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:

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

dove:

- n è lo spazio del dominio dei numerali.
- $\mathcal{E}$ è il range del dominio delle espressioni aritmetiche.
- $+, x, \dots$ 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 $0, 1, \dots$ avremmo potuto usare $zero, uno, \dots$ 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

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

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:

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

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 $c \to c'$ 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:

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

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:

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

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 $\langle e, s\rangle \not\to$.

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

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

Disuguaglianza

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

Operazione 1

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

Operazione 2

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

Le regole di transizione introducono i cambiamenti nella memoria.

Dereferenziazione

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

Assegnamento

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

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

Condizionale

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

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

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

Sequenza

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

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

While

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

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

```
1 l1 ← 1;
2 l2 ← 0;
3 while ¬(!l1 =!l2) do
4 |   l2 :=!l2 + 1;
5 |   l3 :=!l3 + 1;
6 l1 :=!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**.

2.7 Type system

il *type system* è un componente fondamentale nei linguaggi di programmazione, e le sue regole formali sono essenziali per garantire la correttezza e la sicurezza dei programmi. Ecco perché è importante definire queste regole in modo formale e strutturato:

- Evitare errori di runtime: Il principale scopo di un "type system" è evitare errori di runtime. Ciò significa che il sistema è in grado di individuare potenziali errori nel tipo di dati o nell'uso di operatori prima che il programma venga eseguito. Ciò è particolarmente importante poiché gli errori di runtime possono comportare malfunzionamenti del programma, crash o risultati imprevisti.
- Soundness: Un "type system" è considerato "sound" quando garantisce che se un programma è ben tipato, allora non si verificheranno errori di tipo durante l'esecuzione. Questo è un aspetto fondamentale per garantire che i programmi siano corretti dal punto di vista del tipo.
- Incompletezza: Tuttavia, i "type system" sono spesso incompleti, il che significa che ci sono programmi che sono corretti dal punto di vista del tipo ma che vengono respinti dal sistema. Questo può accadere quando il sistema non è in grado di dedurre in modo completo il tipo dell'espressione o quando le regole del "type system" sono troppo conservative. L'incompletezza può portare alla rifiutazione di programmi validi, ma è un compromesso necessario per garantire la sicurezza del tipo.
- Proprietà di progress: L'obiettivo principale del "type system" è garantire che i programmi ben tipati siano in grado di fare progressi durante l'esecuzione, ovvero che non si blocchino o entrino in cicli infiniti. Questo aspetto è strettamente correlato all'obiettivo di non tipare programmi che vanno in regola, e rappresenta un'altra dimensione importante della correttezza dei programmi.

Definiremo la seguente espressione ternaria:

$$\Gamma \vdash e : T$$

L'espressione si legge come: *in un contesto* Γ*, l'espressione e ha tipo* T. Il contesto Γ è un insieme di assegnazioni di variabili a tipi, per esempio:

$$\begin{array}{lcll} \{\} & \vdash & \texttt{if true then 2 else 3 + 4} & : \texttt{int} \\ l_1 : \texttt{intref} & \vdash & \texttt{if } l_1 \geq 3 \texttt{ then } l_1 \texttt{ else } 3 & : \texttt{int} \\ \{\} & \nvdash & 3 + \texttt{false} & : \text{T per ogni T} \\ \{\} & \nvdash & \texttt{if true then 3 else false} & : \text{T per ogni T} \end{array}$$

Da notare che l'ultimo programma non è ben tipato, infatti in alcuni casi il type system dovrebbe assegnare un interno e in altri un booleano. Esso definisce un'approssimazione del comportamento del programma. Vogliamo generalmente che fossero **decidibili**, in modo da garantire che la compilazione sia affidabile.

