Informatica

CdL in Matematica

Parte 4

Roberto Zunino

Linguaggi e Paradigmi

Specifiche

Siano *Input* e *Output* due insiemi arbitrari.

Informalmente, una *specifica* è una funzione da *Input* a *Output* che si desidera calcolare automaticamente.

Esempio. Dato un dizionario e una parola presente al suo interno, trovare in che pagina essa appare.

Algoritmi

Informalmente, un algoritmo è un procedimento automatico, ovvero una *descrizione* informale ma non ambigua che associa as ogni input una sequenza finita di passi di calcolo elementari, tramite il quale è possibile ottenere l'output desiderato, realizzando una data specifica.

N.B. Un algoritmo *non* è una sequenza di passi, ma la *descrizione* di tutte le sequenze di passi necessari a calcolare l'output per ogni dato input.

A input diversi possono corrispondere un numero di passi diversi. Per esempio, cercare in un dizionario tascabile (con poche pagine) richiede intuitivamente meno passi rispetto a cercare in un dizionario più grande.

Algoritmi: esempi

Per cercare una parola in un dizionario:

Esempio. “Ricerca lineare.”

Sia p la parola da cercare. Si parta dall’inizio del dizionario e si scandiscano le parole una per una. Non appena si incontra la parola p, ci si fermi annotando il numero di pagina al quale ci si fermati.

Algoritmi: esempi

Per cercare una parola in un dizionario:

Esempio. "Ricerca binaria."

Sia p la parola da cercare. Si parta dalla metà del dizionario, e sia q la parola lì incontrata. Se $q = p$, si ci fermi annotando il numero di pagina. Se invece $q > p$ secondo l'ordinamento lessicografico (quello "del dizionario"), si restringa la ricerca alla prima metà del dizionario, altrimenti la si restringa alla seconda metà. Ripetere il procedimento di sopra, dimezzando di volta in volta l'intervallo di pagine in cui si cerca, fino a trovare p.

Algoritmi e Specifiche

Si noti che ci possono essere algoritmi *molto diversi* per la stessa specifica.

Linguaggi di programmazione

Un linguaggio formale è un insieme di stringhe (sequenze finite di simboli).

Un linguaggio di programmazione è un linguaggio formale in cui ogni elemento, detto *programma*, descrive un algoritmo precisamente, in modo da poter essere eseguito da un elaboratore.

Sintassi e Semantica

Un linguaggio di programmazione è definito tramite

sintassi e semantica

- sintassi: la descrizione di quali stringhe compongono il linguaggio. Di solito, un insieme di regole per potere costruire programmi validi.

  Esempio. "$4+)*2$" non è un'espressione aritmetica.

- semantica: una applicazione che associa ad ogni programma sintatticamente valido il suo *significato*, ovvero il suo comportamento.

  Esempio L'espressione `x * x` calcola il quadrato.

Nota: sopra `x` e $*$ sono *simboli*, non operatori matematici.

In generale

- Specifica = **cosa** calcolare
- Algoritmo = **come** calcolarlo
- Programma = algoritmo eseguibile, formalizzato in un linguaggio

Programmazione

La programmazione consiste, almeno in prima battuta, nello scrivere programmi (quindi sintatticamente validi) la cui semantica coincida con una data specifica.

Paradigmi

I linguaggi di programmazione vengono classificati in base al loro *paradigma*, ovvero ai concetti fondamentali che il linguaggio usa per descrivere il calcolo automatico.

I principali paradigmi sono:

- funzionale
- imperativo
- logico

Paradigma funzionale

FUORI ESAME

I linguaggi funzionali si fondano sull'idea di *funzione*, e sulla teoria del *lambda calcolo*.

Esempi: Haskell, Ocaml, SML, Erlang, Lisp, Scheme, (Scala), ...

- funzioni definite tramite *espressioni*
- uso frequente della ricorsione
- spesso considerati più leggibili (anche se a volte meno efficienti)
- enfasi sulla correttezza del codice
- ragionamento equazionale: $f(x) + f(x) = 2 * f(x)$
- il concetto di "variabile" è quello matematico
- ricca teoria semantica e legami con la Logica Matematica
- usati anche in ambito finanziario

Paradigma imperativo

FUORI ESAME

I linguaggi imperativi si fondano sull'idea di eseguire *comandi* che modificano lo *stato* del calcolatore.

Esempi:
C, C++, Java, C#, Scala, Python, Perl, Pascal, Ada, Fortran, JavaScript/ECMAScript, ...

- esistenza di uno stato, constantemente modificato
- il valore delle "variabili" può essere modificato dai comandi
- uso frequente dell'iterazione (ripetizione di un comando)
- spesso è possibile definire "funzioni", che però assumono un significato un po' diverso da quello usuale
- ragionamento equazionale arduo: $f(x) + f(x) \neq 2 * f(x)$ in generale, perché $f(x)$ può modificare lo stato.
- molto usati nell'industria, in ogni ambito

Paradigma logico

FUORI ESAME

I linguaggi logici si fondano sull'idea di *dimostrare teoremi* in una logica semplificata.

Esempi: Prolog.

- Il programma definisce predicati tramite una lista di assiomi
- L'utente può fare domande del tipo: "per quali X ho che $p(X, 4, X)$?" e ottenere tutti i valori di X associati
- I predicati non hanno un chiaro concetto di input e output
- Relazioni col metodo di risoluzione della Logica Matematica
- Richiede, in pratica, di utilizzare l'algoritmo di unificazione ad ogni passo

Esempi

FUORI ESAME

Vediamo qualche esempio di programma in diversi linguaggi.

Al momento, non vi si chiede di comprenderli appieno, ma di formare una intuizione, seppur vaga, su come essi operino.

