Informatica

CdL in Matematica A.A. 2014/2015

Parte 6

Roberto Zunino

Introduzione

Iniziamo ora ad affrontare alcuni argomenti pi avanzati di programmazione imperativa, che vanno oltre ai costrutti di base presenti nel linguagigo IMP.

Per rendere la trattazione pi veloce, ci limiteremo ad affrontare questi argomenti solo in termini intuitivi, senza darne una completa descrizione formale come abbiamo fatot invece per IMP.

Per alcuni di essi, segnaliamo il corso opzionale di Linguaggi di Programmazione.

Riferimenti

Sul sito web del corso sono presenti alcuni esempi di programmi Java.

Questi esempi vogliono sia abituarvi alla sintassi del linguaggio in questione, sia mostrarvi l'utilizzo di alcuni costrutti di programmazione più avanzati.

La maggior parte della discussione verrà fatta direttamente dentro questi esempi, mentre useremo le slide solo per aggiungere qualche commento.

Gli esempi proposti non vogliono essere un presentazione esaustiva agli argomenti illustrati: durante le esercitazioni in laboratorio verranno chiariti ulteriori aspetti.

Commenti agli Esempi

Variabili non Intere

Si veda prima il codice `Esempio03.java`

Aggiungendo variabili non intere, la nostra semantica viene modificata come segue:

$$Value = \mathbb{Z} \uplus \mathbb{R} \uplus \mathbb{B} \uplus \mathsf{String} \uplus \ldots$$

$$\sigma \in (Var \to Value)$$

$$(\to_e) \in \mathcal{P}(Exp \times Store \times Value)$$

Funzioni ed Effetti Collaterali

Si veda prima il codice `Esempio09.java`

Aggiungendo le funzioni alle espressioni, di fatto abbiamo modificato la semantica delle espressioni come segue

$$(\rightarrow_e) \in \mathcal{P}(Exp \times Store \times Value \times Store)$$
$$\langle e, \sigma \rangle \rightarrow_e \langle v, \sigma' \rangle$$

Molte funzioni classiche (es: $cos(x)$) in Java chiaramente non modificano lo $Store$, e la loro valutazione risulta in un $\sigma' = \sigma$. Queste vengono chiamate funzioni *pure*.

Nel caso generale, però, la valutazione di una chiamata di funzione può modificare σ. In tal caso si parla di funzione *con effetti collaterali*, o di funzione *impura*.

Allocazione Dinamica

Si veda prima il codice `Esempio10.java`

Per introdurre l'allocazione dinamica per i vettori interi, abbiamo aggiunto le espressioni nuove

$$\begin{array}{rcl} Exp & ::= & \ldots \\ & | & \mathsf{new\ int}[Exp] \\ & | & Exp[Exp] \end{array}$$

Abbiamo anche aggiunto il comando

$$\begin{array}{rcl} Com & ::= & \ldots \\ & | & Exp[Exp] := Exp \end{array}$$

Allocazione Dinamica

La semantica delle espressioni di tipo array richiede di aggiungere ai valori un valore "riferimento":

$$Ref = \mathbb{Z}$$
$$Value = \cdots \cup Ref$$

Dobbiamo infine rappresentare la "memoria allocata".

$$\mu \in Mem = (Ref \to Value \uplus \{\mathsf{libera}\})$$

$$(\to_e) \in \mathcal{P}(Exp \times Store \times Mem \times Value \times Store \times Mem)$$
$$\langle e, \sigma, \mu \rangle \to_e \langle v, \sigma', \mu' \rangle$$

$$(\to_b) \in \mathcal{P}(Com \times Store \times Mem \times Store \times Mem)$$
$$\langle c, \sigma, \mu \rangle \to_b \langle \sigma', \mu' \rangle$$

Allocazione Din.: Espressioni

Regola per l'espressione new:

$$\frac{\langle e, \sigma, \mu\rangle \to_e \langle z, \sigma', \mu'\rangle \qquad \mu'(r) = \cdots \mu'(r+z-1) = \mathsf{libera}}{\langle \mathsf{new\ int}[e], \sigma, \mu\rangle \to_e \langle r, \sigma', \mu'[r \mapsto 0, \ldots, r+z-1 \mapsto 0]\rangle}$$

Sopra, ignoriamo il caso in cui si tenti creare un vettore di lunghezza negativa ($z < 0$). Allo stesso modo, ignoriamo il caso in cui non ci siano z posizioni contigue libere ("errore: memoria esaurita"). Il vettore lo inizializziamo per convenzione con il valore 0.

