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

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

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

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

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

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

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 $c \to c'$ 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

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

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

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

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, $\rightarrow_*$ denota la chiusura riflessiva e transitiva della relazione di riduzione $\rightarrow$.

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 $\langle P, s\rangle \rightarrow_* \langle v, s'\rangle$, dove $\langle v, s\rangle$ è una configurazione terminale.

Teorema 2.5.2 Determinismo
Se $\langle e, s\rangle \rightarrow \langle e_1, s_1\rangle$ e $\langle e, s\rangle \rightarrow \langle e_2, s_2\rangle$, 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**
    $l_2 := !l_2 + 1;$
    $l_3 := !l_3 + 1;$
$l_1 := !3;$

$$[\![-]\!] : \mathit{Exp} \rightarrow (\mathit{Store} \rightharpoonup \mathit{Store})$$

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

Definizione

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

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:

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

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

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 $\langle e, s\rangle \rightarrow \langle e', s'\rangle$, 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 $\langle e, s\rangle \rightarrow^* \langle v', s'\rangle$, allora e' sarà un valore o esisterà e'', s'' tali che $\langle e', s'\rangle \rightarrow \langle e'', s''\rangle$.

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:

```
while true do
    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:

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

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:

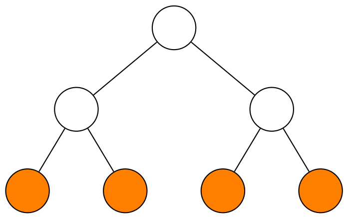
- **Caso base**: Dimostrare che $P(zero)$ è vero.

- **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.

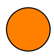
Questa prospettiva strutturale, e la forma associata di induzione, chiamata induzione strutturale, è ampiamente applicabile.

3.3.2 Albero binario attraverso l'induzione strutturale

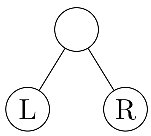


Definizione induttiva

- **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

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

- **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.

3.3.4 Dimostrazioni strutturali

Normalizzazione della big step semantics

Normalizzazione della big step semantics

Per ogni espressione aritmetica E esiste un numero naturale k t.c. $E \downarrow k$.

$$E \in \textbf{\textit{expr}} \ ::= \quad n \mid E_1 + E_2$$

Abbiamo a disposizione due regole di costruzione:

$$\frac{-}{n \downarrow n}\ (\text{B-num}) \qquad\qquad \frac{E_1 \downarrow n_1 \qquad E_2 \downarrow n_2}{E_1 + E_2 \downarrow n_3}\ n_3 = \texttt{add}(n_1, n_2)$$

$$\texttt{Propr} : \forall E \in \texttt{expr} \exists k \quad \textit{numerale} \quad \text{t.c. } E \downarrow k$$

Dimostrazione. Per induzione sulla struttura di E.

- **Caso base**: Sia E un non terminale n, $E \equiv n$, allora il k in questione è proprio n. Per la regola ($\texttt{B-num}$), $n \downarrow n$.

$$\frac{-}{n \downarrow n = k}\ (\text{B-num})$$

- **Passo induttivo**: E è nella forma $E_1 + E_2$ per qualche E_1 e E_2 (*sottostrutture di* E). Per ipotesi induttiva, essendo E_1 e E_2 sottostrutture di E, vale che:
  - $\exists k_1 \quad \textit{numerale} \quad \text{t.c. } E_1 \downarrow k_1$
  - $\exists k_2 \quad \textit{numerale} \quad \text{t.c. } E_2 \downarrow k_2$

  Usando la regola ($\texttt{B-add}$) e scegliendo $k_3 = \texttt{add}(k_1, k_2)$, otteniamo che:

$$\frac{E_1 \downarrow k_1 \qquad E_2 \downarrow k_2}{E \downarrow k_3}\ k_3 = \texttt{add}(k_1, k_2)$$

  Ho quindi trovato il k t.c. $E \downarrow k$ che cercavo.

$\square$

Small step semantics

$E \to F$ implica $E \to_{CH} F$ per ogni espressione aritmetica

Se $E \downarrow n$ e $E \downarrow m$, allora $n \equiv m$.

Dimostrazione. Per induzione sulla struttura di E.

- **Caso base**: Sia $E \equiv k$ per qualche $k \in \mathbb{N}$, allora sia $E \downarrow n$ che $E \downarrow m$ sono derivabili solo per la regola (`B-num`), dove $m = k = n$.

$$\frac{-}{k \downarrow k}\ \text{(B-num)}$$

  Ma allora $n \equiv m$.

$$\frac{-}{m \downarrow k}\ \text{(B-num)} \qquad\qquad \frac{-}{n \downarrow k}\ \text{(B-num)}$$

- **Passo induttivo**: E è nella forma $E_1 + E_2$.
  - Ne consegue che $E \downarrow m$ è stato derivato usando la regola (`B-add`). Esistendo $m_1 + m_2 = m$, per ipotesi induttiva, $E_1 \downarrow m_1$ e $E_2 \downarrow m_2$.

$$\frac{E_1 \downarrow m_1 \qquad E_2 \downarrow m_2}{E \downarrow m}\ m = \texttt{add}(m_1, m_2)$$

  - Ne consegue che $E \downarrow n$ è stato derivato usando la regola (`B-add`). Esistendo $n_1 + n_2 = n$, per ipotesi induttiva, $E_1 \downarrow n_1$ e $E_2 \downarrow n_2$.

$$\frac{E_1 \downarrow n_1 \qquad E_2 \downarrow n_2}{E \downarrow n}\ n = \texttt{add}(n_1, n_2)$$

□

Determinismo forte per la small-step semantics

Determinismo forte per la small-step semantics

Se $E \to F$ e $E \to G$, allora $F \equiv G$.

Dimostrazione. Per induzione sulla struttura di E.

- **Caso base**: $E \equiv n$ per qualche $n \in \mathbb{N}$. Non esiste alcuna E e F t.c. $E \to F$ e $E \to G$. Quindi la conclusione è falsa.
- **Passo induttivo**: E è nella forma $E_1 + E_2$. Sia $E \to F$ per qualche F.
  - Ho usato la regola (`S-left`), allora $\exists E_1'$ t.c. $E_1 \to E_1'$ e $E = E_1 + E_2 \to E_1' + E_2$. Avendo considerato $E \to G$ e avendo utilizzato la regola (`S-left`), dove $E_1 \to \hat{\mathrm{E}}_1$ e $E = E_1 + E_2 \to \hat{\mathrm{E}}_1 + E_2$, perché $E_1 \to \hat{\mathrm{E}}_1$ per ipotesi induttiva su E_1, $\hat{\mathrm{E}}_1 \equiv E_1'$, quindi $F = G$.

- Ho usato la regola (`S-right`), allora $E = m + E_2$ e E_2' t.c $E_2 \to E_2'$ e $E = m + E_2 \to m + E_2'$. Avendo considerato $E \to G$ e avendo utilizzato la regola (`S-right`), dove $E_2 \to \hat{\mathrm{E}}_2$ e $E = m + E_2 \to m + \hat{\mathrm{E}}_2$, perché $E_2 \to \hat{\mathrm{E}}_2$ per ipotesi induttiva su E_2, $\hat{\mathrm{E}}_2 \equiv E_2'$, quindi $F = G$.
- Ho usato la regola (`S-add`), allora $E = E_1 + E_2 = m_1 + m_2$ per qualche m_1, m_2 con $F = m_3 = \mathtt{add}(m_1, m_2)$, inoltre $E \to G$, ma avendo utilizzato la regola (`S-add`), ne consegue che $F = G = m_3$.

□

Determinismo debole per la small-step semantics

> Determinismo forte per la small-step semantics
>
> Se $E \to^* m$ e $E \to^* n$, allora $F \equiv G$.

Per dimostrare il determinismo debole, si utilizza la dimostrazione per induzione strutturale basata sulla dimostrazione del determinismo forte.

3.4 Induzione delle regole

Il comportamento delle espressioni aritmetiche E è completamente definito dalle regole dei suoi componenti. Per questa ragione l'induzione strutturale è sufficientemente potente per dimostrare proprietà per differenti semantiche di `Exp`. Però, in linguaggi più complessi, con operatori di controllo ricorsivi o induttivi, abbiamo bisogno di strumenti più sofisticati. L'idea di base della **induzione delle regole** è di ignorare la struttura degli oggetti e concentrarsi nella **dimensione delle derivazioni**