Esempi

FUORI ESAME

Un programma per calcolare la somma di una sequenza di interi.
Linguaggio: Haskell (funzionale)

```
somma :: [Integer] -> Integer
somma []     = 0
somma (n:s) = n + somma s
```

Esempi

FUORI ESAME

Un programma per calcolare la somma di una sequenza di interi.
Linguaggio: Java (imperativo)

```
int somma(int[] sequenza) {
   int s = 0;
   int i;
   for (i = 0; i < sequenza.length; i++) {
      s = s + sequenza[i];
   }
   return s;
}
```

Esempi

FUORI ESAME

Un programma per calcolare 3^n dato $n \geq 0$.
Linguaggio: Haskell (funzionale)

```
treAlla :: Integer -> Integer
treAlla 0 = 1
treAlla n = 3 * treAlla (n - 1)
```

Esempi

Un programma per calcolare 3^n dato $n \geq 0$.
Linguaggio: Python (imperativo)

```
def treAlla(n):
  r = 1
  while n > 0:
    r = r * 3
    n = n - 1
  return r
```

Esempi

FUORI ESAME

Un programma per calcolare 3^n dato $n \geq 0$.
Linguaggio: Java (imperativo)

```
long treAlla(long n) {
  long r = 1;
  while (n > 0) {
    r = r * 3;
    n = n - 1;
  }
  return r;
}
```

Esempi

FUORI ESAME

Un programma per calcolare 3^n dato $n \geq 0$.
Linguaggio: Haskell (funzionale)

```
-- Algoritmo del ‘‘contadino russo’’
treAlla :: Integer -> Integer
treAlla n = potenza 3 n

potenza :: Integer -> Integer -> Integer
potenza a 0 = 1
potenza a n =
   if even n
   then potenza (a*a) (n ‘div‘ 2)
   else a * potenza a (n-1)
```

Esempi

FUORI ESAME

Un programma per calcolare 3^n dato $n \geq 0$.
Linguaggio: Java (imperativo)

```
long treAlla(long n) {
  long base = 3;
  long r = 1;
  while (n > 0) {
    if (n % 2 == 0) { // pari
      n = n / 2;
      base = base * base;
    } else { // dispari
      r = r * base;
      n = n - 1;
    }
  }
  return r;
}
```

In questo corso

FUORI ESAME

Ci concentreremo sulla programmazione imperativa.

Segnaliamo il corso di Programmazione Funzionale per chi volesse approfondire l'argomento.

Ci concentreremo sulle basi della programmazione imperativa, ignorando il più possibile la programmazione orientata agli oggetti.

Segnaliamo il corso di Programmazione 2 per chi volesse approfondire l'argomento.

Conclusioni

FUORI ESAME

La precedente introduzione è (intrinsecamente) informale.

Non esiste una definizione formale di specifica, algoritmo, o programma (se non in un preciso linguaggio).

Solo una volta fissato uno specifico linguaggio di programmazione è possibile descriverne sintassi e semantica in modo matematicamente rigoroso.

Una descrizione formale di un linguaggio complesso come Java richiederebbe un quantitativo di tempo enorme (e sarebbe di dubbia utilità). Ci concentreremo invece su un linguaggio imperativo "giocattolo", chiamato IMP, nel quale possiamo descrivere tutti i concetti fondamentali della programmazione imperativa.

IMP

un linguaggio di programmazione imperativo

Espressioni di IMP

Variabili e Stato

Indichiamo con Var un insieme di nomi di variabili.

$$Var = \{x, y, \ldots\}$$

Nel linguaggio IMP, per il momento, consentiamo alle variabili di assumere solo valori interi ($\mathbb{Z}$).

Durante l'esecuzione di un programma, lo *stato* (*state*, a volte anche chiamato *store*) delle variabili può essere quindi rappresentato come una funzione

$$\sigma \in State = (Var \to \mathbb{Z})$$

Espressioni di IMP: Sintassi

Una possibile definizione ricorsiva per le espressioni aritmetiche:

$$\frac{}{z}(z \in \mathbb{Z}) \qquad \text{costanti intere}$$

$$\frac{}{x}(x \in Var) \qquad \text{variabili}$$

$$\frac{e_1 \qquad e_2}{e_1 + e_2} \qquad \text{addizione}$$

$$\frac{e_1 \qquad e_2}{e_1 - e_2} \qquad \text{sottrazione}$$

$\ldots$

Espressioni di IMP: Sintassi

Di solito, per la sintassi di un linguaggio, invece di usare le regole di inferenza si preferiscono usare le grammatiche libere dal contesto, che offrono una notazione più compatta.

Ecco come riscrivere le regole precedenti:

$$
\begin{array}{rll}
Exp ::= & z \in \mathbb{Z} & \text{costanti intere} \\
| & x \in Var & \text{variabili} \\
| & Exp + Exp & \text{addizione} \\
| & \cdots & \text{(altri operatori aritmetici)}
\end{array}
$$

Non daremo una definizione formale di grammatica.

Semantica delle espressioni

Semantica delle Espressioni

La *semantica* delle espressioni è una relazione $(\rightarrow_e)$ che associa ad ogni espressione e e ad ogni stato σ il valore di e.

La indicheremo con

$$(\rightarrow_e) \in \mathcal{P}(Exp \times State \times \mathbb{Z})$$
$$\langle e, \sigma \rangle \rightarrow_e v$$

Esempio:

$$\langle x + y + 4, \sigma \rangle \rightarrow_e 11$$

se $\sigma(x) = 2$ e $\sigma(y) = 5$.

Semantica delle Espressioni

Definirla formalmente è molto semplice:

Def. La semantica delle espressioni è definita induttivamente come segue

$$\rightarrow_e \in \mathcal{P}(Exp \times State \times \mathbb{Z})$$

$$\frac{}{\langle z\ ,\ \sigma\rangle \rightarrow_e z}[Lit]$$

$$\frac{}{\langle x\ ,\ \sigma\rangle \rightarrow_e \sigma(x)}[Var]$$

$$\frac{\langle e_1\ ,\ \sigma\rangle \rightarrow_e z_1 \qquad \langle e_2\ ,\ \sigma\rangle \rightarrow_e z_2}{\langle e_1 + e_2\ ,\ \sigma\rangle \rightarrow_e z_1 + z_2}[Plus]$$

$\vdots$

simbolo operazione

Determinismo

Lemma La semantica delle espressioni è *deterministica*.
Ovvero:

$$\langle e, \sigma \rangle \to_e z_1 \ \wedge \ \langle e, \sigma \rangle \to_e z_2 \implies z_1 = z_2$$

per ogni e, σ, z_1, z_2.

