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

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

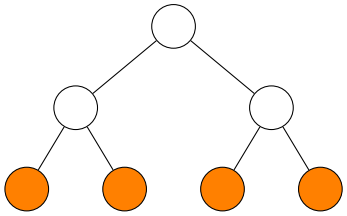
- **Caso base**: Dimostrare che $P(zero)$ è vero.

- **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.

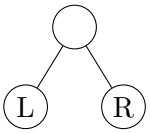
3.3.2 Albero binario attraverso l'induzione strutturale



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.

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. $E \downarrow k$.

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

Abbiamo a disposizione due regole di costruzione:

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

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

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 (`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 (`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.

□

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 $E_2 \to \hat{E}_2$ e $E = m + E_2 \to m + \hat{E}_2$, perché $E_2 \to \hat{E}_2$ per ipotesi induttiva su E_2, $\hat{E}_2 \equiv E_2'$, 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 $F = m_3 = \texttt{add}(m_1, m_2)$, 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:

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

Derivazioni:

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

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:

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

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

induzione matematica sulla dimensione della derivazione della valutazione $n \operatorname{Div} m$ 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 $n \operatorname{Div} m$. 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 $k_1 \operatorname{Div} k_2$ la cui dimensione è inferiore alla dimensione della derivazione di $n \operatorname{Div} m$.

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 $\Gamma \vdash (!l + 2) + 3 : \texttt{int} \implies \phi(\Gamma, (!l + 2) + 3, \texttt{int})$.

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

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 $\mathrm{dom}(s[l \mapsto n]) = \mathrm{dom}(s)$. □ □

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

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

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

Convenzioni

L'applicazione di funzione si associa a sinistra: $e_1e_2e_3 = (e_1e_2)e_3$. 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:

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

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

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 $M_1 M_2$, 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:

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

Call by value

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

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

Call by name

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

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.

$$\Omega \to \Omega \to \Omega \to \ldots \Omega \ldots$$

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:

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

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

Procediamo ora con la strategia call-by-name:

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

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

4.6.1 Call-by-value: small-step semantics

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

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

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

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

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:

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

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

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

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*(Γ) *allora* $\Gamma \vdash e'\{e/x\} : T'$.

Teorema 4.7.3 Normalizzazione Nei sotto-linguaggi senza cicli while o operazioni di store, se $\Gamma \vdash e : T$ ed 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

Per la leggibilità, vogliamo essere in grado di nominare le espressioni e di limitarne la loro portata.

Consideriamo infatti:

$$((1 + 2) \geq (1 + 2) + 4)$$

Questa è una caratteristica molto comune di molti linguaggi di programmazione. Nuova costruzione:

$$\texttt{let y:int} = 1 + 2 \ \texttt{in} \ y \geq (y + 4)$$

Intuizione: prima valutiamo $1 + 2$ per ottenere 3. Poi valutiamo $y \geq (y + 4)$ con y sostituito da 3. Qui, y è un *binder*, lega ogni occorrenza libera di y in $y \geq (y + 4)$.

4.8.1 Sintassi e tipi

Estendiamo la sintassi delle nostre espressioni:

$$e \ ::= \ \ldots \mid \texttt{let} \ x : T = e_1 \ \texttt{in} \ e_2$$

Forniamo la regola di tipizzazione del nuovo costrutto:

$$(\text{let}) \ \frac{\Gamma \vdash e_1 : T \qquad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \texttt{let} \ x : T = e_1 \ \texttt{in} \ e_2 : T'}$$

Si noti che, poiché x è una variabile locale, Γ non contiene una voce per la variabile x. Ciò significa che, come per `fn`, la tipizzazione di un costrutto `let` può richiedere un'alpha conversione .

4.8.2 Intuizione

In un costrutto `let`, le variabili sono segnaposto per quantità sconosciute.

Problema 1: Molte espressioni non hanno senso

$$\texttt{let } y : T = 2 + 3 \texttt{ in } y + z$$
$$\texttt{let } z : T = 2 + x \texttt{ in } z \times z$$

Problema 2: Le variabili possono essere usate in più ruoli

$$\text{let } z : T = 2 + z \texttt{ in } z \times y$$

Problema 3: Dichiarazioni multiple per lo stesso valore

$$\texttt{let } y : T = 1 \texttt{ in let } y : T' = (1 + 2) \texttt{ in } y \times (y + 4)$$

4.8.3 Variabili legate e libere

Intuizione: Nel costrutto `let`, $y : T = 2 + 3$ `in` $y + z$

- y è legato, rappresenta l'espressione $2 + 3$.
- z è libero, non rappresenta alcuna espressione.

Nel costrutto `let`, $z : T = 2 + x$ `in` $z \times (z + y)$

- z è legato, rappresenta l'espressione $2 + x$
- x e y sono libere, non rappresentano alcuna espressione.

Nel costrutto `let`, $z : T = 2 + z$ `in` $z \times (z + y)$

- z è legato, rappresenta l'espressione $2 + z$
- z e y sono libere, non rappresentano alcuna espressione.

4.8.4 Alpha conversion

Come il costrutto `let` è un binder per la variabile locale, possiamo utilizzare l'alfa conversion quando necessario.

Convenzione: ci permetteremo ogni volta, in qualsiasi espressione `let` $x : T = e_1$ `in` e_2 di sostituire il legame x e tutte le occorrenze di x in e_2 che sono legate a tale binder, con qualsiasi altra variabile fresh che non compare altrove:

$$\texttt{let } x : T = e1 \texttt{ in } e_2 \equiv_\alpha \text{let } y : T = e_1 \texttt{ in } e_2\{y/x\}$$

dove y è una variabile fresh, ovvero non compare né in e_1 né in e_2.

La definizione delle variabili libere per il costrutto `let` è come ci si potrebbe aspettare:

$$\text{fv}(\text{let } x : T = e1 \text{ in } e2) \equiv \text{fv}(e1) \cup (\text{fv}(e2) \setminus \{x\})$$

Per quanto riguarda la sostituzione, dobbiamo fare attenzione in quanto potremmo aver bisogno di lavorare fino all'alfa conversion:

$$\begin{array}{lll} (\texttt{let } x:T=e_1 \texttt{ in } e_2)\{e/z\} & \stackrel{\text{def}}{=} & (\texttt{let } x:T=e_1\{e/z\} \texttt{ in } e_2\{e/z\}) \\ & & \text{se } x \notin \texttt{fv}(e) \\ (\texttt{let } x:T=e_1 \texttt{ in } e_2)\{e/z\} & \stackrel{\text{def}}{=} & (\texttt{let } y:T=e_1\{e/z\} \texttt{ in } (e_2\{y/x\}\{e/z\})) \\ & & \text{se } x \in \texttt{fv}(e) \text{ and } y \texttt{ fresh} \\ (\texttt{let } x:T=e_1 \texttt{ in } e_2)\{e/z\} & \stackrel{\text{def}}{=} & (\texttt{let } x:T=e_1\{e/z\} \texttt{ in } e_2) \end{array}$$

Le nostre definizioni utilizzano variabili e non meta-variabili: quindi $x \neq z$!

4.8.5 Small-step semantics

Per quanto riguarda l'applicazione di funzione, possiamo avere almeno un paio di semantica differenti per il costrutto `let`.

