Fondamenti di Linguaggi di Programmazione e Specifica

Alessio Gjergji

Indice

Capitolo 1

Introduzione

1.1 Linguaggi di programmazione

Un linguaggio di programmazione è un linguaggio formale che specifica un insieme di istruzioni che possono essere usate per produrre un insieme di output. Esso è definito da:

- **Sintassi**: specifica la forma delle istruzioni. Ci permette di capire quali stringhe sono ammissibili e quali no mediante diversi strumenti come grammatiche, analizzatori lessicali e sintattici, teoria degli automi.
- **Pragmatica**: specifica l'effetto delle istruzioni. Ci permette di capire le ragioni per introdurre un nuovo linguaggio e di programmazione invece di utilizzarne uno già esistente.
- **Semantica**: specifica il significato dei programmi scritti nel linguaggio, ovvero il loro comportamento a tempo di esecuzione. Ci permette di capire se due programmi apparentemente diversi sono equivalenti.

1.1.1 Benefici di una semantica formale

I benefici dei linguaggi di programmazione diversi, tra cui:

- **Implementazione**: Consente di fornire la specifica (*del comportamento*) dei programmi indipendentemente dalla macchina o dal compilatore utilizzato.
- **Verifica**: una semantica formale consente di ragionare sui programmi e sulle loro proprietà di correttezza.
- **Progettazione di Linguaggio**: spesso una semantica formale consente di scoprire ambiguità all'interno di linguaggi già esistenti. Questo aiuta a progettare nuovi linguaggi in maniera più accurata.

1.2 Un linguaggio per le espressioni aritmetiche: sintassi

Definiamo il seguente linguaggio:

$$\mathcal{E} \quad ::= \quad n \quad | \quad \mathcal{E} + \mathcal{E} \quad | \quad \mathcal{E} * \mathcal{E} \quad | \quad \ldots$$

dove:

- n è lo spazio del dominio dei numerali.
- $\mathcal{E}$ è il range del dominio delle espressioni aritmetiche.
- $+, x, \ldots$ sono simboli del linguaggio.

I numerali sono parte della sintassi del nostro linguaggio e non vanno confusi con i numeri che sono oggetti matematici. Ciò potrebbe significare che nel nostro linguaggio al posto di $0, 1, \ldots$ avremmo potuto usare $zero, uno, \ldots$ e sarebbero potuti essere uguali.

Nel nostro caso assumiamo che esista una corrispondenza ovvia tra il simbolo "numerale" (n) e il numero naturale n. Questo è fatto solo per semplificare la spiegazione. In un altro contesto, il simbolo "numeral" 3 potrebbe essere associato al numero 42!

1.3 Semantica Operazionale

La semantica operazionale ha l'obiettivo di valutare un'espressione aritmetica del linguaggio per ottenere il suo valore numerico associato. Questo può essere fatto in due modi differenti:

- **Semantica Small-Step (*o strutturale*)**: Fornisce un metodo per valutare un'espressione passo dopo passo, considerando le azioni intermedie. Questo approccio fornisce una valutazione dettagliata dell'espressione.
- **Semantica Big-Step (*o naturale*)**: Ignora i passaggi intermedi e fornisce direttamente il risultato finale della valutazione dell'espressione. Questo approccio semplifica la valutazione, concentrando l'attenzione sul risultato finale.

1.3.1 Big-Step Semantics

Valutazione

$E \Downarrow n$

Significato: La valutazione dell'espressione $\mathcal{E}$ produce il numerale n.

Assiomi e regole di inferenza

$$\text{(B-Num)}\ \frac{-}{n \Downarrow n} \qquad\qquad \text{(B-Add)}\ \frac{\mathcal{E}_1 \Downarrow n_1 \qquad \mathcal{E}_2 \Downarrow n_2}{\mathcal{E}_1 + \mathcal{E}_2 \Downarrow n_3}\ n_3 = add(n_1, n_2)$$

Significato:

- (B-Num): Questo è un assioma che afferma che quando valutiamo un singolo numero n, otteniamo lo stesso numero n come risultato. Questo è il caso base della valutazione.
- (B-Add): Questa regola di inferenza afferma che date due espressioni $\mathcal{E}_1$ e $\mathcal{E}_2$:
  - Se è il caso che $\mathcal{E}_1 \Downarrow n_1$ (cioè $\mathcal{E}_1$ si valuta a n_1) e
  - È anche il caso che $\mathcal{E}_2 \Downarrow n_2$ (cioè $\mathcal{E}_2$ si valuta a n_2), allora segue che $\mathcal{E}_1 + \mathcal{E}_2 \Downarrow n_3$, dove n_3 è il numerale associato al numero n_3 tale che $n_3 = add(n_1, n_2)$. Si noti che in questa regola, $E1$, $E2$, $n1$, $n2$, $n3$ sono meta-variabili.

