



SAPIENZA  
UNIVERSITÀ DI ROMA

“SAPIENZA” UNIVERSITÀ DI ROMA  
INGEGNERIA DELL'INFORMAZIONE,  
INFORMATICA E STATISTICA  
DIPARTIMENTO DI INFORMATICA

---

# Linguaggi di Programmazione

---

*Author*  
Simone Bianco

10 novembre 2023

# Indice

<b>Informazioni e Contatti</b>	<b>1</b>
<b>1 Struttura e Rappresentazione</b>	<b>2</b>
1.1 Algebre induttive . . . . .	2
1.1.1 Lemma di Lambek . . . . .	8
1.2 Strutture dati induttive . . . . .	9
1.2.1 Induzione strutturale . . . . .	11
1.3 Sintassi astratta . . . . .	12
<b>2 Paradigma funzionale</b>	<b>14</b>
2.1 <i>Exp</i> : un semplice linguaggio funzionale . . . . .	14
2.2 Valutazione Eager vs Lazy . . . . .	18
2.3 Scoping Statico vs Dinamico . . . . .	20
2.4 <i>Fun</i> : un linguaggio con funzioni . . . . .	23
2.4.1 <i>Fun</i> in Standard ML . . . . .	28
2.5 Lambda calcolo . . . . .	29
2.5.1 <i>Fun</i> vs Lambda calcolo . . . . .	32
<b>3 Paradigma imperativo</b>	<b>36</b>
3.1 <i>Imp</i> : un semplice linguaggio imperativo . . . . .	36
3.2 <i>All</i> : un linguaggio con procedure . . . . .	40
3.2.1 Semantiche di <i>All</i> . . . . .	42
<b>4 Correttezza dei programmi</b>	<b>46</b>
4.1 Correttezza dei programmi imperativi . . . . .	46
4.1.1 Il metodo delle invarianti . . . . .	46
4.1.2 Logica di Hoare . . . . .	49
4.2 Correttezza dei programmi funzionali . . . . .	55
4.2.1 Ricorsione nei programmi funzionali . . . . .	55

# Informazioni e Contatti

Appunti e riassunti personali raccolti in ambito del corso di *Linguaggi di Programmazione* offerto dal corso di laurea in Informatica dell'Università degli Studi di Roma "La Sapienza".

Ulteriori informazioni ed appunti possono essere trovati al seguente link:

<https://github.com/Exyss/university-notes>. Chiunque si senta libero di segnalare incorrettezze, migliorie o richieste tramite il sistema di Issues fornito da GitHub stesso o contattando in privato l'autore :

- Email: [bianco.simone@outlook.it](mailto:bianco.simone@outlook.it)
- LinkedIn: [Simone Bianco](#)

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se le modifiche siano già state apportate nella versione più recente.

## Prerequisiti consigliati per lo studio:

Apprendimento del materiale relativo al corso *Algebra*.

## Licence:

These documents are distributed under the [GNU Free Documentation License](#), a form of copyleft intended for use on a manual, textbook or other documents. Material licensed under the current version of the license can be used for any purpose, as long as the use meets certain conditions:

- All previous authors of the work must be **attributed**.
- All changes to the work must be **logged**.
- All derivative works must be **licensed under the same license**.
- The full text of the license, unmodified invariant sections as defined by the author if any, and any other added warranty disclaimers (such as a general disclaimer alerting readers that the document may not be accurate for example) and copyright notices from previous versions must be maintained.
- Technical measures such as DRM may not be used to control or obstruct distribution or editing of the document.

# Struttura e Rappresentazione

## 1.1 Algebre induttive

### Definizione 1.1.0.1: Assiomi di Peano

L'insieme dei numeri naturali  $\mathbb{N}$  è definito secondo i seguenti **assiomi di Peano**:

1.  $0 \in \mathbb{N}$
2.  $n \in \mathbb{N} \implies \text{succ}(n) \in \mathbb{N}$ , dove  $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$  è la funzione successore
3.  $\forall n, m \in \mathbb{N}, \text{succ}(n) = \text{succ}(m) \implies n = m$ , ossia  $\text{succ}$  è iniettiva
4.  $\nexists n \in \mathbb{N} \mid \text{succ}(n) = 0$
5.  $\forall S \subseteq \mathbb{N} \mid (0 \in S \wedge (n \in S \implies \text{succ}(n) \in S)) \implies S = \mathbb{N}$

### Proposizione 1.1.0.1: Numeri naturali di Von Neumann

I numeri naturali di Von Neumann, indicati con  $\mathcal{N}$ , definiti come:

$$\begin{aligned}
 0_{\mathcal{N}} &:= \{\} \\
 1_{\mathcal{N}} &:= \{\{\}\} \\
 2_{\mathcal{N}} &:= \{\{\}, \{\{\}\}\} \\
 3_{\mathcal{N}} &:= \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\} \\
 &\dots
 \end{aligned}$$

dove  $\text{succ}_{\mathcal{N}} : \mathcal{N} \rightarrow \mathcal{N} : n \mapsto n \cup \{n\}$ , soddisfano gli assiomi di Peano

*Dimostrazione.*

1.  $0_{\mathcal{N}} \in \mathcal{N}$  per definizione stessa di  $\mathcal{N}$
2.  $n \in \mathcal{N} \implies \text{succ}_{\mathcal{N}}(n) \in \mathcal{N}$  per definizione stessa di  $\text{succ}_{\mathcal{N}}$
3. Siano  $n, m \in \mathcal{N}$  tali che  $n \neq m$ . In tal caso, ne segue automaticamente che:

$$n \neq m \implies n \cup \{n\} \neq m \cup \{m\} \iff \text{succ}_{\mathcal{N}}(n) \neq \text{succ}_{\mathcal{N}}(m)$$

Per contro-nominale, dunque, otteniamo che:

$$\text{succ}_{\mathcal{N}}(n) = \text{succ}_{\mathcal{N}}(m) \implies n = m$$

4. Supponiamo per assurdo che  $\exists n \in \mathbb{N} \mid \text{succ}_{\mathcal{N}}(n) = 0_{\mathcal{N}}$ . In tal caso, avremmo che:

$$\text{succ}(n) = 0_{\mathcal{N}} \iff n \cup \{n\} = 0_{\mathcal{N}} \iff n \cup \{n\} = \{\}$$

ma ciò risulta assurdo poiché implicherebbe che l'insieme  $\{\}$  contenga degli elementi. Di conseguenza, l'unica possibilità è che  $\nexists n \in \mathbb{N} \mid \text{succ}_{\mathcal{N}}(n) = 0_{\mathcal{N}}$

5. Supponiamo per assurdo che  $\exists S \subseteq \mathcal{N} \mid (0_{\mathcal{N}} \in S \wedge (n \in S \implies \text{succ}_{\mathcal{N}}(n) \in S)) \wedge S \neq \mathcal{N}$ . Consideriamo quindi  $\mathcal{N} - S = \{n_1, \dots, n_k\}$ . Per via del secondo assioma, ogni elemento di  $\mathcal{N} - S$  deve avere un proprio successore e un proprio predecessore in  $\mathcal{N}$ .

Poiché per ipotesi si ha che  $n \in S \implies \text{succ}_{\mathcal{N}}(n) \in S$ , ne segue che tutti i predecessori degli elementi in  $\mathcal{N} - S$  non possano essere in  $S$ , poiché altrimenti tali elementi sarebbero in  $S$ . Inoltre, poiché  $\text{succ}_{\mathcal{N}}$  è iniettiva, ne segue che i successori degli elementi in  $\mathcal{N} - S$  non possano essere in  $S$ , poiché esiste già un predecessore in  $S$  per ogni elemento in  $S$ .

Di conseguenza, ogni predecessore ed ogni successore degli elementi di  $\mathcal{N} - S$  deve essere in  $\mathcal{N} - S$  stesso. Consideriamo quindi (per comodità) la seguente catena di successori in  $\mathcal{N} - S$ :

$$n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k \rightarrow n_1$$

Notiamo a questo punto che:

$$\text{succ}_{\mathcal{N}}^k(n_1) = n_1 \implies n_1 \in n_1$$

contraddicendo gli assiomi insiemistici per cui un insieme non possa essere contenuto in se stesso. Di conseguenza, l'unica possibilità è che  $S = \mathcal{N}$

□

**Principio 1.1.0.1: Principio di induzione**

Sia  $P$  una proprietà che vale per  $n = 0$ . Dato  $n \in \mathbb{N}$ , se si verifica che la veridicità di  $P$  per  $n$  implica che  $P$  sia vera anche per  $n + 1$ , allora  $P$  vale per tutto  $\mathbb{N}$ . In simboli, abbiamo che:

$$\forall P ((P(0) \wedge (P(n) \implies P(n+1)))) \implies \forall m \in \mathbb{N} P(m)$$

**Osservazione 1.1.0.1**

Il quinto assioma di Peano è equivalente al principio di induzione, poiché basta considerare  $S \subseteq \mathbb{N}$  come l'insieme degli elementi per cui vale la proprietà desiderata

**Osservazione 1.1.0.2**

Dato  $k \in \mathbb{N}$ , il principio di induzione può essere utilizzato per dimostrare che una proprietà  $P$  valga  $\forall n \in \mathbb{N} \mid n \geq k$ . In altre parole, non è necessario che il principio valga per tutti i naturali a partire da 0.