3.4.1 Dimensione delle derivazioni

Per esempio, consideriamo le seguenti regole, che definiscono una relazione binaria sui numeri naturali:

$$(\mathrm{Ax})\ \frac{-}{n\ \mathtt{Div}\ 0} \qquad\qquad (\mathrm{Plus})\ \frac{n\ \mathtt{Div}\ m}{n\ \mathtt{Div}\ (m+n)}$$

Derivazioni:

$$\begin{array}{r c}
 & - \\ \hline
(\mathrm{Ax}) & \text{7 Div 0} \\ \hline
(\mathrm{Plus}) & \text{7 Div 7} \\ \hline
(\mathrm{Plus}) & \text{7 Div 14} \\ \hline
(\mathrm{Plus}) & \text{7 Div 21}
\end{array}
\qquad\qquad
\begin{array}{r c}
 & - \\ \hline
(\mathrm{Ax}) & \text{2 Div 0} \\ \hline
(\mathrm{Plus}) & \text{2 Div 2} \\ \hline
(\mathrm{Plus}) & \text{2 Div 4}
\end{array}$$

Notimo che la derivazione a sinistra è più lunga di quella a destra, infatti `2 Div 4` è più piccolo di `7 Div 21`.

Supponiamo di voler dimostrare una dichiarazione della forma:

$$n\ \mathtt{Div}\ m \to P(n, m)$$

Dove possiamo utilizzare l'induzione sulla dimensione della derivazione di n `Div` m. Supponiamo quindi che n `Div` m, dove $m = n \cdot k$ per qualche $k \in \mathbb{N}$. Ciò significa effettivamente che le regole `(Ax)` e `(Plus)` catturano correttamente il concetto di divisione. Quindi, dimostriamo che $n \,\mathrm{Div}\, m \to P(n, m)$ per

induzione matematica sulla dimensione della derivazione della valutazione $n \,\mathrm{Div}\, m$ dalle regole `(Ax)` e `(Plus)`.

Ciò significa effettivamente che le regole `(Ax)` e `(Plus)` catturano correttamente il concetto di divisione. Quindi, supponiamo di avere una derivazione di $n \,\mathrm{Div}\, m$. Utilizzare l'induzione matematica significa avere un'ipotesi induttiva che afferma che $P(k_1, k_2)$ è vera per qualsiasi k_1, k_2 per i quali esiste una derivazione $k_1 \,\mathrm{Div}\, k_2$ la cui dimensione è inferiore alla dimensione della derivazione di $n \,\mathrm{Div}\, m$.

Quindi n `Div` m può essere derivato solo da `(Ax)` o `(Plus)`.

Abbiamo due possibilità:

- Abbiamo l'applicazione di un assioma `(Ax)`:

$$(\text{Ax}) \, \frac{-}{n \ \texttt{Div} \ 0}$$

  Solo se $m = 0$, quindi $P(n, 0)$ è vera, ma k deve essere 0.

- Abbiamo l'applicazione di una regola `(Plus)`:

$$(\text{Plus}) \, \frac{(...) \, \dfrac{...}{n \ \texttt{Div} \ m_1}}{n \ \texttt{Div} \ (m_1 + n)}$$

  Dove $m = m_1 + n$.

Però questo significa che la valutazione di n `Div` m ha anche una derivazione mediante regole. La dimensione della derivazione è minore della dimensione della derivazione di n `Div` m. Quindi per ipotesi induttiva, sappiamo che esiste un k_1 tale che $m_1 = n \cdot k_1$.

Ora, $P(n, m)$ è una immediata conseguenza di $m = n \cdot (K_1 + 1)$.

3.5 Definizione formale di induzione delle regole

Per dimostrare una proprietà $P(D)$ per ogni derivazione D, è sufficiente fare quanto segue:

(a) **Caso base:** dimostra che $P(A)$ è vera per ogni assioma A (utilizzando fatti matematici noti).

(b) **Caso induttivo:** per ogni regola della forma

$$(\text{Rule}) \, \frac{h_1, \ldots, h_n}{c}$$

dimostra che ogni derivazione che termina con l'uso di questa regola soddisfa la proprietà. Tale derivazione ha sottoderivazioni $D_1, \ldots, D_n$ con conclusioni $h_1, \ldots, h_n$. Per ipotesi induttiva, assumiamo che $P(D_i)$ valga per ogni sotto-derivazione D_i, $1 \leq i \leq n$.

Teorema 3.5.1 **Progress**
Se $\Gamma \vdash e : T$ e $dom(\Gamma) \subseteq dom(s)$ allora e è un valore o esistono e', s' tali che $\langle e, s\rangle \to \langle e', s'\rangle$.

Example

Come esempio, dimostriamo che se $\Gamma \supseteq \{I, \texttt{intref}\}$, allora $\Gamma \vdash (!l + 2) + 3 : \texttt{int} \implies \phi(\Gamma, (!l+2)+3, \texttt{int})$.