In altri termini, il risultato di $(\to_e)$ è unico.

Dim. Per induzione sulle regole di $(\to_e)$.

Determinismo (2)

Basta riscrivere l'enunciato nella forma $(\rightarrow_e) \subseteq p$:

$$\forall e, \sigma, z_1.\ \langle e, \sigma \rangle \rightarrow_e z_1 \implies p(e, \sigma, z_1)$$

dove

$$p(e, \sigma, z_1): \qquad (\forall z_2.\ \langle e, \sigma \rangle \rightarrow_e z_2 \implies z_1 = z_2)$$

e verificare che p sia preservata da ogni regola di $(\rightarrow_e)$. Questo prova che $\hat{\mathcal{R}}(p) \subseteq p$ e quindi che $(\rightarrow_e) \subseteq p$, che è l'enunciato di cui sopra.

Svolgiamo solo il caso della regola $[Plus]$ e lasciamo per esercizio il verificare gli altri casi.

Determinismo (3)

Caso [Plus].

$$\frac{\langle e_1\ ,\ \sigma\rangle \to_e z_1 \qquad \langle e_2\ ,\ \sigma\rangle \to_e z_2}{\langle e_1 + e_2\ ,\ \sigma\rangle \to_e z_1 + z_2}[Plus]$$

Come ipotesi induttive abbiamo

$$IP1:\ p(e_1, \sigma, z_1)$$
$$IP2:\ p(e_2, \sigma, z_2)$$

e dobbiamo dimostrare $p(e_1 + e_2, \sigma, z_1 + z_2)$, ovvero (rinominando $\forall z_2$ in $\forall z$ per non fare confusione):

$$\forall z.\ \langle e_1 + e_2\ ,\ \sigma\rangle \to_e z \implies z_1 + z_2 = z$$

Assumiamo quindi $IP3:\ \langle e_1 + e_2\ ,\ \sigma\rangle \to_e z$ e dimostriamo $z_1 + z_2 = z$.

Determinismo (4)

Invertendo $IP3$, notiamo che può essere derivata solo dalla regola $[Plus]$.

Ne segue che $z = \bar{z}_1 + \bar{z}_2$ per qualche valore $\bar{z}_1, \bar{z}_2 \in \mathbb{Z}$ tale che:

$$IP4: \ \langle e_1 \ , \ \sigma \rangle \rightarrow_e \bar{z}_1$$
$$IP5: \ \langle e_2 \ , \ \sigma \rangle \rightarrow_e \bar{z}_2$$

Da $IP4$ e $IP1$ (scegliendo $z = \bar{z}_1$) si ha $\bar{z}_1 = z_1$. Analogamente, da $IP5$ e $IP2$ (scegliendo $z = \bar{z}_2$) si ha $\bar{z}_2 = z_2$.

Quindi, $z = \bar{z}_1 + \bar{z}_2 = z_1 + z_2$ che è la tesi.

Totalità

Lemma La semantica delle espressioni è *totale*.
Ovvero:

$$\forall e, \sigma.\ \exists z \in \mathbb{Z}.\ \langle e, \sigma \rangle \to_e z$$

Assieme al determinismo, consente di vedere $(\to_e)$ come una funzione $Exp \times State \to \mathbb{Z}$.

Dim. Per induzione sulle regole della sintassi di Exp. (Lasciata per esercizio)

Nota bene: non si può qui procedere per induzione sulle regole di $(\to_e)$! (Perché?)

Comandi di IMP

Sintassi Completa di IMP

Espressioni Aritmetiche

$Exp ::=$	$z \in \mathbb{Z}$		costanti intere
$\mid$	$x \in Var$		variabili
$\mid$	$Exp + Exp$		addizione
$\mid$	$\cdots$		(altri operatori aritmetici)

Comandi

$Com ::=$	skip		"non fare nulla"
$\mid$	$x := Exp$		assegnamento
$\mid$	$Com; Com$		composizione
$\mid$	if $Exp \neq 0$ then Com else Com		condizionale
$\mid$	while $Exp \neq 0$ do Com		ciclo while

Semantica dei comandi “a passi grandi” (big step)

Semantica big step

La *semantica big step* dei comandi di IMP è una relazione $(\rightarrow_b)$ che associa ad ogni comando c e ad ogni stato delle variabili σ, lo stato σ' in cui le variabili si trovano dopo avere eseguito c.

Normalmente σ viene chiamato *stato iniziale* mentre σ' viene chiamato *stato finale*.

Esempio.

$$\langle x := x + 1; y := x * x \ , \sigma \rangle \rightarrow_b \sigma'$$

dove:

$$\sigma(x) = 2 \quad \sigma(y) = 1000$$

$$\sigma'(x) = 3 \quad \sigma'(y) = 9$$

Semantica big step

Def. La semantica big step è definita come segue:

$$\rightarrow_b \in \mathcal{P}(Com \times State \times State)$$

$$\frac{}{\langle \mathsf{skip}\ ,\ \sigma\rangle \rightarrow_b \sigma}[Skip]$$

$$\frac{\langle e, \sigma\rangle \rightarrow_e v}{\langle x := e\ ,\ \sigma\rangle \rightarrow_b \sigma[x \mapsto v]}[Let]$$

stato uguale a σ tranne che su x dove vale v

$$\text{dove } \sigma[x \mapsto v] = \lambda y. \begin{cases} v & \text{se } y = x \\ \sigma(y) & \text{altrimenti} \end{cases}$$

$$\frac{\langle c_1\ ,\ \sigma\rangle \rightarrow_b \sigma' \qquad \langle c_2\ ,\ \sigma'\rangle \rightarrow_b \sigma''}{\langle c_1; c_2\ ,\ \sigma\rangle \rightarrow_b \sigma''}[Comp]$$