Questa regola (B-Add) ci dice come valutare un'addizione tra due espressioni $\mathcal{E}_1$ e $\mathcal{E}_2$ nel contesto della semantica big-step. La regola stabilisce che se possiamo valutare entrambe le espressioni operandi ($\mathcal{E}_1$ e $\mathcal{E}_2$) e otteniamo i numeri n_1 e n_2 rispettivamente, allora possiamo calcolare la somma di $\mathcal{E}_1$ e $\mathcal{E}_2$ come n_3, dove n_3 è il risultato della somma dei numeri n_1 e n_2. Si noti che la funzione di addizione *add* opera sui numeri, non sui numerali.

1.3.2 Small-Step Semantics

Valutazione

$\mathcal{E}_1 \to \mathcal{E}_2$

Significato: Dopo aver eseguito un passo di valutazione su $\mathcal{E}_1$, l'espressione $\mathcal{E}_2$ rimane da valutare.

Assiomi e regole di inferenza

$$\text{(S-Left)}\ \frac{\mathcal{E}_1 \to \mathcal{E}_1'}{\mathcal{E}_1 + \mathcal{E}_2 \to \mathcal{E}_1' + \mathcal{E}_2}$$

$$\text{(S-N.Right)}\ \frac{\mathcal{E}_2 \to \mathcal{E}_2'}{n_1 + \mathcal{E}_2 \to n_1 + \mathcal{E}_2'}$$

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

Fissiamo l'ordine di valutazione da sinistra a destra. Qualcosa di simile non è possibile nella big-step semantics, dove le espressioni sono valutate in un solo passo.

La scelta dell'ordine di valutazione

Assiomi e regole di inferenza

$$\text{(S-Left)}\ \frac{\mathcal{E}_1 \rightarrow_{ch} \mathcal{E}_1'}{\mathcal{E}_1 + \mathcal{E}_2 \rightarrow_{ch} \mathcal{E}_1' + \mathcal{E}_2}$$

$$\text{(S-Right)}\ \frac{\mathcal{E}_2 \rightarrow_{ch} \mathcal{E}_2'}{\mathcal{E}_1 + \mathcal{E}_2 \rightarrow_{ch} \mathcal{E}_1 + \mathcal{E}_2'}$$

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

In questo caso non abbiamo precedenza stabilita per la valutazione delle espressioni. Regole simili possono essere applicate anche con gli altri operatori.

Esecuzione della small-step semantics

La relazione $\rightarrow^k$, per $k \in \mathbb{N}$ è definita per un numero di passi di valutazione definito da k. Mentre la relazione $\rightarrow^*$ è definita per un numero non definito di passi di valutazione.

Capitolo 2

Un semplice linguaggio imperativo

La sintassi del nostro semplice linguaggio imperativo è definita utilizzando la notazione `BNF` come segue:

- `true` e `false` sono booleani.
- I numeri interi n appartengono a $\mathbb{N}$.
- Le locazioni l sono identificatori di variabili.

La sintassi del linguaggio può essere definita dalle seguenti produzioni grammaticali:

$$\begin{array}{rcl} \textit{Operations} & ::= & + \mid \geq \\ \\ \textit{Expressions} & ::= & n \mid b \mid e \text{ op } e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\ & \mid & l := e \mid\ !l \mid \texttt{skip} \mid e\ ;\ e \\ & \mid & \texttt{while } e \texttt{ do } e \end{array}$$

2.1 Valutazione delle espressioni

I valori delle espressioni dipendono dai valori correnti all'interno delle locazioni.

$$!l_1 + !l_2 - 1$$

In questo caso, il valore dell'espressione dipende dai valori correnti nelle locazioni l_1 e l_2.

Quindi, per valutare un'espressione, dobbiamo considerare questi cambiamenti:

- Come valutiamo un'espressione e, in questo caso $!l_1$?
- Come valutiamo un'assegnamento $l := e$?

Abbiamo bisogno di più informazioni relative allo stato della memoria.

2.1.1 Funzioni parziali

Una funzione parziale f è una funzione che può non essere definita per tutti gli input. In questo caso, scriveremo $f(x) \downarrow$ se f è definita per x e $f(x) \uparrow$ se f non è definita per x.