$$\dfrac{(\texttt{op}\ +)\ \dfrac{(\texttt{deref})\ \dfrac{-}{\Gamma \vdash !l : \texttt{int}} \quad (\texttt{int})\ \dfrac{-}{\Gamma \vdash 2 : \texttt{int}}}{\Gamma \vdash (!l + 2) : \texttt{int}} \quad (\texttt{int})\ \dfrac{-}{\Gamma \vdash 3 : \texttt{int}}}{\Gamma \vdash (!l+2)+3 : \texttt{int}}\ (\texttt{op}\ +)$$

3.5.1 Preservazione del tipo

Lemma 3.5.1 *Se $\langle e, s\rangle \to \langle e', s'\rangle$ allora $dom(s) = dom(s')$.*

Dimostrazione. Per induzione sulle regole su perché $\langle e, s\rangle \to \langle e', s'\rangle$. Sia $\Phi(e, s, e', s') = (\mathrm{dom}(s) = \mathrm{dom}(s'))$. Tutte le regole sono utilizzi immediati dell'ipotesi induttiva, tranne la regola (assign1), per la quale notiamo che se $l \in \mathrm{dom}(s)$ allora $\mathrm{dom}(s[l \mapsto n]) = \mathrm{dom}(s)$. $\square$ $\square$

Teorema 3.5.2 **Conservazione del tipo**
Se $\Gamma \vdash e : T$ e $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ e $\langle e, s\rangle \to \langle e', s'\rangle$ allora $\Gamma \vdash e' : T$ e $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s')$.

Capitolo 4

Funzioni

Molti dei linguaggi di programmazione hanno delle notazione per indicare le funzioni, metodi o procedure.

Gli esempi possono essere di vari natura, vediamone alcuni:

- `fn x : int` $\implies$ `x + 1`
- (`fn x : int` $\implies$ `x + 1`)8
- `fn y` $\implies$ (`fn x : int` $\implies$ $x + y$)
- ...

Per semplicità le nostre funzioni saranno **anonime**, prenderanno un solo argomento e restituiranno un solo valore e saranno sempre tipate.

4.1 Funzioni - Estensione della sintassi

$$\begin{aligned}
\textit{Variabili } x \ &\in \mathbb{X}, \quad \text{per } \mathbb{X} = \{x, y, z, \dots\} \\
\textit{Espressioni } e \ &::= \ \texttt{fn x : T} \implies e \mid e\,e \mid x \\
\textit{Tipi } T \ &::= \ \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{T} \to \texttt{T} \\
\textit{Tipi locazioni} T_{loc} \ &::= \ \texttt{intref}
\end{aligned}$$

Convenzioni

L'applicazione di funzione si associa a sinistra: $e_1 e_2 e_3 = (e_1 e_2) e_3$. Il tipo di funzione si associa a destra:

$$T_1 \to T_2 \to T_3 = T_1 \to (T_2 \to T_3)$$

f si estende verso destra quanto più possibile, quindi `fn x : unit` $\Rightarrow x; x$ corrisponde a `fn x : unit` $\Rightarrow (x; x)$ e `fn x : unit` $\Rightarrow$ `fn y : unit` $\Rightarrow x; y$ ha tipo `unit` $\rightarrow$ `int` $\rightarrow$ `int`.

Si osservi che le variabili non rappresentano posizioni ($\mathbb{X} \cap \mathbb{L} = \emptyset$). Pertanto, l'assegnazione come $x := 3$ non è consentita. Inoltre, non è possibile eseguire astrazioni sulle posizioni: $fnl : intref \Rightarrow !l + 5$ non è conforme alla sintassi. Le variabili (*non meta*) x, y, z non sono le stesse delle meta-variabili x, y, z, ... La grammatica dei tipi e la sintassi delle espressioni suggeriscono che il linguaggio includa funzioni di ordine superiore: è possibile astrarsi su una variabile di qualsiasi tipo.

4.2 Shadowing delle variabili

In un linguaggio con definizioni di funzioni annidate, è auspicabile definire una funzione senza essere consapevoli delle variabili nello spazio circostante.

$$\texttt{fn x : int} \Longrightarrow (\texttt{fn x : int} \Longrightarrow x + 1)$$

Lo *shadowing* delle variabili è una tecnica che permette di definire una nuova variabile con lo stesso nome di una variabile già esistente, in modo che la nuova variabile *nasconda* la precedente. Questa tecnica però non è consentita nel linguaggio di programmazione *Java*.

4.3 Alpha conversion, variabile libera e vincolata

