dei naturali.

DEFINIZIONE

Insieme con precedenza. Un insieme con precedenza è una coppia $(S, \triangleleft)$ dove $\triangleleft \subseteq S \times S$ è una relazione; $x \triangleleft y$ si legge "x precede y", e x è predecessore immediato di y . Un elemento x è **minimale** se non esiste $y \in S$ con $y \triangleleft x$. È **minimo** se $x \triangleleft^* y$ per ogni y ; un minimo è minimale, ma non vale il viceversa: il minimo è al più uno, i minimali possono essere molti.

Una **catena discendente** è una successione, anche infinita, $s_0 \triangleright s_1 \triangleright s_2 \triangleright \dots$ in cui s_{i+1} è predecessore immediato di s_i . Un insieme $(S, \triangleleft)$ è **ben fondato** se non ammette catene discendenti infinite; equivalentemente, ogni sottoinsieme non vuoto contiene un elemento minimale. La buona fondatezza dipende dalla relazione, non solo dall'insieme.

INSIEME CON PRECEDENZA	BEN FONDATO?	ELEMENTI MINIMALI
$\mathbb{N}, x \triangleleft y$ se e solo se $y = x + 1$	sì	solo 0
$\mathbb{Z}, x \triangleleft y$ se e solo se $y = x + 1$	no	nessuno: la catena $0, -1, -2, \dots$ è discendente e infinita
$\mathbb{N}, x \triangleleft y$ se e solo se $x = y + 1$	no	nessuno: la catena $0, 1, 2, \dots$ è discendente e infinita
$\mathbb{N}$ meno $\{0\}$, $x \triangleleft y$ se e solo se x divide y e $x \neq y$	sì	solo 1
$\mathbb{Z}, x \triangleleft y$ se e solo se $ x < y $	sì	solo 0, perché ogni catena discendente fa decrescere $ x $ in $\mathbb{N}$

I diagrammi di precedenza rappresentano ciascun elemento come punto e ogni predecessore immediato con una freccia o una linea, letta dal basso verso l'alto: servono a verificare a vista la buona fondatezza.

DEFINIZIONE

Induzione ben fondata. Sia $(S, \triangleleft)$ ben fondato e ϕ una proprietà su S . Se per ogni $x \in S$ vale $(\forall y \triangleleft x. \phi(y)) \rightarrow \phi(x)$, allora $\phi(x)$ vale per ogni $x \in S$.

Il passo si divide in **caso base**, che per un x minimale non offre ipotesi induttive, e **caso induttivo**, per x non minimale, in cui si assume $\phi(y)$ per tutti i predecessori di x . Dimostrazione: se esistesse un x con $\phi(x)$ falsa, l'insieme dei controesempi sarebbe non vuoto e avrebbe un elemento minimale x_0 ; i suoi predecessori soddisfano ϕ , quindi per l'ipotesi del principio $\phi(x_0)$ vale, assurdo. L'induzione matematica è il caso $\mathbb{N}$ con $y \triangleleft x$ se e solo se $x = y + 1$; quella completa è il caso della chiusura transitiva.

COSTRUIRE PRECEDENZE BEN FONDATE

- **Immagine inversa.** Se $(S, \triangleleft)$ è ben fondato e $f : X \rightarrow S$, allora $x_1 \triangleleft x_2$ se e solo se $f(x_1) \triangleleft f(x_2)$ rende ben fondato X (per esempio la lunghezza di una lista).
- **Prodotto.** Su $S_1 \times S_2$: (s_1, s_2) precede (s_1', s_2') se $s_1 \triangleleft_1 s_1'$ e $s_2 \triangleleft_2 s_2'$ è ben fondato.
- **Lessicografica.** Su $S_1 \times S_2$: $(s_1, s_2) \triangleleft (s_1', s_2')$ se $s_1 \triangleleft_1 s_1'$, oppure $s_1 = s_1'$ e $s_2 \triangleleft_2 s_2'$. Se S_1 e S_2 sono ben fondati, lo è anche il prodotto con la precedenza lessicografica.
- La chiusura transitiva di una relazione ben fondata è ben fondata; quella riflessiva e transitiva è un ordinamento parziale.

