

# Linguaggi di programmazione

Riassunto dei principali argomenti

Autore:

**Davide Bianchi**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Esempio di linguaggio basilare</b>	<b>3</b>
2.1	Semantica big-step . . . . .	3
2.1.1	Esempio . . . . .	3
2.2	Semantica small-step . . . . .	4
<b>3</b>	<b>Linguaggio imperativo</b>	<b>5</b>
3.1	Memoria . . . . .	5
3.2	Sistemi di transizione . . . . .	5
3.3	Semantica small-step su un linguaggio imperativo . . . . .	6
3.4	Esecuzione di programmi e proprietà . . . . .	6
3.5	Funzione di valutazione della semantica . . . . .	6
3.6	Possibili varianti del linguaggio . . . . .	7
3.6.1	Inversione dell'ordine di valutazione . . . . .	7
3.6.2	Regole di assegnamento . . . . .	7
3.6.3	Inizializzazione della memoria . . . . .	7
3.6.4	Valori memorizzabili . . . . .	7
3.7	Type systems . . . . .	7
3.7.1	Regole di tipaggio . . . . .	8
3.7.2	Proprietà di tipaggio . . . . .	8
<b>4</b>	<b>Forme di induzione</b>	<b>10</b>
4.1	Induzione matematica . . . . .	10
4.2	Induzione strutturale . . . . .	10
4.2.1	Induzione strutturale su numeri naturali . . . . .	10
4.2.2	Induzione strutturale su strutture complesse . . . . .	10
4.3	Rule induction . . . . .	11
<b>5</b>	<b>Aspetti funzionali</b>	<b>12</b>
5.1	Variabili free e bound . . . . .	12
5.2	Alpha conversion . . . . .	12
5.3	Sostituzioni . . . . .	13
5.3.1	Sostituzioni singole . . . . .	13
5.3.2	Sostituzioni simultanee . . . . .	13
5.4	Lambda calcolo . . . . .	13
5.5	Applicazione di funzioni . . . . .	14
5.5.1	Semantica dell'applicazione di funzioni . . . . .	14
5.6	Tipaggio di funzioni . . . . .	14
5.7	Dichiarazioni locali . . . . .	15
5.7.1	Dichiarazioni locali e alpha conversion . . . . .	15
5.7.2	Dichiarazioni locali, free variables e sostituzioni . . . . .	15
5.7.3	Semantica small-step per dichiarazioni locali . . . . .	16
5.8	Ricorsione . . . . .	16

<b>6</b>	<b>Dati e memoria variabile</b>	<b>17</b>
6.1	Somme e moltiplicazioni tra tipi . . . . .	17
6.2	Record . . . . .	18
6.3	Memoria dinamica . . . . .	18
6.4	Aggiornamento delle proprietà del tipaggio . . . . .	20
<b>7</b>	<b>Sotto-tipaggio</b>	<b>20</b>
<b>8</b>	<b>Equivalenze semantiche</b>	<b>20</b>
<b>9</b>	<b>Concorrenza</b>	<b>20</b>

## 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 operativa di un linguaggio è data attraverso un sistema di regole di inferenza, date come segue:

$$(Assioma) \frac{}{(Conclusione)} \quad (Regola) \frac{(Hyp_1) (Hyp_2) \dots (Hyp_n)}{(Conclusione)}$$

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

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

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:

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

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

#### 2.1.1 Esempio

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

**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  $E_1 \rightarrow E_2$  lo svolgimento di un solo passo di semantica.

$$\begin{array}{l} \text{S-Left} \frac{E_1 \rightarrow E'_1}{E_1 + E_2 \rightarrow E'_1 + E_2} \\ \text{S-N.Right} \frac{E_2 \rightarrow E'_2}{n_1 + E_2 \rightarrow n_1 + E'_2} \\ \text{S-Add} \frac{-}{n_1 + n_2 \rightarrow n_3} n_3 = \text{add}(n_1, n_2) \end{array}$$

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

$$\begin{array}{l} \text{S-Left} \frac{E_1 \rightarrow_{ch} E_2}{E_1 + E_2 \rightarrow_{ch} E'_1 + E_2} \\ \text{S-Right} \frac{E_2 \rightarrow_{ch} E'_2}{E_1 + E_2 \rightarrow_{ch} E_1 + E'_2} \\ \text{S-Add} \frac{-}{n_1 + n_2 \rightarrow_{ch} n_3} n_3 = \text{add}(n_1, n_2) \end{array}$$

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

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

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

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

### 3 Linguaggio imperativo

Definiamo la sintassi di un semplice linguaggio imperativo:

