Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Linguaggi di programmazione

Riassunto dei principali argomenti

Autore:
Davide Bianchi

Anno Accademico 2017/2018

Indice

1 Introduzione

Un linguaggio di programmazione è composto da:

- *Sintassi:* insieme di regole di scrittura del linguaggio;
- *Semantica:* descrizione del comportamento del programma a tempo di esecuzione;
- *Pragmatica:* descrizione delle caratteristiche del linguaggio, delle sue funzionalità ecc.

Gli stili per dare la semantica di un linguaggio sono 3:

- *Operazionale*: la semantica è data con sistemi di transizione, fornendo i passi della computazione passo passo;
- *Denotazionale*: il significato di un programma è dato dalla struttura di un insieme;
- *Assiomatica*: il significato è dato attraverso regole assiomatiche o qualche tipo di logica.

2 Esempio di linguaggio basilare

La semantica operazionale di un linguaggio è data attraverso un sistema di regole di inferenza, date come segue:

$$(Assioma)\ \frac{-}{(Conclusione)} \qquad (Regola)\ \frac{(Hyp_1)\ (Hyp_2)\ ...\ (Hyp_n)}{(Conclusione)}$$

Introduciamo la sintassi di un linguaggio basato solo su espressioni aritmetiche:

$$E := \ n \mid E \mid E + E \mid E * E \ ...$$

La valutazione di programmi generati con questa sintassi può procedere in due modi:

- *Small step:* la semantica fornisce il sistema per procedere nell'esecuzione, passo dopo passo;
- *Big step:* la semantica va subito al risultato finale.

2.1 Semantica big-step

Forniamo la semantica big-step per il linguaggio dato sopra:

$$\text{B-Num}\frac{-}{n \Downarrow n} \qquad \text{B-Add}\frac{E_1 \Downarrow n_1\ E_2 \Downarrow n_2}{E_1 + E_2 \Downarrow n_3}\ n_3 = add(n_1, n_2)$$

2.1.1 Esempio

$$\text{B-Add}\frac{\text{B-Num}\ \dfrac{-}{3 \Downarrow 3} \qquad \text{B-Add}\ \dfrac{\text{B-Num}\ \dfrac{-}{2 \Downarrow 2} \qquad \text{B-Num}\ \dfrac{-}{1 \Downarrow 1}}{2 + 1 \Downarrow 3}}{3 + (2 + 1) \Downarrow 6}$$

2.2 Semantica small-step

Indichiamo con $E_1 \rightarrowtriangle E_2$ lo svolgimento di un solo passo di semantica.

$$\text{S-Left}\frac{E_1 \rightarrowtriangle E_1'}{E_1 + E_2 \rightarrowtriangle E_1' + E_2}$$

$$\text{S-N.Right}\frac{E_2 \rightarrowtriangle E_2'}{n_1 + E_2 \rightarrowtriangle n_1 + E_2'}$$

$$\text{S-Add}\frac{-}{n_1 + n_2 \rightarrowtriangle n_3} n_3 = add(n_1, n_2)$$

Con queste regole l'ordine di valutazione degli statement è fisso, procede sempre da sinistra verso destra. Diamo un'alternativa:

$$\text{S-Left}\frac{E_1 \rightarrowtriangle_{ch} E_2}{E_1 + E_2 \rightarrowtriangle_{ch} E_1' + E_2}$$

$$\text{S-Right}\frac{E_2 \rightarrowtriangle_{ch} E_2'}{E_1 + E_2 \rightarrowtriangle_{ch} E_1 + E_2'}$$

$$\text{S-Add}\frac{-}{n_1 + n_2 \rightarrowtriangle_{ch} n_3} n_3 = add(n_1, n_2)$$

In questo caso l'ordine di valutazione è arbitrario. La notazione utilizzata in generale è la seguente:

- la relazione $\rightarrowtriangle^k$, con $k \in \mathbb{N}$, indica una sequenza di n passi applicando la semantica small-step;
- la relazione $\rightarrowtriangle^*$, indica una sequenza di derivazione lunga un certo numero di passi. Questa relazione è riflessiva ed è la chiusura transitiva di $\rightarrowtriangle$.

3 Linguaggio imperativo

Definiamo la sintassi di un semplice linguaggio imperativo:

$$\begin{aligned} b &:= true \mid false \\ n &:= \{... -1, 0, 1, 2, ...\} \\ l &:= \{l_0, l_1, ...\} \\ op &:= + \mid \geq \\ e &:= n \mid b \mid e\ op\ e \mid \text{if } e \text{ then } e \text{ else } e \mid l := e \mid !l \mid skip \mid e; e \mid \text{while } e \text{ do } e \end{aligned}$$

Nota: lo statement $!l$ indica l'intero memorizzato al momento alla locazione l. Inoltre il linguaggio non è tipato, quindi sono ammesse le sintassi come $2 \geq true$.