ESEMPIO: SOMMA DI GAUSS CON INDUZIONE BEN FONDATA

Per provare $\sum_{i=0}^n i = n(n+1)/2$ conviene usare la precedenza su $\mathbb{N}$ per cui $x \triangleleft y$ se e solo se x e y sono entrambi pari, oppure entrambi dispari, e $x+2 = y$; gli elementi minimali sono 0 e 1. La proprietà si verifica su 0 e 1; per $n \geq 2$ si ipotizza la proprietà per $n-2$, e $\sum_{i=0}^n i = \sum_{i=0}^{n-2} i + (n-1) + n = (n-2)(n-1)/2 + 2n - 1 = n(n+1)/2$.

Verifica di funzioni ricorsive

Per funzioni $f : D \rightarrow C$ definite in un linguaggio funzionale si vogliono provare la correttezza rispetto a una specifica, cioè $\forall x \in D. f(x) = g(x)$, e la totalità. Come nei linguaggi imperativi si usano le triple $\{P\} C \{Q\}$ e le weakest preconditions, qui lo strumento è l'induzione ben fondata su una precedenza adatta di D .

UN LINGUAGGIO FUNZIONALE MINIMALE

Le espressioni sono costruite da numeri naturali, variabili, operazioni aritmetiche e booleane, e dal condizionale `if E then E1 else E2`, un'espressione e non un comando: ha sempre il ramo `else` e i due rami hanno lo stesso tipo. Le funzioni si definiscono con equazioni, anche per casi sul primo argomento (pattern): ogni equazione si prova dall'alto verso il basso. Sulle liste si usano `[]`, il costruttore $x : xs$ e le funzioni `null`, `head` e `tail`. Un identificatore non può comparire due volte nello stesso pattern. Il valore di un'applicazione si ottiene sostituendo i parametri formali con gli argomenti nel corpo; l'ordine di valutazione è irrilevante, salvo per il condizionale, in cui si valuta prima la guardia. La valutazione può non terminare.

ESEMPIO: PARITÀ

La funzione $\text{pari} : \mathbb{N} \rightarrow \{\text{true}, \text{false}\}$ è definita da $\text{pari}(0) = \text{true}$, $\text{pari}(1) = \text{false}$, $\text{pari}(n) = \text{pari}(n-2)$ per $n \geq 2$. Si dimostra che $\text{pari}(n) = \text{true}$ se e solo se $n \bmod 2 = 0$ con l'induzione ben fondata sulla precedenza della somma di Gauss. Base: $\text{pari}(0) = \text{true}$ e $0 \bmod 2 = 0$; $\text{pari}(1) = \text{false}$ e $1 \bmod 2 \neq 0$. Passo, per $n \geq 2$: $\text{pari}(n) = \text{pari}(n-2)$, che per ipotesi induttiva vale true se e solo se $(n-2) \bmod 2 = 0$, cioè se e solo se $n \bmod 2 = 0$. La buona fondatezza della precedenza usata per definire la funzione garantisce anche che essa sia totale.

Induzione sulla lunghezza delle deduzioni

L'induzione completa serve anche a provare proprietà dei sistemi formali: ogni derivazione ha una lunghezza naturale $n \geq 1$, perciò una proprietà delle derivazioni si può dimostrare per induzione su n . Esempio: se $\vdash_{P0} \alpha$ allora α è una tautologia, cioè $\varphi(n) = \text{"se } \alpha \text{ ha una derivazione di lunghezza } n \text{ senza ipotesi, } \alpha \text{ è una tautologia"}$.

- **$\varphi(1)$.** Una derivazione di lunghezza 1 è un solo assioma (non un'ipotesi, perché non ce ne sono, e non una conclusione di *modus ponens*, che richiede due premesse precedenti). Gli assiomi sono tautologie.
- **Passo.** Si assume $\varphi(m)$ per ogni $m < n$. L'ultima fbf α è un assioma, e si conclude come prima, oppure è ottenuta per *modus ponens* da δ e $\delta \rightarrow \alpha$ di posizioni $i, j < n$: i prefissi fino a i e fino a j sono derivazioni di lunghezza minore di n , quindi per ipotesi induttiva δ e $\delta \rightarrow \alpha$ sono tautologie, e allora lo è α .

Qui conviene l'induzione completa: con quella semplice non si potrebbe usare l'ipotesi sui due prefissi, che hanno lunghezze qualsiasi minori di n e non soltanto $n-1$. Lo stesso schema si usa per la dimostrazione del teorema di deduzione di Herbrand.