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

**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  $2 \geq true$ .

#### 3.1 Memoria

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

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

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

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

L'aggiornamento della memoria funziona come segue:

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

#### 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  $\rightarrow \subseteq Config \times Config$ ;

Per indicare un generale passo di semantica si usa la notazione

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

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

$$\begin{array}{c}
 \text{(op+)} \frac{}{\langle n_1 + n_2, s \rangle \rightarrow \langle n, s \rangle} \quad n = \text{add}(n_1, n_2) \qquad \text{(op-geq } ^1) \frac{}{\langle n_1 \geq n_2, s \rangle \rightarrow \langle b, s \rangle} \quad b = \text{geq}(n_1, n_2) \\
 \\
 \text{(op1)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 \text{ op } e_2, s \rangle \rightarrow \langle e'_1 \text{ op } e_2, s' \rangle} \qquad \text{(op2)} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle v \text{ op } e_2, s \rangle \rightarrow \langle v \text{ op } e'_2, s' \rangle} \\
 \\
 \text{(deref)} \frac{}{\langle !l, s \rangle \rightarrow \langle n, s \rangle} \quad \text{if } l \in \text{dom}(s) \text{ and } s(l) = n \qquad \text{(assign1)} \frac{}{\langle l := n, s \rangle \rightarrow \langle \text{skip}, s[l \rightarrow n] \rangle} \quad \text{if } l \in \text{dom}(s) \\
 \\
 \text{(assign2)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle l := e, s \rangle \rightarrow \langle l := e', s' \rangle} \qquad \text{(if-tt)} \frac{}{\langle \text{if true then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_1, s \rangle} \\
 \\
 \text{(if-ff)} \frac{}{\langle \text{if false then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle e_2, s \rangle} \qquad \text{(if)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{if } e \text{ then } e_1 \text{ else } e_2, s \rangle \rightarrow \langle \text{if } e' \text{ then } e_1 \text{ else } e_2, s' \rangle} \\
 \\
 \text{(seq)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1; e_2, s \rangle \rightarrow \langle e'_1; e_2, s' \rangle} \qquad \text{(seq.skip)} \frac{}{\langle \text{skip}; e_2, s \rangle \rightarrow \langle e_2, s \rangle} \\
 \\
 \text{(while)} \frac{}{\langle \text{while } e \text{ do } e_1, s \rangle \rightarrow \langle \text{if } e \text{ then } (e_1; \text{while } e \text{ do } e_1) \text{ else } \text{skip}, s \rangle}
 \end{array}$$

### 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

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

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*

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

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

### 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

$$\llbracket - \rrbracket : \text{Exp} \rightarrow (\text{Store} \rightarrow \text{Store})$$

<sup>1</sup>Problemi con L<sup>A</sup>T<sub>E</sub>X implicano l'uso di un nome diverso.

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

$$\llbracket e \rrbracket(s) = \begin{cases} s' & \text{se } \langle e, s \rangle \rightarrow \langle e', s' \rangle \\ \text{undefined} & \text{altrimenti} \end{cases}$$

### 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:

$$\text{(op1b)} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle e_1 + e_2, s \rangle \rightarrow \langle e_1 + e'_2, s' \rangle} \quad \text{(op2b)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 + v, s \rangle \rightarrow \langle e'_1 + v, s' \rangle}$$

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:

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

#### 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  $l \notin \text{dom}(s)$  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 possibilità 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  $T_{loc}$ :

$$\begin{aligned} T &::= int \mid bool \mid unit \\ T_{loc} &::= intref \end{aligned}$$

### 3.7.1 Regole di tipaggio

$$\begin{array}{ll} \text{(int)} \frac{}{\Gamma \vdash n : int} \text{ per } n \in \mathbb{Z} & \text{(bool)} \frac{}{\Gamma \vdash b : bool} \text{ per } b \in \{true, false\} \\ \text{(op+)} \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int} & \text{(op-geq)} \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \geq e_2 : bool} \\ \text{(if)} \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} & \text{(assign)} \frac{\Gamma \vdash e : int}{\Gamma \vdash l := e : unit} \text{ se } \Gamma(l) = intref \\ \text{(deref)} \frac{}{\Gamma \vdash !l : int} \text{ se } \Gamma(l) = intref & \text{(skip)} \frac{}{\Gamma \vdash skip : unit} \\ \text{(seq)} \frac{\Gamma \vdash e_1 : unit \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} & \text{(while)} \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : unit}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : unit} \end{array}$$

**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*

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

**Teorema 3.4** (Type preservation). *Se  $\Gamma \vdash e : T$  e  $dom(\Gamma) \subseteq dom(s)$  e  $\langle e, s \rangle \rightarrow \langle e', s' \rangle$  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  $\langle e, s \rangle \rightarrow^* \langle e', s' \rangle$  allora  $e'$  è un valore oppure esiste una coppia  $\langle e'', s'' \rangle$  tale che  $\langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$*

**Teorema 3.6** (Type inference). *Dati  $\Gamma, e$ , 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  $\Gamma, e, T$ , 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<sup>2</sup>.