*Dimostrazione.*

- Definendo una proprietà  $Q$  tale che  $P(n) = Q(n - k)$ , si ha che:

$$\forall n - k \in \mathbb{N} \quad Q(n - k) \iff P(n)$$

dunque applicare il principio di induzione per  $P$  partendo da  $k$  equivale ad applicare il principio di induzione per  $Q$  partendo da 0, rispettando quindi il quinto assioma di Peano

□

**Definizione 1.1.0.2: Insieme unità**

Definiamo come **insieme unità** l'insieme  $\mathbb{1} = \{()\}$ , ossia l'insieme composto da una zerupla

**Definizione 1.1.0.3: Funzione nullaria**

Definiamo una funzione  $f : \mathbb{1} \rightarrow S$ , dunque avente  $\mathbb{1}$  come dominio, come **funzione nullaria** (o funzione costante).

Inoltre, per comodità, indichiamo  $f(x)$  direttamente con  $f$ , poiché  $x = ()$

**Esempio:**

- Data la funzione zero :  $\mathbb{1} \rightarrow \mathbb{N} : x \mapsto 0$ , indichiamo zero( $x$ ) direttamente come zero

**Osservazione 1.1.0.3**

Una funzione nullaria è sempre **iniettiva** in quanto esiste un solo elemento nel dominio.

**Definizione 1.1.0.4: Segnatura di una funzione**

Data una funzione  $f$  definiamo  $f : D \rightarrow C$  come **segnatura di  $f$**  dove  $D$  è il **dominio di  $f$**  e  $C$  è il **codominio di  $f$**

**Definizione 1.1.0.5: Algebra**

Definiamo come **algebra** (o struttura algebrica) una  $n$ -upla  $(A, \gamma_1, \dots, \gamma_n)$  dove  $A$  è un insieme non vuoto, detto **dominio**, e  $\gamma_1, \dots, \gamma_n$  sono delle operazioni definite su  $A$  stesso.

**Esempi:**

- La coppia  $(\mathbb{N}, \text{succ})$  è un'algebra
- La coppia  $(\mathbb{N}, \text{zero})$  è un'algebra

**Definizione 1.1.0.6: Segnatura di un'algebra**

Data un'algebra  $(A, \gamma_1, \dots, \gamma_n)$ , definiamo come **segnatura dell'algebra** l'insieme delle segnature delle operazioni definite su essa

**Definizione 1.1.0.7: Segnature equivalenti**

Date due algebre  $(A, \gamma_1, \dots, \gamma_n)$  e  $(B, \delta_1, \dots, \delta_n)$ , definiamo le segnature di tali algebre come **equivalenti** se per ogni operazione  $\gamma$  definita su  $A$  esiste un'operazione  $\delta$  definita su  $B$  per cui invertendo  $B$  con  $A$  all'interno della segnatura di  $\delta$  si ottiene la segnatura di  $\gamma$

**Esempio:**

- Date le due algebre  $(\mathbb{N}, \text{zero}, \text{succ})$  e  $(\mathcal{N}, \text{zero}_{\mathcal{N}}, \text{succ}_{\mathcal{N}})$ , le segnature di tali algebre sono equivalenti poiché:
  - La segnatura di  $\text{zero} : \mathbb{1} \rightarrow \mathbb{N}$  è equivalente alla segnatura di  $\text{zero}_{\mathcal{N}} : \mathbb{1} \rightarrow \mathcal{N}$
  - La segnatura di  $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$  è equivalente alla segnatura di  $\text{succ}_{\mathcal{N}} : \mathcal{N} \rightarrow \mathcal{N}$

**Definizione 1.1.0.8: Algebra induttiva e Costruttori**

Definiamo l'algebra  $(A, \gamma_1, \dots, \gamma_n)$  come **induttiva** (o **iniziale**) se:

- $\gamma_1, \dots, \gamma_n$  sono iniettive
- $\forall i \neq j \quad \text{im}(\gamma_i) \cap \text{im}(\gamma_j) = \emptyset$ , ossia le immagini delle operazioni sono due a due disgiunte
- $\forall S \subseteq A \quad (\forall a_1, \dots, a_k \in S \quad \gamma_i(a_1, \dots, a_k) \in S) \implies S = A$ , ossia è soddisfatto il principio di induzione per ogni operazione

Inoltre, definiamo  $\gamma_1, \dots, \gamma_n$  come **costruttori di  $A$** .

**Esempi:**

- L'algebra  $(\mathbb{N}, +)$  non è un'algebra induttiva poiché  $+: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  non è iniettiva
- Siano  $B = \{\text{true}, \text{false}\}$  e  $\text{not} : B \rightarrow B : x \mapsto \bar{x}$ .

L'algebra  $(B, \text{not})$  non è un'algebra induttiva poiché:

$$\exists \emptyset \subseteq B \mid (\forall x \in \emptyset \quad \text{not}(x) \in \emptyset) \wedge \emptyset \neq B$$

**Proposizione 1.1.0.2: Algebra induttiva dei naturali**

La tripla  $(\mathbb{N}, \text{zero}, \text{succ})$  è un'algebra induttiva

*Dimostrazione.*

- zero risulta essere iniettiva poiché funzione nullaria, mentre succ risulta essere iniettiva grazie al secondo assioma di Peano
- $\text{im}(\text{zero}) \cap \text{im}(\text{succ}) = \{0\} \cap (\mathbb{N} - \{0\}) = \emptyset$
- Sia  $S \subseteq \mathbb{N}$  tale che  $\forall x \in S \quad \text{zero} \in S$  e  $\text{succ}(x) \in S$ . Preso  $x \in \mathbb{N}$ , possiamo esprimere  $x$  come  $x = \text{succ}(\text{succ}(\dots(\text{zero})))$ .

Di conseguenza, poiché  $S$  è chiuso per zero e succ, otteniamo che:

- $\text{zero} \in S \implies \text{succ}(\text{zero}) \in S$
- $\text{succ}(\text{zero}) \in S \implies \text{succ}(\text{succ}(\text{zero})) \in S$
- ...
- $\text{succ}(\dots(\text{zero})) \in S \implies x = \text{succ}(\text{succ}(\dots(\text{zero}))) \in S$

Di conseguenza, otteniamo che  $A \subseteq S$  e dunque che  $S = A$

□



**Osservazione 1.1.0.4**

La terza condizione necessaria delle algebre induttive è equivalente alla seguente:

$$\nexists S \subsetneq A \mid (S, \gamma_1, \dots, \gamma_n) \text{ è algebra induttiva}$$

**Definizione 1.1.0.9: Omomorfismo**

Date due strutture algebriche  $(A, \gamma_1, \dots, \gamma_k)$  e  $(B, \delta_1, \dots, \delta_k)$  dello stesso tipo, definiamo  $f : A \rightarrow B$  come **omomorfismo** se

$$\forall a_1, \dots, a_n \in A, i \in [1, k] \quad f(\gamma_i(a_1, \dots, a_k)) = \delta_i(f(a_1), \dots, f(a_k))$$

**Esempio:**

- Date le due algebre  $(\mathbb{R}, +)$  e  $(\mathbb{R}_{>0}, \cdot)$ , la funzione  $\exp : \mathbb{R} \rightarrow \mathbb{R}_{>0} : x \mapsto e^x$  è un omomorfismo:

$$\exp(x + y) = e^{x+y} = e^x e^y = \exp(x)\exp(y)$$

**Definizione 1.1.0.10: Isomorfismo**

Definiamo come **isomorfismo** un omomorfismo biiettivo. Inoltre, definiamo due algebre  $(A, \gamma_1, \dots, \gamma_n)$ ,  $(B, \delta_1, \dots, \delta_n)$  come **isomorfe**, indicato con  $A \cong B$ , se esiste un isomorfismo tra loro.

**Osservazione 1.1.0.5**

Data una funzione  $f : A \rightarrow B$ , si ha che:

$$f \text{ è biettiva} \iff \exists f^{-1} : B \rightarrow A$$

(*dimostrazione omessa*)

**Osservazione 1.1.0.6**

Data una funzione  $f : A \rightarrow B$ , si ha che:

$$f \text{ è un isomorfismo} \iff f^{-1} \text{ è un isomorfismo}$$

(*dimostrazione omessa*)

**Esempio:**

- Date le due algebre  $(\mathbb{R}, +)$  e  $(\mathbb{R}_{>0}, \cdot)$ , la funzione  $\exp : \mathbb{R} \rightarrow \mathbb{R}_{>0} : x \mapsto e^x$  è un isomorfismo, poiché  $\exp$  è un omomorfismo e  $\exists \ln : \mathbb{R}_{>0} \rightarrow \mathbb{R} \mid \ln(\exp(x)) = x$ , dunque  $f$  è biettiva

### 1.1.1 Lemma di Lambek

#### Proposizione 1.1.1.1: Segnatura equivalente ad un'algebra induttiva

Data un'algebra induttiva  $(A, \gamma_1, \dots, \gamma_n)$ , per ogni algebra  $(B, \delta_1, \dots, \delta_n)$  con la stessa segnatura di  $A$  si ha che

$$\exists! \text{ omomorfismo } f : A \rightarrow B$$

**Nota:** l'algebra di  $B$  non deve necessariamente essere induttiva

(*dimostrazione omessa*)

#### Lemma 1.1.1.1: Lemma di Lambek (versione ridotta)

Date due algebre induttive  $(A, \gamma_1, \dots, \gamma_n)$  e  $(B, \delta_1, \dots, \delta_n)$  con la stessa segnatura, si ha che  $A \cong B$