Regola per la lettura da un vettore:

$$\frac{\begin{array}{c}\langle e_1, \sigma, \mu\rangle \to_e \langle r, \sigma', \mu'\rangle \\ \langle e_2, \sigma', \mu'\rangle \to_e \langle z, \sigma'', \mu''\rangle\end{array}}{\langle e_1[e_2], \sigma, \mu\rangle \to_e \langle \mu''(r+z), \sigma'', \mu''\rangle}$$

Allocazione Dinamica: Comandi

Regola per la scrittura in un vettore:

$$\frac{\begin{array}{c}\langle e_1, \sigma, \mu\rangle \to_e \langle r, \sigma', \mu'\rangle \\ \langle e_2, \sigma', \mu'\rangle \to_e \langle z, \sigma'', \mu''\rangle \\ \langle e_3, \sigma'', \mu''\rangle \to_e \langle v, \sigma''', \mu'''\rangle\end{array}}{\langle e_1[e_2] := e_3, \sigma, \mu\rangle \to_b \langle \sigma''', \mu'''[r + z \mapsto v]\rangle}$$

Allocazione Dinamica

Esercizio. Si ci convinca che i seguenti programmi sono validi (per quanto siano astrusi). Descrivetene informalmente il comportamento.

$$\begin{array}{l}\mathsf{int}\ x;\\ x := (\mathsf{new}\ \mathsf{int}[5])[0]\end{array}$$

$$\begin{array}{l}\mathsf{int}[]\ f(\mathsf{int}\ n)\{\mathsf{return}\ \mathsf{new}\ \mathsf{int}[n+10];\}\\ \ldots\\ x := f(f(5)[6])[3];\end{array}$$

Il secondo dichiara una funzione f che ritorna un vettore.

Algoritmi di Ordinamento

Si veda prima il codice `Esempio15.java`

L'algoritmo merge sort definito nell'esempio ha complessità $O(n \cdot \log n)$ dove n è la lunghezza del vettore che dobbiamo ordinare.

Si può dimostrare che tale andamento asintotico è ottimo, ovvero che nessun algoritmo di ordinamento può avere complessità asintotica migliore (a meno di fare ipotesi che restringono i possibili vettori in ingresso).

Ottimalità del merge sort

FUORI ESAME

Diamo solo una bozza di dimostrazione per l'ottimalità del merge sort. (Gli interessati possono trovare la dismotrazione completa sul libro del Cormen "Introduction to Algortihms".)

Consideriamo il caso in cui venga ordinato un vettore i cui elementi sono tutti distinti. Ordinare il vettore in questo caso è equivalente a trovare la permutazione degli elementi che li mette in ordine crescente.

Il numero di permutazioni su n elementi è $n!$.

(n scelte su chi mettere per primo, $n-1$ scelte per il secondo, $\ldots 1$ sull'ultimo)

Ottimalità del merge sort

FUORI ESAME

All'inizio dell'esecuzione di un algoritmo di ordinamento, non avendo ancora ispezionato il vettore, abbiamo che qualsiasi permutazione delle $n!$ è *ammissibile*, cioè potrebbe essere quella che ordina il vettore.

Ogni volta che un algoritmo di ordinamento fa un test sul vettore (per es., $a[3] > a[5]$) le permutazioni ammissibili si restringono: se il test ha successo, diventano quelle compatibili col test, altrimenti quelle compatibili con la sua negazione.

Ottimalità del merge sort

FUORI ESAME

Osservazione: un algortimo corretto non può terminare prima di avere ristretto le permutazioni ammissibili ad una sola.

Intuitivamente, dovrebbe restituire la permutazione che ordina il vettore senza avere ancora ricavato abbastanza informazioni per capire come ordinarlo.

In altri termini, esisterebbero due input (con elementi distinti) che differiscono solo per l'ordine che però verrebbero ordinati secondo la stessa permutazione – assurdo!

