Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Linguaggi di programmazione

Riassunto dei principali argomenti

Autore:
Davide Bianchi

Anno Accademico 2017/2018

Indice

1 Introduzione

Un linguaggio di programmazione è composto da:

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

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

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

2 Esempio di linguaggio basilare

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

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

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

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

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

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

2.1 Semantica big-step

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

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

La semantica big-step fornisce immediatamente il risultato, dando subito il valore finale dell'espressione che si sta valutando.

2.1.1 Esempio

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

Teorema 2.1 (Determinatezza per semantica big-step)**.** *$E \Downarrow m$ e $E \Downarrow n$ implica $m = n$.*

2.2 Semantica small-step

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

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

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

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

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

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

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

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

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

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

Teorema 2.2 (Determinatezza per semantica small-step)**.** *Definiamo:*

- ***strong determinacy:*** $E \rightarrowtriangle F$ *e* $E \rightarrowtriangle G$ *implica* $F = G$*;*
- ***weak determinacy:*** $E \rightarrowtriangle^* m$ *e* $E \rightarrowtriangle^* n$ *implica* $m = n$*;*

3 Linguaggio imperativo

Definiamo la sintassi di un semplice linguaggio imperativo:

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

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

3.1 Memoria

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

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

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

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

L'aggiornamento della memoria funziona come segue:

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

3.2 Sistemi di transizione

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

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

Per indicare un generale passo di semantica si usa la notazione

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

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

3.3 Semantica small-step su un linguaggio imperativo

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

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

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

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

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

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

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

3.4 Esecuzione di programmi e proprietà

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

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

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

Illustriamo inoltre due importanti proprietà:

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

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

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

3.5 Funzione di valutazione della semantica

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

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

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

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

3.6 Possibili varianti del linguaggio

Nel linguaggio illustrato possono essere introdotte anche diverse varianti.

3.6.1 Inversione dell'ordine di valutazione

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

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

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

3.6.2 Regole di assegnamento

Una piccola variante alla regola dell'assegnamento:

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

3.6.3 Inizializzazione della memoria

Possibili varianti a livello di inizializzazione della memoria potrebbero essere:

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

3.6.4 Valori memorizzabili

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

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

3.7 Type systems

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

- descrivere quando i programmi sono sensati;
- prevenire certi tipi di errore;
- strutturare i programmi;
- dare delle linee guida per la progettazione del linguaggio;

- dare informazioni utili per la fase di ottimizzazione da parte del compilatore;
- rinforzare alcune proprietà di *sicurezza* del programma.

Definiamo la funzione

$$\Gamma \vdash e : T$$

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

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

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

3.7.1 Regole di tipaggio

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

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

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

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

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

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

3.7.2 Proprietà di tipaggio

Teorema 3.3 (Progress)**.** *Se* $\Gamma \vdash e : T$ *e* $dom(\Gamma) \subseteq dom(s)$ *allora* e *è un valore oppure esiste una coppia* $\langle e', s' \rangle$ *tale che*

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

Teorema 3.4 (Type preservation)**.** *Se* $\Gamma \vdash e : T$ *e* $dom(\Gamma) \subseteq dom(s)$ *e* $\langle e, s \rangle \rightarrowtriangle \langle e', s' \rangle$ *allora si ha che* $\Gamma \vdash e' : T$ *e* $dom(\Gamma) \subseteq dom(s')$

Mettendo insieme le due proprietà sopra, si ottiene una nuova proprietà, esplicativa del fatto che programmi ben tipati non vanno mai in deadlock.

Teorema 3.5 (Safety)**.** *Se* $\Gamma \vdash e : T$*,* $dom(\Gamma) \subseteq dom(s)$ *e* $\langle e, s \rangle \rightarrowtriangle^* \langle e', s' \rangle$ *allora* e' *è un valore oppure esiste una coppia* $\langle e'', s'' \rangle$ *tale che* $\langle e', s' \rangle \rightarrowtriangle \langle e'', s'' \rangle$

Teorema 3.6 (Type inference)**.** *Dati* Γ*,* e*, può essere trovato il tipo* T *tale che* $\Gamma \vdash e : T$ *oppure può essere provato che* T *non esiste.*

Teorema 3.7 (Decidibilità del type-checking)**.** *Dati* Γ*,* e*,* T*, è decidibile* $\Gamma \vdash e : T$

Teorema 3.8 (Unicità del tipaggio)**.** *Se vale che* $\Gamma \vdash e : T$ *e* $\Gamma \vdash e : T'$ *allora* $T = T'$*.*

4 Forme di induzione

L'induzione è una tecnica formale che consente di provare delle proprietà su determinate categorie di oggetti, sfruttando la natura di questi oggetti. Esistono 3 tipi di induzione:

- matematica;
- strutturale;
- rule induction[1].

4.1 Induzione matematica

È la forma di induzione più semplice, consiste infatti nel dimostrare una proprietà $P(-)$ su numeri naturali procedendo nel modo seguente:

1. **Caso base:** provare che $P(0)$ è vera, usando qualche procedimento matematico;
2. **Caso induttivo:**
   (a) assumere che l'ipotesi induttiva valga, ovvero che valga $P(k)$;
   (b) dall'ipotesi induttiva dimmostrare che vale $P(k+1)$, usando qualche procedimento matematico.

Se i punti precedenti sono veri, allora $P(n)$ è vera per ogni numero naturale.

4.2 Induzione strutturale

4.2.1 Induzione strutturale su numeri naturali

Per dimostrare una proprietà P su numeri naturali basta applicare il seguente metodo:

- **Caso base:** dimostrare che vale $P(0)$;
- **Caso induttivo:** dimostrare che è vera $P(succ(K))$ assumendo come ipotesi induttiva che valga $P(K)$ per qualche $K \in \mathbb{N}$.