Nelle espressioni della forma `fn x : T` $\Rightarrow e$, la variabile x è **vincolata**. Essa rappresenta il parametro formale della funzione: ogni occorrenza di x in e, che non si trovi all'interno di una definizione di funzione annidata, ha lo stesso significato al di fuori del termine `fn x : T` $\Rightarrow e$. Di conseguenza, la variabile x non ha un significato proprio! Questo implica che non importa quale variabile sia stata scelta come parametro formale: le espressioni `fn x : int` $\Rightarrow x + 2$ e `fn y : int` $\Rightarrow y + 2$ denotano esattamente la stessa funzione!

Diremo che un'occorrenza di una variabile x all'interno di un'espressione e è libera se x non è all'interno di alcun termine della forma `fn x : T` $\Rightarrow \ldots$.

Ad esempio, la variabile x è libera nelle seguenti espressioni:

- 21
- $x + y$
- `fn z : T` $\Rightarrow x + z$

Si noti che, nell'ultimo esempio, la variabile x è libera, ma la variabile z è vincolata dalla definizione di funzione più interna `fn z : T` $\Rightarrow$ ....

Si noti inoltre che, nell'espressione

$$\texttt{fn x : T'} \Rightarrow \texttt{fn z : T} \Rightarrow x + z$$

la variabile x non è più libera, ma è vincolata dalla definizione di funzione più interna `fn x : T'` $\Rightarrow$ ....

Convenzione

Ci consentiremo sempre, in qualsiasi espressione

$$...(\texttt{fn x : T} \Rightarrow e)...$$

di sostituire il vincolo x e tutte le occorrenze di x in e che sono vincolate a quel legante, con qualsiasi altra variabile *fresh* che non appare altrove.

Ad esempio:

- $\texttt{fn x : T} \Rightarrow x + z = \texttt{fn y : T} \Rightarrow y + z$
- $\texttt{fn x : T} \Rightarrow x + y \neq \texttt{fn y : T} \Rightarrow y + y$

Questo processo è chiamato "Alpha conversion".

L'intuizione è che le variabili libere non sono ancora vincolate a nessuna espressione. Definiamo la seguente funzione per induzione:

$$\texttt{fv() : Exp} \rightarrow 2^{\mathbb{X}}$$

$$\begin{array}{lll}
\texttt{fv}(x) & \stackrel{\text{def}}{=} & \{x\} \\
\texttt{fv}(\texttt{fn x : T} \Rightarrow e) & \stackrel{\text{def}}{=} & \texttt{fv}(e) \setminus \{x\} \\
\texttt{fv}(e_1; e_2) = \texttt{fv}(e_1 e_2) & \stackrel{\text{def}}{=} & \texttt{fv}(e_1) \cup \texttt{fv}(e_2) \\
\texttt{fv}(n) = \texttt{fv}(b) = \texttt{fv}(!l) = \texttt{fv}(\texttt{skip}) & \stackrel{\text{def}}{=} & \varnothing \\
\texttt{fv}(e_1 \oplus e_2)\texttt{fv}(\texttt{while } e_1 \texttt{ do } e_2) & \stackrel{\text{def}}{=} & = \texttt{fv}(e_1) \cup \texttt{fv}(e_2) \\
\texttt{fv}(l := e) & \stackrel{\text{def}}{=} & \texttt{fv}(e) \\
\texttt{fv}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & \stackrel{\text{def}}{=} & \texttt{fv}(e_1) \cup \texttt{fv}(e_2) \cup \texttt{fv}(e_3)
\end{array}$$

Un'espressione e si dice **chiusa** se $fv(e) = \varnothing$.

4.3.1 Sostituzione

La semantica delle funzioni coinvolge la sostituzione del parametro effettivo per i parametri formali. Scriviamo $e_2\{e_1/x\}$ per il risultato della sostituzione di e_1 per tutte le occorrenze libere di x in e_2.

Ad esempio:

- $(x \geq x)\{3/x\} = (3 \geq 3)$
- $(\texttt{fn } x : \texttt{int} \Rightarrow x + y)x\{3/x\} = (\texttt{fn } x : \texttt{int} \Rightarrow x + y)3$
- $(\texttt{fn } y : \texttt{int} \Rightarrow x + y)\{y + 2/x\} = \texttt{fn } z : \texttt{int} \Rightarrow (y + 2) + z$

Si noti che nell'ultima sostituzione, lavoriamo con l'ausilio dell'alfa conversion per evitare la cattura di nomi!

Definiamo la sostituzione $\hat{e}\{e/x\}$ come la sostituzione dell'espressione e per ogni occorrenza libera di x in $\hat{e}$.

$$
\begin{array}{lcl}
n\{e/x\} & \stackrel{\text{def}}{=} & n \\
b\{e/x\} & \stackrel{\text{def}}{=} & b \\
\texttt{skip}\{e/x\} & \stackrel{\text{def}}{=} & \texttt{skip} \\
x\{e/x\} & \stackrel{\text{def}}{=} & e \\
y\{e/x\} & \stackrel{\text{def}}{=} & y \\