In generale una funzione parziale può essere definita come segue:

$$f : A \rightharpoonup B$$

dove A è il dominio di f e B è il codominio di f.

Convenzioni

- $dom(f)$ è l'insieme degli elementi nel dominio di f, formalmente:

$$dom(f) = \{x \in A : \exists b \in B \, s.t. \, f(a) = b\}$$

- $ran(f)$ è l'insieme degli elementi nel codominio di f, formalmente:

$$ran(f) = \{b \in B : \exists a \in A \, s.t. \, f(a) = b\}$$

Quindi f è una funzione totale se $dom(f) = A$ e f è una funzione parziale se $dom(f) \subset A$.

2.1.2 Memoria

Nel nostro linguaggio, la memoria è una funzione parziale che mappa locazioni in interi.

$$s : \mathbb{L} \rightharpoonup \mathbb{N}$$

Per esempio: $\{l_1 \mapsto 3, l_3 \mapsto 6, l_3 \mapsto 7\}$.

Aggiornamento della memoria L'aggiornamento della memoria è una funzione che prende in input una memoria s, una locazione l e un valore n e restituisce una nuova memoria s'.

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

Il comportamento dei programmi dipende dallo stato della memoria.

2.2 Sistema di transizione

Un sistema di transizione è composto da un insieme di configurazioni (*Config*) e una relazione binaria ($\subseteq$) su coppie di configurazioni. La relazione rappresenta come una configurazione può effettuare una transizione verso un'altra.

$$\textit{Relazione binaria} \rightarrow \subseteq \textit{Config} \times \textit{Config}$$

In particolare, gli elementi di *Config* sono spesso chiamati configurazioni o stati. La relazione è chiamata relazione di transizione o di riduzione. Adottiamo una notazione infix, quindi $c \rightarrow c'$ dovrebbe essere letto come "la configurazione c può fare una transizione alla configurazione c'".

L'esecuzione completa di un programma trasforma uno stato iniziale in uno stato terminale. Un sistema di transizione è simile a un automa a stati finiti non deterministico (NFA^{ε}) con un alfabeto vuoto, tranne che può avere un numero infinito di stati. Non specifichiamo uno stato di partenza o stati di accettazione.

2.3 Semantica operazionale nel nostro linguaggio imperativo

Le configurazioni sono coppie $\langle e, s\rangle$ di espressioni e e memorie s. Le relazioni di transizione sono definite come segue:

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

dove e' è l'espressione risultante dalla valutazione di e nello stato s e s' è lo stato risultante dalla valutazione di e nello stato s.

Le transizioni rappresentano singoli passi di calcolo. Ad esempio, avremo:

$$\begin{array}{ll}
\rightarrow & \langle l := 2+!l, \{l \mapsto 3\}\rangle \\
\rightarrow & \langle l := 2+3, \{l \mapsto 3\}\rangle \\
\rightarrow & \langle l := 5, \{l \mapsto 3\}\rangle \\
\rightarrow & \langle \text{skip}, \{l \mapsto 5\}\rangle \\
\not\rightarrow &
\end{array}$$

Dove $\langle e, s\rangle$ rappresenta una configurazione, e è un'espressione e s è uno stato. Le transizioni sono passi di calcolo singoli che portano da una configurazione all'altra. La notazione $\langle e, s\rangle$ è "bloccata" o in uno stato di "deadlock" se e non è un valore e $\langle e, s\rangle$ non ha una transizione seguente, ovvero $\langle e, s\rangle \not\rightarrow$.

Ad esempio, $3 + \text{false}$ è "bloccato" o in uno stato di "deadlock" perché $3 + \text{false}$ non è un valore e non può fare una transizione successiva.

2.3.1 Operazioni di base

Somma

$$(\text{op } +)\ \frac{-}{\langle n_1 + n_2, s\rangle \rightarrow \langle n_1 + n_2, s\rangle}$$

Disuguaglianza

$$(\text{op } \geq)\ \frac{-}{\langle n_1 \geq n_2, s\rangle \rightarrow \langle \text{b}, s\rangle}$$

Operazione 1

$$(\text{op } 1)\ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1 \,\texttt{op}\, e_2, s\rangle \rightarrow \langle e_1' \,\texttt{op}\, e_2, s'\rangle}$$

Operazione 2

$$(\text{op } 1)\ \frac{\langle e_2, s\rangle \rightarrow \langle e_2', s'\rangle}{\langle e_1 \,\texttt{op}\, e_2, s\rangle \rightarrow \langle e_1 \,\texttt{op}\, e_2', s'\rangle}$$

Le regole di transizione introducono i cambiamenti nella memoria.

