Capitolo 5: Fondamenti (Cenni di Calcolabilità)

Semantica Statica: permette di fare controlli eseguiti sul testo del programma, senza mandarlo in esecuzione

Semantica Dinamica: permette di analizzare eventuali errori a run-time

Domanda: Possiamo rilevare tutti i possibili errori (statici o dinamici) mediante un analizzatore "statico" (cioè senza eseguire il programma)?

Ovvero: può esistere un compilatore che, staticamente, rileva tutti i possibili errori che si verificheranno a run-time?

Ma, anche volendo essere più generosi, può esistere un programma che, anche eventualmente eseguendo parzialmente il programma oggetto, rileva in tempo finito tutti i possibili errori a run-time?

In altre parole ...

$$\text{check}(P) = \begin{cases} 1 & \text{se } P \text{ è corretto} \\ 0 & \text{se } P \text{ presenta errori} \end{cases}$$

Esiste un programma che calcola Check?

Vediamo un caso specifico, in cui l'errore è la non-terminazione: se il programma è scritto in un ling. sequenziale, ci si aspetta che il suo calcolo termini sempre.

Dato un ling. di programmazione $\mathcal{L}$, proviamo a scrivere in $\mathcal{L}$ un programma H che calcola la seguente funzione

$$H(P, x) = \begin{cases} 1 & \text{se } P(x)\downarrow \text{ "termina"} \\ 0 & \text{se } P(x)\uparrow \text{ "diverge"} \end{cases}$$

Oss: Per rispondere "0", il programma H deve riconoscere in tempo finito che il prog. P con input x non terminerà mai il calcolo!

Ma può esistere un programma H siffatto?

Questo problema è noto in informatica come "problema della fermata"

HALTING PROBLEM

- Supponiamo, per assurdo, che H esista davvero.
- Allora, usando H, possiamo realizzare l'applicazione

$$K(P) = \begin{cases} 1 & \text{se } P(P)\downarrow \\ 0 & \text{se } P(P)\uparrow \end{cases}$$

$$= H(P, P)$$

qui P è usato come dato che il P) usa.

(Vedi compilatori: programmi che usano altri programmi come dato)

<u>Oss</u>: Se H esiste, allora esiste anche K!

- Se esiste K, allora possiamo scrivere un programma G che prende in input un prog. P e calcola

$$G(P) = \begin{cases} 1 & \text{se } K(P) = 0 \\ \uparrow & \text{se } K(P) = 1 \end{cases}$$

<u>Oss</u>: Se K esiste, allora anche G è facilmente programmabile!

- Ma cosa succede se a G diamo in input G?

$$\begin{cases} - \ G(G) = 1 & \text{sse } K(G) = 0 \ \text{ sse } \ G(G)\uparrow \\ - \ G(G) = \uparrow & \text{sse } K(G) = 1 \ \text{ sse } \ G(G)\downarrow \end{cases}$$

contraddizione

ASSURDO! ⇒ G non può esistere ⇒ K non può esistere ⇒ H non può esistere!

H è il primo esempio di funzione non calcolabile (o di problema non risolvibile – Turing 1936)

Ma allora molte altre applicazioni

NON ESISTONO!

$$Z(P) = \begin{cases} 1 & \text{se } P \text{ calcola la funzione costante "zero"} \\ & (\forall x \; P(x) = 0) \\ 0 & \text{altrimenti} \end{cases}$$

Z non è calcolabile perché, se esistesse un prog. per Z, allora potrei costruire un prog. per K! Come?

- Costruiamo una applicazione F tale che

$$F(P)(x) = \begin{cases} 0 & \text{se } P(P)\downarrow \\ \uparrow & \text{se } P(P)\uparrow \end{cases}$$

<u>Oss:</u> F è calcolabile! Basta avere un interprete che permetta di eseguire P su P (come dato).

- Costruiamo $K(P) = Z(F(P))$ Infatti

$$Z(F(P)) = \begin{cases} 1 & \text{se } \forall x \; F(P)(x) = 0 \\ 0 & \text{altrimenti} \end{cases}$$

$$= \begin{cases} 1 & \text{se } P(P)\downarrow \\ 0 & \text{se } P(P)\uparrow \end{cases} = K(P)$$

- Dato che sappiamo che K non è calcolabile, deve essere anche Z non calcolabile!

$$\mathrm{Equiv}(P, Q) = \begin{cases} 1 & \text{se } \forall x \; P(x) = Q(x) \; (\text{o } P(x) = \uparrow = Q(x)) \\ 0 & \text{altrimenti} \end{cases}$$

Problema dell'equivalenza di due programmi.

Equiv non è calcolabile! Se lo fosse, potrei calcolare Z, cioè costruire un programma per Z.

$$Z(P) = \mathrm{Equiv}(P, \mathrm{zero})$$
$$= \begin{cases} 1 & \text{se } \forall x \; P(x) = \mathrm{zero}(x) = 0 \\ 0 & \text{altrimenti} \end{cases}$$

Poiché già sappiamo che non può esistere un programma per Z, possiamo concludere che non può esistere un programma per Equiv!

Possiamo risolvere questi problemi considerando un diverso linguaggio di programmazione?