Semantica big step

$$\frac{\langle e, \sigma\rangle \rightarrow_e v \neq 0 \qquad \langle c_1, \sigma\rangle \rightarrow_b \sigma'}{\langle \text{if } e \neq 0 \text{ then } c_1 \text{ else } c_2 \ , \ \sigma\rangle \rightarrow_b \sigma'}[If - true]$$

$$\frac{\langle e, \sigma\rangle \rightarrow_e 0 \qquad \langle c_2, \sigma\rangle \rightarrow_b \sigma'}{\langle \text{if } e \neq 0 \text{ then } c_1 \text{ else } c_2 \ , \ \sigma\rangle \rightarrow_b \sigma'}[If - false]$$

Semantica big step

$$\frac{\langle e, \sigma\rangle \rightarrow_e v \neq 0 \qquad \langle c; \text{while } e \neq 0 \text{ do } c\ ,\ \sigma\rangle \rightarrow_b \sigma'}{\langle \text{while } e \neq 0 \text{ do } c\ ,\ \sigma\rangle \rightarrow_b \sigma'}[While - true]$$

$$\frac{\langle e, \sigma\rangle \rightarrow_e 0}{\langle \text{while } e \neq 0 \text{ do } c\ ,\ \sigma\rangle \rightarrow_b \sigma}[While - false]$$

(Fine delle regole)

Esempio

Sia

$$c = \Big(x := x + 4;\ y := x + y\Big)$$

e sia σ tale che $\sigma(x) = 1, \sigma(y) = 2$.
Eseguire c partendo da σ termina in $\sigma' = \sigma[x \mapsto 5][y \mapsto 7]$

$$\frac{\dfrac{\langle x + 4\ , \sigma\rangle \to_e 5}{\langle x := x + 4\ , \sigma\rangle \to_b \sigma[x \mapsto 5]}[Let]\quad \dfrac{\langle x + y\ , \sigma[x \mapsto 5]\rangle \to_e 7}{\langle y := x + y\ , \sigma[x \mapsto 5]\rangle \to_b \sigma'}[Let]}{\langle x := x + 4; y := x + y\ , \sigma\rangle \to_b \sigma'}[Comp]$$

Esempio: somma 1..n

Calcoliamo $1 + 2 + \cdots + n$ nel modo banale.

Sia $\sigma(n) \geq 0$ e sia

$$c = \Big(x := 0;\ \mathsf{while}\ n \neq 0\ \mathsf{do}\ (x := x + n;\ n := n - 1)\Big)$$

Allora

$$\langle c\ ,\ \sigma\rangle \to_b \sigma[n \mapsto 0][x \mapsto \sum_{i=1}^{\sigma(n)} i]$$

Notate che, intuitivamente, la somma viene calcolata dal programma "in ordine inverso", cioè come $n + (n-1) + \cdots + 1$.

Derivazione per un while

$c = (x := 0;\ c') \quad c' = \mathsf{while}\ n \neq 0\ \mathsf{do}\ c'' \quad c'' = (x := x + n;\ n := n - 1)$

Sotto scriviamo σ_b^a per $\sigma[x \mapsto a][n \mapsto b]$.

$$\dfrac{\langle x := 0, \sigma_2^a\rangle \to_b \sigma_2^0 \qquad \dfrac{\langle n, \sigma_2^0\rangle \to_e 2 \neq 0 \qquad \dfrac{\dfrac{\vdots}{\langle c'', \sigma_2^0\rangle \to_b \sigma_1^2}[C] \quad \dfrac{D}{\langle c', \sigma_1^2\rangle \to_b \sigma_0^3}}{\langle c'; c', \sigma_2^0\rangle \to_b \sigma_0^3}[C]}{\langle c', \sigma_2^0\rangle \to_b \sigma_0^3}[Wt]}{\langle c, \sigma_2^a\rangle \to_b \sigma_0^3}[C]$$

dove

$$D = \dfrac{\langle n, \sigma_1^2\rangle \to_e 1 \neq 0 \qquad \dfrac{\dfrac{\vdots}{\langle c'', \sigma_1^2\rangle \to_b \sigma_0^3}[C] \quad \dfrac{\langle n, \sigma_0^3\rangle \to_e 0}{\langle c', \sigma_0^3\rangle \to_b \sigma_0^3}[Wf]}{\langle c'; c', \sigma_1^2\rangle \to_b \sigma_0^3}[C]}{\langle c', \sigma_1^2\rangle \to_b \sigma_0^3}[Wt]$$

Esempi

Calcoliamo $1^2 + 2^2 + \cdots + n^2$ nel modo banale.

Sia $\sigma(n) \geq 0$ e sia

$$c = \Big(x := 0;\ \mathsf{while}\ n \neq 0\ \mathsf{do}\ (x := x + n * n;\ n := n - 1)\Big)$$

Allora

$$\langle c\ ,\ \sigma\rangle \to_b \sigma[n \mapsto 0][x \mapsto \sum_{i=0}^{\sigma(n)} i^2]$$

Dimostrazioni di Correttezza

Le affermazioni fatte riguardo agli esempi precedenti sono facili da verificare quando $\sigma(n) > 0$ è un valore noto.

Richiedono "solo" la costruzione di una (lunga e noiosa) derivazione per $\langle c, \sigma \rangle \rightarrow_b \sigma'$.

Dimostrarle invece per ogni valore di $\sigma(n) > 0$ è possibile, ma meno banale. Un modo potrebbe essere quello di procedere per induzione sul naturale $\sigma(n)$, e fare vedere che per ogni valore di $\sigma(n)$ il programma funziona.