(\texttt{fn}\ x : T \Rightarrow e_1)\{e/z\} & \stackrel{\text{def}}{=} & \texttt{fn}\ x : T \Rightarrow e_1\{e/z\} \text{ se } x \notin \texttt{fv}(e) \\
(\texttt{fn}\ x : T \Rightarrow e_1)\{e/z\} & \stackrel{\text{def}}{=} & (\texttt{fn}\ y : T \Rightarrow (e_1\{y/x\})\{e/z\}) \text{ se } x \in \texttt{fv}(e) \text{ e } y \ \texttt{fresh} \\
(\texttt{fn}\ x : T \Rightarrow e_1)\{e/x\} & \stackrel{\text{def}}{=} & \texttt{fn}\ x : T \Rightarrow e_1 \\
(e_1 e_2)\{e/x\} & \stackrel{\text{def}}{=} & (e_1\{e/x\} e_2\{e/x\})
\end{array}
$$

D'altra parte, la sostituzione è un omomorfismo per le altre espressioni.

4.3.2 Sostituzioni simultanee

Le sostituzioni possono essere facilmente generalizzate per sostituire contemporaneamente più variabili. Più precisamente, una sostituzione simultanea σ è una funzione parziale:

$$\sigma : X \rightharpoonup \text{Exp}$$

Data un'espressione e, scriviamo $e\sigma$ per indicare l'espressione risultante dalla sostituzione simultanea di ciascun $x \in \texttt{dom}(\sigma)$ con l'espressione corrispondente $\sigma(x)$.

Notazione: scriviamo σ come $\{e_1/x_1, \ldots, e_k/x_k\}$ invece di $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\}$. Useremo $e\sigma$ per indicare l'espressione e che è stata influenzata dalla sostituzione σ.

4.4 Lambda calcolo

Il λ calcolo può essere arricchito in una varietà di modi. È spesso conveniente aggiungere costrutti speciali per caratteristiche come numeri, booleani, tuple, record, ecc. Tuttavia, tutte queste funzionalità possono essere codificate nel λ calcolo, quindi rappresentano solo "zucchero sintattico". Tali estensioni portano infine a linguaggi di programmazione come `Lisp` (*McCarthy, 1960*), `ML` (*Milner et al., 1990*), `Haskell` (*Hudak et al., 1992*) o `Scheme` (*Sussman and Steele, 1975*).

Nel λ calcolo, abbiamo fondamentalmente esteso il linguaggio con i seguenti costrutti primitivi:

$$M \in \mathit{Lambda} \quad ::= x \mid \lambda x.M \mid MM$$

dove:

- x è una variabile, utilizzata per definire parametri formali
- $\lambda x.M$ è chiamato λ-abstraction e definisce funzioni anonime; questo costrutto lega la variabile x nel corpo della funzione M
- MM è l'applicazione della funzione M all'argomento M

Così, $(\lambda x.M)N$ evolve in $M\{N/x\}$, dove l'argomento N sostituisce ogni occorrenza (*libera*) di x in M. Nel λ calcolo puro, le funzioni sono gli unici valori. Gli interi, i booleani e altri valori di base possono essere facilmente codificati e non sono primitivi nel λ calcolo.

4.5 Applicazione di funzioni

Per valutare M_1M_2, Prima valutiamo M_1 come una funzione $\lambda x.M$. Quindi, la strategia di valutazione dipende dalla strategia di valutazione adottata.

Call by value

Valutiamo prima M_2 ad un valore v e poi valutiamo $M\{M_2/x\}$. Quindi M_2 viene valutato prima di essere passato alla funzione.

Call by name

Valutiamo $M\{M_2/x\}$, perciò M_2 verrà valutato se e solo se è necessario.

4.5.1 Semantica formale

La regola dell'applicazione è la seguente:

$$(\text{App})\ \frac{M_1 \to M_1'}{M_1M_2 \to M_1'M_2}$$

Call by value

$$(\text{CBV.A})\ \frac{M_2 \to M_2'}{(\lambda x.M)M_2 \to (\lambda x.M)M_2'} \qquad (\text{CBV.B})\ \frac{\text{-}}{(\lambda x.M)v \to M\{v/x\}}$$

Dove $v \in Val ::= \lambda x.M$.

Call by name

$$(\text{CBN})\ \frac{\text{-}}{(\lambda x.M)M_2 \to M\{M_2/x\}}$$

Dove M_2 è un termine chiuso.

4.5.2 Applicazione ricorsiva

È possibile esprimere programmi non terminanti nel lambda calcolo? Sì, naturalmente! Ad esempio, il combinatore di divergenza ω definito come

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

contiene un solo *redex* e la riduzione di questo *redex* produce nuovamente Ω.

$$\Omega \to \Omega \to \Omega \to \dots \Omega \dots$$