Call-by-value

$$\text{(CBV-let1)}\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle \texttt{let } x:T=e_1 \texttt{ in } e_2, s\rangle \to \langle \texttt{let } x:T=e_1' \texttt{ in } e_2, s'\rangle}$$

$$\text{(CBV-let2)}\ \frac{-}{\langle \texttt{let } x:T=v \texttt{ in } e_2, s\rangle \to \langle \texttt{let } x:T=e_2\{v/x\}, s\rangle}$$

Call-by-name

$$\text{(CBN-let)}\ \frac{-}{\langle \texttt{let } x:T=e_1 \texttt{ in } e_2, s\rangle \to \langle e_2\{e_1/x\}, s\rangle}$$

4.9 Ricorsione

Nel contesto del nostro linguaggio, la modellazione delle funzioni ricorsive solleva importanti questioni sulle loro proprietà e sulla loro presenza libera. È innegabile che, grazie al teorema di normalizzazione, abbiamo acquisito la consapevolezza che senza istruzioni di ciclo o memorizzazione non possiamo sostenere calcoli infiniti, rendendo così impossibile la presenza diretta di meccanismi di ricorsione.

Tuttavia, possiamo notare che mediante l'uso combinato di costrutti come `if`, `while` e la memorizzazione, siamo in grado di implementare la ricorsione all'interno del nostro linguaggio. È importante sottolineare che, pur offrendo questa possibilità, l'utilizzo di tali costrutti può diventare complesso e richiedere una pianificazione attenta.

D'altra parte, è interessante osservare che abbiamo già intrapreso questa strada. Pertanto, per garantire un'implementazione efficiente e strutturata delle funzioni ricorsive, è auspicabile prendere come ispirazione il λ-calcolo.

4.9.1 Punto fisso

In matematica, un punto fisso p (*noto anche come punto invariante*) di una funzione f è un punto che viene mappato su se stesso, cioè $f(p) = p$. I punti fissi rappresentano il cuore di ciò che è noto come teoria della ricorsione.

Teoremi di ricorsione di Kleene

I teoremi di ricorsione di Kleene sono un paio di risultati fondamentali sull'applicazione di funzioni calcolabili alle proprie descrizioni. I due teoremi di ricorsione possono essere applicati per costruire punti fissi di determinate operazioni su funzioni calcolabili, per generare quines e per costruire funzioni definite tramite definizioni ricorsive.

Teorema 4.9.1 Rice per ogni proprietà non banale delle funzioni parziali, non esiste un metodo generale ed efficace per decidere se un algoritmo calcoli una funzione parziale con tale proprietà.

Quindi, è molto importante dimostrare che il λ-calcolo può esprimere i punti fissi. Utilizziamo il combinatorio di Turing per derivare i punti fissi:

$$A \stackrel{\text{def}}{=} \lambda x.\lambda y.y(xxy)$$

$$\texttt{fix} \stackrel{\text{def}}{=} AA$$

`fix` è una funzione ricorsiva che, dato un termine M, restituisce un punto fisso di M, indicato con fix$M$. Infatti, per qualsiasi termine M, utilizzando una valutazione call-by-name, abbiamo:

$$\begin{array}{rcl} \texttt{fix}\ M & \to & (\lambda y.y(AAy))M \\ & \to & M(\texttt{fix}\ M) \end{array}$$

Ora, se scegliamo M nella forma $\lambda f.\lambda x.B$, per qualche corpo B, allora, in una semantica call-by-name, abbiamo:

$$\begin{array}{rcl} \texttt{fix}(\lambda f.\lambda x.B) & \to^* & (\lambda f.\lambda y(B))(\texttt{fix}(\lambda f.\lambda x.B)) \\ & \to & \lambda y.B\{\texttt{fix}(\lambda f.\lambda x.B)/f\} \end{array}$$

Definizioni ricorsive

Quindi, se definiamo:

$$\texttt{rec}f.B \textit{ come abbreviazione per } \texttt{fix}(\lambda f.\lambda x.B)$$

allora possiamo riscrivere la riduzione precedente come:

$$\texttt{rec}f.B \to^* \lambda x.B\{\texttt{rec}f.B/f\}$$

Punto fisso nella semantica call-by-name

Abbiamo quindi trovato un modo per esprimere funzioni ricorsive nel λ-calcolo, secondo la semantica call-by-name. Questo meccanismo funziona anche con la semantica call-by-value? Proviamo a vedere:

$$\begin{array}{lcl} \texttt{rec}f.B & \stackrel{\text{def}}{=} & \texttt{fix}(\lambda f.\lambda x.B) \\ & \to^* & (\lambda f.\lambda x.B)(\texttt{fix}(\lambda f.\lambda x.B)) \\ & \to^* & (\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\texttt{fix}(\lambda f.\lambda x.B)) \\ & \to^* & (\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\lambda f.\lambda x.B)(\texttt{fix}(\lambda f.\lambda x.B)) \\ & \to^* & \ldots \text{ all'infinito!} \end{array}$$

Dobbiamo quindi fermare l'espansione indefinita di $\texttt{fix}(\lambda f.\lambda x.B)$.

Redifiniamo quindi il combinatore di punto fisso come:

Call-by-name fixed-point combinator

$$A \stackrel{\text{def}}{=} \lambda x.\lambda y.y(xxy) \quad \texttt{fix} \stackrel{\text{def}}{=} AA \quad \texttt{fix}\ M \to^* M(\texttt{fix}\ M)$$

Call-by-value fixed-point combinator

$$A_v \stackrel{\text{def}}{=} \lambda x.\lambda y.y(\lambda z.(xxy)z) \quad \texttt{fix}_v \stackrel{\text{def}}{=} A_v A_v \quad \texttt{fix}_v M \to^* M(\lambda z.(\texttt{fix}_v M)z)$$

Call-by-value nelle funzioni ricorsive

Sia $\texttt{rec}_v f.B$ un'abbreviazione per $\texttt{fix}_v(\lambda f.\lambda x.B)$, allora

$$\texttt{rec}_v f.B \to^* \lambda x.B\{\lambda z.(\texttt{rec}_v f.B)z/f\}$$

Notiamo che non diverge più.

4.10 Punto fisso nel linguaggio funzionale

Dobbiamo adottare uno dei combinatori di punto fisso precedenti? No, nessuno dei combinatori precedenti è ben tipizzato per il teorema di Normalizzazione.

Sintassi

$$e \ ::= \ \ldots \mid \texttt{fix}\ \text{e}$$

Tipizzazione

$$\text{(T-Fix)}\ \frac{\Gamma \vdash e : (T_1 \to T_2) \to (T_1 \to T_2)}{\Gamma \vdash \texttt{fix}\ e : T_1 \to T_2}$$

Semantica

$$\text{(Fix-cbn)}\ \frac{-}{\texttt{fix}.e \to e(\texttt{fix}.e)} \qquad \text{(Fix-cbv)}\ \frac{e \equiv \texttt{fn}\ f : T_1 \to T_2 \implies e_2}{\texttt{fix}\ e \to e(\texttt{fn}\ x : T_1 \implies (\texttt{fix}.e)x)}$$

4.10.1 La regola (*Fix-cbv*) non funziona per la semantica call-by-value

Se utilizziamo la regola (*Fix-cbv*) per la semantica call-by-value, otteniamo:

$$\begin{aligned} \texttt{fact}\ 1 &\to \texttt{Fact(fact)}1 \\ &\to \texttt{Fact(Fact(fact))}1 \\ &\to \ldots \\ &\to \texttt{Fact(Fact(Fact(}\ldots\texttt{fact)))}1 \end{aligned}$$

La ricorsione call-by-value ha bisogno di un meccanismo per interrompere la valutazione della iterazione successiva. Ecco perché abbiamo definito una regola diversa (`Fix-cbv`) da utilizzare nella semantica call-by-value.

4.10.2 Decodifica del `while`

Avendo definito la ricorsione come primitiva, potremmo rimuovere il costrutto `while` se lo volessimo:

$$W \stackrel{\text{def}}{=} \texttt{fn w :}\ \ \texttt{unit} \to \texttt{unit.fn}\ y : \texttt{unit}\ \text{se}\ e_1\ \text{allora}\ (e_2; (w\ \texttt{skip}))\ \text{altrimenti}\ \texttt{skip}$$

Per w e y non presenti in $\texttt{fv}(e_1) \cup \texttt{fv}(e_2)$.