3.1 Memoria

La memoria è necessaria per poter valutare gli statement di lettura da una locazione. In particolare definiamo

$$\begin{aligned} dom(f) &= \{a \in A \mid \exists b \in B\ s.t.\ f(a) = b\} \\ ran(f) &= \{b \in B \mid \exists a \in A\ s.t.\ f(a) = b\} \end{aligned}$$

Lo store del linguaggio imperativo in questione è un insieme di funzioni parziali che vanno dalle locazioni di memoria nei numeri interi:

$$s : \mathbb{L} \to \mathbb{Z}$$

L'aggiornamento della memoria funziona come segue:

$$s[l \to n](l') = \begin{cases} n & \text{if } l = l' \\ s(l') & \text{altrimenti} \end{cases}$$

3.2 Sistemi di transizione

Le semantiche operazionali sono date attraverso sistemi di transizione, ovvero strutture composte da:

- un insieme $Config$ di configurazioni;
- una relazione binaria $\twoheadrightarrow \subseteq Config \times Config$;

Per indicare un generale passo di semantica si usa la notazione

$$\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle$$

che rappresenta una trasformazione di un programma e con una memoria s in un programma e' con memoria associata s'. I singoli passi di computazione sono singole applicazioni di regole della semantica.

3.3 Semantica small-step su un linguaggio imperativo

$$\text{(op+)}\frac{-}{\langle n_1 + n_2, s \rangle \twoheadrightarrow \langle n, s \rangle}\ n = add(n_1, n_2) \qquad \text{(op-geq}^1\text{)}\frac{-}{\langle n_1 \geq n_2, s \rangle \twoheadrightarrow \langle b, s \rangle}\ b = geq(n_1, n_2)$$

$$\text{(op1)}\frac{\langle e_1, s \rangle \twoheadrightarrow \langle e_1', s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \twoheadrightarrow \langle e_1' \text{ op } e_2, s' \rangle} \qquad \text{(op2)}\frac{\langle e_2, s \rangle \twoheadrightarrow \langle e_2', s' \rangle}{\langle v \text{ op } e_2, s \rangle \twoheadrightarrow \langle v \text{ op } e_2', s' \rangle}$$

$$\text{(deref)}\frac{-}{\langle !l, s \rangle \twoheadrightarrow \langle n, s \rangle}\ \text{if } l \in dom(s) \text{ and } s(l) = n \qquad \text{(assign1)}\frac{-}{\langle l := n, s \rangle \twoheadrightarrow \langle skip, s[l \to n] \rangle}\ \text{if } l \in dom(s)$$

$$\text{(assign2)}\frac{\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle}{\langle l := e, s \rangle \twoheadrightarrow \langle l := e', s' \rangle} \qquad \text{(if-tt)}\frac{-}{\langle \text{if true then } e_1 \text{ else } e_2, s \rangle \twoheadrightarrow \langle e_1, s \rangle}$$

$$\text{(if-ff)}\frac{-}{\langle \text{if false then } e_1 \text{ else } e_2, s \rangle \twoheadrightarrow \langle e_2, s \rangle} \qquad \text{(if)}\frac{\langle e, s \rangle \twoheadrightarrow \langle e', s' \rangle}{\langle \text{if } e \text{ then } e_1 \text{ else } e_2, s \rangle \twoheadrightarrow \langle \text{if } e' \text{ then } e_1 \text{ else } e_2, s' \rangle}$$

$$\text{(seq)}\frac{\langle e_1, s \rangle \twoheadrightarrow \langle e_1', s' \rangle}{\langle e_1; e_2, s \rangle \twoheadrightarrow \langle e_1'; e_2, s' \rangle} \qquad \text{(seq.skip)}\frac{-}{\langle skip; e_2, s \rangle \twoheadrightarrow \langle e_2, s \rangle}$$

$$\text{(while)}\frac{-}{\langle \text{while } e \text{ do } e_1, s \rangle \twoheadrightarrow \langle \text{if } e \text{ then } (e_1; \text{while } e \text{ do } e_1) \text{ else } skip, s \rangle}$$

3.4 Esecuzione di programmi e proprietà

L'esecuzione di programmi con questa semantica consiste nel trovare una memoria s' tale per cui valga che

$$\langle P, s\rangle \rightarrowtriangle^* \langle v, s'\rangle$$

ovvero che si raggiunga una configurazione terminale in un certo numero di passi.