*Dimostrazione.*

- Per la proposizione precedente, si ha che:

$$\exists! \text{ omomorfismo } f : A \rightarrow B$$

$$\exists! \text{ omomorfismo } g : B \rightarrow A$$

- Consideriamo quindi la funzione  $g \circ f : A \rightarrow A : x \mapsto g(f(x))$  e verifichiamo che essa sia un omomorfismo

$$g \circ f(x + y) = g(f(x + y)) = g(f(x) + f(y)) = g(f(x)) + g(f(y)) = g \circ f(x) + g \circ f(y)$$

- Notiamo che per ogni algebra esiste sempre l'isomorfismo identità  $\text{id} : A \rightarrow A : x \mapsto x$  e poiché per il lemma precedente esiste necessariamente un unico omomorfismo tra  $A$  e  $A$ , ne segue necessariamente che  $g \circ f = \text{id}$

- Di conseguenza, si ha che

$$g \circ f = \text{id} \iff g = f^{-1} \implies g, f \text{ biettive} \implies g, f \text{ isomorfismi} \implies A \cong B$$

□

#### Esempio:

- Date le due algebre induttive  $(\mathbb{N}, \text{zero}, \text{succ})$  e  $(\mathcal{N}, \text{zero}_{\mathcal{N}}, \text{succ}_{\mathcal{N}})$  sono isomorfe tra loro poiché aventi la stessa segnatura algebrica
- Difatti, come già dimostrato,  $\mathbb{N}$  e  $\mathcal{N}$  sono solamente due modi diversi per rappresentare lo stesso identico concetto algebrico

## 1.2 Strutture dati induttive

### Definizione 1.2.0.1: Insieme delle liste finite

Definiamo  $\text{List}\langle T \rangle$  come l'insieme delle liste finite di elementi di  $T$

**Esempio:**

- Dato  $\text{List}\langle \text{Int} \rangle$ , si ha che  $[3 \rightarrow 5 \rightarrow 1] \in \text{List}\langle \text{Int} \rangle$

### Proposizione 1.2.0.1: Algebra induttiva delle liste finite

La tripla  $(\text{List}\langle T \rangle, \text{empty}, \text{cons})$ , dove:

- $\text{empty} : \mathbb{1} \rightarrow \text{List}\langle T \rangle : x \mapsto []$  è la funzione nullaria che restituisce la **lista vuota**
- $\text{cons} : \text{List}\langle T \rangle \times T \rightarrow \text{List}\langle T \rangle : x, ([x_1 \rightarrow \dots \rightarrow x_n]) \mapsto [x \rightarrow x_1 \rightarrow \dots \rightarrow x_n]$  è la funzione di **costruzione delle liste**

è un'algebra induttiva

*Dimostrazione.*

1. La funzione  $\text{empty}$  risulta essere iniettiva poiché nullaria.

Dati  $\ell_1, \ell_2 \in \text{List}\langle T \rangle$  e  $x_1, x_2 \in T$ , supponiamo che:

$$\text{cons}(y_1, \ell_1) = \text{cons}(y_2, \ell_2) = [x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n]$$

Per definizione stessa di  $\text{cons}$ , si ha che:

$$\text{cons}(y_1, \ell_1) = \text{cons}(y_2, \ell_2) = [x \rightarrow x_1 \rightarrow \dots \rightarrow x_n]$$

$$\implies y_1 = y_2 = x, \ell_1 = \ell_2 = [x_1 \rightarrow \dots \rightarrow x_n]$$

dunque anche  $\text{cons}$  risulta iniettiva

2.  $\text{im}(\text{empty}) \cap \text{im}(\text{cons}) = \{[]\} \cap (\text{List}\langle T \rangle - \{[]\}) = \emptyset$
3. Sia  $S \subseteq \text{List}\langle T \rangle$  tale che  $\forall x \in T, \ell \in \text{List}\langle T \rangle \text{ } \text{cons}(x, \ell) \in S$  e  $\text{empty} \in S$ .

Preso  $\ell := [x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n] \in \text{List}\langle T \rangle$ , possiamo esprimere  $\ell$  come

$$\ell = \text{cons}(x_1, \text{cons}(x_2, \dots \text{cons}(x_n, \text{empty})))$$

Di conseguenza, poiché  $S$  è chiuso per  $\text{cons}$  e  $\text{empty}$  e poiché  $\text{empty} \in S$ , otteniamo che ogni valore della catena sia contenuto in  $S$ , implicando che  $x \in S$  e quindi che  $\text{List}\langle T \rangle \subseteq S$ , concludendo che  $S = \text{List}\langle T \rangle$

□

**Osservazione 1.2.0.1**

La tripla  $(\text{List}\langle T \rangle_\infty, \text{empty}, \text{cons})$ , dove  $\text{List}\langle T \rangle_\infty$  è l'insieme delle liste infinite di elementi di  $T$  **non è un'algebra induttiva**, poiché  $\text{List}\langle T \rangle \subsetneq \text{List}\langle T \rangle_\infty$  e poiché  $(\text{List}\langle T \rangle, \text{empty}, \text{cons})$  è un'algebra induttiva

**Osservazione 1.2.0.2**

Tramite i costruttori di un'algebra induttiva è possibile definire le ulteriori operazioni "aggiuntive" di tale algebra

**Esempio:**

- Data l'algebra induttiva  $(\text{List}\langle T \rangle, \text{empty}, \text{cons})$ , definiamo la seguente operazione

$$\text{concat} : \text{List}\langle T \rangle \times \text{List}\langle T \rangle \rightarrow \text{List}\langle T \rangle$$

dove:

$$\begin{cases} \text{concat}(\text{empty}, \ell) = \ell \\ \text{concat}(\text{cons}(n, \ell), \ell') = \text{cons}(n, \text{concat}(\ell, \ell')) \end{cases}$$

- Ad esempio, in  $\text{List}\langle \text{Int} \rangle$ , abbiamo che:

$$\begin{aligned} \text{concat}([1 \rightarrow 5], [7 \rightarrow 2]) &= \text{concat}(\text{cons}(1, [5], [7 \rightarrow 2])) = \text{cons}(1, \text{concat}([5], [7 \rightarrow 2])) = \\ &= \text{cons}(1, \text{concat}(\text{cons}(5, \text{empty}), [7 \rightarrow 2])) = \text{cons}(1, \text{cons}(5, \text{concat}(\text{empty}, [7 \rightarrow 2]))) = \\ &= \text{cons}(1, \text{cons}(5, [7 \rightarrow 2])) = \text{cons}(1, [5 \rightarrow 7 \rightarrow 2]) = [1 \rightarrow 5 \rightarrow 7 \rightarrow 2] \end{aligned}$$

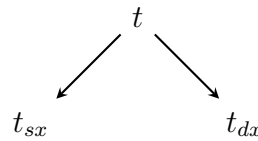
**Definizione 1.2.0.2: Insieme degli alberi binari finiti**

Definiamo **BinTree** come l'insieme degli alberi binari finiti

**Proposizione 1.2.0.2: Algebra induttiva degli alberi binari finiti**

La tripla  $(\text{BinTree}, \text{leaf}, \text{branch})$ , dove:

- $\text{leaf} : \mathbb{1} \rightarrow \text{BinTree} : x \mapsto \circ$  è la funzione nullaria che restituisce una **foglia**
- $\text{branch} : \text{BinTree} \times \text{BinTree} \rightarrow \text{BinTree} : (t_{sx}, t_{dx}) \mapsto t$  è la funzione di **costruzione dei rami**, ossia tale che



è un'algebra induttiva

(*dimostrazione omessa*)

**Esempio:**

- Il seguente albero



corrisponde a:

$$a = \text{branch}(\text{leaf}, \text{branch}(\text{leaf}, \text{leaf}))$$

**1.2.1 Induzione strutturale****Definizione 1.2.1.1: Induzione strutturale**

Definiamo come **induzione strutturale** il metodo dimostrativo generalizzante il principio di induzione e basato sulle proprietà di un'algebra induttiva.