Allora

$$\texttt{while}\ e_1\ \texttt{do}\ e_2 \to \texttt{fix}\ W\ \texttt{skip}$$

e l'operatore `while` non è più primitivo.

Capitolo 5

Dati e store mutabili

Sino ad ora abbiamo considerato solo tipi di dati molto semplici e di base: `int`, `bool`, `unit` e funzioni su di essi. Ora esploreremo dati più strutturati, mantenendoli nella forma più semplice possibile, e rivisiteremo la semantica dello storage mutabile. Iniziamo con due tipi di dati strutturati di base: il tipo prodotto e il tipo somma. Il tipo prodotto $T1 \times T2$ ci permette di raggruppare insieme valori di tipo $T1$ e $T2$. In `C` ciò è realizzato con le strutture, mentre in `Java` si può usare una classe. Il tipo somma $T1 + T2$ consente di formare un'unione disgiunta, con un valore del tipo somma che può essere un valore di tipo $T1$ o un valore di tipo $T2$. In C ciò si realizza con l'`union`, mentre in `Java` una classe può implementare più interfacce (*sebbene possa estenderne solo una*).

Nella maggior parte dei linguaggi queste caratteristiche compaiono in forme più complesse: record con etichette invece che semplici prodotti, o varianti etichettate, o in `ML` tipi di dati con costruttori denominati, anziché semplici somme.

5.1 Prodotto

Estendiamo la grammatica con le espressioni e i tipi prodotto:

$$e \ ::= \quad \ldots (e_1, e_2) \mid \#1e \mid \#2e$$

$$T \ ::= \quad \ldots T_1 * T_2$$

Scelte progettuali per semplicità:

- abbiamo sia `int * (bool * unit)` che `(int * bool) * unit`, ma non abbiamo `int * bool * unit`;
- abbiamo le proiezioni #1 e #2, non il pattern matching;
- non abbiamo $\#e\ e'$ (*non può essere tipizzato*).

5.1.1 Tipizzazione del prodotto

$$\text{(pair)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2}$$

$$\text{(proj1)}\ \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#1\ e : T_1}$$

$$\text{(proj2)}\ \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#2\ e : T_2}$$

5.1.2 Semantica operazionale del prodotto

Estendiamo i possibili valori come segue:

$$v \ ::= \quad \ldots \mid (v_1, v_2)$$

e le regole di valutazione:

$$\text{(pair1)}\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle (e_1, e_2), s\rangle \to \langle (e_1', e_2), s'\rangle} \qquad \text{(pair2)}\ \frac{\langle e_2, s\rangle \to \langle e_2', s'\rangle}{\langle (v, e_2), s\rangle \to \langle (v, e_2'), s'\rangle}$$

$$\text{(proj1)}\ \frac{-}{\langle \#1\ (v_1, v_2), s\rangle \to \langle v_1, s\rangle} \qquad \text{(proj2)}\ \frac{-}{\langle \#2\ (v_1, v_2), s\rangle \to \langle v_2, s\rangle}$$

$$\text{(proj3)}\ \frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle \#1\ e, s\rangle \to \langle \#1\ e', s'\rangle} \qquad \text{(proj4)}\ \frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle \#2\ e, s\rangle \to \langle \#2\ e', s'\rangle}$$

Abbiamo scelto la valutazione da sinistra a destra per mantenere consistenza.

5.2 Records

Una generalizzazione del prodotto è il record, dove ogni campo ha associato un'etichetta.

$$\textit{Labels } lab \in \mathbb{LAB} \textit{ per un insieme } \mathbb{LAB} = \{q, p, \ldots\}$$

Estendiamo nuovamente la sintassi delle espressioni e dei tipi:

$$e \ ::= \quad \ldots \mid \{lab_1 = e_1, \ldots, lab_k = e_k\} \mid \#lab\ e$$

$$T \ ::= \quad \ldots \mid \{lab_1 : T_1, \ldots, lab_k : T_k\}$$

Dove in ogni record le etichette sono distinte, e non ci sono campi duplicati.

5.2.1 Tipizzazione dei record

$$\text{(record)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_k : T_k}{\Gamma \vdash \{lab_1 = e_1, \ldots, lab_k = e_k\} : \{lab_1 : T_1, \ldots, lab_k : T_k\}}$$

$$\text{(recordproj)}\ \frac{\Gamma \vdash e : \{lab_1 : T_1, \ldots, lab_k : T_k\}}{\Gamma \vdash \#lab\ e : T_i}$$

Qui l'ordine dei campi conta, per esempio:

$$(\texttt{fn}\ x : \{l_1 : \texttt{int}, l_2 : \texttt{bool}\} \implies x)\{l_2 = \texttt{true}, l_1 = 17\}$$

È mal tipato.

5.2.2 Semantica operazionale dei record

Estendiamo nuovamente la grammatica dei valori come segue:

$$v \ ::= \quad \ldots \mid \{lab_1 = v_1, \ldots, lab_k = v_k\}$$

E la semantica operazionale:

$$\text{(record1)}\ \frac{\langle e_i, s\rangle \to \langle e_i', s'\rangle}{\langle \{lab_1 = e_1, \ldots, lab_k = e_k\}, s\rangle \to \langle \{lab_1 = e_1', \ldots, lab_k = e_k\}, s'\rangle}$$

$$\text{(record2)}\ \frac{-}{\langle \{lab_1 = v_1, \ldots, lab_k = v_k\}, s\rangle \to \langle v_i, s\rangle}$$

$$\text{(record3)}\ \frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle \#lab\ e, s\rangle \to \langle \#lab\ e', s'\rangle}$$

5.3 Store mutabili

La maggior parte dei linguaggi hanno alcuni tipi di store mutabili. Ci sono due scelte principali:

1. Quello che abbiamo visto fino ad ora nel nostro linguaggio è:

   $$e \ ::= \quad \ldots \mid l := e \mid !l \mid x$$

   - le **locazioni** memorizzano valori mutabili: utilizziamo il costrutto di assegnazione per modificare il valore associato a una locazione.
   - le **variabili** si riferiscono a un valore precedentemente calcolato: una volta che associamo un valore a una variabile, non possiamo più cambiarlo.
   - la **deferenziazione** è esplicita solo per le locazioni.

     $$\texttt{fn}\ x : \texttt{int} \implies l := !l + x; \ldots$$

2. In altri linguaggi, come `C` e `Java`, le variabili possono riferirsi a valori precedentemente calcolati ed è possibile sovrascrivere tale valore. Inoltre la deferenziazione è implicita. La funzione precedentemente descritta in `Java` è:

   $$\texttt{void}\ foo(x : \texttt{int})\{l := l + x; \ldots\}$$

Nel nostro linguaggio ci fermiamo alla prima scelta.