Corrispondenza tra induzione e ricorsione

Dimostrare e programmare sono, in sostanza, la stessa attività (capitolo 9). All'induzione matematica corrisponde la ricorsione sui naturali: si specifica il valore per 0 e, per un generico n , il valore supponendo noto il risultato su $n - 1$. Per esempio la somma dei numeri da 0 a n è $s(0) = 0$, $s(n) = s(n - 1) + n$. All'induzione completa corrisponde la ricorsione che usa i risultati su numeri minori: i numeri di Fibonacci con $\text{fib}(0) = 1$, $\text{fib}(1) = 1$ e $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$ per $n \geq 2$. All'induzione ben fondata corrisponde la ricorsione generale sugli insiemi ben fondati.

INSIEMI DEFINITI INDUTTIVAMENTE

Un insieme si definisce induttivamente quando si forniscono gli elementi più semplici e un modo di costruirne altri a partire da elementi dell'insieme. Le liste di numeri sono definite da $[]$ e da $x : xs$. Gli alberi binari localmente completi (ABLC) hanno un nodo come albero più semplice e si ottengono unendo due ABLC con una nuova radice:

```
data ABLC = Node | AggRad ABLC ABLC
```

La relazione “è un elemento con cui si è costruito” è la precedenza associata al procedimento induttivo, ed è ben fondata. L'altezza di un ABLC si definisce con $\text{altezza}(\text{Node}) = 1$ e $\text{altezza}(\text{AggRad } t_1 \ t_2) = 1 + \max(\text{altezza}(t_1), \text{altezza}(t_2))$. La somma degli elementi di una lista è $\text{sommanums}([]) = 0$, $\text{sommanums}(n : l) = n + \text{sommanums}(l)$; la ricorsione si riflette sulla precedenza “ $L_1 \triangleleft L_2$ se e solo se L_1 ha meno elementi di L_2 ”.

Non sempre la precedenza usata da una ricorsione è quella del procedimento induttivo. La funzione che somma gli elementi di una lista in un altro modo, con $\text{snums}([]) = 0$, $\text{snums}(0 : l) = \text{snums}(l)$, $\text{snums}(n : l) = 1 + \text{snums}((n - 1) : l)$ per $n > 0$, chiama se stessa sia sulla coda l , sia sulla lista che ha un primo elemento più piccolo. La relazione adatta è la precedenza lessicografica: $L_1 \triangleleft L_2$ se e solo se $|L_1| < |L_2|$, oppure $|L_1| = |L_2|$ e il primo elemento di L_1 è minore del primo di L_2 . È una versione della precedenza lessicografica del prodotto, ed è ben fondata.

DA RICORDARE

Per una funzione ricorsiva è importante individuare la relazione di precedenza, che coincide con quella del tipo di dato oppure è una più complessa e va dimostrata ben fondata. Se la definizione ricorsiva è guidata da una precedenza ben fondata, la funzione è totale. Sull'induzione ben fondata e sul progetto di funzioni ricorsive si veda Sassone (sezioni 1 a 3).

Semantica dei linguaggi di programmazione: il linguaggio WHILE

La sintassi di un programma è la sequenza di caratteri con cui lo si scrive, la semantica ne descrive il significato. La semantica operativa strutturata definisce il significato di un programma con un sistema formale di regole di derivazione.

Che cosa è la semantica

Che cosa significa un programma? Si potrebbe dire: ciò che accade eseguendolo con un interprete o un compilatore, ma interpreti e compilatori possono contenere errori, quindi non si può prenderli come definizione. Si può guardare al manuale del linguaggio, che però è una descrizione informale. Si può infine dare una definizione formale e matematica della semantica.

VANTAGGIO DELLA SEMANTICA FORMALE	SIGNIFICATO
Meno ambigua	il comportamento del linguaggio è più chiaro per chi scrive programmi, realizza compilatori o interpreti, aggiunge costrutti
Più concisa	la notazione matematica descrive in modo breve il linguaggio e le restrizioni sui programmi legali
Argomentazioni formali	permette di enunciare e dimostrare proprietà: l'assenza di certi errori a tempo di esecuzione, o la correttezza di un programma rispetto alla sua specifica

Lo svantaggio è che i modelli matematici diventano complessi: per questo pochi linguaggi reali hanno una semantica formale. Occorre inoltre distinguere la sintassi concreta, cioè come analizzare in modo univoco una stringa, dalla sintassi astratta, che descrive la struttura delle frasi anche in modo ambiguo; la semantica lavora sulla sintassi astratta, assumendo noto l'albero sintattico.