Se il ling. è abbastanza ricco (cioè Turing-completo), allora NO!

Il problema della fermata, per i normali ling. di programmazione, è indecidibile: non esiste alcuna procedura che lo risolva.

Procedura di decisione

È una procedura che
- funziona per argomenti arbitrari
- Risponde sì o No in tempo finito

<u>Esempio</u> $w \in L(G)$? è decidibile per, ad es. grammatiche regolari

Procedura di semi-decisione

È una procedura che

- funziona per argomenti arbitrari
- risponde sì in tempo finito
- non è in grado di rispondere NO in tempo finito

<u>Esempio</u> $H'(P,x) = \begin{cases} 1 & \text{se } P(x)\downarrow \\ \uparrow & \text{se } P(x)\uparrow \end{cases}$

La maggior parte dei problemi interessanti per i linguaggi di programmazione sono <u>indecidibili</u>, (a volte semi-decidibili, ma poco utili in pratica)

Proprietà Indecidibili

- Terminazione $H(P,x) = \begin{cases} 1 & \text{se } P(x)\downarrow \\ 0 & \text{se } P(x)\uparrow \end{cases}$

  è però semidecidibile

- Divergenza $D(P,x) = \begin{cases} 1 & \text{se } P(x)\uparrow \\ 0 & \text{se } P(x)\downarrow \end{cases}$

  non è nemmeno semidecidibile!

- Equivalenza di programmi

- Calcolo di una costante (cioè se P calcola una funzione costante)

- Generazione di errori a run-time

  ⋮

Se consideriamo ling. con limitato potere espressivo, allora alcune di queste proprietà possono essere decise.

Ad es. Se $\mathcal{L}$ non ha while (iterazione indeterminata) né ricorsione, allora tutti i suoi programmi terminano, $\Longrightarrow$ Terminazione è decidibile!

Se un ling. è Turing-completo, allora le proprietà sopra sono INDECIDIBILI!!

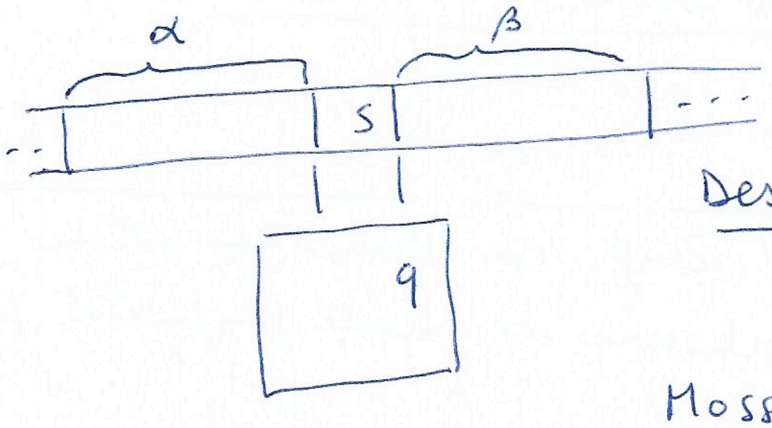
Ma cosa vuol dire Turing-completo?

Macchina di Turing

MdT $M = (Q, A, B, \delta, q_0, q_f)$ dove

- Q è un insieme finito di stati
- A è l'alfabeto finito dell'input (tipicamente le cifre 0-9)
- B è l'alfabeto finito del nastro ($A \subset B$, $\square \in B$)
  casella vuota
- q_0 è lo stato iniziale
- q_f è lo stato finale
- $\delta : Q \times B \multimap Q \times B \times \{sx, dx\}$
  - funzione parziale
  - MdT deterministica

  tale che $\delta(q_f, b)$ è indefinita $\forall b \in B$



nastro infinito!

Descrizione istantanea

$$\alpha q s \beta$$

Mosse:

- $\alpha q s \beta \vdash \alpha s' q' \beta$ se $\delta(q, s) = (q', s', dx)$
- $\alpha \bar{s} q s \beta \vdash \alpha q' \bar{s} s' \beta$ se $\delta(q, s) = (q', s', sx)$

$$L[M] = \{ w \in A^* \mid q_0 w \vdash^* \alpha q_f \beta \}$$

ma può non raggiungere mai né q_f, né uno stato di blocco "erroneo", cioè può divergere

Una MdT deterministica su alfabeto $A = [0-9]$ può essere vista come una macchina che calcola funzioni parziali binarie:

$$f_M(w) = \begin{cases} 1 & \text{se } q_0 w \vdash^* \alpha q_F \beta \\ 0 & \text{se } q_0 w \vdash^* \alpha q' \beta \not\vdash \text{ e } q' \neq q_F \\ \uparrow & \text{altrimenti} \end{cases}$$

Se consideriamo l'insieme di tutte le funzioni parziali a valori binari

$$\mathcal{F} = \{ f \mid f : \mathbb{N} \multimap \{0,1\} \}$$

possiamo concludere che $f \in \mathcal{F}$ è <u>Turing-calcolabile</u> se $\exists$ MdT M tale che $f_M = f$.