In particolare, viene ipotizzato che una proprietà  $P$  valga per ogni argomento di ogni costruttore dell'algebra e tramite il terzo assioma viene dimostrato che tale proprietà valga per tutti gli elementi dell'algebra stessa

**Teorema 1.2.1.1: Relazione tra nodi e foglie**

Dato  $t \in \text{BinTree}$  avente  $n$  foglie, il numero di nodi di  $t$  è pari a  $2n - 1$

*Dimostrazione per induzione strutturale.*

- Definiamo l'operazione

$$\text{leaves} : \text{BinTree} \rightarrow \mathbb{N} : t \mapsto \text{Numero di foglie in } t$$

dove:

$$\begin{cases} \text{leaves}(\text{leaf}) = 1 \\ \text{leaves}(\text{branch}(b_1, b_2)) = \text{leaves}(b_1) + \text{leaves}(b_2) \end{cases}$$

- Dato  $t \in \text{BinTree}$ , sia  $k$  il numero di nodi di  $t$  e sia  $n = \text{leaves}(t)$

*Caso base.* Se  $t = \text{leaf}$ , allora  $t$  è composto da  $k = 1$  nodi e  $n = \text{leaves}(\text{leaf}) = 1$  foglie. Difatti, si ha che  $k = 1 = 2n - 1$

*Ipotesi induttiva.* Ogni argomento  $t'$  di ogni costruttore possiede  $k' = 2\text{leaves}(t') - 1$  nodi

*Passo induttivo.* Se  $t \neq \text{leaf}$ , allora  $\exists t_1, t_2 \in \text{BinTree} \mid t = \text{branch}(t_1, t_2)$  dove  $t_1$  e  $t_2$  possiedono rispettivamente  $k_1$  e  $k_2$  nodi. Inoltre, si ha che  $k = k_1 + k_2 + 1$

In quanto  $t_1$  e  $t_2$  sono argomenti del costruttore `branch`, per ipotesi induttiva si ha che:

$$\begin{aligned} k &= k_1 + k_2 + 1 = 2\text{leaves}(t_1) - 1 + 2\text{leaves}(t_2) - 1 + 1 = 2(\text{leaves}(t_1) + \text{leaves}(t_2)) - 1 = \\ &= 2(\text{leaves}(\text{branch}(t_1, t_2))) - 1 = 2(\text{leaves}(t)) - 1 \end{aligned}$$

□

## 1.3 Sintassi astratta

### Definizione 1.3.0.1: Linguaggio

Definiamo come **linguaggio** un insieme di stringhe

### Definizione 1.3.0.2: Grammatica

Definiamo come **grammatica** un insieme di regole, dette **termini**, che definiscono come poter manipolare le stringhe di un linguaggio.

La **forma di Backus-Naur** è una notazione utilizzata per descrivere grammatiche ed è definita come:

$$\langle \text{symbol} \rangle ::= \_ \text{expression} \_$$

dove:

- $\langle \text{symbol} \rangle$  è una simbolo non-terminale espresso dalla grammatica
- L'operatore  $::=$  indica che ciò che si trova alla sua sinistra possa essere sostituito con ciò che si trova alla sua destra
- $\langle \_ \text{expression} \_ \rangle$  consiste in una o più sequenze di simboli terminali o non-terminali dove ogni sequenza è separata da una barra verticale (ossia  $|$ ) indicante una scelta possibile per l'operatore  $::=$

### Esempio:

- Consideriamo il linguaggio  $L$  espresso dalla grammatica:

$$M, N ::= 0 \mid 1 \mid \dots \mid M + N \mid M * N$$

Tale grammatica indica che i simboli non-terminali  $M$  e  $N$  possono essere sostituiti con:

- Un numero naturale
- Un'espressione  $M + N$  o  $M * N$  dove  $M$  e  $N$  sono due ulteriori simboli terminali o non-terminali

- Ad esempio, abbiamo che la stringa "5 + 7" sia ben definita dalla grammatica, mentre la stringa "5 + +" non lo sia

### Definizione 1.3.0.3: Sintassi astratta

La **sintassi astratta** di un linguaggio è una definizione induttiva di un insieme  $T$  di termini, permettendo di definire strutture algebriche senza dover necessariamente definire concretamente le sue operazioni

#### Esempio:

- Consideriamo ancora il linguaggio  $L$  definito dalla grammatica

$$M, N ::= 0 \mid 1 \mid \dots \mid M + N \mid M * N$$

- Definiamo quindi la funzione  $\text{eval} : L \rightarrow \mathbb{N}$  in grado di valutare le espressioni del linguaggio:

$$\text{eval}("0") = 0$$

$$\text{eval}("1") = 1$$

...

$$\text{eval}("M + N") = \text{eval}("M") + \text{eval}("N")$$

$$\text{eval}("M * N") = \text{eval}("M") * \text{eval}("N")$$

- Notiamo quindi che la grammatica definisca in modo astratto (ma concretamente tramite  $\text{eval}$ ) le seguenti operazioni:

$$0 : \mathbb{1} \rightarrow \mathbb{N} : x \mapsto 0$$

$$1 : \mathbb{1} \rightarrow \mathbb{N} : x \mapsto 1$$

...

$$\text{plus} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} : (m, n) \mapsto m + n$$

$$\text{times} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} : (m, n) \mapsto m \cdot n$$

- Notiamo però che le operazioni  $\text{plus}$  e  $\text{times}$  non risultano essere né iniettive né con immagini disgiunte. Di conseguenza, la funzione  $\text{eval}$  non ci permette di definire un'algebra induttiva.
- Tuttavia, per tale linguaggio è comunque possibile definire (in qualche modo, ad esempio fissando una precedenza per le operazioni rompendo proprietà come l'associatività e la commutatività) una funzione che possa descrivere un'algebra induttiva.

### Teorema 1.3.0.1: Algebra induttiva dei termini

Dato un linguaggio  $L$  con una sintassi astratta con termini definiti in  $T$ , esiste sempre un'algebra induttiva  $(T, \alpha)$ . Di conseguenza, **tutte le proprietà** di un linguaggio sono dimostrabili tramite l'induzione strutturale sulla sua algebra dei termini.

(*dimostrazione omessa*)

# 2

## Paradigma funzionale

### 2.1 *Exp*: un semplice linguaggio funzionale

#### Definizione 2.1.0.1: Il linguaggio *Exp*

Definiamo come *Exp* il linguaggio rappresentato dalla seguente grammatica:

$$M, N ::= k \mid x \mid M + N \mid \text{let } x = M \text{ in } N$$

dove:

- $k \in \{0, 1, \dots\}$  ossia è una **costante**
- $x \in \text{Var} = \{x, y, z, \dots\}$  ossia è una **variabile**
- $+$  :  $\text{Exp} \times \text{Exp} \rightarrow \text{Exp}$  la quale **somma le due espressioni**
- $\text{let} : \text{Var} \times \text{Exp} \times \text{Exp} \rightarrow \text{Exp}$  la quale **assegna** alla variabile  $x$  l'espressione  $M$  all'interno della **valutazione** di  $N$ . Inoltre,  $x$  prende il nome di variabile locale all'interno di  $N$ .
- $\text{Val} = \{0, 1, \dots\}$  è l'**insieme dei valori** in cui un'espressione può essere valutata

#### Esempi:

- L'espressione  $\text{let } x = 3 \text{ in } x + 1$  indica che la variabile  $x$  assuma valore 3 all'interno della valutazione di  $x + 1$ . Di conseguenza, il risultato della valutazione dell'espressione è 4
- L'espressione  $\text{let } x = 3 \text{ in } 7$  viene valutata come 7
- L'espressione  $\text{let } y = 9 \text{ in } (\text{let } x = (\text{let } y = 2 \text{ in } y + 1) \text{ in } x + y)$  viene valutata come 12 (si consiglia di cercare di capire come le clausole interne sovrascrivano i valori delle clausole esterne. Se ciò risultasse complesso, più avanti verranno forniti strumenti matematici per valutare in modo corretto le clausole *let* annidate)



**Definizione 2.1.0.2: Scope di una variabile**

Data un'espressione e una variabile  $x$ , definiamo come **scope di  $x$**  la porzione la porzione dell'espressione all'interno della quale una variabile può essere riferita, ossia per cui ne è definito il valore.

Una variabile il cui valore non è assegnato in una porzione dell'espressione viene detta **variabile libera**

**Definizione 2.1.0.3: Variabile libera**

Data un'espressione  $expr \in Exp$ , definiamo  $x \in expr$  come **libera** se  $x$  non ha un valore assegnato durante la valutazione di  $expr$ .

**Esempio:**

- L'espressione  $let\ x = (let\ y = 2\ in\ y + 1)\ in\ x + y$  non è coerente con la grammatica di *Exp*, poiché  $y$  non è definito durante la valutazione di  $x + y$ . Di conseguenza, non è possibile valutare tale espressione.

**Proposizione 2.1.0.1: Variabili libere in *Exp***

Dato il linguaggio *Exp*, la funzione

$$free : Exp \rightarrow \mathcal{P}(Var)$$

restituisce l'insieme di tutte le **variabili libere** di un'espressione dove:

$$\begin{cases} free(k) = \emptyset \\ free(x) = \{x\} \\ free(M + N) = free(M) \cup free(N) \\ free(let\ x = M\ in\ N) = free(M) \cup (free(N) - \{x\}) \end{cases}$$

**Nota:**  $\mathcal{P}(Var)$  è l'insieme delle parti di  $Var$ , ossia l'insieme contenente tutti i suoi sottoinsiemi possibili

**Esempio:**

- Riprendendo l'esempio precedente, notiamo che:

$$\begin{aligned} & free(let\ x = (let\ y = 2\ in\ y + 1)\ in\ x + y) = \\ & = free(let\ y = 2\ in\ y + 1) \cup (free(x + y) - \{x\}) = \\ & = free(let\ y = 2\ in\ y + 1) \cup ((free(x) \cup free(y)) - \{x\}) = \\ & = free(let\ y = 2\ in\ y + 1) \cup ((\{x\} \cup \{y\}) - \{x\}) = \\ & = free(let\ y = 2\ in\ y + 1) \cup \{y\} = \end{aligned}$$

$$\begin{aligned}
 &= (\text{free}(2) \cup (\text{free}(y+1) - \{y\})) \cup \{y\} = \\
 &= ((\text{free}(y)) - \{y\}) \cup \{y\} = \\
 &= \{y\}
 \end{aligned}$$

dunque l'espressione è invalutabile

#### Definizione 2.1.0.4: Insieme degli ambienti in *Exp*

Dato il linguaggio *Exp*, definiamo come **insieme degli ambienti di *Exp***, indicato con *Env*, l'insieme delle funzioni parziali (ossia non necessariamente definite su tutto il dominio) che associano ogni variabile al proprio valore:

$$Env = \{f \mid f : Var \xrightarrow{fin} Val\}$$

#### Definizione 2.1.0.5: Concatenazione di ambienti

Dato il linguaggio *Exp*, definiamo l'operazione di **concatenazione di ambienti**, ossia:

$$\cdot : Env \times Env \rightarrow Env$$

dove:

$$(E_1 E_2)(x) = \begin{cases} E_2(x) & \text{se } x \in \text{dom}(E_1) \\ E_1(x) & \text{altrimenti} \end{cases}$$

