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

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

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

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

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

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

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

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

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:

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

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

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

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

$$\text{(op+)}\frac{-}{\langle n_1 + n_2, s\rangle \rightarrow \langle n, s\rangle}\ n = add(n_1, n_2) \qquad \text{(op-geq [1])}\frac{-}{\langle n_1 \geq n_2, s\rangle \rightarrow \langle b, s\rangle}\ b = geq(n_1, n_2)$$

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

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

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

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

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

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

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

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

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

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

[1] Problemi con LaTeX implicano l'uso di un nome diverso.

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:

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

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

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

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

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

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

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

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[2].

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 $tree(L, R)$.

[2] 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 $1 \leq i \leq n$.

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:

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

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

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:

$$\begin{aligned}
n\{^e/_x\} &\triangleq n \\
b\{^e/_x\} &\triangleq b \\
\texttt{skip}\{^e/_x\} &\triangleq \texttt{skip} \\
x\{^e/_x\} &\triangleq e \\
y\{^e/_x\} &\triangleq y \\
(\texttt{fn x:T} \Rightarrow e_1)\{^e/_z\} &\triangleq (\texttt{fn x:T} \Rightarrow e_1\{^e/_z\}) \text{se } x \notin fv(e) \\
(\texttt{fn x:T} \Rightarrow e_1)\{^e/_z\} &\triangleq (\texttt{fn x:T} \Rightarrow (e_1\{^e/_z\})\{^e/_z\}) \text{se } x \in fv(e) \wedge y \text{ è fresh}^3 \\
(\texttt{fn x:T} \Rightarrow e_1)\{^e/_x\} &\triangleq (\texttt{fn x:T} \Rightarrow e_1) \\
(\texttt{e}_1\texttt{;e}_2)\{^e/_x\} &\triangleq (e_1\{^e/_x\}e_2\{^e/_x\})
\end{aligned}$$

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 $\sigma : \mathbb{X} \to Exp$.

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

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

[3] Operando con alpha-conversion.

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:

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

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

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

Inoltre, in modalità call-by-value:

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

mentre in modalità call-by-name:

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

6 Dati e memoria variabile

7 Sotto-tipaggio