L'induzione strutturale consiste quindi nell'assumere che l'ipotesi induttiva valga per la *sottostruttura* di $succ(K)$.

4.2.2 Induzione strutturale su strutture complesse

Prendiamo come esempio la costruzione di alberi binari. Diamo la seguente grammatica per costruire gli alberi:

$$T ::= leaf \mid tree(T, T)$$

In tal caso partiamo col presupposto che:

- **Caso base:** una foglia sia un albero binario;
- **Caso induttivo:** se L e R sono alberi binari, allora lo è anche $tree(L, R)$.

[1] Appena avrò una traduzione valida la metterò.

4.3 Rule induction

L'idea di base della rule induction consiste nell'ignorare la struttura di ciò che si deriva per fare induzione sulla dimensione dell'albero di derivazione.

Per provare formalmente una proprietà $P(D)$ su una derivazione D, si procede come segue:

1. **Caso base:** dimostrare che $P(A)$ è vera, per ogni assioma A;
2. **Caso induttivo:** per ogni regola della forma

$$(\text{regola})\,\frac{h_1\ h_2\ ...\ h_n}{c}$$

si dimostra che ogni derivazione che termina con l'utilizzo di questa regola soddisfa la proprietà. Questa derivazione ha sottoderivazioni $D_1, D_2, ..., D_n$ che terminano con le ipotesi $h_1, h_2, ..., h_n$. Per ipotesi induttiva si assume che valga $P(D_i)$ con $1 \leqslant i \leqslant n$.

5 Aspetti funzionali

Estendiamo la sintassi data in 3 con:

$$\text{Variabili } x \in \mathbb{X}, \text{con } x = \{x, y, z, ...\}$$
$$\text{Espressioni } e ::= \ ... \mid \text{fn } x : T \Rightarrow e \mid ee \mid x$$

e i tipi con:

$$T \ ::= \ ... \mid T \to T$$

Assumiamo che:

- l'applicazione di funzioni ee è associativa a sinistra;
- i tipi delle funzioni sono associativi a destra;
- il corpo di fn si estende a sinistra quanto più è possibile;
- $fn\ x : unit \Rightarrow fn\ y : int \Rightarrow x; y$ è di tipo $unit \to int \to int$.

5.1 Variabili free e bound

Intuitivamente, diciamo che una variabile è *free* in e se x non occorre in nessun termine della forma `fn x:T` $\Rightarrow$ ...

Più formalmente definiamo le variabili free come una funzione:

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

dove $\mathbb{X}$ è l'insieme delle variabili. La funzione fv è definita come:

$$\begin{aligned}
fv(x) &\triangleq \{x\} \\
fv(\texttt{fn x:T} \Rightarrow e) &\triangleq fv(e) \backslash \{x\} \\
fv(e_1 e_2) = fv(e_1; e_2) &\triangleq fv(e_1) \cup fv(e_2) \\
fv(n) = fv(b) = fv(!l) = fv(\texttt{skip}) &\triangleq \varnothing \\
fv(e_1 \texttt{op} e_2) = fv(\texttt{ while } e_1 \texttt{ do } e_2) &\triangleq fv(e_1) \cup fv(e_2) \\
fv(l \texttt{:=} e) &\triangleq fv(e) \\
fv(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) &\triangleq fv(e_1) \cup fv(e_2) \cup fv(e_3)
\end{aligned}$$

Definizione 5.1. *Un'espressione e si dice **chiusa** se $fv(e) = \varnothing$.*

5.2 Alpha conversion

Nell'espressione `fn x : T` $\Rightarrow e$ la variabile x è *bound* in e. Diciamo che:

- x è il parametro formale della funzione, quindi ogni occorrenza di x che non è in una funzione annidata alla funzione attuale indica la stessa cosa;
- al di fuori della definizione della funzione, la variabile x non ha significato;

Basandoci su quanto detto nei due punti precedenti, possiamo concludere che il nome del parametro formale nella funzione **non modifica** il comportamento della funzione. Assumiamo quindi di poter *rimpiazzare il vincolo su x e tutte le occorrenze di x in una certa espressione e con una variabile **fresh** che non occorra da nessun'altra parte.*

5.3 Sostituzioni

5.3.1 Sostituzioni singole

Un perno della semantica delle funzioni è dato dalle sostituzioni, ovvero il rimpiazzo a runtime di un parametro attuale con un parametro formale. Una sostituzione è indicata con

$$e_2\{^{e_1}/_x\}$$

e indica la sostituzione di x con e_2 nell'espressione e_1, per tutte le occorrenze libere di x. Le sostituzioni funzionano come segue:

$$\begin{aligned}
n\{^{e}/_x\} &\triangleq n \\
b\{^{e}/_x\} &\triangleq b \\
\texttt{skip}\{^{e}/_x\} &\triangleq \texttt{skip} \\
x\{^{e}/_x\} &\triangleq e \\
y\{^{e}/_x\} &\triangleq y \\
\texttt{(fn x:T} \Rightarrow e_1\texttt{)}\{^{e}/_z\} &\triangleq \texttt{(fn x:T} \Rightarrow e_1\{^{e}/_z\}\texttt{)} \text{ se } x \notin fv(e) \\
\texttt{(fn x:T} \Rightarrow e_1\texttt{)}\{^{e}/_z\} &\triangleq \texttt{(fn x:T} \Rightarrow (e_1\{^{e}/_z\})\{^{e}/_z\}\texttt{)} \text{ se } x \in fv(e) \wedge y \text{ è fresh}^2 \\
\texttt{(fn x:T} \Rightarrow e_1\texttt{)}\{^{e}/_x\} &\triangleq \texttt{(fn x:T} \Rightarrow e_1\texttt{)} \\
\texttt{(e}_1\texttt{;e}_2\texttt{)}\{^{e}/_x\} &\triangleq (e_1\{^{e}/_x\}e_2\{^{e}/_x\})
\end{aligned}$$

Le altre espressioni dopo una sostituzione rimangono invariate per il semplice motivo che non sono funzioni, quindi non va sostituito alcun parametro.