**Nota:** tale operazione può essere interpretata come una sovrascrittura in  $E_1$  di tutte le variabili definite in  $E_2$

**Esempio:**

- Dati gli ambienti  $E_1 = \{(x, 4), (y, 3)\}$  e  $E_2 = \{(x, 5)\}$ , si ha che

$$(E_1 E_2)(x) = 5$$

$$(E_1 E_2)(y) = 3$$

#### Proposizione 2.1.0.2: Regola di inferenza

Data la proposizione:

$$\text{Premessa 1} \wedge \dots \wedge \text{Premessa n} \implies \text{Conclusione}$$

definiamo come **regola di inferenza** la notazione alternativa:

$$\frac{\text{Premessa 1} \quad \dots \quad \text{Premessa n}}{\text{Conclusione}}$$

**Definizione 2.1.0.6: Semantica operativa di *Exp***

Data la seguente relazione detta **semantica operativa**, ossia:

$$\leadsto \subseteq Env \times Exp \times Val$$

definiamo come **giudizio operativo** la tripla  $(E, M, v) \in \leadsto$  descritta dalla notazione

$$E \vdash M \leadsto v$$

la quale viene letta come "nell'ambiente  $E$ ,  $M$  viene valutato come  $v$ ".

**Proposizione 2.1.0.3: Regole operative di *Exp***

Definiamo come **regole operative** le regole di inferenza che dettano le valutazioni effettuate dalla semantica operativa:

- Per le **costanti** si ha che:

$$\forall E \in Env \quad E \vdash k \leadsto k$$

- Dato  $E \in Env$ , per le **variabili** si ha che:

$$E(x) = v \implies E \vdash x \leadsto v$$

- Dato  $E \in Env$ , per la **somma** si ha che:

$$u = v + v' \implies \frac{E \vdash M \leadsto v \quad E \vdash N \leadsto v'}{E \vdash M + N \leadsto u}$$

- Per l'espressione **let** si ha che:

$$\frac{E \vdash M \leadsto v \quad E\{(x, v)\} \vdash N \leadsto v'}{E \vdash \text{let } x = M \text{ in } N \leadsto v'}$$

**Osservazione 2.1.0.1: Ambiente iniziale**

A meno che non vi siano variabili esternamente assegnate, all'interno di un'espressione l'**ambiente iniziale** corrisponde sempre a  $\emptyset \subseteq Env$ .

**Osservazione 2.1.0.2: Variabili invalutabili**

Dato un ambiente  $E \in Env$ , se  $x \notin \text{dom}(E)$ , ossia se  $x$  non è definita nell'ambiente  $E$ , allora  $x$  è una **variabile libera** e dunque è **invalutabile** in  $E$ , ossia:

$$\nexists v \in Val \text{ t.c. } E \vdash x \leadsto v$$

**Esempio:**

- L'espressione  $x + 4$  è invalutabile, poiché  $x \notin \text{dom}(\emptyset)$ , dunque:

$$\nexists v' \in \text{Val t.c. } v = v' + 1 \wedge \frac{\emptyset \vdash x \rightsquigarrow v' \quad \emptyset \vdash 1 \rightsquigarrow 1}{\emptyset \vdash x + 1 \rightsquigarrow v}$$

- L'espressione  $\text{let } x = 1 \text{ in } x + 4$  è valutabile, poiché  $x \in \text{dom}(\{(x, 1)\})$ , dunque:

$$\frac{\emptyset \vdash 1 \rightsquigarrow 1 \quad \frac{\{(x, 1)\} \vdash x \rightsquigarrow 1 \quad \{(x, 1)\} \vdash 4 \rightsquigarrow 4}{\{(x, 1)\} \vdash x + 1 \rightsquigarrow 5}}{\emptyset \vdash \text{let } x = 1 \text{ in } x + 4 \rightsquigarrow 5}$$

**Definizione 2.1.0.7: Albero di derivazione**

Definiamo come **albero di derivazione** l'albero generato dalla valutazione concatenata di più regole di inferenza.

**Esempio:**

- L'espressione  $\text{let } y = 3 \text{ in } (\text{let } x = 7 \text{ in } x + y)$  viene valutata dal seguente albero di derivazione:

$$\frac{\emptyset \vdash 3 \rightsquigarrow 3 \quad \frac{\{y, 3\} \vdash 7 \rightsquigarrow 7 \quad \frac{\{(y, 3), (x, 7)\} \vdash x \rightsquigarrow 7 \quad \{(y, 3), (x, 7)\} \vdash y \rightsquigarrow 3}{\{(y, 3), (x, 7)\} \vdash x + y \rightsquigarrow 10}}{\{(y, 3)\} \vdash \text{let } x = 7 \text{ in } x + y \rightsquigarrow 10}}{\emptyset \vdash \text{let } y = 3 \text{ in } (\text{let } x = 7 \text{ in } x + y) \rightsquigarrow 10}$$

- Notiamo quindi come, per valutare l'intera espressione, ci basti in realtà valutare i termini "più in alto" dell'albero di derivazione

## 2.2 Valutazione Eager vs Lazy

Consideriamo la seguente espressione per il linguaggio *Exp*:

$$\text{let } x = \sqrt{397^5 + \int_3^{15} y^2 dy + \log_{\sqrt{37}}(479)} \text{ in } 3$$

Notiamo come nonostante l'espressione assegnata ad  $x$  sia di grandi dimensioni, richiedendo un enorme albero di derivazione, la valutazione dell'espressione sia totalmente indipendente da tale valutazione in quanto la variabile  $x$  non venga neanche utilizzata per la valutazione del secondo termine dell'espressione *let*.

Utilizzando le regole di valutazione previste dalla metodologia di valutazione, detta *eager* (trad: *affrettata*), vista nella sezione precedente, andremmo a valutare delle espressioni del tutto inutili.

Una metodologia di valutazione alternativa, detta *lazy*, è costituita da regole operazionali atte al *ritardare* la valutazione dei termini fino a quando non sia strettamente necessario.

### Definizione 2.2.0.1: Valutazione eager

Definiamo una modalità di valutazione come **eager** se la valutazione di una sua espressione viene effettuata non appena essa viene legata ad una variabile, associandone immediatamente il risultato alla variabile stessa.

### Definizione 2.2.0.2: Valutazione lazy

Definiamo una modalità di valutazione come **lazy** se la valutazione di una sua espressione viene effettuata solo quando si richiede il valore di un'espressione che da essa dipende.

### Proposizione 2.2.0.1: Linguaggio *Exp* lazy

L'uso di una valutazione lazy necessita la ridefinizione dell'insieme *Env* e di alcune regole operazionali definite per la valutazione eager:

- L'insieme *Env* viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Exp\}$$

- Dato  $E \in Env$ , per le variabili si ha che:

$$x \in dom(E) \wedge E(x) = M \implies \frac{E \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v}$$

- Per l'espressione *let* si ha che:

$$\frac{E\{(x, M)\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$

### Osservazione 2.2.0.1

È necessario puntualizzare che non sempre la valutazione lazy sia più ottimale della eager

**Esempio:**

- Consideriamo la seguente espressione

$$let\ x = M\ in\ x + x$$

- Utilizzando la valutazione eager otteniamo il seguente albero di derivazione:

$$\frac{\frac{\dots}{\emptyset \vdash M \rightsquigarrow v'} \quad \frac{\{(x, v')\} \vdash x \rightsquigarrow v' \quad \{(x, v')\} \vdash x \rightsquigarrow v'}{\{(x, v')\} \vdash x + x \rightsquigarrow v}}{\emptyset \vdash \text{let } x = M \text{ in } x + x \rightsquigarrow v}$$

dove  $v = v' + v'$

- Utilizzando la valutazione lazy, invece, otteniamo il seguente albero di derivazione:

$$\frac{\frac{\dots}{\{(x, M)\} \vdash M \rightsquigarrow v'} \quad \frac{\{(x, M)\} \vdash M \rightsquigarrow v'}{\{(x, M)\} \vdash x \rightsquigarrow v'} \quad \frac{\dots}{\{(x, M)\} \vdash M \rightsquigarrow v'} \quad \frac{\{(x, M)\} \vdash M \rightsquigarrow v'}{\{(x, M)\} \vdash x \rightsquigarrow v'}}{\{(x, M)\} \vdash x + x \rightsquigarrow v} \\ \frac{}{\emptyset \vdash \text{let } x = M \text{ in } x + x \rightsquigarrow v}$$

dove  $v = v' + v'$

- Notiamo quindi che l'espressione  $M$  venga valutata una sola volta nella valutazione eager ma due volte nella valutazione lazy