La non terminazione è intrinseca nel lambda calcolo!

4.5.3 Differenze tra call by value e call by name

Notiamo che, a differenza del linguaggio precedente, avere definizioni di funzioni tra i costrutti può portare a risultati diversi.

Danno risultati diversi:

- $(\lambda x.0)(\Omega) \rightarrow_{cbn} = 0$
- $(\lambda x.0)(\Omega) \rightarrow_{cbv} = (\lambda x.0)(\Omega) \rightarrow_{cbv} \cdots \rightarrow_{cbv} \ldots$

Il risultato è ancora più sorprendentemente:

- $(\lambda x.\lambda y.x)(Id0) \rightarrow^*_{cbn} = \lambda y.(Id0)$
- $(\lambda x.\lambda y.x)(Id0) \rightarrow^*_{cbv} = \lambda y.0$

4.6 Comportamento delle funzioni

Consideriamo l'espressione:

$$e \stackrel{\text{def}}{=\!=} \texttt{fn}\ \ x : \texttt{unit} \Rightarrow (l := 1); x(l := 2)$$

Poi consideriamo una esecuzione nello store dove l è associato a 0:

$$\langle e, \{l \mapsto 0\}\rangle_* \langle \texttt{skip}, \{l \mapsto ???\}\rangle$$

Il risultato dello store potrebbe variare a seconda della valutazione dell'espressione e, poiché potrebbero essere presenti dichiarazioni o assegnamenti che possono influire sul risultato della locazione l, quindi avere *side effects*.

Come valutiamo una chiamata di funzione $e_1\ e_2$?

Call-by-value (*detta anche valutazione eager*)

1. Valutiamo e_1 a una funzione fn $x : T \Rightarrow e$
2. Valutiamo e_2 a un valore v
3. Sostituiamo il **parametro attuale** v per il **parametro formale** x nel corpo della funzione e
4. Valutiamo $e\{v/x\}$

Questo metodo è utilizzato in molti linguaggi come `C`, `Scheme`, `ML`, `OCaml`, `Java`, ecc. (*ci sono diverse varianti di call-by-value*).

C'è un altro metodo per valutare $e_1\ e_2$

Call-by-name (*detta anche valutazione lazy*)

1. Valutiamo e_1 a una funzione fn $x : T \Rightarrow e$
2. Sostituiamo il **parametro attuale** e_2 per il **parametro formale** x nel corpo della funzione e
3. Valutiamo $e\{e_2/x\}$

Varianti del call-by-name sono state utilizzate in alcuni linguaggi di programmazione ben noti, in particolare `Algol-60` (*Naur et al., 1963*) e `Haskell` (*Hudak et al., 1992*). Haskell utilizza effettivamente una versione ottimizzata nota come call-by-need (*Wadsworth, 1971*) che, anziché rivalutare un argomento ogni volta che viene utilizzato, sovrascrive tutte le occorrenze dell'argomento con il suo valore la prima volta che viene valutato.

Valutiamo il nostro esempio precedente in una strategia call-by-value:

$$e = (\text{fn } x : \text{unit} \Rightarrow (l := 1); x)(l := 2)$$

Poi:

$$\begin{aligned}
\langle e, \{l \mapsto 0\}\rangle &\rightarrow \langle(\texttt{fn } x : \texttt{unit} \Rightarrow (l := 1); x)\texttt{skip}, \{l \mapsto 2\}\rangle \\
&\rightarrow \langle(l := 1; \texttt{skip}), \{l \mapsto 2\}\rangle \\
&\rightarrow \langle \texttt{skip}; \texttt{skip}, \{l \mapsto 1\}\rangle \\
&\rightarrow \langle \texttt{skip}, \{l \mapsto 1\}\rangle
\end{aligned}$$

Alla fine della valutazione, la locazione l è associata a 1.

Procediamo ora con la strategia call-by-name:

$$\begin{aligned}
\langle e, \{l \mapsto 0\}\rangle &\rightarrow \langle(l := 1); l := 2, \{l \mapsto 0\}\rangle \\
&\rightarrow \langle \texttt{skip}; l := 2, \{l \mapsto 1\}\rangle \\
&\rightarrow \langle l := 2, \{l \mapsto 1\}\rangle \\
&\rightarrow \langle \texttt{skip}, \{l \mapsto 2\}\rangle
\end{aligned}$$

Notiamo quindi che le due strategie di valutazione hanno prodotto risultati diversi.

4.6.1 Call-by-value: small-step semantics

$$\textit{Valori } v \ ::= \ b \mid n \mid \texttt{skip} \mid \texttt{fn } x : T \Rightarrow e$$

$$(\text{CBV-app1}) \ \frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1 \ e_2, s\rangle \rightarrow \langle e_1' \ e_2, s'\rangle}$$