### 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 dimostrare 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  $P(\text{succ}(K))$  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  $\text{succ}(K)$ .

#### 4.2.2 Induzione strutturale su strutture complesse

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

$$T ::= \text{leaf} \mid \text{tree}(T, T)$$

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  $\text{tree}(L, R)$ .

---

<sup>2</sup>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

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

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

## 5 Aspetti funzionali

Estendiamo la sintassi data in 3 con:

$$\begin{aligned} &\text{Variabili } x \in \mathbb{X}, \text{ con } x = \{x, y, z, \dots\} \\ &\text{Espressioni } e ::= \dots \mid \text{fn } x : T \Rightarrow e \mid ee \mid x \end{aligned}$$

e i tipi con:

$$T ::= \dots \mid T \rightarrow T$$

Assumiamo che:

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

### 5.1 Variabili free e bound

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

Più formalmente definiamo le variabili free come una funzione:

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

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

$$\begin{aligned} fv(x) &\triangleq \{x\} \\ fv(\text{fn } x : T \Rightarrow e) &\triangleq fv(e) \setminus \{x\} \\ fv(e_1 e_2) &= fv(e_1; e_2) \triangleq fv(e_1) \cup fv(e_2) \\ fv(n) &= fv(b) = fv(!l) = fv(\text{skip}) \triangleq \emptyset \\ fv(e_1 \text{ope}_2) &= fv(\text{while } e_1 \text{ do } e_2) \triangleq fv(e_1) \cup fv(e_2) \\ fv(l := e) &\triangleq fv(e) \\ fv(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &\triangleq fv(e_1) \cup fv(e_2) \cup fv(e_3) \end{aligned}$$

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

### 5.2 Alpha conversion

Nell'espressione  $\text{fn } x : T \Rightarrow e$  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:

$$\begin{aligned} n\{e/x\} &\triangleq n \\ b\{e/x\} &\triangleq b \\ \text{skip}\{e/x\} &\triangleq \text{skip} \\ x\{e/x\} &\triangleq e \\ y\{e/x\} &\triangleq y \\ (\text{fn } x:T \Rightarrow e_1)\{e/z\} &\triangleq (\text{fn } x:T \Rightarrow e_1\{e/z\}) \text{ se } x \notin \text{fv}(e) \\ (\text{fn } x:T \Rightarrow e_1)\{e/z\} &\triangleq (\text{fn } x:T \Rightarrow (e_1\{e/z\})\{e/z\}) \text{ se } x \in \text{fv}(e) \wedge y \text{ è fresh}^3 \\ (\text{fn } x:T \Rightarrow e_1)\{e/x\} &\triangleq (\text{fn } x:T \Rightarrow e_1) \\ (e_1; e_2)\{e/x\} &\triangleq (e_1\{e/x\}e_2\{e/x\}) \end{aligned}$$

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  $\sigma : \mathbb{X} \rightarrow \text{Exp}$ .

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

$$\{e_1/x_1, \dots, e_k/x_k\}$$

### 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  $\text{fn } x:T \Rightarrow e$  diventano funzioni del tipo  $\lambda x : T.e$ .

La sintassi del lambda-calcolo non tipato è:

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

---

<sup>3</sup>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

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

La semantica funziona come di seguito:

$$\text{App} \frac{M_1 \rightarrow M'_1}{M_1 M_2 \rightarrow M'_1 M_2}$$

Inoltre, in modalità call-by-value:

$$\text{CBV-app1} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 e_2, s \rangle \rightarrow \langle e'_1 e_2, s' \rangle} \quad \text{CBV-app2} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle v e_2, s \rangle \rightarrow \langle v e'_2, s' \rangle} \quad \text{CBV-fn} \frac{-}{\langle (\text{fn } x : T \Rightarrow e) v, s \rangle \rightarrow \langle e\{v/x\}, s \rangle}$$

mentre in modalità call-by-name:

$$\text{CBN-app} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 e_2, s \rangle \rightarrow \langle e'_1 e_2, s' \rangle} \quad \text{CBN-fn} \frac{-}{\langle (\text{fn } x : T \Rightarrow e) e_2, s \rangle \rightarrow \langle e\{e_2/x\}, s \rangle}$$