## 2.3 Scoping Statico vs Dinamico

Consideriamo la seguente espressione:

$$\text{let } x = 3 \text{ in } (\text{let } y = x \text{ in } (\text{let } x = 7 \text{ in } y + x))$$

Prima di tutto, valutiamo tale espressione tramite valutazione eager:

$$\frac{\frac{\frac{\frac{\frac{\emptyset \vdash 3 \rightsquigarrow 3}{\{(x, 3)\} \vdash x \rightsquigarrow 3} \quad \frac{\frac{E \vdash 7 \rightsquigarrow 7 \quad \frac{E\{(x, 7)\} \vdash y \rightsquigarrow 3 \quad E\{(x, 7)\} \vdash x \rightsquigarrow 7}{E\{(x, 7)\} \vdash y + x \rightsquigarrow 10}}{\{(x, 3), (y, 3)\} \vdash \text{let } x = 7 \text{ in } y + x \rightsquigarrow 10}}{\{(x, 3)\} \vdash \text{let } y = x \text{ in } (\text{let } x = 7 \text{ in } y + x) \rightsquigarrow 10}}{\emptyset \vdash \text{let } x = 3 \text{ in } (\text{let } y = x \text{ in } (\text{let } x = 7 \text{ in } y + x)) \rightsquigarrow 10}}$$

dove  $E := \{(x, 3), (y, 3)\}$

Valutiamo ora invece tale espressione utilizzando una valutazione lazy:

$$\frac{\frac{\frac{\frac{\frac{E\{(x, 7)\} \vdash 7 \rightsquigarrow 7}{E\{(x, 7)\} \vdash x \rightsquigarrow 7} \quad \frac{E\{(x, 7)\} \vdash 7 \rightsquigarrow 7}{E\{(x, 7)\} \vdash y \rightsquigarrow 7}}{\frac{E\{(x, 7)\} \vdash x \rightsquigarrow 7 \quad E\{(x, 7)\} \vdash y \rightsquigarrow 7}{E\{(x, 7)\} \vdash y + x \rightsquigarrow 14}}{\{(x, 3), (y, x)\} \vdash \text{let } x = 7 \text{ in } y + x \rightsquigarrow 14}}{\{(x, 3)\} \vdash \text{let } y = x \text{ in } (\text{let } x = 7 \text{ in } y + x) \rightsquigarrow 14}}{\emptyset \vdash \text{let } x = 3 \text{ in } (\text{let } y = x \text{ in } (\text{let } x = 7 \text{ in } y + x)) \rightsquigarrow 14}$$

dove  $E := \{(x, 3), (y, x)\}$

Notiamo quindi che le due valutazioni abbiano prodotto un risultato diverso. Tuttavia, vorremmo che le due valutazioni siano differenti solo a livello "*implementativo*", ossia che venga solo ritardata la valutazione dei termini. Difatti, tale problematica non è dovuta alla metodologia di valutazione utilizzata ma bensì dal tipo di *scoping*.

#### Definizione 2.3.0.1: Scoping statico

Definiamo un linguaggio come linguaggio a **scoping statico** se durante la valutazione di un'espressione viene utilizzato l'ambiente definito al tempo in cui viene interpretata (ma non valutata) l'espressione stessa.

#### Definizione 2.3.0.2: Scoping dinamico

Definiamo un linguaggio come linguaggio a **scoping dinamico** se durante la valutazione di un'espressione viene utilizzato l'ambiente definito al tempo di valutazione stesso.

Difatti, nell'esempio precedente ci troviamo in due situazioni:

- Nella valutazione eager, la variabile  $y$  viene valutata con l'ambiente  $\{(x, 3), (y, x)\}$  definito al tempo in cui viene interpretata l'espressione  $let\ y = x\ in\ \dots$  (scoping *statico*)
- Nella valutazione lazy, la variabile  $y$  viene valutata con l'ambiente  $\{(x, 3), (y, x), (x, 7)\}$  definito al tempo della sua valutazione (scoping *dinamico*)

Per tanto, è necessario precisare che le due precedenti versioni viste del linguaggio *Exp* siano rispettivamente la versione **eager statica** e la versione **Exp lazy dinamica**.

#### Proposizione 2.3.0.1: Linguaggio *Exp* lazy statico

L'uso di una semantica lazy statica necessita la ridefinizione dell'insieme  $Env$  e di alcune regole operazionali definite per la semantica lazy dinamica:

- L'insieme  $Env$  viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Exp \times Env\}$$

- Dato  $E \in Env$ , per le variabili si ha che:

$$x \in dom(E) \wedge E(x) = (M, E') \implies \frac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v}$$

- Per l'espressione  $let$  si ha che:

$$\frac{E\{(x, (M, E))\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$





## 2.4 *Fun*: un linguaggio con funzioni

### Definizione 2.4.0.1: Il linguaggio *Fun*

Definiamo come *Fun* il linguaggio rappresentato dalla seguente grammatica:

$$M, N ::= k \mid x \mid M + N \mid \text{let } x = M \text{ in } N \mid \text{fn } x \Rightarrow M \mid MN$$

dove:

- $k \in \{0, 1, \dots\}$  ossia è una **costante**
- $x \in \text{Var} = \{x, y, z, \dots\}$  ossia è una **variabile**
- $+$  :  $\text{Fun} \times \text{Fun} \rightarrow \text{Fun}$  la quale **somma le due espressioni**
- $\text{let}$  :  $\text{Var} \times \text{Fun} \times \text{Fun} \rightarrow \text{Fun}$  la quale **assegna** alla variabile  $x$  l'espressione  $M$  all'interno della **valutazione** di  $N$ . Inoltre,  $x$  prende il nome di variabile locale all'interno di  $N$
- $\text{fn}$  :  $\text{Var} \times \text{Fun} \rightarrow \text{Fun}$  la quale restituisce una **funzione** avente un parametro il quale influenza l'espressione valutata dalla funzione
- Data l'espressione  $\text{fn } x \Rightarrow M$ , definiamo la coppia  $(x, M) \in \text{Var} \times \text{Fun}$  come **chiusura** di tale espressione
- $\cdot$  :  $\text{Fun} \times \text{Fun} \rightarrow \text{Fun}$  la quale **applica** il termine sinistro al termine destro. In particolare, è necessario che il termine sinistro sia una funzione
- $\text{Val} = \{0, 1, \dots\} \cup (\text{Var} \times \text{Fun})$  è l'**insieme dei valori** in cui un'espressione può essere valutata, ossia costanti e chiusure

### Esempi:

- L'espressione  $(\text{fn } x \Rightarrow x + 1) 7$  viene valutata come 8, poiché la funzione sinistra  $\text{fn } x \Rightarrow x + 1$  viene applicata al termine destro 7 (dunque 7 viene utilizzato come argomento della funzione per il parametro  $x$ )
- L'espressione  $(\text{fn } x \Rightarrow x 3) 7$  è invalutabile, poiché l'argomento 7 viene passato come parametro  $x$  della funzione, ma all'interno di quest'ultima non è possibile valutare  $x 3$  visto che 7 non è applicabile a 3
- L'espressione  $(\text{fn } x \Rightarrow x 3)(\text{fn } x \Rightarrow x + 1)$  viene valutata come 4, poiché l'argomento  $\text{fn } x \Rightarrow x + 1$  viene passato come parametro  $x$  della funzione  $\text{fn } x \Rightarrow x 3$ , per poi valutare l'applicazione  $x 3$  passando l'argomento 3 come parametro per la funzione contenuta in  $x$  (ossia  $\text{fn } x \Rightarrow x + 1$ ).

Informalmente, possiamo dire che:

$$(\text{fn } x \Rightarrow x 3)(\text{fn } x \Rightarrow x + 1) \longrightarrow (\text{fn } x \Rightarrow x + 1) 3 \longrightarrow 4$$

**Osservazione 2.4.0.1**

Nel caso in cui si abbia un'espressione con doppio operatore di applicazione  $MNL$ , essa verrà valutata come  $(MN)L$

**Esempio:**

- Le due espressioni  $(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x + 1)\ 7$  e  $[(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x + 1)]\ 7$  sono equivalenti

**Definizione 2.4.0.2: Curryficazione**

Definiamo come **curryficazione** la contrazione sintattica  $fn\ x_1 x_2 \dots x_n \Rightarrow M$  equivalente alla seguente espressione:

$$fn\ x_1 \Rightarrow (fn\ x_2 \Rightarrow \dots (fn\ x_n \Rightarrow M) \dots)$$

**Esempi:**

- L'uncurryficazione dell'espressione  $(fn\ xy \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x + 1)$  corrisponde a:

$$(fn\ x \Rightarrow fn\ y \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x + 1)$$

e viene pertanto valutata come 8. Difatti, informalmente, possiamo dire che:

$$(fn\ x \Rightarrow fn\ y \Rightarrow yx)\ 7\ (fn\ x \Rightarrow x + 1) \longrightarrow (fn\ y \Rightarrow y\ 7)(fn\ x \Rightarrow x + 1) \longrightarrow 8$$

**Osservazione 2.4.0.2**

Trattandosi di un'estensione del linguaggio *Exp*, il linguaggio *Fun* **eredita le regole operazionali** delle semantiche di *Exp*

**Proposizione 2.4.0.1: Linguaggio *Fun* eager dinamico**

La semantica eager dinamica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme *Env* viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Val\}$$

- Dato  $E \in Env$ , per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M)$$