2.7.1 Tipi per il linguaggio `while`

$$T \ ::= \ \texttt{int} \mid \texttt{bool} \mid \texttt{unit}$$

I tipi delle locazioni saranno:

$$T_{loc} \ ::= \ \texttt{intref}$$

Dove `intref` rappresenta un tipo utilizzato per riferimenti a valori interi nel programma.

L'ambiente dei tipi, indicato come Γ, è un insieme di funzioni parziali che associano le localizzazioni (L) ai tipi di localizzazione (T_{loc}). Per una rappresentazione più chiara, possiamo esprimere Γ nel seguente formato:

$$\Gamma = \{l_1 : \texttt{intref}, \ldots, l_k : \texttt{intref}\}$$

Questo ambiente dei tipi associa le localizzazioni $l_1, \ldots, l_k$ al tipo `intref`. In un contesto più avanzato, T_{loc} potrebbe contenere tipi più complessi, ma per ora, consideriamo solo il tipo `intref`.

2.7.2 Definizione delle valutazioni dei tipi

$$\text{(int)} \ \frac{-}{\Gamma \vdash n : \texttt{int}} \ \texttt{for } n \in \mathbb{Z}$$

$$\text{(bool)} \ \frac{-}{\Gamma \vdash b : \texttt{bool}} \ \texttt{for } b \in \{\texttt{true, false}\}$$

$$\text{(op +)} \ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

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

$$\text{(if)} \ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T}$$

Con le regole di tipaggio scartiamo quindi le espressioni che non hanno senso, a tempo di compilazione.

Nel processo di tipizzazione, utilizzo Γ per portare con me informazioni parziali sul programma. Questo è fondamentale per scoprire errori a tempo di compilazione.

Per ora, il tipo dell'assegnamento sarà semplicemente di tipo `unit`.

$$\text{(assign)}\ \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash l := e : \texttt{unit}}\ \Gamma(l) = \texttt{intref}$$

$$\text{(deref)}\ \frac{-}{\Gamma \vdash !l : \texttt{int}}\ \Gamma(l) = \texttt{intref}$$

Di seguito, riporto le condizioni e sopra le regole induttive, le quali dipendono dal tipaggio del sottoprogramma.

Si ricordi che il tipo delle locazioni è rappresentato da `intref`.

$$\text{(skip)}\ \frac{-}{\Gamma \vdash \texttt{skip} : \texttt{unit}}$$

$$\text{(seq)}\ \frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T}$$

Nel caso della composizione sequenziale, il tipo del secondo termine sarà uguale al tipo del primo, che in questo caso è `unit`.

Nel caso del ciclo "while":

$$\text{(while)}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{unit}}{\Gamma \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 : \texttt{unit}}$$

Considerando la regola di tipizzazione, il tipo del corpo del ciclo "while" sarà anch'esso di tipo `unit`.

2.8 Proprietà

2.8.1 Teorema della Progressione

Il Teorema della Progressione afferma che:

> Se $\Gamma \vdash e : T$ e $dom(\Gamma) \subseteq dom(s)$, allora e è un valore o esiste e', s' tali che $\langle e, s\rangle \to \langle e', s'\rangle$.

Quindi, durante l'esecuzione di un programma, siamo in grado di fare progressi. Se un'espressione ha un tipo valido e il contesto è adeguato, o l'espressione è già un valore (*cioè non può essere valutata ulteriormente*), oppure possiamo effettuare una transizione a uno stato successivo.

2.8.2 Teorema della Preservazione del Tipo

Il Teorema della Preservazione del Tipo stabilisce che: Se $\Gamma \vdash e : T$ e $dom(S) \subseteq dom(s)$, e se $\langle e, s\rangle \to \langle e', s'\rangle$, allora $\Gamma \vdash e' : T$ e $dom(\Gamma) \subseteq dom(s')$.

Questo teorema ci assicura che, durante l'esecuzione di un programma, se un'espressione ha un tipo valido e il contesto è adeguato, allora qualsiasi transizione non romperà la coerenza dei tipi. Il tipo delle espressioni sarà preservato durante l'esecuzione.

Mettendo insieme i due teoremi, otteniamo un nuovo teorema, il **Teorema della Safety**.

2.8.3 Teorema della Safety

Il Teorema della Safety ci dice che: Se $\Gamma \vdash e : T$ e $dom(\Gamma) \subseteq dom(s)$, e se $\langle e, s\rangle \to^* \langle v', s'\rangle$, allora e' sarà un valore o esisterà e'', s'' tali che $\langle e', s'\rangle \to \langle e'', s''\rangle$.

In parole povere, se prendiamo una configurazione iniziale e la facciamo avanzare attraverso un numero qualsiasi di passi, avremo due possibilità: o raggiungeremo una configurazione finale, oppure saremo in grado di effettuare un ulteriore passo.

Quando diciamo che $dom(\Gamma) \subseteq dom(s)$, intendiamo che il dominio dello store (*lo spazio in cui sono memorizzati i valori delle variabili*) deve essere contenuto almeno nel dominio del contesto (*l'insieme delle variabili dichiarate nel programma*).

Ecco un esempio che illustra questi concetti:

```
1 while true do
2 |_ skip
```

In base alle regole di tipizzazione, questo programma avrà il tipo `unit`, ma non impedisce al programma di essere in uno stato di blocco infinito, in cui continua a eseguire l'istruzione "skip" all'infinito.

2.8.4 Type Checking, Typeability e Type Inference

Problema del Controllo di Tipo

Dato un type system, un tipo ambiente Γ, un'espressione e, e un tipo T, stabilire se $\Gamma \vdash e : T$ è derivabile?

Problema di Type Inference

Dato un type system, un tipo ambiente Γ, e un'espressione e, trovare un tipo T tale che $\Gamma \vdash e : T$ è derivabile, oppure mostrare che non esiste.

Il secondo problema è più difficile del primo, in quanto devo trovare un tipo T che soddisfi la proprietà, mentre nel primo caso devo solo stabilire se esiste un tipo T che soddisfi la proprietà.

2.8.5 Type Checking

> Dato Γ, e e T, possiamo trovare un T tale che $\Gamma \vdash e : T$ o mostrare che non esiste.

Il Teorema del Type Checking afferma che, dati un ambiente di tipi Γ, un'espressione e e un tipo T, siamo in grado di determinare se $\Gamma \vdash e : T$ è derivabile. In altre parole, possiamo verificare se un'espressione e è ben tipata rispetto a un ambiente di tipi Γ, e se è così, possiamo anche calcolare il tipo T che la verifica.

2.8.6 Preservazione del Tipo

> Dato Γ, e e T, si può decidere se $\Gamma \vdash e : T$.

Il Teorema della Preservazione del Tipo afferma che, dati un ambiente di tipi Γ, un'espressione e e un tipo T, possiamo determinare se $\Gamma \vdash e : T$ è derivabile. In altre parole, questo teorema ci assicura che possiamo verificare se il tipo di un'espressione e rimane coerente durante l'esecuzione del programma.

2.8.7 Unicità del Tipo

> Se $\Gamma \vdash e : T$ e $\Gamma \vdash e : T'$ allora $T = T'$.

Il Teorema dell'Unicità del Tipo afferma che se un'espressione e ha due tipi T e T' rispetto a un ambiente di tipi Γ, allora T e T' devono essere identici. In altre parole, non può esistere una situazione in cui un'espressione abbia più di un tipo ben formato.

Capitolo 3

Induzione

3.1 Induzione come principio di dimostrazione

L'induzione è un principio di dimostrazione che consente di dimostrare proprietà su insiemi infiniti di elementi. Questo principio si basa su passi finiti di calcolo, che su basano su insiemi che hanno una struttura ben definita. Questa struttura ci aiuta a ridurre il problema complesso in una serie di passaggi più semplici.

Esistono diversi tipi di induzione tra cui:

- Induzione matematica: Utilizzata per dimostrare affermazioni sui numeri naturali. Si basa sulla dimostrazione di una proprietà per un valore iniziale e sulla dimostrazione che, se la proprietà è vera per un certo numero, lo è anche per il numero successivo.
- Induzione strutturale: Utilizzata per dimostrare affermazioni su strutture ricorsive come alberi. La dimostrazione inizia dimostrando la proprietà per il caso base (*ad esempio, un albero vuoto*) e successivamente dimostra che se la proprietà è vera per le componenti strutturali, lo è anche per la struttura complessiva.
- Induzione delle regole: Spesso utilizzata per dimostrare proprietà delle regole in un sistema formale. La dimostrazione inizia dimostrando la proprietà per ciascuna regola e successivamente dimostra che la proprietà si conserva quando si applicano le regole in sequenza.

3.1.1 I numeri naturali $\mathbb{N}$