5.3.1 Estensione dello store

Di seguito superiamo alcune limitazioni sui riferimenti del nostro linguaggio. In particolare, ricordiamo che al momento possiamo solo memorizzare valori interi, non possiamo creare nuove posizioni (*sono staticamente determinate*), non possiamo scrivere funzioni che agiscano su posizioni, come ad esempio

$$\texttt{fn}\ l : \texttt{intref} \implies !l$$

Estendiamo la sintassi e i tipi per superare tali limitazioni:

$$e \ ::= \ \ldots \mid \cancel{l := e} \mid \cancel{M} \mid e_1 := e_2 \mid !e \mid \texttt{ref}\ \ e \mid l$$

$$T \ ::= \ \ldots \mid \texttt{ref}\ \ \mathrm{T}$$

$$T_{loc} \ ::= \ \cancel{\texttt{intref}} \mid \texttt{ref}\ \ T$$

5.3.2 Tipizzazione dei riferimenti

$$(\text{ref})\ \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{ref}\ \ e : \texttt{ref}\ \ T}$$

$$(\text{assign})\ \frac{\Gamma \vdash e_1 : \texttt{ref}\ \ T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1 := e_2) : \texttt{unit}}$$

$$(\text{deref})\ \frac{\Gamma \vdash e : \texttt{ref}\ \ T}{\Gamma \vdash !e : T}$$

$$(\text{intref})\ \frac{-}{\Gamma \vdash l : \texttt{ref}\ \ T}\ \Gamma(l) = \texttt{ref}\ \ T$$

5.3.3 Semantica operazionale dei riferimenti

Una locazione è un valore:

$$v \ ::= \ \ldots \mid l$$

Fino ad ora, lo store s era una mappa parziale finita da $\mathbb{L}$ a $\mathbb{Z}$. Da ora in poi:

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

Vediamo le regole della semantica:

$$(\text{ref1})\ \frac{-}{\langle \texttt{ref}\ \ v, s\rangle \to \langle l, s[l \mapsto v]\rangle}\ l \notin dom(s)$$

$$(\text{ref2})\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle \texttt{ref}\ \ e, s\rangle \to \langle \texttt{ref}\ \ e', s'\rangle}$$

Notiamo che la regola (ref1) rappresenta l'**allocazione dinamica della memoria**.

$$(\text{deref1})\ \frac{-}{\langle !l, s\rangle \to \langle s(l), s\rangle}\ se\ l \in dom(s)\ \ e\ \ s(l) = v$$

$$\text{(deref2)}\ \frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle !e, s\rangle \to \langle !e', s'\rangle}$$

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

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

$$\text{(assign3)}\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1 := e_2, s\rangle \to \langle e_1' := e_2, s'\rangle}$$

5.3.4 Cambiamenti avvenuti

Un'espressione della forma $ref(v)$ deve fare qualcosa durante l'esecuzione: dovrebbe restituire una nuova (*fresh*) posizione associata al valore v.

Le funzioni possono astrarre sulle locazioni: `fn` x : `ref` $T \Rightarrow !x$. Quando i programmi iniziano, non hanno locazioni: devono creare nuove locazioni durante l'esecuzione. La tipizzazione e la semantica operazionale consentono alle locazioni di contenere altre locazioni, ad esempio $ref(ref3)$. In questa semantica, la proprietà del determinismo è persa per una ragione tecnica: le nuove posizioni vengono scelte in modo arbitrario. Per ripristinare il determinismo, dovremmo lavorare "fino alla alpha conversion per le posizioni". All'interno del nostro linguaggio non è consentito eseguire operazioni aritmetiche sulle posizioni, solo assegnazioni (*che possono essere fatte in* `C` *ma non in* `Java`) o testare se una posizione è più grande di un'altra. La nostra memoria cresce durante il calcolo; in un linguaggio di programmazione reale avremmo bisogno di un raccoglitore di rifiuti.

5.3.5 Controllo dei tipi

Prima di introdurre i riferimenti nelle nostre proprietà dei tipi, usiamo la condizione:

$$\textit{dom}(\Gamma) \subseteq \textit{dom}(s)$$

Per esprimere che tutte le locazioni menzionate nel contesto sono presenti nello store.

Ora, con l'introduzione dei riferimenti, abbiamo bisogno di una nuova condizione:

$$\textit{per ogni } l \in \textit{dom}(s),\ \textit{abbiamo che } s(l) \textit{ è tipabile}$$

Ciò significa che $s(l)$ può contenere funzioni o altri riferimenti ad altre locazioni.

5.3.6 Proprietà dei tipi

Store ben tipato

Scriviamo $\Gamma \vdash s$ se:

- $\textit{dom}(\Gamma) = \textit{dom}(s)e$
- per ogni $l \in \textit{dom}(s)$, se $\Gamma(l) =$ `ref` T allora $\Gamma \vdash s(l) : T$.

Progress (*riformulata*)

Se e è chiuso e $\Gamma \vdash e : T$ e $\Gamma \vdash s$ allora:

- o e è un valore, oppure
- esistono e', s' tale che $\langle e, s\rangle \to \langle e', s'\rangle$

Preservazione dei tipi

Se e è chiuso e $\Gamma \vdash e : T$ e $\Gamma \vdash s$ e $\langle e, s\rangle \to \langle e', s'\rangle$ allora e' è chiuso per qualche Γ' con un dominio disgiunto a di Γ, abbiamo quindi $\Gamma', \Gamma \vdash e' : T$ e $\Gamma', \Gamma \vdash s'$.

Capitolo 6

Subtyping

Il subtyping è un concetto che rende facile la vita del programmatore, facendo si che possa riutilizzare il codice. Prende una particolare valenza nella programmazione ad oggetti, rendendo meno rigido il type system, consentendo di compilare programmi che a runtime non darebbero problemi nonostante non siano tipabili.

6.1 Polimorfismo

- **Polimorfismo ad-hoc**: si tratta del classico overloading, data per esempio dalla possibilità di sommare due interi o due float.
- **Polimorfismo parametrico**: per esempio, una funzione
- che prende in input un argo,mento di tipo `list` α e la computa per una parametrica lunghezza.
- **Polimorfismo di subtyping**: un tipo può essere usato al posto di un altro, e noi ci focalizzeremo su questo.

Ridefiniamo le regole di subtyping per il nostro linguaggio funzionale:

$$(\text{fun})\ \frac{\Gamma, x, T \vdash e : T'}{\Gamma \vdash \lambda x.e : T \to T'}$$

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

Il type system presentato introduce una funzione rappresentata come:

$$\Gamma \vdash (\texttt{fn}\ x : \texttt{left :}\ \ \texttt{int} \implies \#\texttt{left}\ x) : \{\texttt{left :}\ \ \texttt{int}\} \to \texttt{int}$$

Questa funzione accetta un argomento x con un campo `left` di tipo `int` e restituisce il valore associato a `left`, dichiarato come $\#\texttt{left}\ x$, producendo un risultato di tipo `int`. Tuttavia,

è evidente che non è possibile applicare la stessa funzione a un argomento x che ha un tipo diverso da $\{\texttt{left : int}\}$.

Nel secondo esempio, si cerca di applicare la stessa funzione a un argomento di tipo

$$\{\texttt{left = 3,right = 5}\}$$

che è chiaramente incompatibile con il tipo richiesto dalla funzione. Questo viene espresso come:

$$\Gamma \not\vdash (\texttt{fn } x : \texttt{left : int} \implies \#\texttt{left } x) : \{\texttt{left = 3, right = 5}\}$$

Per gestire casi in cui si desidera consentire l'utilizzo di tipi più specifici in contesti che richiedono tipi più generali, viene introdotto il concetto di subtyping. La relazione di subtyping $<:$ viene definita in modo che un tipo possa essere considerato come un sottotipo di un altro. Nell'esempio fornito, il tipo $\{\texttt{left : int, right : int}\}$ è un sottotipo di $\{\texttt{left : int}\}$, che a sua volta è un sottotipo di $\{\}$.

$$\{\texttt{left : int, right : int}\} <: \{\texttt{left : int}\} <: \{\}$$

Tuttavia, l'introduzione di subtyping può portare alla perdita dell'unicità dei tipi. La regola di subtyping (`sub`) è aggiunta al sistema di tipi per consentire la sostituzione di un tipo con uno dei suoi sottotipi, ma ciò può portare a ambiguità nelle assegnazioni di tipo, poiché un'espressione potrebbe avere più tipi validi.