- Dato  $E \in Env$ , per le applicazioni si ha che:

$$\frac{E \vdash M \rightsquigarrow (x, L) \quad E \vdash N \rightsquigarrow v' \quad E\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

**Proposizione 2.4.0.2: Linguaggio *Fun* eager statico**

La semantica eager statica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme *Env* viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Val \times Env\}$$

- Dato  $E \in Env$ , per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M, E)$$

- Dato  $E \in Env$ , per le applicazioni si ha che:

$$\frac{E \vdash M \rightsquigarrow (x, L, E') \quad E \vdash N \rightsquigarrow v' \quad E'\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

**Lemma 2.4.0.1**

A differenza del linguaggio *Exp*, per la sua estensione *Fun* si ha che:

$$Fun\ eager\ dinamico \not\equiv Fun\ eager\ statico$$

*Dimostrazione.*

- Consideriamo l'espressione  $let\ x = 7\ in\ ((fn\ y \Rightarrow let\ x = 3\ in\ yx)(fn\ z \Rightarrow x))$
- Utilizzando la semantica eager dinamica, l'albero di derivazione corrisponde a:

$$\begin{array}{c} (*) \quad \frac{E' \vdash 3 \rightsquigarrow 3 \quad \frac{E'' \vdash y \rightsquigarrow (z, x) \quad E'' \vdash x \rightsquigarrow 3 \quad E''\{(z, 3)\} \vdash x \rightsquigarrow 3}{E'' \vdash yx \rightsquigarrow 3}}{E' \vdash M \rightsquigarrow 3} \\ \frac{\emptyset \vdash 7 \rightsquigarrow 7 \quad \frac{E \vdash fn\ y \Rightarrow M \rightsquigarrow (y, M) \quad E \vdash fn\ z \Rightarrow x \rightsquigarrow (z, x) \quad (*)}{E \vdash (fn\ y \Rightarrow M)(fn\ z \Rightarrow x) \rightsquigarrow 3}}{\emptyset \vdash let\ x = 7\ in\ ((fn\ y \Rightarrow M)(fn\ z \Rightarrow x)) \rightsquigarrow 3} \end{array}$$

dove  $M := let\ x = 3\ in\ yx$ ,  $E := \{(x, 7)\}$ ,  $E' := E\{(y, (z, x))\}$  e  $E'' := E'\{(x, 3)\}$

- Utilizzando la semantica eager statica, invece, l'albero di derivazione corrisponde a:

$$(*) \quad \frac{E' \vdash 3 \rightsquigarrow 3 \quad \frac{E'' \vdash y \rightsquigarrow (z, x, E) \quad E'' \vdash x \rightsquigarrow 3 \quad E\{(z, 3)\} \vdash x \rightsquigarrow 7}{E'' \vdash yx \rightsquigarrow 7}}{E' \vdash M \rightsquigarrow 7}$$

$$\frac{\emptyset \vdash 7 \rightsquigarrow 7 \quad \frac{E \vdash fn\ y \Rightarrow M \rightsquigarrow (y, M, E) \quad E \vdash fn\ z \Rightarrow x \rightsquigarrow (z, x, E) \quad (*)}{E \vdash (fn\ y \Rightarrow M)(fn\ z \Rightarrow x) \rightsquigarrow 7}}{\emptyset \vdash let\ x = 7\ in\ ((fn\ y \Rightarrow M)(fn\ z \Rightarrow x)) \rightsquigarrow 7}$$

dove  $M := let\ x = 3\ in\ yx$ ,  $E := \{(x, 7)\}$ ,  $E' := E\{(y, (z, x, E))\}$  e  $E'' := E'\{(x, 3)\}$

- Poiché l'espressione restituisce due valutazioni diverse, le due semantiche non sono equivalenti

□

### Proposizione 2.4.0.3: Linguaggio *Fun* lazy dinamico

La semantica lazy dinamica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme  $Env$  viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Fun\}$$

- Dato  $E \in Env$ , per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M)$$

- Dato  $E \in Env$ , per le applicazioni si ha che:

$$\frac{E \vdash M \rightsquigarrow (x, L) \quad E'\{(x, N)\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

### Proposizione 2.4.0.4: Linguaggio *Fun* lazy statico

La semantica lazy statica del linguaggio *Fun* prevede l'aggiunta di alcune regole operazionali:

- L'insieme  $Env$  viene ridefinito come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Fun \times Env\}$$

- Dato  $E \in Env$ , per le funzioni si ha che:

$$E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M, E)$$

- Dato  $E \in Env$ , per le applicazioni si ha che:

$$\frac{E \vdash M \rightsquigarrow (x, L, E') \quad E'\{(x, N, E)\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

**Osservazione 2.4.0.3**

Come per il linguaggio *Exp*, per la sua estensione *Fun* si ha che:

$$Fun \text{ lazy dinamico} \neq Fun \text{ lazy statico}$$

**Definizione 2.4.0.3: Espressione  $\omega$** 

Dato il linguaggio *Fun*, definiamo come **espressione omega**, indicata con  $\omega$ , la seguente espressione:

$$\omega := (fn\ x \Rightarrow xx)(fn\ x \Rightarrow xx)$$

In particolare, l'espressione  $\omega$  è **invalutabile per qualsiasi semantica**

**Esempio:**

- Analizziamo l'albero di derivazione di  $\omega$  utilizzando una semantica eager statica:

$$\begin{array}{c}
 (*) \quad \emptyset \vdash x \rightsquigarrow (x, xx, \emptyset) \quad \emptyset \vdash x \rightsquigarrow (x, xx, \emptyset) \quad \frac{(*)}{(x, \{(x, xx, \emptyset)\}) \vdash xx \rightsquigarrow v} \\
 \hline
 \emptyset \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \emptyset) \quad \emptyset \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \emptyset) \quad \frac{(*)}{(x, \{(x, xx, \emptyset)\}) \vdash xx \rightsquigarrow v} \\
 \hline
 \emptyset \vdash (fn\ x \Rightarrow xx)(fn\ x \Rightarrow xx) \rightsquigarrow v
 \end{array}$$

- Notiamo quindi che affinché la valutazione del termine  $(x, \{(x, xx, \emptyset)\}) \vdash xx \rightsquigarrow v$  richieda che esso stesso venga valutato, creando così un albero di derivazione infinito.

**Lemma 2.4.0.2**

Dato il linguaggio *Fun*, si ha che:

$$Fun \text{ eager statico} \neq Fun \text{ lazy statico}$$

$$Fun \text{ eager dinamico} \neq Fun \text{ lazy dinamico}$$

*Dimostrazione.*

- Consideriamo l'espressione *let*  $x = \omega$  *in* 42. Utilizzando una semantica eager (statica o dinamica), verrebbe richiesta immediatamente la valutazione del termine  $\omega$ , il quale tuttavia è invalutabile. Utilizzando una semantica lazy (statica o dinamica), invece, il termine  $\omega$  non verrà mai valutato, restituendo 42 come risultato.

□

**Teorema 2.4.0.1: Equivalenze semantiche di *Fun***

Dato il linguaggio *Fun*, **non esistono due semantiche equivalenti**

**Proposizione 2.4.0.5: Variabili libere in *Fun***

Dato il linguaggio *Fun*, la funzione  $\text{free} : \text{Fun} \rightarrow \mathcal{P}(\text{Var})$  è definita come:

$$\begin{cases} \text{free}(k) = \emptyset \\ \text{free}(x) = \{x\} \\ \text{free}(M + N) = \text{free}(M) \cup \text{free}(N) \\ \text{free}(\text{let } x = M \text{ in } N) = \text{free}(M) \cup (\text{free}(N) - \{x\}) \\ \text{free}(\text{fn } x \Rightarrow M) = \text{free}(M) - \{x\} \\ \text{free}(MN) = \text{free}(M) \cup \text{free}(N) \end{cases}$$

**2.4.1 *Fun* in Standard ML**

La grammatica prevista dal linguaggio *Fun* mostrato fino ad ora è utilizzabile all'interno del **linguaggio SML (Standard Model Language)**, il quale prevede una sintassi leggermente diversa:

- L'operatore  $\text{let } x = M \text{ in } N$  corrisponde a `let val x = M in N end`
- L'operatore  $\text{fn } x \Rightarrow M$  corrisponde a `fn x => M;`
- L'operatore  $MN$  corrisponde a `MN` (potrebbe essere necessario introdurre uno spazio tra `M` ed `N` affinché l'interprete riesca a distinguere i due termini)
- La **curryficazione non è prevista**
- L'espressione va terminata da un **punto e virgola**
- La semantica utilizzata è **eager statica**

Ad esempio, l'espressione:

$$\text{let } x = 7 \text{ in } ((\text{fn } y \Rightarrow \text{let } x = 3 \text{ in } yx)(\text{fn } z \Rightarrow x))$$

corrisponde al comando:

```
let val x = 7 in (fn y => let val x = 3 in y x end) end;
```

Inoltre, il linguaggio SML permette di assegnare variabili, alle quali possono essere assegnate anche funzioni. Ad esempio, definendo:

```
val id = fn x => x;
```

il seguente comando restituisce 7:

```
id 7;
```

Per utilizzare il linguaggio SML, si consiglia l'uso del programma `smlnj` o dell'emulatore online `SOSML`.

## 2.5 Lambda calcolo

### Definizione 2.5.0.1: Lambda calcolo

Il **lambda calcolo** è un sistema formale in logica matematica per esprimere il calcolo basato sull'**astrazione** e l'applicazione di **funzioni**.