Introduciamo inoltre una nuova semantica che raccoglie le due precedenti:

$$\text{BETA-app1} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 e_2, s \rangle \rightarrow \langle e'_1 e_2, s' \rangle} \quad \text{BETA-app2} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle e_1 e_2, s \rangle \rightarrow \langle e_1 e'_2, s' \rangle}$$

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

## 5.6 Tipaggio di funzioni

Estendiamo l'insieme dei tipi come segue:

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

tale che:

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

Definiamo le regole di tipaggio:

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

Qui di seguito definiamo alcune proprietà del tipaggio:

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

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

**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  $s$ ,*

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

## 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  $y : \text{int} = 1 + 2$  in  $y \geq y + 4$`

La valutazione procede alla maniera seguente:

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

Estendiamo la sintassi con

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

e diamo la relativa regola di tipaggio

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

Dal momento che  $x$  è una variabile locale,  $\Gamma$  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:

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

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

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

e le sostituzioni nel seguente modo:

$$\begin{aligned} (\text{let } x : T = e_1 \text{ in } e_2)\{e/z\} &= (\text{let } x : T = e_1\{e/z\} \text{ in } e_2\{e/z\}) && \text{if } x \notin fv(e) \\ (\text{let } x : T = e_1 \text{ in } e_2)\{e/z\} &= (\text{let } x : T = e_1\{e/z\} \text{ in } (e_2\{y/x\})\{e/z\}) && \text{if } x \in fv(e) \wedge y \text{ fresh} \\ (\text{let } x : T = e_1 \text{ in } e_2)\{e/x\} &= (\text{let } x : T = e_1\{e/x\} \text{ in } e_2) \end{aligned}$$



### 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

$$\begin{aligned} \text{(CBV-let1)} \quad & \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle \text{let } x : T = e_1 \text{ in } e_2, s \rangle \rightarrow \langle \text{let } x : T = e'_1 \text{ in } e_2, s' \rangle} \\ \text{(CBV-let2)} \quad & \frac{-}{\langle \text{let } x : T = v \text{ in } e_2, s \rangle \rightarrow \langle e_2\{v/x\}, s \rangle} \end{aligned}$$

mentre in modalità call-by-name:

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

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:

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

## 5.8 Ricorsione

La ricorsione è implementata tramite punti fissi<sup>4</sup>. Estendiamo il linguaggio con il costrutto `fix`:

$$e ::= \dots \mid \text{fix}.e$$

La regola per il tipaggio di `fix` è la seguente:

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

La semantica in sè è relativamente semplice:

$$\begin{aligned} \text{(Fix-cbn)} \quad & \frac{-}{\text{fix}.e \rightarrow e(\text{fix}.e)} \\ \text{(Fix-cbv)} \quad & \frac{e \equiv fn \ f : T_1 \rightarrow T_2 = e_1 \Rightarrow e_2}{\text{fix}.e \rightarrow e(fn \ x : T_1 \Rightarrow \text{fix}.e \ x)} \end{aligned}$$

---

<sup>4</sup>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:

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

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

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

$$\begin{array}{ll} \text{(pair1)} \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle (e_1, e_2), s \rangle \rightarrow \langle (e'_1, e_2), s' \rangle} & \text{(pair2)} \frac{\langle e_2, s \rangle \rightarrow \langle e'_2, s' \rangle}{\langle (v, e_2), s \rangle \rightarrow \langle (v, e'_2), s' \rangle} \\ \text{(proj1)} \frac{-}{\langle \#1(v_1, v_2), s \rangle \rightarrow \langle v_1, s \rangle} & \text{(proj2)} \frac{-}{\langle \#2(v_1, v_2), s \rangle \rightarrow \langle v_2, s \rangle} \\ \text{(proj3)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \#1 \ e, s \rangle \rightarrow \langle \#1 \ e', s \rangle} & \text{(proj4)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \#2 \ e, s \rangle \rightarrow \langle \#2 \ e', s \rangle} \end{array}$$

Il tipaggio invece procede come segue:

$$\begin{array}{ll} \text{(pair)} \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2} & \text{(proj1)} \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#1 \ e : T_1} \\ \text{(proj2)} \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \#2 \ e : T_2} \end{array}$$