In breve, mentre il subtyping offre maggiore flessibilità nell'uso dei tipi, è importante considerare l'effetto sulla chiarezza e sull'unicità dei tipi nel contesto del sistema specifico che si sta definendo.

$$\text{(sub)}\ \frac{\Gamma \vdash e : T \qquad T <: T'}{\Gamma \vdash e : T}$$

6.1.1 Il sottotipo relazione $T <: T'$

Si tratta di una relazione riflessiva e transitiva.

$$\text{(s-refl)}\ \frac{-}{T <: T}$$

$$\text{(s-trans)}\ \frac{T <: T' \qquad T' <: T''}{T <: T''}$$

In accordo con il riordino dei campi:

$$\text{(rec-perm)}\ \frac{\pi \text{ una permutazione di } 1, 2, \ldots, k}{\{p_1 : T'_1, \ldots, p_n : T_k\} <: \{p_{\pi(1)} : T_{\pi(1)}, \ldots, p_{\pi(k)} : T_{\pi(k)}\}}$$

La relazione di subtyping sui record è antisimmetrica: un preordine e non un ordine parziale. Dimenticandoci dei campi a destra otteniamo:

$$\text{(rec-width)}\ \frac{-}{\{p_1 : T_1, \ldots, p_k : T_k, p_{k+1} : T_{k+1}, \ldots, p_z : T_z\} <: \{p_1 : T_1, \ldots, p_k : T_k\}}$$

Quindi se volessimo riordinare prima, possiamo dimenticarci dei campi e in accordo con il riordino dei campi otteniamo:

$$\text{(rec-perm)}\ \frac{T_1 <: T_1' \ldots T_k <: T_k'}{\{p_1 : T_1, \ldots, p_k : T_k\} <: \{p_1 : T_1', \ldots, p_k : T_k'\}}$$

Si dice infatti che il subtyping sui record è **covariante**.

Per quanto riguarda il sistema di tipi precedentemente discusso, l'introduzione del sottotipaggio sui record aggiunge ulteriore flessibilità nel trattamento dei tipi di dati strutturati.

6.2 Subtyping e funzioni

Contesto in cui si aspetta un certo algoritmo. Potrei avere una funzione che si presta a funzionare in quel contesto, come tipo argomento vuole di meno rispetto a quello che gli passo. Il subtyping è un modo per dire che una funzione che si aspetta un tipo, può essere usata con un tipo più generale.

La regola di sottotipo è la seguente:

$$\text{(fun-sub)}\ \frac{T_1 :> T_1' \qquad T_2 <: T_2'}{T_1 \to T_2 <: T_1' \to T_2'}$$

Diremo che il sottotipaggio sulle funzioni è **controvariante** a sinistra della freccia e **covariante** a destra della freccia.

Così, se $f : T_1 \to T_2$ allora possiamo usare f in ogni contesto dove forniamo ad f un qualsiasi argomento di tipo T_1' dove $T_1 <: T_1'$, e usiamo il risultato di f in ogni contesto dove ci aspettiamo un risultato di tipo T_2' dove $T_2 <: T_2'$.

Per istanza, definiamo f in un costruttore `let`:

$$\texttt{let}\ \ f : T = \texttt{fn}\ \ x : \{p : \texttt{int}\} \to \{a = \#p\, x, b = 28\}\ \ \texttt{in}\ \ e$$

allora quando tipiamo e dobbiamo usare il tipe environment Γ tale che:

$$\Gamma(f) = T = \{p : \texttt{int}\} \to \{a : \texttt{int}, b : \texttt{int}\}$$

Dalle regole di sottotipaggio usiamo f in e tale che abbia uno dei seguenti tipi:

$$\begin{array}{lcl} \{p : \texttt{int}\} & \to & \{a : \texttt{int}\} \\ \{p : \texttt{int}, q : \texttt{int}\} & \to & \{a : \texttt{int}, b : \texttt{int}\} \\ \{p : \texttt{int}, q : \texttt{int}\} & \to & \{a : \texttt{int}\} \end{array}$$

perché:

$$\begin{array}{lcl} \{p : \texttt{int}\} & <: & \{p : \texttt{int}, q : \texttt{int}\} \\ \{a : \texttt{int}, b : \texttt{int}\} & <: & \{a : \texttt{int}\} \end{array}$$

D'altro canto, nel programma:

$$\texttt{let}\ \ f : \hat{T} = \texttt{fn}\ \ x : \{p : \texttt{int}, q : \texttt{int}\} \to \{a = (\#p\, x) + (\#q\, x)\}\ \ \texttt{in}\ \ e$$

quando tipiamo e dobbiamo usare il tipe environment Γ tale che:

$$\Gamma(f) = \hat{T} = \{p : \texttt{int}, q : \texttt{int}\} \to \{a : \texttt{int}\}$$

Dalle regole di sottotipaggio usiamo f in e tale che abbiamo come tipo:

$$\{p : \texttt{int}, q : \texttt{int}\} \to \{a : \texttt{int}\}$$

Tuttavia, non possiamo usare f in e con tipi:

$$\begin{array}{lcl} \{p : \texttt{int}\} & \to & T \quad \textit{per ogni}\ \Gamma\ e\ T \\ T & \to & \{a : \texttt{int}\} \quad \textit{per ogni}\ \Gamma\ e\ T \end{array}$$

6.3 Subtyping e prodotto e somma

Il subtyping sul prodotto e sulla somma è controvariante sia a sinistra che a destra, ovvero per entrambe le componenti.

$$\text{(prod-sub)}\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

$$\text{(sum-sub)}\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 + T_2 <: T_1' + T_2'}$$

Per quanto riguarda le referenze, non introdurremmo le regole di sottotipaggio per evitare inconsistenza durante la tipizzazione.

Cosa cambia realmente con il subtyping? Il subtyping è un modo per affermare che un tipo può essere impiegato in un contesto dove è atteso un tipo più generale. La semantica rimane invariata, dato che non abbiamo modificato la grammatica delle espressioni. Le proprietà di preservazione del tipo e di progressione restano valide, mentre l'implementazione varia. L'inferenza del tipo è ora più sottile, poiché le regole di tipizzazione non sono dirette dalla sintassi. Ottenere una buona implementazione a runtime è anche più complicato, specialmente con la riordinazione dei campi.

6.4 Down-cast

La regola di subsumption (*sub*) permette l'up-casting in qualsiasi momento: se $T <: T'$ qualsiasi espressione di tipo T può essere usata in qualsiasi contesto dove è attesa un'espressione di tipo T'. Per quanto riguarda il down-casting, ovvero il casting da un tipo più generale ad uno più specifico, supponiamo di aggiungere alla grammatica il seguente costrutto:

$$e \ ::= \quad \ldots \mid (T)e$$

con la seguente regola di tipizzazione:

$$\text{(down-cast)}\ \frac{\Gamma \vdash e : T' \qquad T <: T'}{\Gamma \vdash (T)e : T}$$

In questo è possibile controllare il tipo $(T)e$ solo dinamicamente, perché la correttezza del down-casting dipende dal "vero tipo" a runtime di e.

Vediamo un esempio:

$$\{\texttt{left} : \texttt{int}, \texttt{right} : \texttt{int}\} <: \{\texttt{left} : \texttt{int}\} <: \{\}$$

Supponiamo di disporre del seguente frammento di codice:

$$l :=\!!m; e$$

Dove a runtime, la locazione m ci sarà uno store con l'espressione di uno dei tre tipi menzionati sopra. Ora, possiamo controllare staticamente il tipo della seguente espressione?

$$\{(\texttt{left} : \texttt{int})\}!l$$

Ovviamente no, perché non sappiamo se $!l$ a runtime ritornerà un valore di tipo $\{\texttt{left} : \texttt{int}, \texttt{right} : \texttt{int}\}$, però $\{\texttt{left} : \texttt{int}\} \not<: \{\texttt{left} : \texttt{int}, \texttt{right} : \texttt{int}\}$. Concludiamo che il down-casting può eseguire il controllo solo a runtime.