5.3.2 Sostituzioni simultanee

Le sostituzioni possono essere implementate in modo da poter essere anche simultanee, ovvero con lo scopo di poter sostituire più variabili contemporaneamente. In generale, una sostituzione simultanea è una funzione parziale $\sigma : \mathbb{X} \to Exp$.

Sintatticamente viene indicato con $e\sigma$ l'espressione risultante dalla sostituzione simultanea di ogni $x \in dom(\sigma)$ con la corrispondente espressione $\sigma(x)$. La sostituzione dei parametri viene indicata con

$$\{^{e_1}/_{x_1}, ..., ^{e_k}/_{x_k}\}$$

5.4 Lambda calcolo

Il lambda calcolo è un linguaggio dove:

- ogni termine è una funzione;
- ogni termine può essere applicato ad un altro termine;
- le funzioni del linguaggio `fn x:T` $\Rightarrow e$ diventano funzioni del tipo $\lambda x : T.e$.

La sintassi del lambda-calcolo non tipato è:

$$M \in Lambda ::= \ x \mid \lambda x.M \mid MM$$

[2] Operando con alpha-conversion.

5.5 Applicazione di funzioni

Intuitivamente, nell'espressione $M_1 M_2$, per applicare M_2 a M_1 si procede prima risolvendo M_1 ad un termine del tipo $\lambda x.M$, poi si procede in base a una delle strategie di valutazione possibili:

- *call-by-value:* si valuta M_2 ad un valore v, poi si valuta $M\{^{v}/_{x}\}$;
- *call-by-name:* si valuta direttamente $M\{^{M_2}/_{x}\}$.

5.5.1 Semantica dell'applicazione di funzioni

Estendiamo l'insieme dei valori con

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

La semantica funziona come di seguito:

$$\text{App}\frac{M_1 \rightarrowtriangle M_1'}{M_1 M_2 \rightarrowtriangle M_1' M_2}$$

Inoltre, in modalità call-by-value:

$$\text{CBV-app1}\frac{\langle e_1, s\rangle \rightarrowtriangle \langle e_1', s'\rangle}{\langle e_1 e_2, s\rangle \rightarrowtriangle \langle e_1' e_2, s'\rangle} \quad \text{CBV-app2}\frac{\langle e_2, s\rangle \rightarrowtriangle \langle e_2', s'\rangle}{\langle v e_2, s\rangle \rightarrowtriangle \langle v e_2', s'\rangle} \quad \text{CBV-fn}\frac{-}{\langle(\text{fn } x : T \Rightarrow e)v, s\rangle \rightarrowtriangle \langle e\{^{v}/_{x}\}, s\rangle}$$

mentre in modalità call-by-name:

$$\text{CBN-app}\frac{\langle e_1, s\rangle \rightarrowtriangle \langle e_1', s'\rangle}{\langle e_1 e_2, s\rangle \rightarrowtriangle \langle e_1' e_2, s'\rangle} \quad \text{CBN-fn}\frac{-}{\langle(\text{fn } x : T \Rightarrow e)e_2, s\rangle \rightarrowtriangle \langle e\{^{e_2}/_{x}\}, s\rangle}$$

Introduciamo inoltre una nuova semantica che raccoglie le due precedenti:

$$\text{BETA-app1}\frac{\langle e_1, s\rangle \rightarrowtriangle \langle e_1', s'\rangle}{\langle e_1 e_2, s\rangle \rightarrowtriangle \langle e_1' e_2, s'\rangle} \quad \text{BETA-app2}\frac{\langle e_2, s\rangle \rightarrowtriangle \langle e_2', s'\rangle}{\langle e_1 e_2, s\rangle \rightarrowtriangle \langle e_1 e_2', s'\rangle}$$

$$\text{BETA-fn1}\frac{-}{\langle(\text{fn } x : T \Rightarrow e)e_2, s\rangle \rightarrowtriangle \langle e\{^{e_2}/_{x}\}, s\rangle} \quad \text{BETA-fn2}\frac{-}{\langle \text{fn } x : T \Rightarrow e, s\rangle \rightarrowtriangle \langle \text{fn } x : T \Rightarrow e', s\rangle}$$

5.6 Tipaggio di funzioni

Estendiamo l'insieme dei tipi come segue:

$$TypeEnv = \mathbb{L} \cup \mathbb{X} \rightharpoonup T_{loc} \cup T$$

tale che:

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

Definiamo le regole di tipaggio:

$$\text{var}\frac{-}{\Gamma \vdash x : T}\text{se } \Gamma(x) = T \quad \text{fn}\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \text{fn } x : T \Rightarrow e : T \to T'} \quad \text{app}\frac{\Gamma \vdash e_1 : T \to T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'}$$

Qui di seguito definiamo alcune proprietà del tipaggio:

Teorema 5.1 (Progress). *Se e è chiusa, $\Gamma \vdash e : T$ e $dom(\Gamma) \subseteq dom(s)$ allora e è un valore oppure esiste una configurazione $\langle e', s' \rangle$ tale $\langle e, s \rangle \rightarrowtriangle \langle e', s' \rangle$.*