Vedremo più avanti strumenti più facili da usare per dimostrare la correttezza di un programma. Per ora, ci basiamo sull'intuizione e non dimostriamo nulla.

Esempi

Problema: scambiare il valore di due variabili x, y.

Il programma banale non funziona:

$$c = \Big(x := y;\ y := x\Big)$$

perché ho che, se per esempio $\sigma(x) = 1$ e $\sigma(y) = 2$:

$$\langle c\ ,\ \sigma\rangle \to_b \sigma[x \mapsto 2][y \mapsto 2]$$

Sovrascrivere x con y fa perdere il valore originale di x.

Esempi

Problema: scambiare il valore di due variabili x, y.

Usiamo una variabile t "di appoggio":

$$c = \Big(t := x;\ x := y;\ y := t\Big)$$

Ora abbiamo che (con $\sigma(x) = 1$, $\sigma(y) = 2$ e $\sigma(t) = 1000$):

$$\langle c\ ,\ \sigma\rangle \to_b \sigma[x \mapsto 2][y \mapsto 1][t \mapsto 1]$$

Notate che si è perso il valore originale di t.

Esercizio

Esercizio. Scrivete una derivazione per il programma appena visto.

Esempi

Problema: scambiare il valore di due variabili x, y.

Senza variabile di appoggio:

$$c = \Big(x := x + y;\ y := x - y;\ x := x - y\Big)$$

Ora ho che (con $\sigma(x) = 1$, $\sigma(y) = 2$):

$$\langle c\ ,\ \sigma\rangle \to_b \sigma[x \mapsto 2][y \mapsto 1]$$

Seppur tecnicamente corretto, questo programma è più complicato (e meno leggibile) del precedente. Si preferisce usare la variabile di appoggio che sfruttare "trucchi" come quello di sopra.

Alcune proprietà della semantica di IMP

Determinismo

Lemma. La semantica big step di IMP è *deterministica*.

$$\langle c, \sigma \rangle \to_b \sigma_1 \ \wedge \ \langle c, \sigma \rangle \to_b \sigma_2 \implies \sigma_1 = \sigma_2$$

per ogni $c, \sigma, \sigma_1, \sigma_2$.
Dim.(bozza) L'enunciato è equivalente a:

$$\forall c, \sigma, \sigma_1. \langle c, \sigma \rangle \to_b \sigma_1 \implies (\forall \sigma_2. \langle c, \sigma \rangle \to_b \sigma_2 \implies \sigma_1 = \sigma_2) \quad (*)$$

quindi basta definire $p \in \mathcal{P}(Com \times State \times State)$ come:

$$p(c, \sigma, \sigma_1) \iff (\forall \sigma_2. \langle c, \sigma \rangle \to_b \sigma_2 \implies \sigma_1 = \sigma_2)$$

e notare che $(*)$ è equivalente a $(\to_b) \subseteq p$, e quindi procedere per induzione sulla semantica.

Esercizio. Provate a fare il caso $[If - true]$.

Non totalità

Lemma. La semantica big step di IMP è *non totale*.
Per esempio:

$$\nexists \sigma'.\ \langle \mathsf{while}\ 1 \neq 0\ \mathsf{do}\ \mathsf{skip}\ ,\ \sigma \rangle \to_b \sigma'$$

Dim. Basta definire:

$$\begin{array}{l} w = \mathsf{while}\ 1 \neq 0\ \mathsf{do}\ \mathsf{skip} \\ p \in \mathcal{P}(Com \times State \times State) \\ p(c, \sigma, \sigma') \iff c \neq w \wedge c \neq (\mathsf{skip};\ w) \end{array}$$

e dimostrare che

$$(\to_b) \subseteq p$$

il che, per induzione, segue da

$$\hat{\mathcal{R}}(p) \subseteq p$$

Non totalità

Basta quindi vedere che p è preservata da ogni regola di $(\rightarrow_b)$. Vediamo solo qualche caso:

Caso [Comp]

$$\frac{\langle c_1 \ , \ \sigma\rangle \rightarrow_b \sigma' \qquad \langle c_2 \ , \ \sigma'\rangle \rightarrow_b \sigma''}{\langle c_1; c_2 \ , \ \sigma\rangle \rightarrow_b \sigma''}[Comp]$$

Qui abbiamo come ipotesi induttive

$$p(c_1, \sigma, \sigma') \qquad \text{e} \qquad p(c_2, \sigma', \sigma'')$$

e dobbiamo dimostrare $p(c_1; c_2, \sigma, \sigma'')$, cioè che 1) $c_1; c_2 \neq w$ e 2) che $c_1; c_2 \neq (\mathsf{skip}; w)$. La 1) è banale, la 2) segue da $c_2 \neq w$ che segue da $p(c_2, \sigma', \sigma'')$.

Non totalità

Caso [While-true]

$$\frac{\langle e, \sigma\rangle \to_e v \neq 0 \qquad \langle c; \mathsf{while}\ e \neq 0\ \mathsf{do}\ c\ ,\ \sigma\rangle \to_b \sigma'}{\langle \mathsf{while}\ e \neq 0\ \mathsf{do}\ c\ ,\ \sigma\rangle \to_b \sigma'}[While{-}true]$$

chiamiamo $w' = (\mathsf{while}\ e \neq 0\ \mathsf{do}\ c)$. Qui sappiamo che $\langle e, \sigma\rangle \to_e v \neq 0$ e inoltre abbiamo come ipotesi induttiva

$$p(c; w', \sigma, \sigma') \quad \text{cioè} \quad c; w' \neq w \wedge c; w' \neq \mathsf{skip}; w$$

e dobbiamo dimostrare $p(w', \sigma, \sigma')$, cioè che 1) $w' \neq w$ e 2) che $w' \neq (\mathsf{skip}; w)$.

La 2) è banale. Per la 1): se per assurdo fosse $w' = w$ avremmo anche $c = \mathsf{skip}$ ma allora $(c; w') = (\mathsf{skip}; w)$ contraddice l'ipotesi induttiva.