Capitolo 7

Equivalenza semantica

La semantica formale di un linguaggio di programmazione ci autorizza a ragionare su proprietà di programmi, basati sul loro linguaggio di programmazione.

Intuizione

Due programmi P_1 e P_2 si dicono **semanticamente equivalenti**, $P_1 \simeq P_2$, se entrambi possono essere rimpiazzati l'uno con l'altro in qualsiasi contesto senza alterare il comportamento del programma.

Con una buona equivalenza semantica possiamo:

- si possa capire cosa fa un programma;
- dimostrare se una particolare espressione (*ad esempio, un algoritmo efficiente*) è equivalente ad un'altra; questa operazione è detta **verifica di programmi**;
- dimostrare che le ottimizzazioni di un compilatore non alterano il comportamento del programma (*soundsness*);
- capire le differenze semantiche tra programmi.

Esempio

Possiamo dire che:

$$(l := 0; 4) \simeq (l := 1; 3+!l)?$$

I due frammenti producono lo stesso risultato a partire da uno stato iniziale. Il fatto è che non possiamo rimpiazzare arbitrariamente i due frammenti di codice in ogni contesto, ma solo in alcuni. Vediamo perché:

$$C[\cdot] \stackrel{\text{def}}{=\!=} C[\cdot]+!l$$

allora:

$$C(l := 0; 4) \not\simeq C[l := 1; 3+!l]$$

$$(l := 0; 4)+!l \not\simeq (l := 1; 3+!l)+!l$$

Di fatto, $C[l := 0; 4]$ ritorna 4 mentre $C[l := 1; 3+!l]$ ritorna 5.

7.1 Il significato di buona equivalenza semantica

Cosa significa per $\simeq$ essere "buona"?

- I programmi che risultano in valori osservabilmente diversi (*partento da un certo store iniziale*) non devono essere equivalenti.

$$\exists s, s_1, s_2, v_1, v_2 \quad . \quad \langle e, s\rangle \to^* \langle v_1, s_1\rangle \quad \wedge \quad \langle e_2, s\rangle \to^* \langle v_2, s_2\rangle \quad \wedge \quad v_1 \neq v_2 \implies e_1 \not\simeq e_2$$

- I programmi che terminano non devono essere equivalenti a quelli che non terminano.
- La relazione $\simeq$ deve essere una relazione di equivalenza:

$$e \simeq e \qquad e_1 \simeq e_2 \implies e_2 \simeq e_1 \qquad e_1 \simeq e_2 \simeq e_3 \implies e_1 \simeq e_3$$

- La relazione $\simeq$ deve essere una congruenza, cioè preservata
- dai contesti del programma: se $e_1 \simeq e_2$ allora per ogni contesto $C[\cdot]$ si ha che $C[e_1] \simeq C[e_2]$.
- La relazione $\simeq$ deve essere la più grande possibile, cioè mettere in relazione il maggior numero di programmi possibile.

7.2 Il contesto di un programma

Il contesto di un programma, ovvero $C[\cdot]$, è un programma non completamente definito. In gergo, si dice che $C[\cdot]$ denota un programma con un "buco" $[\cdot]$ che ha bisogno di essere riempito con un qualche programma P. Scrivendo $C[P]$ denotiamo un qualche programma ottenuto riempiendo il codice mancate in $C[\cdot]$ con il codice di P.

Ad esempio, nel linguaggio While, il contesto di un programma è definito dalla seguente grammatica:

$$\begin{array}{rcl} C[\cdot] \in \texttt{Cxt} & ::= & [\cdot] \mid C[\cdot]\ op\ e_2 \mid e_1\ op\ C[\cdot] \mid l := C[\cdot] \\ & \mid & \texttt{if}\ C[\cdot]\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{if}\ e_1\ \texttt{then}\ C[\cdot]\ \texttt{else}\ e_3 \\ & \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ C[\cdot] \mid C[\cdot]\text{;e\_2} \mid \text{e\_1;}C[\cdot] \\ & \mid & \texttt{while}\ C[\cdot]\ \texttt{do}\ \text{e\_2} \mid \texttt{while}\ e_1\ \texttt{do}\ C[\cdot] \end{array}$$

Per esempio, se $C[\cdot]$ è nel contesto $\texttt{while}\,!l = 0\,\texttt{do}\,[\cdot]$, allora $C[l :=!l + 1]$ è $\texttt{while}\,!l = 0\,\texttt{do}\,l := !l + 1$.

7.2.1 Congruenza rispetto ai contesti

È fondamentale che l'equivalenza di programma sia una congruenza rispetto ai contesti. Ciò significa che, se $e_1 \simeq e_2$, allora $C[e_1] \simeq C[e_2]$ per ogni contesto $C[\cdot]$. Immaginiamo di avere un

ampio programma, `Sys`, che controlla un vasto sistema e include un sottoprogramma P. Potremmo esprimere `Sys` come $\texttt{Sys} \stackrel{\text{def}}{=} C[P]$, per un contesto appropriato $C[\cdot]$, e ipotizziamo che ci venga chiesto di sviluppare una versione ottimizzata di P, chiamata `Pfast`. Come possiamo assicurarci che il comportamento dell'intero programma resti inalterato sostituendo il sottoprogramma P con `Pfast`? Dovremmo verificare che $C[\texttt{Pfast}] \simeq C[P]$, ma i due sistemi potrebbero essere molto estesi. Tuttavia, se l'uguaglianza è una **congruenza**, allora è sufficiente dimostrare che i due sottoprogrammi sono equivalenti, ovvero che $\texttt{Pfast} \simeq P$. Di conseguenza, l'uguaglianza dei sistemi interi, cioè $C[\texttt{Pfast}] \simeq C[P]$, ne consegue automaticamente.

7.2.2 Equivalenza semantica basata sulle tracce per il linguaggio While

Consideriamo il nostro linguaggio While tipato, senza funzioni:

Equivalenza delle tracce $\simeq_\Gamma^T$

Definiamo $e_1 \simeq_\Gamma^T e_2$ se e solo se, per tutti gli store s tali che $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(s)$, abbiamo $\Gamma \vdash e_1 : T$, $\Gamma \vdash e_2 : T$, e

- $\langle e1, s\rangle_* \langle v, s'\rangle \implies \langle e2, s\rangle_* \langle v, s'\rangle$,
- $\langle e2, s\rangle_* \langle v, s'\rangle \implies \langle e1, s\rangle_* \langle v, s'\rangle$.

Proprietà di congruenza

La relazione di equivalenza $\simeq_T^\Gamma$ gode della proprietà di congruenza perché, ogniqualvolta $e_1 \simeq_T^\Gamma e_2$, abbiamo che, per tutti i contesti C e i tipi T', se $\Gamma \vdash C[e_1] : T'$ e $\Gamma \vdash C[e_2] : T'$, allora $C[e_1] \simeq_{T'}^\Gamma C[e_2]$.

7.2.3 Equivalenza delle tracce su $\simeq_\Gamma^T$

Dato $e_1 \simeq_\Gamma^T e_2$ allora:

- Se una delle due configurazioni diverge da uno store s allora l'altra configurazione deve divergere con lo stesso store s.
- Dato uno store s se una delle due configurazioni converge, allora deve essere nello stesso valore v e nello stesso store s.

Supponiamo che dato uno store s, le due configurazioni $\langle e_1, s\rangle$ e $\langle e_2, s\rangle$ convergano, rispettivamente a $\langle v_1, s_1\rangle$ e $\langle v_2, s_2\rangle$, con $s_1(l) \neq s_2(l)$ per qualche l e v di tipo T. Allora il contesto distintivo potrebbe essere il seguente:

- Se $T = \texttt{unit}$ allora definiamo $C[\cdot] \stackrel{\text{def}}{=} [\cdot]; !\texttt{l}$
- Se $T = \texttt{bool}$ allora definiamo $C[\cdot] \stackrel{\text{def}}{=} \texttt{if } [\cdot] \texttt{ then } !\texttt{l } \texttt{ else } !\texttt{l}$
- Se $T = \texttt{int}$ allora definiamo $C[\cdot] \stackrel{\text{def}}{=} l_1 := [\cdot]; \texttt{!l}$

dove $\langle C[e_1], s\rangle \to_* \langle v_1, s_1'\rangle$ e $\langle C[e_2], s\rangle \to_* \langle v_2, s_2'\rangle$, con $v_1 \neq v_2$.