Teorema 5.2 (Type preservation). *Se e è chiusa, $\Gamma \vdash e : T$, $dom(\Gamma) \subseteq dom(s)$ e $\langle e, s \rangle \rightarrowtriangle \langle e', s' \rangle$ allora vale che $\Gamma \vdash e : T'$, e' è chiusa e $dom(\Gamma) \in dom(s')$.*

Teorema 5.3 (Normalization). *Nei sottolinguaggi senza cicli while o operazioni sulla memoria, se vale che $\Gamma \vdash e : T$ e e è chiusa, allora esiste un valore v tale che, per ogni memoria s,*

$$\langle e, s \rangle \rightarrowtriangle^* \langle v, s \rangle$$

5.7 Dichiarazioni locali

Le dichiarazioni locali sono usate per aumentare la leggibilità del codice e consistono nel nominare una certa espressione restringendo al contempo il suo scope. Il costrutto da introdurre è `let` y `:int` $= 1 + 2$ `in` $y \geqslant y + 4$

La valutazione procede alla maniera seguente:

- si valuta $1 + 2$ a 3;
- si valuta $y \geqslant y + 4$ sostituendo a y il valore 3;

Estendiamo la sintassi con

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

e diamo la relativa regola di tipaggio

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

Dal momento che x è una variabile locale, Γ non contiene una entry per x, quindi potrebbe essere necessario ricorrere ad alpha-conversion.

5.7.1 Dichiarazioni locali e alpha conversion

Analogamente a quanto detto in precedenza, l'alpha-conversion è applicabile anche al costrutto `let`. In sostanza, si opera come segue:

$$(\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2) =_\alpha \texttt{let}\ y : T = e_1\ \texttt{in}\ e_2\{^y/_x\}$$

dove y è una fresh variable, ovvero non si trova nè in e_1, nè in e_2.

5.7.2 Dichiarazioni locali, free variables e sostituzioni

Definiamo le variabili libere per il costrutto `let` come

$$fv(\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2) = fv(e_1) \cup (fv(e_2) \backslash \{x\})$$

e le sostituzioni nel seguente modo:

$$(\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2)\{^e/_z\} = (\texttt{let}\ x : T = e_1\{^e/_z\}\ \texttt{in}\ e_2\{^e/_z\}) \qquad \text{if } x \notin fv(e)$$
$$(\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2)\{^e/_z\} = (\texttt{let}\ x : T = e_1\{^e/_z\}\ \texttt{in}\ (e_2\{^y/_x\})\{^e/_z\}) \quad \text{if } x \in fv(e) \wedge y\text{fresh}$$
$$(\texttt{let}\ x : T = e_1\ \texttt{in}\ e_2)\{^e/_x\} = (\texttt{let}\ x : T = e_1\{^e/_x\}\ \texttt{in}\ e_2)$$

5.7.3 Semantica small-step per dichiarazioni locali

La semantica del costrutto `let` funziona in modi differenti a seconda del tipo di chiamata. In modalità call-by-value

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

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

mentre in modalità call-by-name:

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

In realtà il costrutto `let` funge semplicemente da espansione sintattica, nel senso che ciò che si esprime con il costrutto `let` può tranquillamente essere implementato con altri costrutti, ad esempio:

$$(fn\ x : \text{unit} \Rightarrow e_2)e_1 \equiv \texttt{let } x : \text{unit} = e_1 \texttt{ in } e_2 \equiv e_1; e_2$$

5.8 Ricorsione

La ricorsione è implementata tramite punti fissi[3]. Estendiamo il linguaggio con il costrutto fix:

$$e ::= \ldots \mid fix.e$$

La regola per il tipaggio di fix è la seguente:

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

La semantica in sè è relativamente semplice:

$$\text{(Fix-cbn)}\frac{-}{fix.e \twoheadrightarrow e(fix.e)}$$

$$\text{(Fix-cbv)}\frac{e \equiv fn\ f : T_1 \to T_2 = e_1 \Rightarrow e_2}{fix.e \twoheadrightarrow e(fn\ x : T_1 \Rightarrow fix.e\ x)}$$

[3]Un punto fisso x per un operatore f è tale se vale che $f(x) = x$.

6 Dati e memoria variabile

Fino ad ora abbiamo lavorato solo su tipi di dato semplici, quali interi, booleani, unit e funzioni. Ora procediamo introducendo dati strutturati, e modificando la concezione di memoria mutabile che abbiamo usato fino ad ora.

Introduciamo quindi il prodotto tra tipi $T_1 * T_2$, che consente di lavorare su *tuple*, e la somma di tipi $T_1 + T_2$, che rappresenta l'unione disgiunta, con il valore risultante di tipo T_1 *oppure* di tipo T_2.

6.1 Somme e moltiplicazioni tra tipi

Estendiamo il linguaggio utilizzato fino ad ora con:

$$\begin{aligned} e &::= \ldots \mid (e_1, e_2) \mid \sharp 1\ e \mid \sharp 2\ e \\ T &::= \ldots \mid T_1 * T_2 \\ v &::= \ldots \mid (v_1, v_2) \end{aligned}$$

Nota: le tuple sono ordinate, non sono arbitrarie. Inoltre sono state aggiunte delle proiezioni ($\sharp$), che estraggono elementi dai record.

