Corso di Linguaggi di Programmazione

Lezione 2

Alberto Ceselli
alberto.ceselli@unimi.it

Università degli Studi di Milano

01 Marzo 2013

1. Controlli automatici di correttezza dei programmi
2. Il punto di vista imperativo
3. Calcolabilità e funzioni

Controlli automatici di correttezza dei programmi

- Obiettivo chiave dei LdP: fornire costrutti che riducano la probabilità di lasciare bug nei programmi
- ... ovvero: il *programma* implementa effettivamente l'*algoritmo* per cui è progettato.
- Perchè non costruire *verificatori automatici* della correttezza dei programmi, e poi utilizzare il linguaggio che preferiamo?
- Ci sono risultati teorici che dimostrano che *non è possibile costruire verificatori perfetti*.

Verificatore della terminazione di programmi C

- supponiamo di avere un programma *term* (alla lavagna ...)
- in ogni caso, *term* si comporta diversamente dalle specifiche.
- infatti, term *non può essere realizzato!*.

Programma per il controllo dell'utilizzo degli indici negli array

(alla lavagna)

Riassumendo

E' impossibile verificare in modo consistente e completo qualsiasi proprietà di un programma che dipenda dall'esito della computazione

- Si possono condurre controlli consistenti, ma incompleti (es. errori in compilazione)
- Si possono condurre controlli completi, ma inconsistenti (es. warning)
- es. si possono individuare alcuni segmenti di codice non raggiungibile (ma mai garantire di individuare tutto e solo il codice irraggiungibile in qualsiasi programma)
- es. si può controllare il matching dei tipi nel passaggio di parametri a procedura, ma la chiamata potrebbe non avvenire
- si possono condurre controlli a run-time (ma l'*errore* è già avvenuto)

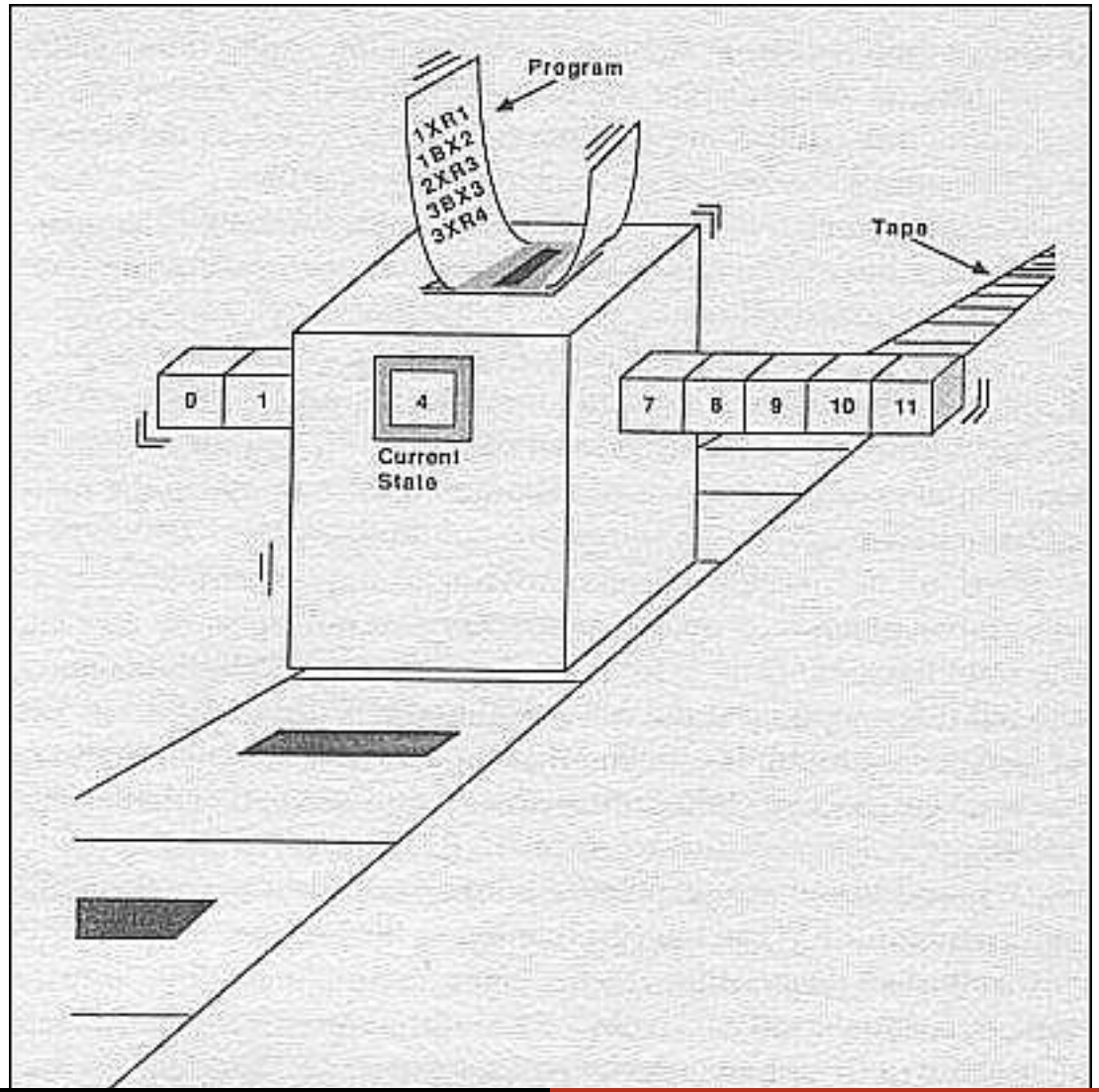
Linguaggi di programmazione general purpose