$$(\text{CBV-app2}) \ \frac{\langle e_2, s\rangle \rightarrow \langle e_2', s'\rangle}{\langle v_1 \ e_2, s\rangle \rightarrow \langle v_1 \ e_2', s'\rangle}$$

$$(\text{CBV-fn}) \ \frac{-}{\langle(\texttt{fn } x : T \Rightarrow e)v, s\rangle \rightarrow \langle e\{v/x\}, s\rangle}$$

La valutazione della funzione non tocca lo store, infatti in un linguaggio funzionale puro non avremmo bisogno di uno store. In $e\{v/x\}$ il valore v verrebbe copiato tante volte quante sono le occorrenze libere di x in e. Le implementazioni reali non fanno questo.

4.6.2 Call-by-name: small-step semantics

$$\text{(CBN-app)}\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1\ e_2, s\rangle \to \langle e_1'\ e_2, s'\rangle}$$

$$\text{(CBN-fn)}\ \frac{-}{\langle (\texttt{fn}\ x : T \Rightarrow e)e_2, s\rangle \to \langle e\{e_2/x\}, s\rangle}$$

Non valutiamo l'argomento della funzione, ma lo sostituiamo direttamente nel corpo, in questo modo se tale argomento non verrà utilizzato non verrà mai valutato, ma se verrà utilizzato più volte verrà valutato ogni volta che viene utilizzato.

4.7 Tipizzazione delle funzioni

Fino ad ora, un ambiente di tipi Γ fornisce il tipo delle locazioni dello store. Da adesso in poi, deve anche fornire assunzioni sul tipo delle variabili usate nelle funzioni, ad esempio

$$\Gamma = \{l_1 : \text{int ref}, x : \text{int}, y : \text{bool} \to \text{int}\}$$

Quindi, estendiamo il set `TypeEnv` degli ambienti dei tipi come segue:

$$\texttt{TypeEnv} \stackrel{\text{def}}{=} \mathbb{L} \cup \mathbb{X} \rightharpoonup T_{loc} \cup T$$

Tale che:

- $\forall l \in \texttt{dom}(\Gamma).\Gamma(l) \in T_{loc}$
- $\forall x \in \texttt{dom}(\Gamma).\Gamma(x) \in T$

Notazioni: se $x \notin \text{dom}(\Gamma)$, scriviamo $\Gamma, x : T$ per la funzione parziale che mappa x su T ma altrimenti è come Γ.

Si noti che con l'introduzione delle funzioni ci sono più configurazioni bloccate (ad esempio 2, `true`, `true fn` $x : T \Rightarrow e$, ecc.). Il nostro sistema di tipi respingerà queste configurazioni tramite le seguenti regole:

$$\text{(var)}\ \frac{-}{\Gamma \vdash x : T}\ \textit{se}\ \Gamma(x) = T$$

$$\text{(fn)}\ \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \texttt{fn}\ x : T \Rightarrow e : T \to T'}$$

$$\text{(app)}\ \frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'}$$

4.7.1 Proprietà del sistema di tipi

Consideriamo solo esecuzioni di programmi chiusi, senza variabili libere.

Teorema 4.7.1 **Progress**
Se e è chiuso e $\Gamma \vdash e : T$ e $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$, allora o c'è un valore o esiste e', s' tale che $\langle e, s\rangle \to \langle e', s'\rangle$.

Teorema 4.7.2 **Conservazione del tipo**
Se e è chiuso e $\Gamma \vdash e : T$ e $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s)$ e $\langle e, s\rangle \to \langle e', s'\rangle$ allora $\Gamma \vdash e' : T$ e e' è chiuso e $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s')$.

Questo richiede il seguente lemma:

Lemma 4.7.1 **Sostituzione**
Se $\Gamma \vdash e : T$ *e* $\Gamma, x : T \vdash e' : T'$ *con* $x \notin dom(\Gamma)$ *allora* $\Gamma \vdash e'\{e/x\} : T'$.

Teorema 4.7.3 **Normalizzazione**
Nei sotto-linguaggi senza cicli while o operazioni di store, se $\Gamma \vdash e : T$ e e è chiuso, allora c'è un valore v tale che, per ogni store, $\langle e, s\rangle \to^* \langle v, s\rangle$. In altre parole, se consideriamo un linguaggio funzionale puro, come il calcolo lambda, la sua versione tipizzata non è più Turing-completa.

4.8 Dichiarazioni locali

4.8.1 Sintassi e tipi

4.8.2 Intuizione

4.8.3 Variabili legate e libere

4.8.4 Alpha conversion

4.8.5 Small-step semantics

4.9 Ricorsione

4.9.1 Punto fisso

4.10 Punto fisso nel linguaggio funzionale