Moltiplicazioni. La semantica dei costrutti di moltiplicazione appena introdotti funziona come segue:

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

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

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

Il tipaggio invece procede come segue:

$$\text{(pair)}\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2} \qquad \text{(proj1)}\frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \sharp 1\ e : T_1}$$

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

Somme. Per quanto riguarda la somma di tipi, l'estensione da eseguire è la seguente:

$$\begin{aligned} e &::= \ldots \mid \text{inl}\ e : T \mid \text{inr}\ e : T \\ &\quad \mid (\text{case}\ e\ \text{of inl}\ (x_1 : T_1) \Rightarrow e_1 \mid \text{inr}\ (x_2 : T_2) \Rightarrow e_2) \\ T &::= \ldots \mid T_1 + T_2 \\ v &::= \ldots \mid \text{inl}\ v : T \mid \text{inr}\ v : T \end{aligned}$$

La semantica delle somme di tipi è definita come segue:

$$\text{(inl)}\frac{\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle}{\langle \text{inl}\ e : T, s\rangle \rightarrowtriangle \langle \text{inl}\ e' : T, s'\rangle} \qquad \text{(inr)}\frac{\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle}{\langle \text{inr}\ e : T, s\rangle \rightarrowtriangle \langle \text{inr}\ e' : T, s'\rangle}$$

$$\text{(case1)}\frac{\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle}{\langle \text{case}\ e\ \text{of inl}\ (x_1 : T_1) \Rightarrow e_1 \mid \text{inr}\ (x_2 : T_2) \Rightarrow e_2, s\rangle \rightarrowtriangle \langle \text{case}\ e'\ \text{of inl}\ (x_1 : T_1) \Rightarrow e_1 \mid \text{inr}\ (x_2 : T_2) \Rightarrow e_2, s'\rangle}$$

$$\text{(case2)}\frac{-}{\langle \text{case inl } v : T \text{ of inl } (x_1 : T_1) \Rightarrow e_1 \mid \text{inr } (x_2 : T_2) \Rightarrow e_2, s\rangle \rightarrowtriangle \langle e_1\{^v/_{x_1}\}, s\rangle}$$

$$\text{(case3)}\frac{-}{\langle \text{case inr } v : T \text{ of inl } (x_1 : T_1) \Rightarrow e_1 \mid \text{inr } (x_2 : T_2) \Rightarrow e_2, s\rangle \rightarrowtriangle \langle e_2\{^v/_{x_2}\}, s\rangle}$$

Il tipaggio delle somme procede nel seguente modo:

$$\text{(inl)}\frac{\Gamma \vdash e : T_1}{\Gamma \vdash (\text{inl } e : T_1 + T_2) : T_1 + T_2} \qquad \text{(inr)}\frac{\Gamma \vdash e : T_2}{\Gamma \vdash (\text{inr } e : T_1 + T_2) : T_1 + T_2}$$

$$\text{(case)}\frac{\Gamma \vdash e : T_1 + T_2 \quad \Gamma, x_1 : T_1 \vdash e_1 : T \quad \Gamma, x_2 : T_2 \vdash e_2 : T}{\Gamma \vdash (\text{case } e \text{ of inl } (x_1 : T_1) \Rightarrow e_1 \mid \text{inr } (x_2 : T_2) \Rightarrow e_2) : T}$$

6.2 Record

I record costituiscono una generalizzazione del prodotto tra tipi, che viene esteso fino a generare una tupla di lunghezza arbitraria.

Un record è quindi una struttura del tipo:

$$\begin{aligned} e &::= \ \ldots \mid \{lab_1 = e_1, \ldots, lab_n = e_n\} \mid \sharp lab\ e \\ T &::= \ \ldots \mid \{lab_1 : T_1, \ldots, lab_n : T_n\} \\ v &::= \ \ldots \mid \{lab_1 = v_1, \ldots, lab_n = v_n\} \end{aligned}$$

Nota: in ogni record un'etichetta non occorre mai più si una volta.

La semantica dei record è abbastanza semplice:

$$\text{(record1)}\frac{\langle e_i, s\rangle \rightarrowtriangle \langle e_i', s'\rangle}{\langle \{lab_1 = v_1, \ldots, lab_i = e_i, \ldots, lab_k = e_k\}, s\rangle \rightarrowtriangle \langle \{lab_1 = v_1, \ldots, lab_i = e_i', \ldots, lab_k = e_k\}, s'\rangle}$$

$$\text{(record2)}\frac{-}{\langle \sharp lab_i\{lab_1 = e_1 \ldots., lab_i = e_i, \ldots, lab_k = e_k\}, s\rangle \rightarrowtriangle \langle v_i, s\rangle} \qquad \text{(record3)}\frac{\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle}{\langle \sharp lab\ e, s\rangle \rightarrowtriangle \langle \sharp lab\ e', s'\rangle}$$

Il tipaggio dei record invece:

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

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

6.3 Memoria dinamica

I linguaggi moderni possiedono un concetto di memoria dinamica che supera i problemi che ci sono nella memoria del nostro linguaggio fino ad ora, ovvero:

- si possono salvare solo valori interi;
- non si possono creare nuove locazioni;

- non si possono scrivere funzioni che astraggano sulle locazioni di memoria, come ad esempio $fn\ l : intref \Rightarrow !l$.

Per rimuovere queste limitazioni, modifichiamo la sintassi come segue[4]:

$$\begin{aligned} e &::= \ \ldots \mid \cancel{l := e} \mid \cancel{M} \mid e_1 := e_2 \mid !e \mid ref\ e \mid l \\ T &::= \ \ldots \mid ref\ T \\ T_{loc} &::= \ \cancel{intref} \mid ref\ T \\ v &::= \ \ldots \mid l \end{aligned}$$

Aggiorniamo la funzione di *Store* ridefinendola come segue:

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

La semantica dei costrutti appena introdotti è le seguente:

$$\text{(ref1)}\frac{-}{\langle \text{ref } v, s\rangle \twoheadrightarrow \langle l, s\,[l \mapsto v]\rangle}\ l \notin dom(s) \quad \text{(ref2)}\frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle \text{ref } e, s\rangle \twoheadrightarrow \langle \text{ref } e', s'\rangle}$$

$$\text{(deref1)}\frac{-}{\langle !l, s\rangle \twoheadrightarrow \langle v, s\rangle}\ \text{if } l \in dom(s) \text{ and } s(l) = v \quad \text{(deref2)}\frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle !e, s\rangle \twoheadrightarrow \langle !e', s'\rangle}$$

$$\text{(assign1)}\frac{-}{\langle l := v, s\rangle \twoheadrightarrow \langle \text{skip}, s[l \mapsto v]\rangle}\ \text{if } l \in dom(s) \quad \text{(assign2)}\frac{\langle e, s\rangle \twoheadrightarrow \langle e', s'\rangle}{\langle l := e, s\rangle \twoheadrightarrow \langle l := e', s'\rangle}$$

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

Le corrispondenti regole di tipaggio sono:

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

$$\text{(deref)}\frac{\Gamma \vdash e : \text{ref } T}{\Gamma \vdash !e : T} \quad \text{(loc)}\frac{-}{\Gamma \vdash l : \text{ref } T}\ \Gamma(l) = \text{ref } T$$

Le principali novità sono le seguenti:

- l'espressione ref v ora ritorna una nuova locazione di memoria;
- all'inizio un programma non ha locazioni già disponibili, sono create a runtime;
- è possibile il costrutto ref(ref(3));
- la determinatezza non è più valida, in quanto le locazioni di memoria sono decise in maniera arbitraria;
- la memoria cresce sempre più durante l'esecuzione (si rende necessario un *garbage collector*).

[4]Le espressioni barrate sono **rimosse** dal linguaggio.

6.4 Aggiornamento delle proprietà del tipaggio

Teorema 6.1 (Memoria ben tipata)**.** *Scriviamo che* $\Gamma \vdash s$ *se:*

- $dom(\Gamma) = dom(s)$*;*
- $\forall l \in dom(s)$*, se* $\Gamma(l) = ref\, T$ *allora* $\Gamma \vdash s(l) : T$*.*

Teorema 6.2 (Progress)**.** *Se* e *è chiusa, inoltre valgono* $\Gamma \vdash e : T$ *e* $\Gamma \vdash s$ *allora* e *è un valore oppure esistono* e', s' *tali che* $\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle$*.*

Teorema 6.3 (Type preservation)**.** *Se* e *è chiusa,* $\Gamma \vdash e : T$*,* $\Gamma \vdash s$ *e* $\langle e, s\rangle \rightarrowtriangle \langle e', s'\rangle$ *allora* e' *è chiusa e, per qualche* Γ' *con dominio disgiunto da* Γ *si ha che* $\Gamma, \Gamma' \vdash e' : T$ *e* $\Gamma, \Gamma' \vdash s$*.*

7 Sotto-tipaggio

La motivazione principale per la creazione del sotto tipaggio è sostanzialmente il fatto di non poter tipare (con i type systems visti fino ad ora) i programmi che, aspettandosi un certo argomento, ricevono come parametro un argomento più ricco dal punto di vista strutturale. Ad esempio, se una certa funzione si aspetta un argomento

$$\{left : \text{int}\}$$

le si può passare tranquillamente un argomento del tipo

$$\{left : \text{int}, right : \text{int}\}$$

senza avere problemi nell'accesso alla struttura dati in quanto un campo con la label $left$ si trova in entrambi i parametri.

Introduciamo quindi una regola di sussunzione, ovvero:

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

Nella regola di sussunzione, T è sottotipo di T', quindi **un oggetto di tipo T può sempre essere usato dovunque ci si aspetti un oggetto di tipo T'**.

La relazione di sottotipaggio è **riflessiva** e **transitiva**.

Le principali modifiche apportate dal subtyping si riflettono sul tipaggio dei programmi: vengono infatte mantenute le proprietà di type-preservation e progress, ma le regole di tipaggio non sono più syntax-directed.

7.1 Sottotipaggio sui record

Con il sottotipaggio si può esprimere il riordinamento dei campi dei record, il passaggio di record più grandi come parametro oppure il sottotipaggio dei singoli campi di uno stesso record:

$$\text{(rec-perm)}\frac{\pi \text{ permutazione di } 1, \ldots, k}{\{p_1 : T_1, \ldots, p_k : T_k\} <: \{p_{\pi(1)} : T_1, \ldots, p_{\pi(k)} : T_k\}}$$

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

$$\text{(rec-depth)}\frac{T_1 <: T'_1 \quad \ldots \quad T_k <: T'_k}{\{p_1 : T_1, \ldots, p_k : T_k\} <: \{p_1 : T'_1, \ldots, p_k : T'_k\}}$$

7.2 Sottotipaggio su funzioni

La regola di sottotipaggio su funzioni è la seguente:

$$\text{(fun-sub)}\frac{T_1 >: T'_1 \quad T_2 <: T'_2}{T_1 \to T_2 <: T'_1 \to T'_2}$$

Il sottotipaggio su funzioni è detto essere:

- *covariante* sulla sinistra di $\to$;
- *controvariante* in caso contrario.

7.3 Sottotipaggio su somme e prodotti

Il sottotipaggio è covariante sia sulla somme che sui prodotti:

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

7.4 Downcasting

La regola di sussunzione permette l'upcasting al momento: si possono passare argomenti "più ampi" al posto di quelli "più piccoli". Per il downcast, supponiamo di aggiungere alle espressioni la seguente:

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

e aggiungiamo la regola di tipaggio

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

L'espressione $(T)e$ **non può essere tipata a tempo di compilazione**, ma solo a runtime, in quanto bisogna sapere il "vero tipo" di e. Ad esempio, avendo un down-cast come questo:

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

non si saprà mai a compile-time il tipo ritornato da $!l$, ma lo si scoprirà solo a runtime. Notare che il tipo ritornato potrebbe anche non essere "castabile" a $\{left : \text{int}\}$.