TRE APPROCCI

SEMANTICA	IDEA	IL SIGNIFICATO DEL PROGRAMMA È
Denotazionale	si modella un programma come funzione matematica	una funzione
Assiomatica	si definisce il comportamento con formule logiche soddisfatte prima e dopo	un legame tra condizioni della memoria iniziale e finale
Operazionale	si descrive come un programma si esegue su una macchina astratta	una descrizione delle computazioni

Nella semantica operativa strutturata (SOS, Structural Operational Semantics) il significato è definito da un insieme di regole. Una regola ha la forma di premesse $A_1, \dots, A_n$ sopra una linea e conclusione A sotto: significa che un'istanza dell'asserzione A è vera se lo sono le istanze corrispondenti di $A_1, \dots, A_n$. Se $n = 0$ la regola è un assioma, e ogni sua istanza è sempre vera. Si distinguono due stili: nella semantica **big step** i giudizi descrivono il comportamento di un programma su più passi, fino al risultato; nella semantica **small step** descrivono un solo passo di computazione. Qui si usa lo stile big step.

Sintassi del linguaggio WHILE

Il linguaggio WHILE è un piccolo linguaggio imperativo con variabili che contengono numeri naturali, tre sole assegnazioni, un test di disuguaglianza e un ciclo. Si definisce con la forma normale di Backus, dove le parole tra parentesi acute sono i simboli non terminali e quelle in grassetto i terminali; le regole della BNF sono le regole di produzione.

```
{variabile} ::= X {indice}
{indice} ::= {digit} | {indice}{digit}
{assegnazione} ::= {variabile} := 0 | {variabile} := succ({variabile})
| {variabile} := pred({variabile})
{test} ::= {variabile} ≠ {variabile}
{istruzione} ::= {assegnazione} | while {test} do {istruzione}
| while {test} do {programma}
{sequenza} ::= {istruzione} | {istruzione} ; {sequenza}
{programma} ::= begin end | begin {sequenza} end
```

La variabile X seguita da un indice (digit, definito come $0 \mid 1 \mid \dots \mid 9$) non è definita in modo rigoroso dalla grammatica. Si assume che in un programma con k variabili queste siano $X_1, \dots, X_k$.

Semantica operativa di WHILE

Il significato di un programma dipende dal testo e dal valore delle variabili. I valori si memorizzano in un **vettore di stato**, una sequenza di naturali $a = (a_1, \dots, a_k)$, dove a_i è il valore di X_i ; a e b indicano vettori generici.

DEFINIZIONE

Giudizio. Le regole dimostrano asserzioni del tipo $a, P \Downarrow b$, dove P è un programma e a, b sono vettori di stato di lunghezza maggiore o uguale al numero di variabili di P . Si legge: “il programma P , se inizia l'esecuzione con il vettore di stato a , termina e alla fine il vettore di stato è b ”. Se non esiste alcuna dimostrazione di $a, P \Downarrow b$ per nessun b , il programma non termina sul vettore a , e si scrive $a, P \Uparrow$.

COSTRUTTO	PREMESSE	CONCLUSIONE
Programma vuoto	nessuna	$a, \text{begin end} \Downarrow a$
Sequenza, con $n \geq 1$	$a, \text{begin } \delta_1 \text{ end} \Downarrow a' \text{ e } a', \text{begin } \delta_2 ; \dots ; \delta_n \text{ end} \Downarrow b$	$a, \text{begin } \delta_1 ; \dots ; \delta_n \text{ end} \Downarrow b$
Azzeramento	nessuna	$(a_1, \dots, a_k), \text{begin } X_i := 0 \text{ end} \Downarrow \text{stato in cui la componente } i \text{ è } 0 \text{ e le altre sono invariate}$
Successore	nessuna	$(a_1, \dots, a_k), \text{begin } X_i := \text{succ}(X_i) \text{ end} \Downarrow \text{stato in cui la componente } i \text{ è } a_i + 1 \text{ e le altre sono invariate}$
Predecessore	nessuna	$(a_1, \dots, a_k), \text{begin } X_i := \text{pred}(X_i) \text{ end} \Downarrow \text{stato in cui la componente } i \text{ è } a_i - 1 \text{ (sottrazione aritmetica: } 0 \text{ se } a_i = 0) \text{ e le altre sono invariate}$
Ciclo, test vero	$a_i \neq a_j, a, \text{begin } \delta \text{ end} \Downarrow b, \text{ e } b, \text{begin while } X_i \neq X_j \text{ do } \delta \text{ end} \Downarrow c$	$a, \text{begin while } X_i \neq X_j \text{ do } \delta \text{ end} \Downarrow c$
Ciclo, test falso	$a_i = a_j$	$a, \text{begin while } X_i \neq X_j \text{ do } \delta \text{ end} \Downarrow a$