Non totalità

Caso [While-false]

$$\frac{\langle e, \sigma\rangle \to_e 0}{\langle \mathsf{while}\ e \neq 0\ \mathsf{do}\ c\ ,\ \sigma\rangle \to_b \sigma}[While - false]$$

chiamiamo $w' = (\mathsf{while}\ e \neq 0\ \mathsf{do}\ c)$. Qui non abbiamo nessuna ipotesi induttiva, ma sappiamo che $\langle e, \sigma\rangle \to_e 0$. Da questo dobbiamo dimostrare $p(w', \sigma, \sigma)$, cioè che 1) $w' \neq w$ e 2) che $w' \neq (\mathsf{skip}; w)$.

La 2) è banale.
Per la 1): se per assurdo fosse $w' = w$, avrei che $e = 1$, ma in tal caso sarebbe impossibile avere $\langle e, \sigma\rangle \to_e 0$.

Non totalità

Esercizio. Gli altri casi sono banali: verificateli.

Non totalità

Una dimostrazione meno formale ma più intuitiva per la non totalità può essere fatta osservando che non è possibile costriure una derivazione per $\langle w, \sigma \rangle \to_b \sigma'$.

Se esistesse, dovrebbe essere della forma

$$\dfrac{\langle 1, \sigma \rangle \to_e 1 \neq 0 \qquad \dfrac{\langle \mathsf{skip}, \sigma \rangle \to_b \sigma \; [Skip] \qquad \dfrac{\vdots}{\langle w, \sigma \rangle \to_b \sigma'}\,[Wt]}{\langle \mathsf{skip}; w, \sigma \rangle \to_b \sigma'}\,[C]}{\langle w, \sigma \rangle \to_b \sigma'}\,[Wt]$$

ma al posto dei puntini ho bisogno proprio della derivazione stessa, creando un circolo vizioso.

Non totalità

Esercizio. Perché nella prima dimostrazione abbiamo preso una relazione p così complicata?

$$p(c, \sigma, \sigma') \iff c \neq w \wedge c \neq \mathsf{skip}; w$$

Avremmo potuto prendere la seguente più semplice:

$$p'(c, \sigma, \sigma') \iff c \neq w$$

e da $(\rightarrow_b) \subseteq p'$ avremmo *comunque* ricavato il lemma!

Convincetevi che p' è vera (è implicata da p !) ma non è induttiva, cioè:

$$\hat{\mathcal{R}}(p') \not\subseteq p'$$

Suggerimento: cosa succede nel caso $[While - true]$?

Semantica come Funzione

FUORI ESAME

Abbiamo visto che, dato uno stato iniziale (di input), lo stato finale (di output) dell'esecuzione di un programma può non esistere (non totalità), ma se lo fa è unico (determinismo).

Questo fa sì che l'esecuzione di un programma non possa essere correttamente descritta come una funzione

$$Input \to Output$$

ma solo come una funzione

$$Input \to Output \cup \{\text{“nessun risultato”}\}$$

Semantica come Funzione

FUORI ESAME

Si usa a volte la notazione $Output_\perp$ per intendere appunto l'insieme $Output$ arricchito del caso "nessun risultato", denotato col simbolo $\perp$ (o a volte $\uparrow$).

Le funzioni

$$Input \to Output_\perp$$

vengono anche chiamate "funzioni parziali".

Nota: alcuni risultati in Teoria della Ricorsione fanno capire che la parzialità dei programmi è una caratteristica ineludibile della computazione. Ogni linguaggio di programmazione "sufficientemente potente" deve avere una semantica non totale.

Equivalenza

Def. Due programmi c_1, c_2 vengono detti equivalenti ($c_1 \equiv c_2$) quando

$$\langle c_1 \ , \sigma \rangle \rightarrow_b \sigma'$$
$$\Longleftrightarrow$$
$$\langle c_2 \ , \sigma \rangle \rightarrow_b \sigma'$$

per ogni σ, σ'.

Esercizi

Esercizio. Dimostrate che la composizione tra comandi è associativa:

$$(c_1; c_2); c_3 \;\equiv\; c_1; (c_2; c_3)$$

(Osservate le derivazioni...)

Esercizio. Dimostrate che, in generale, non è vero che

$$\mathsf{while}\ e \neq 0\ \mathsf{do}\ (c_1; c_2) \qquad \equiv \qquad (\mathsf{while}\ e \neq 0\ \mathsf{do}\ c_1); c_2$$

Esercizio

Esercizio. Quali delle seguenti equivalenze valgono?

$$\mathsf{if}\ e \neq 0\ \mathsf{then}\ (c_1; c)\ \mathsf{else}\ (c_2; c) \quad \equiv \quad (\mathsf{if}\ e \neq 0\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2); c$$

$$\mathsf{if}\ e \neq 0\ \mathsf{then}\ (c; c_1)\ \mathsf{else}\ (c; c_2) \quad \equiv \quad c; (\mathsf{if}\ e \neq 0\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)$$

Dimostrate quelle che valgono osservando le derivazioni, e trovate i controesempi per le altre.