8 Equivalenze semantiche

Intuitivamente, diciamo che due programmi P_1 e P_2 sono semanticamente equivalenti ($P_1 \simeq P_2$) se uno dei due può essere rimpiazzato dall'altro, in un qualsiasi contesto.

Con una buona equivalenza semantica, si può quindi:

- comprendere cosa sia un programma;
- dimostrare alcune particolari proprietà (*program verification*);
- dimostrare che le ottimizzazioni di alcuni compilatori sono *sound*;
- comprendere le differenze semantiche tra due programmi.

In particolare per avere una buona relazione di equivalenza si deve avere che:

1. programmi che convergono a valori diversi, iniziando dalla stessa memoria, non devono essere equivalenti, quindi

$$(\exists s, s_1, s_2, v_1, v_2. \langle e_1, s\rangle \rightarrowtriangle^* \langle v_1, s_1\rangle \wedge \langle e_2, s_2\rangle \rightarrowtriangle^* \langle v_2, s_2\rangle \wedge v_1 \neq v_2) \implies e_1 \not\simeq e_2$$

2. i programmi che terminano non devono essere simili ai programmi che non terminano;
3. $\simeq$ deve essere una relazione di equivalenza;
4. $\simeq$ deve essere una *congruenza*, ovvero se vale che $e_1 \simeq e_2$ per un qualche contesto $C[\cdot]$, allora deve valere che $C[e_1] \simeq C[e_2]$;
5. $\simeq$ deve valere per quanti più programmi possibili.

Un contesto $C[\cdot]$ è un programma non completo, ovvero un programma con una parte mancante, la quale deve essere istanziata con un qualche programma P. Inoltre, se vale che $\simeq$ è una congruenza, allora avendo che $P \simeq Q$ si ottiene che $C[P] \simeq C[Q]$.

Considerando il semplice linguaggio imperativo, diamo qualche definizione formale.

Definizione 8.1 (Trace equivalence)**.** *Definiamo che* $e_1 \simeq^T_\Gamma e_2$ *vale se,* $\forall s.dom(\Gamma) \subseteq dom(s)$*, si ha che* $\Gamma \vdash e_1 : T$*,* $\Gamma \vdash e_2 : T$ *e*

- $(\langle e_1, s\rangle \rightarrowtriangle^* \langle v, s'\rangle) \implies (\langle e_2, s\rangle \rightarrowtriangle^* \langle v, s'\rangle)$*;*
- $(\langle e_2, s\rangle \rightarrowtriangle^* \langle v, s'\rangle) \implies (\langle e_1, s\rangle \rightarrowtriangle^* \langle v, s'\rangle)$*;*

Definizione 8.2 (Proprietà di congruenza)**.** *La relazione di equivalenza* $\simeq^T_\Gamma$ *soddisfa la proprietà della congruenza, in quanto, dati due programmi* e_1 *e* e_2 *qualsiasi che soddisfano* $e_1 \simeq^T_\Gamma e_2$*, per ogni contesto* C *e tipo* T'*, se vale che* $\Gamma \vdash C[e_1] : T'$ *e* $\Gamma \vdash C[e_2] : T'$*, allora* $C[e_1] \simeq^T_\Gamma C[e_2]$*.*

Diamo qui di seguito alcune leggi basilari sulla trace equivalence.

Teorema 8.1 (Associatività di ;)**.** *Vale:*

$$e_1; (e_2; e_3) \simeq^T_\Gamma (e_1; e_2); e_3$$

per qualsiasi Γ, T, e_1, e_2 *ed* e_3 *tali per cui* $\Gamma \vdash e_1 : unit, \Gamma \vdash e_2 : unit$ *e* $\Gamma \vdash e_3 : unit$*.*

Teorema 8.2 (Rimozione di skip)**.** *Valgono:*

- $e_2 \simeq^T_{\Gamma_2} (skip; e_2)$
- $e_1; skip \simeq^{unit}_{\Gamma_1} e_1$

per qualunque $\Gamma_1, \Gamma_2, T, e_1, e_2$ *tali che* $\Gamma_2 \vdash e_2 : T$ *e* $\Gamma_1 \vdash e_1 : unit$*.*

Teorema 8.3 (if-false)**.**

$$if\ false\ then\ e_1\ else\ e_2 \simeq^T_\Gamma e_2$$

per qualsiasi Γ, T, e_1, e_2 *tali per cui* $\Gamma \vdash e_1 : T, \Gamma \vdash e_2 : T$*.*

Teorema 8.4 (if-true)**.**

$$if\ true\ then\ e_1\ else\ e_2 \simeq^T_\Gamma e_2$$

per qualsiasi Γ, T, e_1, e_2 *tali per cui* $\Gamma \vdash e_1 : T, \Gamma \vdash e_2 : T$*.*

Teorema 8.5 (Distributività dell'if rispetto a ;)**.**

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

per qualsiasi Γ, T, e_1, e_2, e_3 *tali che* $\Gamma \vdash e_1 : bool, \Gamma \vdash e_2 : unit, \Gamma \vdash e_3 : unit, \Gamma \vdash e : T$*.*

Teorema 8.6 (Distributività di ; rispetto all'if)**.**

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

per qualsiasi Γ, T, e_1, e_2, e_3 *tali che* $\Gamma \vdash e_1 : bool, \Gamma \vdash e_2 : T, \Gamma \vdash e_3 : T, \Gamma \vdash e : unit$*.*

Diamo ora due definizioni più formali di trace equivalence, che considerano la cosa da un altro punto di vista.