ESEMPIO 1: UN PROGRAMMA CHE TERMINA

Sia $P_1 = \text{begin while } X1 \neq X2 \text{ do } X1 := \text{succ}(X1) \text{ end}$, con due variabili. Con lo stato $(1, 2)$: poiché $1 \neq 2$, si usa la regola del ciclo con test vero. L'assegnazione dà $(1, 2)$, $\text{begin } X1 := \text{succ}(X1) \text{ end} \Downarrow (2, 2)$; poi dallo stato $(2, 2)$, siccome $2 = 2$, il ciclo termina con $(2, 2)$, $\text{begin while } X1 \neq X2 \text{ do } X1 := \text{succ}(X1) \text{ end} \Downarrow (2, 2)$. Si ottiene così la dimostrazione $(1, 2), P_1 \Downarrow (2, 2)$.

ESEMPIO 2: UN PROGRAMMA CHE NON TERMINA

Con lo stato $(3, 2)$: $3 \neq 2$, e dopo un passo si arriva a $(4, 2)$, ancora con test vero, poi $(5, 2)$ e così via. La ricerca della dimostrazione non finisce mai, perché la prima componente cresce e non raggiunge mai 2. Quindi $(3, 2), P_1 \Uparrow$.

Funzioni calcolate da un programma

DEFINIZIONE

Estensione dello stato. Sia P un programma con k variabili e a un vettore di naturali di lunghezza k . Per ogni vettore di naturali b , indicando con ab la concatenazione dei due vettori: $a, P \Downarrow c$ se e solo se $ab, P \Downarrow cb$, e $a, P \Uparrow$ se e solo se $ab, P \Uparrow$. Le componenti oltre le k variabili di P non influenzano l'esecuzione.

Ogni programma calcola infinite funzioni, una per ogni arità $n \geq 1$. Per un programma P con k variabili, la funzione ψ di arità n calcolata da P è definita così, per ogni $(a_1, \dots, a_n)$:

- se $n \leq k$: se $(a_1, \dots, a_n, 0, \dots, 0), P \Downarrow b$, con k meno n zeri, allora $\psi(a_1, \dots, a_n) = b_1$, la prima componente di b ; se il programma non termina, ψ è indefinita;
- se $n > k$: se $(a_1, \dots, a_k, a_{k+1}, \dots, a_n), P \Downarrow b$, allora $\psi(a_1, \dots, a_n) = b_1$; i parametri dal $(k + 1)$ esimo in poi non influenzano il risultato. Se non termina, ψ è indefinita per ogni $a_{k+1}, \dots, a_n$.

ESEMPIO 3

Il programma P_1 ha due variabili. Per ogni s , $(0, 0, 0, \dots, 0), P_1 \Downarrow (0, 0, 0, \dots, 0)$; per $n \leq m$, $(n, m, 0, \dots, 0), P_1 \Downarrow (m, m, 0, \dots, 0)$; per $n > m$, $(n, m, 0, \dots, 0), P_1 \Uparrow$. Le funzioni calcolate sono:

ARITÀ	FUNZIONE CALCOLATA
1	$\psi(n) = 0$ se $n = 0$, indefinita altrimenti
2 o più	$\psi(n_1, n_2, \dots) = n_2$ se $n_1 \leq n_2$, indefinita altrimenti

DA RICORDARE

La semantica operativa strutturata riduce il significato di un programma all'esistenza di una derivazione in un sistema formale: un programma termina su uno stato se e solo se esiste una dimostrazione di $a, P \Downarrow b$; altrimenti diverge.