- Abbiamo dimostrato che non esiste nessun *programma C* che *in ambiente UNIX* verifichi la terminazione di un programma (system, exit ...)
- Domanda: dato un linguaggio di programmazione ed un problema "calcolabile"
- posso sempre scrivere un programma in quel linguaggio che risolva il problema?
- Linguaggi di programmazione con questa caratteristica vengono definiti *general purpose*
- es. C, C++, Java, LISP, ML, Prolog ... ma NON SQL
- Per caratterizzare i linguaggi general purpose abbiamo bisogno di *formalità* e *generalità*.

Modelli di calcolo astratti

- Come dimostrare qualcosa che valga *indipendentemente* dal linguaggio utilizzato?
- ... con modelli di calcolo astratti
- es. macchina RAM, *macchina di Turing*

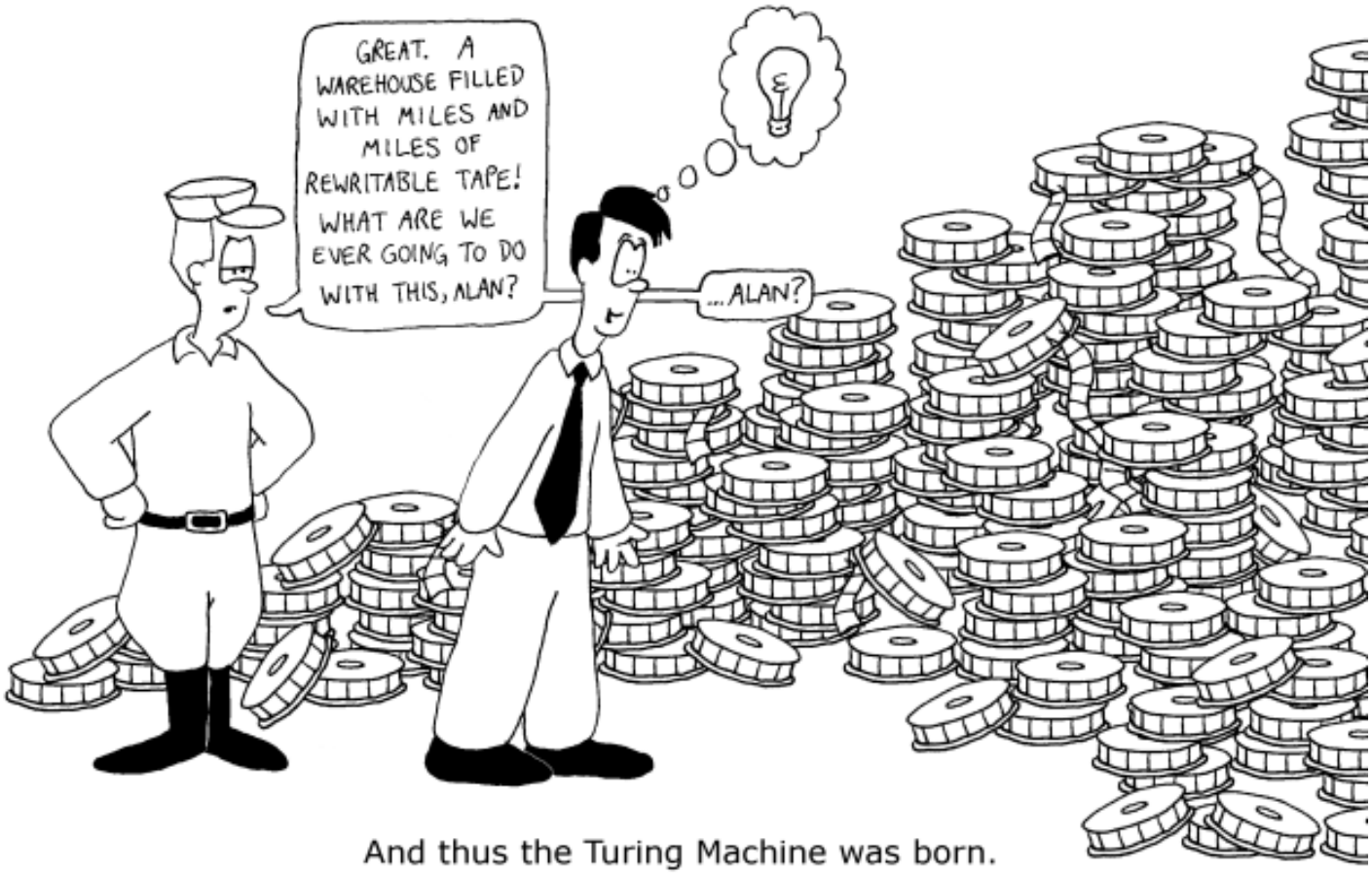
La macchina di Turing



La macchina di Turing

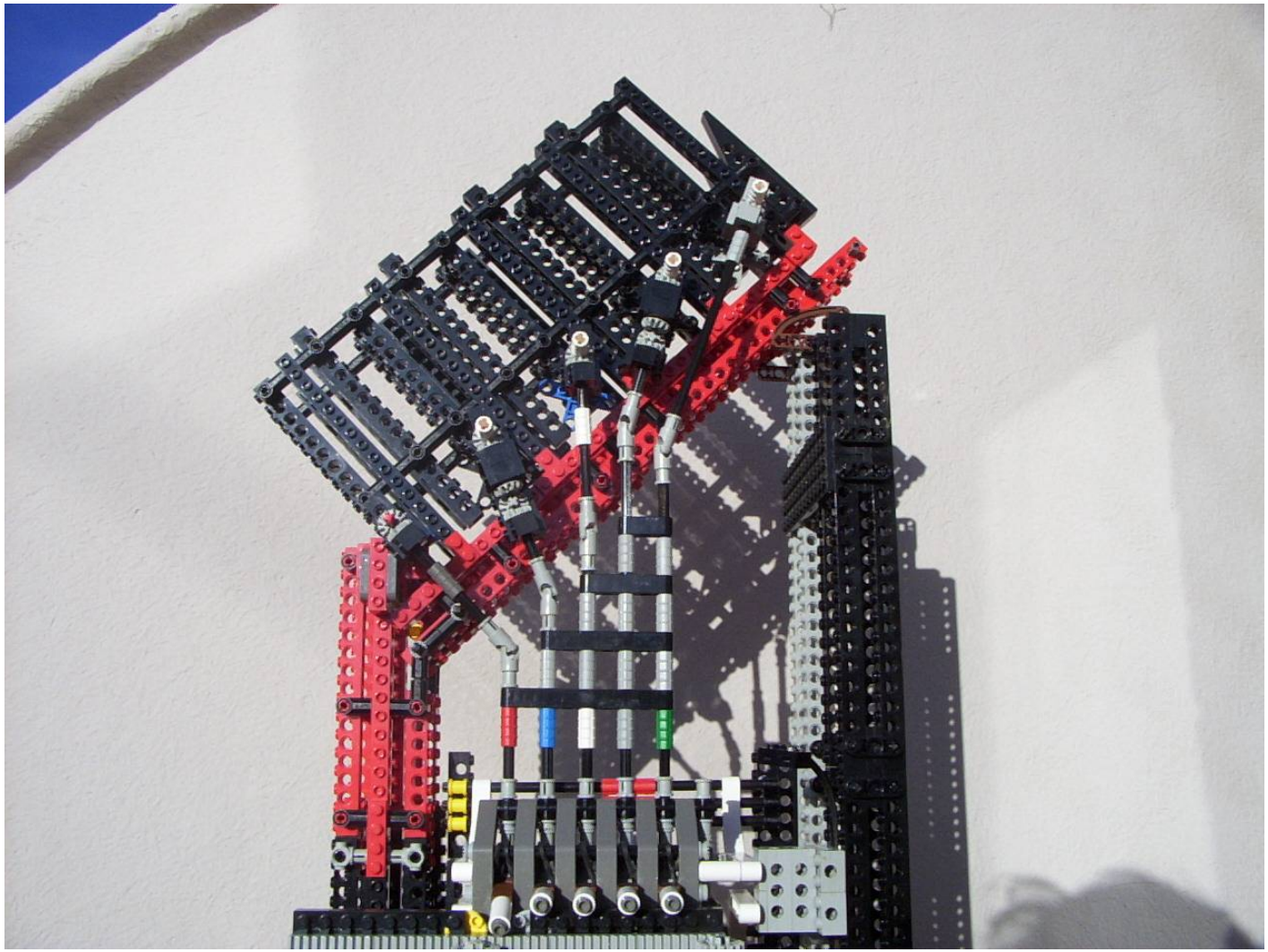
- oggetto matematico, definito in modo completo, formale e molto compatto
- rudimentale, ma potente almeno quanto gli attuali computer (nastro infinito)
- i programmi delle MdT non hanno dettagli inutili: il loro comportamento non è ambiguo
- il comportamento del processore è definito matematicamente, come i programmi