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

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

La semantica delle somme di tipi è definita come segue:

$$\begin{array}{ll} \text{(inl)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{inl } e : T, s \rangle \rightarrow \langle \text{inl } e' : T, s' \rangle} & \text{(inr)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{inr } e : T, s \rangle \rightarrow \langle \text{inr } e' : T, s' \rangle} \\ \text{(case1)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{case } e \text{ of inl } (x_1 : T_1) \Rightarrow e_1 \mid \text{inr } (x_2 : T_2) \Rightarrow e_2, s \rangle \rightarrow \langle \text{case } e' \text{ of inl } (x_1 : T_1) \Rightarrow e_1 \mid \text{inr } (x_2 : T_2) \Rightarrow e_2, s' \rangle} \end{array}$$

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

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

Il tipaggio delle somme procede nel seguente modo:

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

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

## 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:

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

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

La semantica dei record è abbastanza semplice:

$$\text{(record1)} \frac{\langle e_i, s \rangle \rightarrow \langle e'_i, s' \rangle}{\langle \{lab_1 = v_1, \dots, lab_i = e_i, \dots, lab_k = e_k\}, s \rangle \rightarrow \langle \{lab_1 = v_1, \dots, lab_i = e'_i, \dots, lab_k = e_k\}, s' \rangle}$$

$$\text{(record2)} \frac{-}{\langle \#lab_i \{lab_1 = e_1, \dots, lab_i = e_i, \dots, lab_k = e_k\}, s \rangle \rightarrow \langle v_i, s \rangle} \quad \text{(record3)} \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \#lab \ e, s \rangle \rightarrow \langle \#lab \ e', s' \rangle}$$

Il tipaggio dei record invece:

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

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

## 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  $fn\ l : \text{intref} \Rightarrow !l$ .

Per rimuovere queste limitazioni, modifichiamo la sintassi come segue<sup>5</sup>:

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

Aggiorniamo la funzione di *Store* ridefinendola come segue:

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

La semantica dei costrutti appena introdotti è le seguente:

$$\begin{aligned} (\text{ref1}) \frac{}{\langle \text{ref } v, s \rangle \rightarrow \langle l, s[l \mapsto v] \rangle} \quad & l \notin \text{dom}(s) \quad (\text{ref2}) \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle \text{ref } e, s \rangle \rightarrow \langle \text{ref } e', s' \rangle} \\ (\text{deref1}) \frac{}{\langle !l, s \rangle \rightarrow \langle v, s \rangle} \quad & \text{if } l \in \text{dom}(s) \text{ and } s(l) = v \quad (\text{deref2}) \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle !e, s \rangle \rightarrow \langle !e', s' \rangle} \\ (\text{assign1}) \frac{}{\langle l := v, s \rangle \rightarrow \langle \text{skip}, s[l \mapsto v] \rangle} \quad & \text{if } l \in \text{dom}(s) \quad (\text{assign2}) \frac{\langle e, s \rangle \rightarrow \langle e', s' \rangle}{\langle l := e, s \rangle \rightarrow \langle l := e', s' \rangle} \\ (\text{assign3}) \frac{\langle e_1, s \rangle \rightarrow \langle e'_1, s' \rangle}{\langle e_1 := e_2, s \rangle \rightarrow \langle e'_1 := e_2, s' \rangle} \end{aligned}$$

Le corrispondenti regole di tipaggio sono:

$$\begin{aligned} (\text{ref}) \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : \text{ref } T} \quad & (\text{assign}) \frac{\Gamma \vdash e_1 : \text{ref } T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1 := e_2) : \text{unit}} \\ (\text{deref}) \frac{\Gamma \vdash e : \text{ref } T}{\Gamma \vdash !e : T} \quad & (\text{loc}) \frac{}{\Gamma \vdash l : \text{ref } T} \quad \Gamma(l) = \text{ref } T \end{aligned}$$

Le principali novità sono le seguenti:

- l'espressione  $\text{ref } v$  ora ritorna una nuova locazione di memoria;
- all'inizio un programma non ha locazioni già disponibili, sono create a runtime;
- è possibile il costrutto  $\text{ref}(\text{ref}(3))$ ;
- 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*).

---

<sup>5</sup>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:*

- $\text{dom}(\Gamma) = \text{dom}(s)$ ;
- $\forall l \in \text{dom}(s)$ , se  $\Gamma(l) = \text{ref } T$  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  $\langle e, s \rangle \rightarrow \langle e', s' \rangle$ .*

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

- 7 Sotto-tipaggio
- 8 Equivalenze semantiche
- 9 Concorrenza