In queste definizioni, $C[\cdot]$ rappresenta un contesto distintivo che varia in base al tipo di T, e le espressioni $e1$ ed $e2$ sono inserite in questo contesto per testare la loro equivalenza.

Esempi

- $2 + 2 \simeq^{\text{int}}_{\Gamma} 4$ per ogni contesto Γ.
- $(l := 0; 4) \not\simeq^{\text{int}}_{\Gamma} (l := 1; 3{+}!l)$ per ogni contesto Γ.
- $(l :=!l + 1); (l :=!l - 1) \simeq^{\text{int}}_{\Gamma} (l :=!l)$ per ogni contesto $\Gamma \supseteq \{l : \texttt{intref}\}$.
- $(l :=!l{+}1); (k :=!j{+}1) \simeq^{\text{int}}_{\Gamma} (k :=!j{+}1); (l :=!l{+}1)$ per ogni contesto $\Gamma \supseteq \{l : \texttt{intref}, j : \texttt{intref}\}$.

7.3 Regole generali

Associatività del ;

$$e_1; (e_2; e_3) \simeq (e_1; e_2); e_3$$

Per ogni Γ, T, e_1, e_2 e e_3 tale che $\Gamma \vdash e_1 : \texttt{unit}$, $\Gamma \vdash e_2 : \texttt{unit}$ e $\Gamma \vdash e_3 : T$.

rimozione dello `skip`

- $e_2 \simeq^{T}_{\Gamma_2} \texttt{skip}; e_1$
- $e_1; \texttt{skip} \simeq^{T}_{\Gamma_1} e_1$

Per ogni $\Gamma_1, \Gamma_2, T, e_1$ e e_2 tali che $\Gamma_2 \vdash e_2 : T$, $\Gamma_1 \vdash e_1 : \texttt{unit}$.

`if true`

$$\texttt{if true then } e_1 \texttt{ else } e_2 \simeq^{T}_{\Gamma} e_1$$

Per ogni Γ, T, e_1 e e_2 tali che $\Gamma \vdash e_1 : T$ e $\Gamma \vdash e_2 : T$.

`if false`

$$\texttt{if false then } e_1 \texttt{ else } e_2 \simeq^{T}_{\Gamma} e_2$$

Per ogni Γ, T, e_1 e e_2 tali che $\Gamma \vdash e_1 : T$ e $\Gamma \vdash e_2 : T$.

Distributività dell'`if` rispetto all';

$$(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3); e \simeq^{T}_{\Gamma} (\texttt{if } e_1 \texttt{ then } e_2; e \texttt{ else } e_3; e)$$

per ogni Γ, T, e_1, e_2 e e_3 tali che $\Gamma \vdash e_1 : \texttt{bool}$, $\Gamma \vdash e_2 : \texttt{unit}$, $\Gamma \vdash e_3 : \texttt{unit}$ e $\Gamma \vdash e : T$.

Distributività dell'; rispetto all'`if`

$$e; (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \simeq^T_\Gamma (\texttt{if } e_1 \texttt{ then } e; e_2 \texttt{ else } e; e_3)$$

per ogni Γ, T, e_1, e_2 e e_3 tali che $\Gamma \vdash e_1 : \texttt{bool}$, $\Gamma \vdash e_2 : \texttt{unit}$, $\Gamma \vdash e_3 : \texttt{unit}$ e $\Gamma \vdash e : T$.

7.3.1 Regole errate

Sappiamo che:

$$(e; \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \not\simeq^T_\Gamma (\texttt{if } e_1 \texttt{ then } e; e_2 \texttt{ else } e; e_3)$$

Per esempio, consideriamo che:

- e sia $l := 1$
- e_1 sia $!l = 0$
- e_2 sia `skip`
- e_3 sia `while true do skip`

Allora, in qualsiasi store s, dove la locazione l è associata a 0, l'espressione a sinistra diverge, mentre quella a destra converge.

7.4 Approccio alla simulazione

Simulazione

Diciamo che e_1 è simulato da e_2, ovvero $e_1 \sqsubseteq e_2$, se e solo se:

- $\Gamma \vdash e_1 : T$ e $\Gamma \vdash e_2 : T$ per qualche Γ e T.
- Per ogni store s con $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(s)$, se $\langle e_1, s\rangle \to^* \langle e_1', s_1'\rangle$ allora esiste e_2' tale che $\langle e_2, s\rangle \to^* \langle e_2', s_2'\rangle$, con $e_1' \sqsubseteq^T_\Gamma e_2'$ e $s_1' = s_2'$.

Bisimulazione

Diciamo che e_1 è bisimile da e_2, ovvero $e_1 \approx e_2$ se e solo se:

- $\Gamma \vdash e_1 : T$ e $\Gamma \vdash e_2 : T$ per qualche Γ e T.
- Per ogni store s con $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(s)$, se $\langle e_1, s\rangle \to^* \langle e_1', s_1'\rangle$ allora esiste e_2' tale che $\langle e_2, s\rangle \to^* \langle e_2', s_2'\rangle$, con $e_1' \approx^T_\Gamma e_2'$ e $s_1' = s_2'$.
- Per ogni store s con $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(s)$, se $\langle e_2, s\rangle \to^* \langle e_2', s_2'\rangle$ allora esiste e_1' tale che $\langle e_1, s\rangle \to^* \langle e_1', s_1'\rangle$, con $e_1' \approx^T_\Gamma e_2'$ e $s_1' = s_2'$.

Capitolo 8

Concorrenza

8.1 Introduzione

Il fulcro della nostra ricerca finora è stata la semantica delle computazioni sequenziali. Tuttavia, è importante riconoscere che numerosi sistemi complessi operano in modalità non sequenziale. Questa osservazione è particolarmente pertinente nell'era dell'hardware intrinsecamente parallelo, che include:

- Sistemi con architetture multi-processore.
- Il multi-threading, sia in contesti di processori singoli che multipli.
- Sistemi distribuiti e macchine collegate in rete.
- Sistemi ciber-fisici integrati.
- Dispositivi nell'ambito dell'Internet of Things (IoT).

La concorrenza, in generale, può notevolmente migliorare le prestazioni di un programma, permettendo l'esecuzione parallela di task indipendenti su hardware multicore. Inoltre, favorisce lo sviluppo di interfacce utente dinamiche e reattive.

La complessità dei sistemi concorrenti si manifesta anche nell'espansione dello spazio degli stati. Con n thread, ognuno dei quali può assumere uno di due stati, il sistema può teoricamente raggiungere 2^n stati distinti. Oltre alla vastità dello spazio degli stati, emergono ulteriori sfide, come:

- La necessità di gestire l'accesso alle risorse condivise tramite meccanismi di mutua esclusione per prevenire situazioni di *deadlock* o *starvation*.
- La natura intrinsecamente non deterministica delle computazioni, che si accentua in assenza di meccanismi di sincronizzazione, poiché i diversi thread possono operare a velocità variabili.

- Problemi specifici della programmazione concorrente, tra cui le race condition, ovvero l'accesso contemporaneo a dati condivisi da parte di thread multipli, che può portare a comportamenti imprevedibili o errati.