La macchina di Turing



And thus the Turing Machine was born.

La macchina di Turing



La macchina di Turing

Overview: la MdT

- è una n-upla $(S, \Sigma, I, P, s_0, e_+, e_-)$
- S è l'insieme di possibili *stati* in cui si può trovare una macchina
- Σ è l'alfabeto di lavoro della macchina
- $I \subseteq \Sigma$ è l'alfabeto di input della macchina
- s_0 è lo stato iniziale della macchina
- e_+ è lo stato finale "successo" della macchina
- e_- è lo stato finale "fallimento" della macchina
- P è il programma della macchina

Programmi per MdT

Un programma P per una MdT è

- un insieme finito di tuple $(s, \sigma, m^i, \bar{\sigma}, m^w, \bar{s})$
- s è lo stato in cui si deve trovare la MdT, σ il carattere sul nastro di input
- m^i la direzione in cui spostare la testina del nastro di input, m^w la direzione in cui spostare la testina del nastro di lavoro $(m^i, m^w \in M = \{l, r, h\})$
- $\bar{\sigma}$ il carattere da scrivere sul nastro di lavoro, $\bar{s}$ lo stato al termine dell'istruzione
- in altri termini: $P : S \times I \to M \times \Sigma \times M \times S$

Come dimostrare la Turing-equivalenza?

- dato un linguaggio di programmazione
- realizzare un simulatore di MdT in quel linguaggio
- progettare una MdT interprete per quel linguaggio
- anche con traduzioni intermedie ...
- semplice, dato che i compilatori producono linguaggio macchina

- una MdT può calcolare *tutto quello che sappiamo calcolare* (e solo quello).
- def. **linguaggio general purpose:** è un linguaggio *Turing equivalente*
- N.B. diverso dall'*equivalenza polinomiale* (legata alla complessità dei problemi)

Questioni di cardinalità

- Th: qualsiasi sequenza di naturali può essere codificata con un singolo naturale.
- Dim: codifico per diagonali (alla lavagna)
- possiamo anche costruire una formula

$$\ll x, y \gg = \binom{x + y + 1}{2} + y$$

  che è una bijezione (invertibile)
- ... ed il gioco può essere ripetuto (per qualsiasi sequenza *finita* di naturali)
- il processo di assegnare un naturale (numero di Gödel) ad ogni elemento di un insieme numerabile è detto *Gödelizzazione*.

Questioni di cardinalità

- Le MdT sono *tante quanti i naturali* (alla lavagna)
- I problemi sono *più dei naturali* (alla lavagna)
- ... non posso avere corrispondenza 1 a 1 tra funzioni e MdT!
- Th. esiste una MdT *universale* u, che avendo sul nastro di input la descrizione di una qualsiasi MdT k (codificabile con un naturale) e l'input per quella MdT (i) emula il comportamento di k su input i.

Funzioni ricorsive

Quali sono le *funzioni* calcolabili?