Esercizio

Esercizio. Dimostrate la "proprietà di unfolding" del while:

$$\mathsf{while}\ e \neq 0\ \mathsf{do}\ c$$
$$\equiv$$
$$\mathsf{if}\ e \neq 0\ \mathsf{then}\ (c;\ \mathsf{while}\ e \neq 0\ \mathsf{do}\ c)\ \mathsf{else}\ \mathsf{skip}$$

Esercizi

Esercizio. È vero che il while è idempotente nel senso che segue?

$$\mathsf{while}\ e \neq 0\ \mathsf{do}\ c$$
$$\equiv$$
$$(\mathsf{while}\ e \neq 0\ \mathsf{do}\ c); (\mathsf{while}\ e \neq 0\ \mathsf{do}\ c)$$

Esercizio. Dimostrare se

$$\mathsf{while}\ e \neq 0\ \mathsf{do}\ c$$
$$\equiv$$
$$\mathsf{while}\ e \neq 0\ \mathsf{do}\ (c; c)$$

Esercizi

Esercizio. Riscrivere i programmi che seguono in una forma più leggibile ed equivalente.

```
if x ≠ 0 then
        x := 0
else
        skip
```

```
x := x * x + 10;
while x − 2 ≠ 0 do
        x := x − 1
```

Esercizio

Esercizio. Aggiungete ad IMP le espressioni booleane

$$\begin{array}{rcl} Bexp & ::= & Exp \leq Exp \\ & | & Exp = Exp \\ & | & Bexp \wedge Bexp \\ & | & \neg Bexp \\ & | & \mathsf{true} \\ & | & \ldots \end{array}$$

Definite un insieme di regole per la loro semantica attraverso una relazione $\rightarrow_{be}$ (che tipo di relazione è?).

Modificate sintassi e semantica di if e while in modo da consentire aritrarie espressioni booleane al posto di "$e \neq 0$".

Semantica dei comandi “a passi piccoli” (small step)

Semantica small step

La semantica big step definisce la corrispondenza tra stati iniziali e stati finali di un programma.

Ci consente di dire che, per esempio

$$c_1 = \quad n := 100;\ x := 0;\ \textsf{while}\ n \neq 0\ \textsf{do}\ (x := x + n;\ n := n - 1)$$
$$c_2 = \quad n := 0; x := 5050$$

sono equivalenti.

Nonostante l'equivalenza semantica, intuivamente eseguire c_1 "richiede più calcoli" di eseguire c_2. Infatti c_1 svolge 100 addizioni, mentre c_2 nemmeno una.

Semantica small step

Per differenziare i comandi

$$c_1 = \quad n := 100;\ x := 0;\ \textsf{while}\ n \neq 0\ \textsf{do}\ (x := x + n;\ n := n - 1)$$
$$c_2 = \quad n := 0;\ x := 5050$$

abbiamo bisogno di una semantica più fine, che tenga conto dei singoli passi elementari di calcolo.

La semantica small step $(\rightarrow_s)$ è una relazione

$$(\rightarrow_s) \in \mathcal{P}(Com \times State \times Com \times State)$$
$$\langle c, \sigma \rangle \rightarrow_s \langle c', \sigma' \rangle$$

che, dato $\langle c, \sigma \rangle$, esegue solo il "primo" passo elementare di calcolo, restituendo $\langle c', \sigma' \rangle$ dove σ' è lo stato modificato da questo primo passo, mentre c' è ciò che resta da eseguire di c (la cosidetta "continuazione" di c).

Semantica small step

Esempio.

$$
\begin{aligned}
& \langle x := 10;\ x := x + 1;\ x := x * 2\ ,\ \sigma \rangle \\
\rightarrow_s \quad & \langle \mathsf{skip};\ x := x + 1;\ x := x * 2\ ,\ \sigma[x \mapsto 10] \rangle \\
\rightarrow_s \quad & \langle x := x + 1;\ x := x * 2\ ,\ \sigma[x \mapsto 10] \rangle \\
\rightarrow_s \quad & \langle \mathsf{skip};\ x := x * 2\ ,\ \sigma[x \mapsto 11] \rangle \\
\rightarrow_s \quad & \langle x := x * 2\ ,\ \sigma[x \mapsto 11] \rangle \\
\rightarrow_s \quad & \langle \mathsf{skip}\ ,\ \sigma[x \mapsto 22] \rangle \\
\not\rightarrow_s \quad &
\end{aligned}
$$

Semantica small step

FUORI ESAME

Def. La semantica small step è definita come segue:

$$\rightarrow_s \in \mathcal{P}(Com \times State \times Com \times State)$$

$$\frac{\langle e, \sigma \rangle \rightarrow_e v}{\langle x := e \ , \ \sigma \rangle \rightarrow_s \langle \mathsf{skip} \ , \ \sigma[x \mapsto v] \rangle}[Let]$$

$$\frac{}{\langle \mathsf{skip}; c_2 \ , \ \sigma \rangle \rightarrow_s \langle c_2 \ , \ \sigma \rangle}[Comp1]$$

$$\frac{\langle c_1 \ , \ \sigma \rangle \rightarrow_s \langle c_1' \ , \ \sigma' \rangle}{\langle c_1; c_2 \ , \ \sigma \rangle \rightarrow_s \langle c_1'; c_2 \ , \ \sigma' \rangle}[Comp2]$$

Semantica small step

FUORI ESAME

$$\frac{\langle e, \sigma\rangle \rightarrow_e v \neq 0}{\langle \text{if } e \neq 0 \text{ then } c_1 \text{ else } c_2 \ , \ \sigma\rangle \rightarrow_s \langle c_1 \ , \ \sigma\rangle}[If - true]$$

$$\frac{\langle e, \sigma\rangle \rightarrow_e 0}{\langle \text{if } e \neq 0 \text{ then } c_1 \text{ else } c_2 \ , \ \sigma\rangle \rightarrow_s \langle c_2 \ , \ \sigma\rangle}[If - false]$$

Semantica small step

FUORI ESAME

$$\frac{\langle e, \sigma\rangle \rightarrow_e v \neq 0}{\langle \text{while } e \neq 0 \text{ do } c \ , \ \sigma\rangle \rightarrow_s \langle c; \text{while } e \neq 0 \text{ do } c \ , \ \sigma\rangle}[While - true]$$

$$\frac{\langle e, \sigma\rangle \rightarrow_e 0}{\langle \text{while } e \neq 0 \text{ do } c \ , \ \sigma\rangle \rightarrow_s \langle \text{skip} \ , \ \sigma\rangle}[While - false]$$

(Fine delle regole)