I numeri naturali sono costruiti seguendo due regole fondamentali:

- **Regola di base**: Il numero 0 appartiene all'insieme dei numeri naturali, indicato come $0 \in \mathbb{N}$.
- **Passo induttivo**: Se un numero k è un numero naturale, allora il successore di k, ovvero $k + 1$, è anch'esso un numero naturale.

Esempio

Per definire una funzione $f : \mathbb{N} \to X$, è necessario seguire due passi:

- **Regola di base**: Si descrive il risultato di f per il valore iniziale, ovvero 0.
- **Passo induttivo**: Si assume che f sia definita per un valore k, e si descrive il risultato di f per $k + 1$ in termini di $f(k)$. Questo approccio di definizione ricorsiva è spesso utilizzato nella programmazione, in particolare nell'ambito del "pattern matching".

Esempio

Nel contesto della semantica "small step" nel caso deterministico, possiamo definire una funzione `red` come segue:

$$\texttt{red} : \texttt{Exp} \times \mathbb{N} \to \text{Exp}$$

- **Regola di base**: $\texttt{red}(E, 0) = E$ per ogni espressione E.
- **Passo induttivo**: $\texttt{red}(E, k + 1) = E''$ se esiste un'espressione E' tale che $\text{red}(E, k) = E'$ e $E' \to E''$. Questa definizione permette di rappresentare l'espressione ottenuta riducendo E per k passi.

La dimostrazione per induzione è un metodo formale per dimostrare affermazioni matematiche o logicamente corrette su insiemi infiniti, come i numeri naturali. L'obiettivo è dimostrare una proprietà P per un numero arbitrario k e dimostrare che la proprietà è vera anche per il successivo $k + 1$:

$$\forall k \in \mathbb{N}. P(k) \Rightarrow P(k + 1)$$

Nel processo di dimostrazione per induzione, è essenziale dimostrare che la proprietà è vera per il caso base (spesso $k = 0$) e che il passaggio all'elemento successivo è valido. Questo garantisce che la proprietà sia valida per tutti i numeri naturali.

3.2 Induzione matematica

Il metodo più semplice di induzione è l'induzione matematica, che è un tipo di induzione basato sui numeri naturali. Il principio può essere descritto come segue. Dato un'affermazione $P(-)$ sui numeri naturali, vogliamo dimostrare che $P(n)$ è vera per ogni numero naturale n:

- **Caso base**: dimostrare che $P(0)$ è vera (*utilizzando alcuni fatti matematici noti*).
- **Caso induttivo**: assumere l'ipotesi induttiva, cioè che $P(k)$ è vera. A partire dall'ipotesi induttiva, dimostrare che segue $P(k + 1)$ (*utilizzando alcuni fatti matematici noti*).

Se (a) e (b) vengono stabiliti, allora $P(n)$ è vera per ogni numero naturale n.

L'induzione matematica è un principio valido perché ogni numero naturale può essere "costruito" a partire da 0 come punto di partenza e usando l'operazione di aggiunta di uno per creare nuovi numeri.

3.3 Induzione strutturale

L'induzione strutturale è un principio di dimostrazione che consente di dimostrare proprietà su elementi di un insieme costruito induttivamente. Questo tipo di induzione è particolarmente utile quando si tratta di oggetti con una struttura ricorsiva.

Un concetto importante nell'ambito dell'induzione strutturale è l'isomorfismo. Due oggetti si dicono isomorfi se esiste una funzione biunivoca tra di essi, cioè una funzione che stabilisce una corrispondenza uno a uno tra gli elementi degli oggetti.

Un esempio comune di passaggio dall'induzione matematica a quella strutturale coinvolge la seguente grammatica:

$$N \ ::= \ \texttt{zero} \mid \texttt{SUCC}(\text{N})$$

Nell'induzione strutturale, possiamo dimostrare proprietà sugli elementi di questa grammatica in questo modo:

- Caso base: Dimostrare che la proprietà è vera per `zero`.
- Passo induttivo: Assumere che la proprietà sia vera per un generico elemento `N` e dimostrare che è vera anche per `SUCC`(N).

Questo principio ci consente di affrontare dimostrazioni relative a strutture ricorsive in modo sistematico e rigoroso.

Somma (*sum*)

Il principio di definizione delle funzioni per induzione può essere applicato a questa rappresentazione dei numeri naturali allo stesso modo di prima. Vediamo un esempi:

- Regola di base: $sum(zero) = zero$
- Regola induttiva: $sum(succ(N)) = succ^{(n+1)}(sum(N))$, dove $N = succ(...succ(zero))$ per un certo numero naturale n.

Ciò significa che il caso base è definito per *zero*, e nel caso induttivo, applichiamo la funzione *sum* in modo ricorsivo a $succ(N)$ aggiungendo *succ* ripetutamente $n + 1$ volte.

3.3.1 Induzione strutturale per i numeri naturali

Il principio di induzione afferma che per dimostrare $P(N)$ per tutti i numeri N, è sufficiente fare due cose:

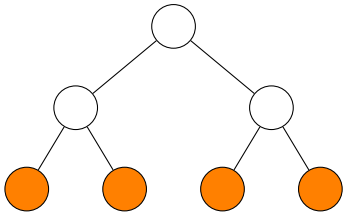
- **Caso base**: Dimostrare che $P(zero)$ è vero.

- **Caso induttivo**: Dimostrare che $P(succ(K))$ segue dall'assunzione che $P(K)$ sia vero per un certo numero K.

È importante notare che nel tentativo di dimostrare $P(succ(K))$, l'ipotesi induttiva ci dice che possiamo assumere che la proprietà sia valida per la **sottostruttura** di $succ(K)$, ovvero possiamo supporre che $P(K)$ sia vero.

Questa prospettiva strutturale, e la forma associata di induzione, chiamata induzione strutturale, è ampiamente applicabile.

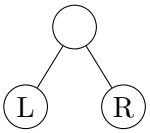
3.3.2 Albero binario attraverso l'induzione strutturale



Definizione induttiva

- **Caso base**: `leaf` è un albero binario.

- **Passo induttivo**: se L e R sono alberi binari allora `node`(L, R) è un albero binario.



3.3.3 Regole di costruzione

$$T \in \textit{bTree} ::= \texttt{leaf} \mid \texttt{node}(\mathrm{T}, \mathrm{T})$$

- **Caso base**: `leaf` è un albero binario.
- **Passo induttivo**: se T_1 e T_2 sono alberi binari, allora `node`(T_1, T_2) è un albero binario.

Definizione induttiva

$$f : \texttt{bTree} \to X$$

Per definire una funzione f che mappa alberi binari a elementi di un insieme X, possiamo seguire il principio della definizione induttiva:

- **Regola di base:** Descriviamo il risultato dell'applicazione di f a una foglia terminale, ad esempio $f(\texttt{leaf})$.
- **Regola induttiva:** Supponiamo che $f(T1)$ e $f(T2)$ siano già definiti. Ora, descriviamo il risultato dell'applicazione di f all'albero binario $\texttt{node}(T1, T2)$.

Con queste regole, possiamo definire la funzione f per ogni possibile albero binario, indipendentemente dalla sua complessità. Ogni passo nella definizione si basa sui passi precedenti, garantendo che la funzione sia ben definita per tutti gli alberi binari.

Funzione che conta il numero di foglie

$$\texttt{leaves}(\texttt{leaf}) = 1$$

$$\texttt{leaves}(\texttt{tree}(T_1, T_2)) = \texttt{leaves}(T_1) + \texttt{leaves}(T_2)$$

Funzione che conta il numero di rami

$$\texttt{branches}(\texttt{leaf}) = 0$$

$$\texttt{branches}(\texttt{tree}(T_1, T_2)) = \texttt{branches}(T_1) + \texttt{branches}(T_2) + 1$$