- def. funzione $f : X \to Y$: una qualsiasi relazione che mette in corrispondenza ad ogni elemento $x \in X$ uno ed un solo elemento $y \in Y$
- def. Funzione *Parziale* (FP) $g : X \to Y$: è una relazione che *può essere non definita* per qualche $x \in X$ (in questo caso $g(x) = \perp$)
- possiamo immaginare che $\perp$ corrisponda a "loop infinito" in un programma
- (N.B. $g()$ non è totale, quindi NON è una funzione!)
- per convenzione $g(\perp) = \perp$.

FP ed indecidibilità algoritmica

- Idea: ad ogni FP provo ad assegnare un indice (quello della MdT che realizza il calcolo corrispondente): $\varphi_k()$ (ruolo dei filename)
- (purtroppo) verifico che ad alcune FP interessanti non riesco ad assegnare indice
- (per fortuna) riesco a caratterizzare le FP che hanno indice: le Funzioni Parziali Ricorsive (FPR)
- Esiste una MdT *universale* u, quindi esiste una *FPR universale* $\varphi_u()$ (ruolo di "system")

FPR ed indecidibilità algoritmica

- Th. la funzione parziale "inversione"

$$f(x) = \begin{cases} 1 & \text{se } \varphi_u(x,x) = \perp \\ \perp & \text{altrimenti} \end{cases}$$

non ha indice (non è FPR, dim. alla lavagna)

- Th. la funzione "term"

$$h(x,y) = \begin{cases} 0 & \text{se } \varphi_u(x,y) = \perp \\ 1 & \text{altrimenti} \end{cases}$$

non ha indice (non è FPR, dim. alla lavagna)

La tesi di Church

Infatti: tesi di Church

- Le funzioni calcolabili algoritmicamente sono le funzioni ricorsive.
- Le funzioni semi-calcolabili algoritmicamente sono le funzioni parziali ricorsive.
- N.B. La semi-calcolabilità è principalmente utile come concetto di carattere "intuitivo".
- ... non perdo nè guadagno nulla rinunciando al concetto di stato della MdT, o restringendomi all'utilizzo della ricorsione!

Funzioni parziali ricorsive

La classe delle Funzioni Parziali Ricorsive (FPR) è la più piccola classe di funzioni parziali da $\mathbb{N}^n$ in $\mathbb{N}$ tale per cui

1. la funzione costante 0, successore $s()$ e le proiezioni i-esime $\pi_i^{(n)}$ appartengono a FPR (descrizione tralasciata)
2. applicando *sostituzioni* e *minimalizzazione* a FPR si ottengono FPR (descrizione tralasciata)
3. date due FPR $g : \mathbb{N}^{n-1} \to \mathbb{N}$ e $h : \mathbb{N}^{n+1} \to \mathbb{N}$, la funzione parziale $f()$ ottenuta tramite *ricorsione primitiva*

$$\begin{aligned} f(0, x_2 \ldots x_n) &= g(x_2 \ldots x_n) \\ f(y+1, x_2, \ldots x_n) &= h(f(y, x_2 \ldots x_n), x_2, \ldots, x_n, y) \end{aligned}$$

appartiene a FPR.

Riassumendo

Riassumendo, alcuni dei principali compiti dei LdP sono:

- fornire strumenti per programmare senza errori (o per accorgersi di aver commesso errori già in fase di compilazione)
- fornire strutture adeguate a facilitare le fasi di testing
- ... perchè non è possibile realizzare verificatori automatici *universali e perfetti* di correttezza dei programmi.

Riassumendo

Abbiamo infatti mostrato come alcuni problemi quali la *terminazione* dei programmi siano *indecidibili* (indipendentemente dal paradigma di programmazione o dalle risorse di calcolo dedicate)

- costruendo particolari controesempi in C
- descrivendo le Macchine di Turing e caratterizzando in modo formale i linguaggi *Turing equivalenti*
- dimostrando che le funzioni, e quindi i problemi sono più delle Macchine di Turing
- caratterizzando le funzioni parziali e la classe delle funzioni parziali ricorsive, e mettendo in corrispondenza le funzioni parziali ricorsive con le Macchine di Turing
- dimostrando, ad esempio, che la funzione "terminazione" non è ricorsiva

Riferimenti:

Disponibili in rete:

- P. Bonatti (1999) Cenni informali di calcolabilità
- S. Vigna (2006) Dispense per il corso di informatica teorica