Semantica small step

FUORI ESAME

Lemma. La semantica small step di IMP è *totale*, tranne per skip. Cioè:

$$\forall c \neq \mathsf{skip}, \sigma.\ \exists c', \sigma'.\ \langle c, \sigma \rangle \to_s \langle c', \sigma' \rangle$$

Lemma. La semantica small step di IMP è *deterministica*.

$$\langle c, \sigma \rangle \to_s \langle c_1, \sigma_1 \rangle \wedge \langle c, \sigma \rangle \to_s \langle c_2, \sigma_2 \rangle \implies c_1 = c_2 \wedge \sigma_1 = \sigma_2$$

(Non li dimostriamo)

Corrispondenza small/big step

FUORI ESAME

Teorema. Le semantiche small step e big step di IMP sono equivalenti.

chiamata "stella di Kleene"

$$\langle c, \sigma \rangle \to_b \sigma' \iff \langle c, \sigma \rangle \to_s^* \langle \mathsf{skip}, \sigma' \rangle$$

dove $\to_s^*$ sta per la chiusura transitiva e riflessiva della relazione $\to_s$, cioè:

$$\langle c, \sigma \rangle \to_s^* \langle c', \sigma' \rangle$$
$$\iff$$
$$\langle c, \sigma \rangle \underbrace{\to_s \langle c_1, \sigma_1 \rangle \to_s \langle c_2, \sigma_2 \rangle \to_s \langle c_3, \sigma_3 \rangle \to_s \cdots \to_s}_{n \geq 0 \text{ passi}} \langle c', \sigma' \rangle$$

incluso zero $(c = c', \sigma = \sigma')$

Corrispondenza small/big step

FUORI ESAME

Non lo dimostriamo.

Esercizio. Provate a dimostrare il verso $\Longrightarrow$ per induzione, facendo solo qualche caso.

Perché due semantiche

La semantica big step è in genere più comoda per dimostrare proprietà dei programmi.

La semantica small step in compenso ci dà un'informazione aggiuntiva rispetto alla big step, in quanto ci consente di definire quanti passi elementari di calcolo servono a un programma per raggiungere il suo risultato.

La small step è anche usata per poter definire "macchine" che siano in grado di eseguire i programmi.

Complessità Computazionale

Def. Dato un programma (o un algoritmo) che opera su un input n, si definisce la *complessità computazionale* del programma come la funzione $T(n)$ che associa ad ogni input n il numero di passi di $(\rightarrow_s)$ che sono necessari per completare l'esecuzione del programma (cioè raggiungere $\langle \mathsf{skip}, \sigma' \rangle$).

Nota: a volte, si preferisce usare n per indicare non l'input, ma una qualche "taglia" o "misura" della dimensione dell'input. Per esempio, se un programma ricerca una parola in un dizionario, n di solito indica il numero dei lemmi nel dizionario piuttosto che il dizionario stesso.

Complessità Computazionale

Esempio:

$$\begin{aligned} c_1 = &\quad x := 0;\ \textsf{while}\ n \neq 0\ \textsf{do}\ (x := x + n;\ n := n - 1) \\ c_2 = &\quad x := n \cdot (n+1)/2 \end{aligned}$$

Il programma c_1 ha complessità:

$$T(n) = 2 + 5 \cdot n + 1 = 5 \cdot n + 3$$

Il programma c_2 invece ha:

$$T(n) = 1$$

Notazione O grande

Di solito è difficile dare il valore esatto di $T(n)$, ma è possibile invece stimarne l'andamento asintotico.

Def. (Notazione "O grande") Siano date due funzioni $f, g \in (\mathbb{N} \to \mathbb{N})$. Diciamo che "$f$ è $O(g(n))$" quando

$$\exists k > 0, \bar{n}.\ \forall n > \bar{n}.\ f(n) \leq k \cdot g(n)$$

cioè quando f è definitivamente superata da g, a meno di una costante moltiplicativa.

Complessità Asintotica

Riprendendo l'esempio precedente:

$$\begin{aligned} c_1 = &\quad x := 0;\ \textsf{while}\ n \neq 0\ \textsf{do}\ (x := x + n;\ n := n - 1) \\ c_2 = &\quad x := n \cdot (n+1)/2 \end{aligned}$$

Diciamo che c_1 ha complessità:

$$T(n) = O(n)$$

Il programma c_2 invece ha:

$$T(n) = O(1)$$

Algoritmo per la Potenza

Problema: dati $a, b \geq 0$ calcolare a^b (supponendo $0^0 = 1$).

Algoritmo banale: fare b moltiplicazioni

$r := 1;$
while $b \neq 0$ do
    $r := r * a;$
    $b := b - 1$

La complessità asintotica dipende solo da b:

$$T(b) = O(b)$$

Algoritmo “del Contadino Russo”

```
r := 1;
while b ≠ 0 do
    if b pari then
        a := a * a;
        b := b/2
    else
        r := r * a;
        b := b − 1
```

(supponiamo che esista l’operatore “pari”)

Si noti che dopo ogni ciclo con b dispari ve ne è uno con b pari, in cui b viene dimezzato.

Quindi la complessità asintotica diventa:

$$T(b) = O(\log_2 b) = O(\log b)$$

(si ricorda che tutti i logaritmi con base > 1 differiscono per una costante, quindi la base è irrilevante)

Valutare un Polinomio

Problema: dati i coefficienti di un polinomio p di grado n e un punto x, valutare $p(x)$.

$$p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n$$

Esercizio. Verificare che la valutazione ingenua della formula di sopra usa $O(n^2)$ moltiplicazioni.

Esercizio. Verificare che usando l'algoritmo del contadino russo posso ridurre la complessità a $O(n \cdot \log n)$.

Esercizio. Verificare che si può ulteriormente migliorare la complessità fino a $O(n)$ come segue:

$$p(x) = a_0 + x \cdot \Big(a_1 + x \cdot \Big(a_2 + x \cdot \Big(\cdots + x \cdot a_n \Big)\Big)\Big)$$