In aggiunta a questi aspetti, i sistemi non sequenziali e distribuiti devono affrontare sfide quali:

- Fallimenti parziali (*di alcuni processi, dispositivi in una rete o dispositivi di storage persistente*); la necessità di meccanismi di transazione.
- Comunicazione tra ambienti diversi con risorse locali differenti (*ad esempio, diversi store locali o librerie*); la necessità di meccanismi di coerenza.
- Comunicazione tra domini amministrativi con fiducia parziale (*o, in effetti, assenza di fiducia*); la protezione contro attacchi informatici.
- La gestione della complessità contingente.

8.2 Composizione parallela

Consideriamo il seguente linguaggio concorrente:

$$
\begin{array}{lrcl}
\textit{Booleans} & b \in \mathbb{B} & = & \texttt{true} \mid \texttt{false} \\
\textit{Integers} & n \in \mathbb{N} & = & \{\ldots, \text{-1}, 0, 1, \ldots\} \\
\textit{Locations} & l \in \mathbb{L} & = & \{\text{l}, \text{l\_0}, \text{l\_1}, \ldots\} \\
\textit{Operations} & \textit{op} & ::= & + \mid \geq \\
\textit{Expressions} & e \in \textit{Exp} & ::= & n \mid b \mid e\ \texttt{op}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid & l := e \mid\ !l \mid \texttt{skip} \mid e; e \\
 & & \mid & \texttt{while}\ e\ \texttt{do}\ e \mid e \parallel e \\
\textit{Types} & T & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \texttt{proc} \\
 & T_{loc} & ::= & \texttt{intref}
\end{array}
$$

Il costrutto $e \parallel e$ è chiamato composizione parallela.

8.2.1 Le nostre scelte di design

Nel contesto della programmazione concorrente, è importante anche considerare le seguenti caratteristiche dei thread:

- I thread non restituiscono un valore.

- I thread sono anonimi, cioè non hanno un'identità.
- La terminazione di un thread non può essere direttamente osservata all'interno di un programma.
- I processi, in generale, sono costituiti da un pool di thread concorrenti.
- I thread non possono essere terminati esternamente.

Vediamo ora i cambiamenti apportati alla semantica operazionale.

$$\text{(T-sq1)}\ \frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \texttt{unit}}{\Gamma \vdash e_1; e_2 : \texttt{unit}}$$

$$\text{(T-sq2)}\ \frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \texttt{proc}}{\Gamma \vdash e_1; e_2 : \texttt{proc}}$$

$$\text{(T-par)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \parallel e_2 : \texttt{proc}}\ T_1, T_2 \in \{\texttt{unit, proc}\}$$

$$\text{(par-L)}\ \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1 \parallel e_2, s\rangle \to \langle e_1' \parallel e_2, s'\rangle}$$

$$\text{(par-R)}\ \frac{\langle e_2, s\rangle \to \langle e_2', s'\rangle}{\langle e_1 \parallel e_2, s\rangle \to \langle e_1 \parallel e_2', s'\rangle}$$

$$\text{(end-L)}\ \frac{-}{\langle \texttt{skip} \parallel e_2, s\rangle \to \langle e_2, s\rangle}$$

$$\text{(end-R)}\ \frac{-}{\langle e_1 \parallel \texttt{skip}, s\rangle \to \langle e_1, s\rangle}$$

Dove $\Gamma \vdash e : \texttt{unit}$ è essenzialmente multithread, mentre $\Gamma \vdash e : \texttt{proc}$ è essenzialmente single-thread.

Come in ogni linguaggio concorrente:

- I thread eseguono in modo asincrono: la semantica permette qualsiasi interleaving (intreccio) delle riduzioni dei thread.
- Tutti i thread possono leggere e scrivere nella memoria condivisa.

Di conseguenza, la proprietà di determinatezza (*Determinacy*) non vale. Ad esempio:

$$\langle l := 1 \parallel l := 2, \{l \mapsto 0\}\rangle \to \langle \text{skip} \parallel l := 2, \{l \mapsto 1\}\rangle \to \langle \text{skip} \parallel \text{skip}, \{l \mapsto 2\}\rangle \to \langle \text{skip}, \{l \mapsto 2\}\rangle$$

Ma anche:

$$\langle l := 1 \parallel l := 2, \{l \mapsto 0\}\rangle \to \langle l := 1 \parallel \text{skip}, \{l \mapsto 2\}\rangle \to \langle \text{skip} \parallel \text{skip}, \{l \mapsto 1\}\rangle \to \langle \text{skip}, \{l \mapsto 1\}\rangle$$

Nell'ambito della programmazione concorrente, è fondamentale sottolineare che sia le operazioni di *assegnamento* che di *dereferenziazione* sono considerate operazioni atomiche. Questo

significa che, in una data configurazione, lo store risultante può associare la locazione l unicamente ai valori 1 o 2, escludendo così la possibilità di combinazioni anomale di questi valori.

Tuttavia, quando si considera l'espressione $(l := e) \parallel e'$, emerge una complessità aggiuntiva. In questo contesto, i passi semantici necessari per valutare e ed e' possono essere sovrapposti o intrecciati. Di conseguenza, questo intreccio può generare sfide significative, come illustrato nell'esecuzione del programma $(l := 1+!l) \parallel (l := 7+!l)$. In questo scenario, si possono verificare delle condizioni *race conditions*, portando a un output potenzialmente inaspettato e non allineato con le intenzioni di ciascun thread.

Per esemplificare ulteriormente, consideriamo le seguenti tre configurazioni finali possibili per l'espressione $\langle (l := 1+!l) \parallel (l := 7+!l), \{l \mapsto 0\}\rangle$:

1. $\langle \text{skip}, \{l \mapsto 1\}\rangle$
2. $\langle \text{skip}, \{l \mapsto 7\}\rangle$
3. $\langle \text{skip}, \{l \mapsto 8\}\rangle$

È importante notare che le configurazioni (1) e (2) sono il risultato di interferenze durante l'esecuzione delle assegnazioni. Al contrario, solo la configurazione (3) rappresenta il risultato di una sequenza di operazioni coerente e correttamente pianificata.

La programmazione concorrente introduce una complessità notevole a causa della vasta gamma di possibili risultati. Infatti, l'esecuzione di programmi concorrenti può portare a una vera e propria esplosione combinatoria degli stati possibili. Questo fenomeno rende l'analisi e la rappresentazione di questi stati, per esempio attraverso diagrammi dello spazio degli stati, praticabile solo per esempi molto semplici.

È evidente, quindi, la necessità di sviluppare metodologie più efficaci per analizzare i programmi concorrenti.

Un aspetto cruciale da considerare è che, quasi certamente, come programmatore non si desidera che tutti i possibili esiti siano effettivamente realizzabili. Questo pone l'accento sulla necessità di sviluppare idiomi o costrutti di programmazione più raffinati, che consentano di limitare e controllare meglio le possibili evoluzioni dei programmi concorrenti. La scelta di tali strumenti e tecniche diventa quindi un punto fondamentale per garantire che il comportamento dei programmi concorrenti sia non solo efficace ma anche prevedibile e allineato alle intenzioni originali del programmatore.

Per gestire efficacemente la concorrenza nei programmi, è essenziale disporre di meccanismi per sincronizzare i thread, in modo da garantire la mutua esclusione per l'accesso ai dati condivisi.

Sebbene sia possibile affidarsi al supporto integrato fornito dallo scheduler, come i mutex o le variabili di condizione, o addirittura a operazioni a basso livello come *test-and-set* (*tas*) o *compare-and-set* (*cas*), ciò non toglie l'importanza di comprendere e sviluppare algoritmi di sincronizzazione intrinseci al linguaggio. Questo non solo aiuta a comprendere meglio i

principi fondamentali della concorrenza, ma offre anche una maggiore flessibilità e controllo nella gestione delle interazioni tra thread.

8.3 Aggiunta delle primitive di mutex nel linguaggio