Definizione 8.3 (Simulazione)**.** *Diciamo che* e_1 *è simulato da* e_2*, indicando* $e_1 \sqsubseteq^T_\Gamma e_2$*, se:*

- $\Gamma \vdash e_1 : T$ *e* $\Gamma \vdash e_2 : T$ *per qualche* T*;*
- *per qualsiasi* s *con* $dom(\Gamma) \subseteq dom(s)$*, se* $\langle e_1, s\rangle \rightarrowtriangle \langle e'_1, s'_1\rangle$ *allora esiste un* e'_2 *tale che* $\langle e_2, s\rangle \rightarrowtriangle^* \langle e'_2, s'_2\rangle$*, con* $e'_1 \sqsubseteq^T_\Gamma e'_2$ *e* $s'_1 = s'_2$*.*

Definizione 8.4 (Bisimulazione)**.** *Diciamo che* e_1 *è bisimile ad* e_2*, indicando* $e_1 \approx^T_\Gamma e_2$ *se e solo se:*

- $\Gamma \vdash e_1 : T$ *e* $\Gamma \vdash e_2 : T$ *per qualche* T*;*
- *per qualsiasi* s *con* $dom(\Gamma) \subseteq dom(s)$*, se* $\langle e_1, s\rangle \rightarrowtriangle \langle e'_1, s'_1\rangle$ *allora esiste un* e'_2 *tale che* $\langle e_2, s\rangle \rightarrowtriangle^* \langle e'_2, s'_2\rangle$*, con* $e'_1 \approx^T_\Gamma e'_2$ *e* $s'_1 = s'_2$*;*
- *per qualsiasi* s *con* $dom(\Gamma) \subseteq dom(s)$*, se* $\langle e_2, s\rangle \rightarrowtriangle \langle e'_2, s'_2\rangle$ *allora esiste un* e'_1 *tale che* $\langle e_1, s\rangle \rightarrowtriangle^* \langle e'_1, s'_1\rangle$*, con* $e'_1 \approx^T_\Gamma e'_2$ *e* $s'_1 = s'_2$*.*

9 Concorrenza

Fino ad ora il nostro focus è stato quello di un ambito di programmazione puramente sequenziale. In realtà i sistemi moderni non viaggiano mai in linea sequenziale, ma sono programmati in ambito concorrente.

La programmazione concorrente può incrementare anche di molto le prestazioni, con alcuni effetti collaterali, quali:

- lo spazio degli stati diventa enormemente più ampio e complesso;
- si pongono i problemi di *deadlock*, *starvation* e *data race*, che possono portare a comportamenti erronei;
- le computazioni possono diventare non deterministiche, a meno non sia imposto un qualche sistema di sincronizzazione;
- problemi di comunicazione tra processi con differenti risorse locali (memorie, librerie condivise,...);
- problemi legati alla sicurezza.

9.1 Definizione del linguaggio

Per cominciare, definiamo il linguaggio sul quale si lavorerà: è composto da un linguaggio imperativo banale come in 3 e alcuni altri costrutti, quali:

$$\begin{aligned} e &::= \ \ldots \mid e||e \\ T &::= \ \ldots \mid proc \end{aligned}$$

Il linguaggio gode di alcune caratteristiche, quali:

- la composizione parallela $e\ ||e$;
- le thread sono anonime, non ritornano valore e non possono essere uccise esternamente;
- i processi sono costituiti da un *pool* di thread concorrenti;
- la terminazione di thread non può essere osservata direttamente da un programma.

Inoltre, come in ogni altro linguaggio concorrente, le thread vengono eseguite in maniera asincrona e possono scrivere e leggere su memoria condivisa, di conseguenza **si perde la proprietà di determinatezza**.

9.2 Semantica della composizione parallela

La semantica della composizione parallela procede come segue:

$$\text{(par-L)}\frac{\langle e_1, s\rangle \rightarrowtriangle \langle e_1', s'\rangle}{\langle e_1 \mid\mid e_2, s\rangle \rightarrowtriangle \langle e_1' \mid\mid e_2, s'\rangle} \qquad \text{(par-R)}\frac{\langle e_2, s\rangle \rightarrowtriangle \langle e_2', s'\rangle}{\langle e_1 \mid\mid e_2, s\rangle \rightarrowtriangle \langle e_1 \mid\mid e_2', s'\rangle}$$

$$\text{(end-L)}\frac{-}{\langle skip \mid\mid e, s\rangle \rightarrowtriangle \langle e, s\rangle} \qquad \text{(end-R)}\frac{-}{\langle e \mid\mid skip, s\rangle \rightarrowtriangle \langle e, s\rangle}$$

9.3 Tipaggio della composizione parallela

Il tipaggio della composizione parallela procede come segue:

$$\text{(T-sq1)}\frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash e_1; e_2 : unit} \qquad \text{(T-sq2)}\frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : proc}{\Gamma \vdash e_1; e_2 : proc}$$

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

Nota: dire che $\Gamma \vdash e_1 : unit$ equivale a dire che e_1 è single-threaded, mentre se è di tipo *proc* è multi-threaded.

9.4 Race conditions

Le race condition sono un fenomeno che si verifica quando più thread vanno a modificare una zona di memoria condivisa senza un adeguato metodo di accesso, finendo con l'ottenere risultati inconsistenti e mal composti. Il problema risiede nel fatto che l'esecuzione di una thread può essere interrotta anche a metà di un'operazione, che, senza un sistema di sincronizzazione, causa risultati privi di senso.

È necessario quindi introdurre costrutti di sincronizzazione più sofisticati, che possano garantire l'accesso in mutua esclusione ai dati condivisi di interesse.