Ottimalità del merge sort

FUORI ESAME

Quanti test devo fare quindi per ridurre le permutazioni ammissibili a una sola?

Supponendo di avere m permutazioni ammissibili, un test potrebbe dividermi le permutazioni in una sola compatibile e $m-1$ incompatibile. Se il test ha successo, posso concludere velocemente, ma altrimenti non avrei ridotto di molto m.

Eliminare le permutazioni una ad una mi costerebbe $n!$ test.

Ottimalità del merge sort

FUORI ESAME

Osservazione: se voglio ottimizzare i miei test in modo da ridurne il numero, indipendentemente dal loro successo, dovrei usare test che mi dividono le m permutazioni ammissibili a metà.

Tali test "ideali" mi consentirebbero infatti di minimizzare il numero di permutazioni ammissibili rimaste "nel caso pessimo".

Anche avendo a disposizione tali test "ideali", ne dovrei comunque fare $\log_2(n!)$.

Ottimalità del merge sort

FUORI ESAME

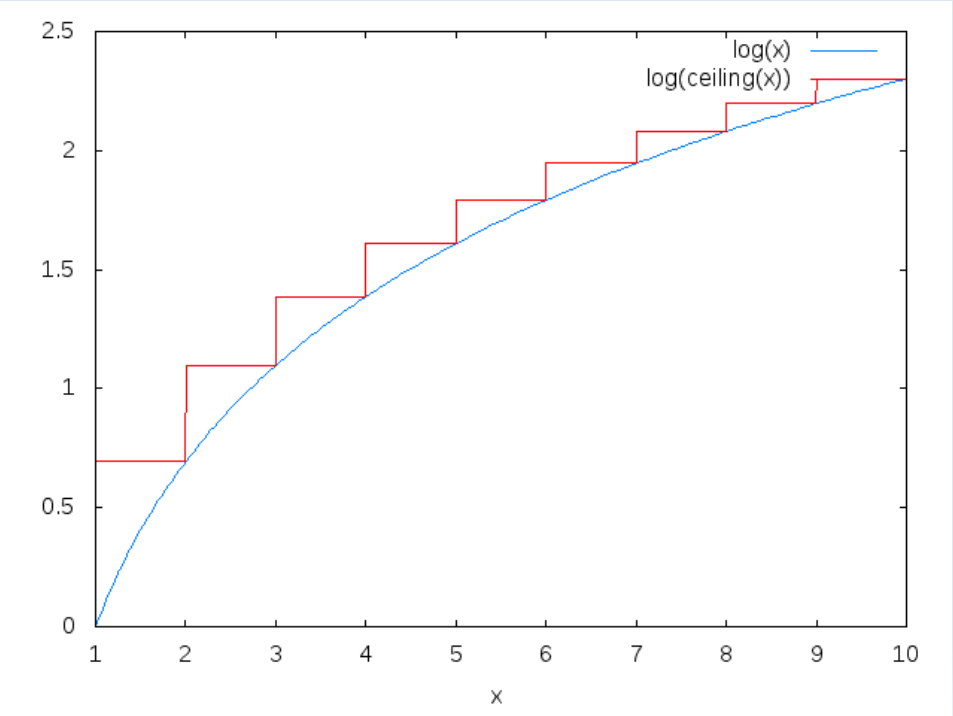
Dobbiamo fare $\log_2(n!)$ test ma:

Esercizio. Dimostrare che, per n sufficientemente grande,

$$\log_2(n!) \geq \frac{1}{2} \cdot n \cdot \log_2(n)$$

Suggerimento:

$$\log(n!) = \sum_{i=1}^{n} \log(i) \geq \int_1^n \log(x)\ \mathrm{d}x$$



Ottimalità del merge sort

FUORI ESAME

Conclusione: a meno di una costante moltiplicativa, servono sempre almeno $n \cdot \log(n)$ test per ordinare un vettore. Quindi l'algoritmo merge sort è ottimo.

Esercizio

FUORI ESAME

Esercizio. (subdolo) Definite un algoritmo per ordinare un vettore di *booleani* in tempo $O(n)$. (Suggerimento: contate prima quanti `false` ci sono.)

Spiegate quindi come mai in questo caso si può fare di meglio di $O(n \cdot \log(n))$?