Dereferenziazione

$$(\text{deref})\ \frac{-}{\langle !l, s\rangle \rightarrow \langle s(l), s\rangle} \qquad se\, l \in dom(s)\ e\, s(l) = n$$

Assegnamento

$$(\text{assign1})\ \frac{-}{\langle l := n, s\rangle \rightarrow \langle \texttt{skip}, s[l \mapsto n]\rangle} \quad se \quad l \in dom(s)$$

$$(\text{assign2})\ \frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle l := e, s\rangle \ \rightarrow \langle l := e', s'\rangle}$$

Condizionale

$$(\text{if\_tt})\ \frac{-}{\langle \texttt{if true then } e_1 \texttt{ else } e_2, s\rangle \rightarrow \langle e_1, s\rangle}$$

$$(\text{if\_ff})\ \frac{-}{\langle \texttt{if false then } e_1 \texttt{ else } e_2, s\rangle \rightarrow \langle e_2, s\rangle}$$

$$(\text{if})\ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, s\rangle \rightarrow \langle \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3, s'\rangle}$$

Sequenza

$$(\text{seq})\ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \rightarrow \langle e_1'; e_2, s'\rangle}$$

$$(\text{seq.Skip})\ \frac{-}{\langle \texttt{skip;}\ e_2, s\rangle \rightarrow \langle e_2, s\rangle}$$

While

$$(\text{while})\ \frac{-}{\langle \texttt{while } e_1 \texttt{ do } e_2, s\rangle \rightarrow \langle \texttt{if } e_1 \texttt{ then } e_2;\ \texttt{while } e_1 \texttt{ do } e_2 \texttt{ else skip}, s\rangle}$$

Questa è una regola di riscrittura chiamata anche *unwinding*, che consente di rivalutare la l'espressione e_1 ad ogni iterazione del ciclo.

2.4 Esecuzione di un programma

Per eseguire un programma P a partire da uno stato s, è possibile trovare uno stato s' tale che

$$\langle P, s \rangle \to_* \langle v, s' \rangle$$

per $v \in \mathbb{V} = \mathbb{B} \cup \mathbb{Z} \cup \{\texttt{skip}\}$.

Le configurazioni della forma $\langle v, s \rangle$ sono considerate terminali. Qui, $\to_*$ denota la chiusura riflessiva e transitiva della relazione di riduzione $\to$.

2.5 Proprietà del linguaggio

Teorema 2.5.1 **Normalizzazione forte**
Per ogni stato s e per ogni programma P, esistono degli stati s' tali che $\langle P, s \rangle \to_* \langle v, s' \rangle$, dove $\langle v, s \rangle$ è una configurazione terminale.

Teorema 2.5.2 **Determinismo**
Se $\langle e, s \rangle \to \langle e_1, s_1 \rangle$ e $\langle e, s \rangle \to \langle e_2, s_2 \rangle$, allora $\langle e_1, s_1 \rangle = \langle e_2, s_2 \rangle$.

2.5.1 Funzione di interpretazione semantica

Possiamo usare la semantica operazionale per fornire una semantica formale del seguente programma: Quindi:

Algorithm 1: Esempio

$l_1 \leftarrow 1;$
$l_2 \leftarrow 0;$
while $\neg(!l_1 = !l_2)$ **do**
  $l_2 := !l_2 + 1;$
  $l_3 := !l_3 + 1;$
$l_1 := !3;$

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

Dove forniamo una espressione arbitraria e, $[\![e]\!]$ è una funzione parziale che mappa uno stato s in un nuovo stato s'.

Definizione

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

Il nostro programma d'esempio possiamo descriverlo come segue:

$$[\![P]\!] = \begin{cases} s(l_1) - 1 & \text{se } l \in \{l_1, l_3\} \text{ e } s(l_1) > 0 \\ s(l_1) & \text{se } l = l_2 \text{ e } s(l_1) > 0 \\ s(l) & \text{se } l \notin \{l_1, l_2, l_3\} \text{ e } s(l_1) > 0 \end{cases}$$

2.6 Espressività del linguaggio

Un linguaggio si dice espressivo se è possibile esprimerci qualsiasi funzione calcolabile. Per esempio, il linguaggio imperativo è Turing completo, quindi esprime qualsiasi funzione calcolabile.

Il nostro linguaggio è però troppo espressivo perché è possibile esprimere funzioni di questa tipologia $3+true$, il modo per evitare questo problema è quello di introdurre il **type system**.