<u>Oss:</u> Ogni MdT deterministica calcola una funzione. Allora come sarà l'insieme di tutte le funzioni Turing-calcolabili? Coincide con $\mathcal{F}$? O esistono funzioni in $\mathcal{F}$ che non sono Turing-calcolabili?

La maggior parte delle funzioni in $\mathcal{F}$ non sono Turing-calcolabili!!!

Argomento di Cantor

Supponiamo che le funzioni binarie (considerando solo le totali, per semplicità) siano enumerabili. Dimostreremo che ciò è impossibile!

	0	1	2	3	4	...
f_1	0	0	1	1	1	...
f_2	0	1	0	1	0	...
f_3	1	0	0	0	0	...
f_4	0	1	0	1	1	...
⋮						

Definiamo la funzione

$\overline{f}_d(i) = \overline{f_i(i)}$ cioè se $f_i(i) = 1$ allora $\overline{f_i}(i) = 0$ e viceversa

Ora, per ogni $i \in \mathbb{N}$, $\overline{f}_d(i) \neq f_i(i)$

cioè $\overline{f}_d$ non compare nell'elenco!

In realtà, si può dimostrare che l'insieme $\mathcal{F}$ ha cardinalità pari ai numeri reali $\mathbb{R}$, quindi più grande della cardinalità di $\mathbb{N}$.

D'altro lato, le MdT sono enumerabili!

$M_0 \ M_1 \ M_2 \ \ldots$

$\Rightarrow$ di MdT ce ne sono tante quante $\mathbb{N}$.

$\Downarrow$

La stragrande maggioranza delle funzioni non è calcolabile! (con MdT almeno)

Oss: Analogia con linguaggi, ovvero $\mathcal{P}(A^*) \cong \mathbb{R}$ e le grammatiche, $\cong \mathbb{N}$, già visto ad inizio corso

---

Se cambiamo formalismo riusciamo a calcolare di più??

Formalismo Turing-completo: ha la stessa potenza espressiva delle MdT — calcola le stesse funzioni calcolabili con MdT.

Negli anni '30, '40, '50 del secolo scorso

- MdT
- Lambda calcolo
- Funzioni generali ricorsive
- Random Access Machines (RAM)
- Counter Machines

⋮

Tutti dimostrato equivalenti, attraverso compilazione di un formalismo in un altro.

E i linguaggi di programmazione?

C, Java, Pascal, Lisp, ML, Prolog, ...

Tutti Turing-completi, purché sia possibile usare tutta la memoria di cui possono necessitare.
(Un computer ha solo una memoria finita, per cui non può essere equivalente a una MdT)

Teorema di Jacopini - Böhm (1966)

Un ling. di program. imperativo che contenga

- if-then-else (condizionale)
- while (iterazione indeterminata)
- ; (composizione sequenziale)

(+ istruzione di assegnamento) è Turing-completo!

Oss: il ling. usato per spiegare la semantica SOS è Turing-completo!

Oss: Questo teorema ha avuto un benefico effetto nel promuovere la "programmazione strutturata", cioè quel principio di programmazione secondo il quale un ling. doveva contenere solo operatori ben strutturati, la cui semantica era definibile "localmente", solo guardando i propri argomenti (Non è così per il GOTO!)

Tesi di Church-Turing (1936-1937)

Se una funzione può essere calcolata algoritmicamente in un qualche formalismo, allora è calcolabile con MdT.

- Tesi perché:
  - non c'è una definizione formale di cosa è "algoritmicamente" calcolabile
  - è implicita una quantificazione universale su tutti i possibili formalismi! Infinite prove? E se domani uno arriva con un nuovo formalismo? ...

- Considerata vera perché in 80 anni nessuno è riuscito a confutarla!!

- Criterio di equivalenza tra linguaggi sequenziali: Se $\mathcal{L}_1$ e $\mathcal{L}_2$ sono entrambi Turing-completi, allora sono equamente espressivi per la Tesi di Church-Turing

- Nel caso dei ling. concorrenti, la questione è diversa perché i programmi concorrenti non calcolano solo funzioni, ma risolvono problemi, offrono servizi, ecc...

[Gerarchia di Macchine]

Espressività vs. Analizzabilità

MdT $\longleftrightarrow$ gram. generali $\longleftrightarrow$ $w \in L(M)/L(G)$?
è solo semidecidibile

Ma abbiamo visto formalismi più deboli ma significativi

PDA $\longleftrightarrow$ gr. libere $\longleftrightarrow$ $w \in L[N]/L(G)$?
è decidibile!
Ma il problema dell'equival.

$$E(G_1, G_2) = \begin{cases} 1 & \text{se } L(G_1) = L(G_2) \\ 0 & \text{altrimenti} \end{cases}$$

è indecidibile!

DFA $\longleftrightarrow$ gr. regolari $\longleftrightarrow$ $E(G_1, G_2)$ è decidibile!

Oss: - circuiti logici sono DFA
- vending machine sono DFA
- funzioni find/replace in text-editors sono DFA

$\Rightarrow$ esistono molte "funzioni" interessanti che si possono calcolare in formalismi non Turing-completi!
E su questi possiamo decidere molte proprietà interessanti!