Illustriamo inoltre due importanti proprietà:

Teorema 3.1 (Strong normalization)**.** *Per ogni memoria s e ogni programma P esiste una qualche memoria s' tale che*

$$\langle P, s\rangle \rightarrowtriangle^* \langle v, s'\rangle$$

Teorema 3.2 (Determinatezza)**.** *Se $\langle e, s\rangle \rightarrowtriangle \langle e_1, s_1\rangle$ e $\langle e, s\rangle \rightarrowtriangle \langle e_2, s_2\rangle$ allora $\langle e_1, s_1\rangle = \langle e_2, s_2\rangle$.*

3.5 Funzione di valutazione della semantica

Date le regole nella sezione 3.3, possiamo dire che in generale, per valutare una porzione di programma, viene applicata la regola

$$[\![-]\!] : Exp \to (Store \rightharpoonup Store)$$

dove, data una generica espressione e, la funzione $[\![\cdot]\!]$ prende una memoria e ne ritorna una aggiornata dopo la valutazione di e.

$$[\![e]\!](s) = \begin{cases} s' & \text{se } \langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle \\ undefined & \text{altrimenti} \end{cases}$$

3.6 Possibili varianti del linguaggio

Nel linguaggio illustrato possono essere introdotte anche diverse varianti.

3.6.1 Inversione dell'ordine di valutazione

È possibile ad esempio introdurre un ordine di valutazione *right-to-left*, ossia:

$$\text{(op1b)}\frac{\langle e_2, s\rangle \to \langle e_2', s'\rangle}{\langle e_1 + e_2, s\rangle \rightarrowtriangle \langle e_1 + e_2', s'\rangle} \qquad \text{(op2b)}\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1 + v, s\rangle \rightarrowtriangle \langle e_1' + v, s'\rangle}$$

Aggiungendo queste due regole alla semantica ovviamente salta la regola della determinatezza.

3.6.2 Regole di assegnamento

Una piccola variante alla regola dell'assegnamento:

$$\text{(assign1b)}\frac{-}{\langle l := n, s\rangle \rightarrowtriangle \langle n, s[l \to n]\rangle} \text{ if } l \in dom(s) \qquad \text{(seq.skip.b)}\frac{-}{\langle v; e_2, s\rangle \rightarrowtriangle \langle e_2, s\rangle}$$

3.6.3 Inizializzazione della memoria

Possibili varianti a livello di inizializzazione della memoria potrebbero essere:

- inizializzare implicitamente tutte le locazioni a 0;
- permettere assegnamenti ad una locazione l tale che $l \notin dom(s)$ per inizializzare quella locazione.

3.6.4 Valori memorizzabili

Altre estensioni relative alla memoria (qui definita staticamente, ovvero l'insieme delle locazioni possibili è fisso) possono includere:

- la possibilità di memorizzare anche altri tipi di dato (non solo interi come in questo caso);
- la possibilità di avere una memoria definita dinamicamente, quindi dare la possiblità di avere sempre nuove locazioni disponibili oltre a quelle già in uso.

3.7 Type systems

Un type system è una struttura i cui usi principali sono:

- descrivere quando i programmi sono sensati;
- prevenire certi tipi di errore;
- strutturare i programmi;
- dare delle linee guida per la progettazione del linguaggio;
- dare informazioni utili per la fase di ottimizzazione da parte del compilatore;
- rinforzare alcune proprietà di *sicurezza* del programma.

Definiamo la funzione

$$\Gamma \vdash e : T$$

che sostanzialmente assegna il tipo T all'espressione e, per qualche tipo T del linguaggio.

Aggiungiamo al linguaggio i tipi delle espressioni T e i tipi delle locazioni T_{loc}:

$$T ::= int \mid bool \mid unit$$
$$T_{loc} ::= intref$$

3.7.1 Regole di tipaggio

$$\text{(int)}\frac{-}{\Gamma \vdash n : int} \text{ per } n \in \mathbb{Z} \qquad \text{(bool)}\frac{-}{\Gamma \vdash b : bool} \text{ per } b \in \{true, false\}$$

$$\text{(op+)}\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int} \qquad \text{(op-geq)}\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \geq e_2 : bool}$$

$$\text{(if)}\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \qquad \text{(assign)}\frac{\Gamma \vdash e : int}{\Gamma \vdash l := e : unit} \text{ se } \Gamma(l) = intref$$

$$\text{(deref)}\frac{-}{\Gamma \vdash !l : int} \text{ se } \Gamma(l) = intref \qquad \text{(skip)}\frac{-}{\Gamma \vdash skip : unit}$$

$$\text{(seq)}\frac{\Gamma \vdash e_1 : unit \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} \qquad \text{(while)}\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : unit}$$

Nota: le regole di tipaggio sono *syntax-directed*, ovvero per ogni regola della sintassi astratta si ha una regola di tipaggio.