Nella forma più semplice di lambda calcolo, i termini sono costruiti utilizzando solo le seguenti regole:

- Una **variabile** è rappresentata da un carattere (es:  $x$ )
- Una **funzione** è rappresentata da una **lambda astrazione**, ossia una stringa composta dal simbolo  $\lambda$  seguito dai parametri della funzione separati con un punto dal corpo della funzione stessa (es:  $\lambda x.M$ )
- L'**applicazione** di una funzione  $M$  ad un argomento  $N$  viene rappresentata come  $M N$

#### Esempi:

- La lambda astrazione  $\lambda x.x + 1$  corrisponde alla funzione  $f(x) = x + 1$
- La lambda astrazione  $\lambda xy.x + y$  corrisponde alla funzione  $f(x, y) = x + y$
- La lambda astrazione  $(\lambda x.x) 3$  corrisponde all'applicazione della funzione  $f(x) = x$  all'argomento 3, restituendo quindi 3
- La lambda astrazione  $(\lambda x.x)(\lambda x.x)$  restituisce  $\lambda x.x$
- La lambda astrazione  $\lambda xy.x(xy)$  applica due volte sull'argomento  $y$  la funzione  $x$  passata anch'essa come argomento

### Osservazione 2.5.0.1: Curryficazione in lambda calcolo

La lambda astrazione  $\lambda x_1 \dots x_n.M$  è la contrazione sintattica della seguente lambda astrazione:

$$\lambda x_1. \dots \lambda x_n.M$$

### Definizione 2.5.0.2: Sostituzione

Definiamo come **sostituzione**, indicata con  $M[N/x]$ , l'operazione tramite cui all'interno di un'espressione  $M$  tutte le occorrenze di una variabile  $x$  vengono rimpiazzate con il termine  $N$

#### Esempi:

- La sostituzione  $(xy)[\lambda z.z/x]$  corrisponde a  $((\lambda z.z)y)$
- La sostituzione  $(fn\ x \Rightarrow xy)[x/y]$  corrisponde a  $(fn\ x \Rightarrow xx)$

**Osservazione 2.5.0.2: Cattura di variabili**

L'operazione di sostituzione potrebbe legare una variabile precedentemente libera o viceversa. Tale fenomeno viene detto **cattura di variabili** ed è necessario accertarsi che esso non si verifichi affinché la sostituzione sia corretta

**Esempio:**

- L'espressione  $(\lambda y.M)[N/x]$  è equivalente all'espressione  $\lambda y.(M[N/x])$  solo se  $y \notin \text{free}(N)$ . Difatti, la sostituzione  $(\lambda y.x)[y/x]$  risulta essere "scorretta" in quanto  $(\lambda y.y)$  ha una valutazione differente rispetto all'espressione originale

**Definizione 2.5.0.3: Alfa conversione**

Definiamo come **alfa conversione**, indicata con  $\xrightarrow{\alpha}$ , la regola secondo cui all'interno di una lambda astrazione  $\lambda x.M$  ogni occorrenza della variabile  $x$  (incluso il parametro) possa essere rimpiazzata dalla variabile  $y$ :

$$\lambda x.M \xrightarrow{\alpha} \lambda y.(M[y/x])$$

**Esempi:**

- Data la lambda astrazione  $\lambda x.(xy)$ , si ha che:

$$\lambda x.xy \xrightarrow{\alpha} \lambda z.zy$$

- Data la lambda astrazione  $\lambda x.x(\lambda z.zw)$ , si ha che:

$$\lambda x.x(\lambda z.zw) \xrightarrow{\alpha} \lambda z.z(\lambda z.zw)$$

**Definizione 2.5.0.4: Alfa equivalenza**

Due lambda astrazioni  $\lambda x.M$  e  $\lambda y.N$  vengono dette **alfa equivalenti**, indicato con  $\equiv$ , se:

$$\lambda x.M \equiv \lambda y.N \iff \lambda x.M \xrightarrow{\alpha} \lambda y.N \wedge \lambda y.N \xrightarrow{\alpha} \lambda x.M$$

**Esempi:**

- Date le due lambda astrazioni  $\lambda x.(xy)$  e  $\lambda z.zy$ , si ha che:

$$\lambda x.xy \xrightarrow{\alpha} \lambda z.zy \wedge \lambda z.zy \xrightarrow{\alpha} \lambda x.xy \implies \lambda x.xy \equiv \lambda z.zy$$

- Date le due lambda astrazioni  $\lambda x.x(\lambda z.zw)$  e  $\lambda z.z(\lambda z.zw)$ , si ha che:

$$\lambda x.x(\lambda z.zw) \xrightarrow{\alpha} \lambda z.z(\lambda z.zw)$$

$$\lambda z.z(\lambda z.zw) \not\xrightarrow{\alpha} \lambda x.x(\lambda z.zw)$$

dunque ne concludiamo che:

$$\lambda x.x(\lambda z.zw) \not\equiv \lambda z.z(\lambda z.zw)$$



**Definizione 2.5.0.5: Beta conversione**

Definiamo come **beta conversione** (o *beta riduzione*), indicata con  $\xrightarrow{\beta}$ , la regola secondo cui all'interno di una lambda espressione  $(\lambda x.M)N$  ogni occorrenza della variabile  $x$  all'interno di  $M$  possa essere rimpiazzata dal termine  $N$ :

$$(\lambda x.M)N \xrightarrow{\beta} M[N/x]$$

**Osservazione 2.5.0.3**

La beta riduzione corrisponde esattamente ad singolo **passo computazionale**

**Esempio:**

- Data la lambda espressione  $(\lambda x.xy)(\lambda z.z)$ , si ha che:

$$(\lambda x.xy)(\lambda z.z) \xrightarrow{\beta} (\lambda z.z)y \xrightarrow{\beta} y$$

**Osservazione 2.5.0.4**

La beta riduzione utilizza implicitamente la **valutazione lazy**

**Esempio:**

- Data la lambda espressione  $(\lambda x.7)\omega$ , si ha che:

$$(\lambda x.7)\omega \xrightarrow{\beta} 7$$

dunque la valutazione è necessariamente lazy, poiché altrimenti il termine  $\omega$  sarebbe stato valutato (il quale ricordiamo essere invalutabile)

**Definizione 2.5.0.6: Eta conversione**

Definiamo come **eta conversione**, indicata con  $\xrightarrow{\eta}$ , la regola secondo cui la lambda espressione  $(\lambda x.Mx)$  possa essere rimpiazzata con il termine  $M$  solo se  $x \notin \text{free}(M)$ :

$$x \notin \text{free}(M) \implies \lambda x.Mx \xrightarrow{\eta} M$$

**Esempi:**

- Consideriamo la lambda espressione  $\lambda x.(\lambda y.y)x$ .
- Poiché:

$$\text{free}(\lambda y.y) = \{\text{free}(y) - \{y\}\} = \{y\} - \{y\} = \emptyset \implies x \notin \text{free}(\lambda y.y)$$

è possibile applicare l'eta conversione:

$$\lambda x.(\lambda y.y)x \xrightarrow{\eta} \lambda y.y$$

### 2.5.1 *Fun* vs Lambda calcolo

Avendo trattato le componenti principali del lambda calcolo, possiamo rappresentare quest'ultimo tramite la seguente grammatica:

$$M, N ::= x \mid fn\ x \Rightarrow M \mid MN$$

notiamo come il linguaggio *Fun* corrisponda ad un **sovra-linguaggio** del lambda calcolo stesso. Difatti, essendo il lambda calcolo già **turing completo**, alcuni termini del linguaggio *Fun* risultano "*ridondanti*".

In particolare, le seguenti due espressioni:

$$let\ x = M\ in\ N \qquad (fn\ x \Rightarrow N)M$$

risultano essere **operativamente equivalenti**, ossia vengono sempre valutate nello stesso risultato indipendentemente dalla semantica utilizzata (sebbene esse differiscano in termini di "implementazione" delle loro regole operazionali, dunque non sono effettivamente la stessa espressione).

#### Osservazione 2.5.1.1

La lambda astrazione  $\lambda x_1. \dots \lambda x_n. M$ , corrisponde all'espressione:

$$fn\ x_1 \dots x_n \Rightarrow M$$

In modo analogo a Von Neumann, il matematico Church diede una propria definizione alternativa dei **numeri naturali**: il numero  $n \in \mathbb{N}$  corrisponde all'applicazione per  $n$  volte di un'operazione  $x$  su un valore  $y$ .

In particolare, notiamo che tale definizione data da Church possa essere espressa in termini di **lambda calcolo**. Ad esempio, il numero naturale 3 corrisponderà alla lambda astrazione  $\lambda xy. x(x(xy))$

#### Proposizione 2.5.1.1: Numeri naturali di Church

I numeri naturali di Church, indicati con  $\mathcal{N}_\lambda$ , definiti come:

$$0_{\mathcal{N}_\lambda} := \lambda xy. y$$

$$1_{\mathcal{N}_\lambda} := \lambda xy. xy$$

$$2_{\mathcal{N}_\lambda} := \lambda xy. x(xy)$$

$$3_{\mathcal{N}_\lambda} := \lambda xy. x(x(xy))$$

...

dove  $\text{succ}_{\mathcal{N}_\lambda} : \mathcal{N}_\lambda \rightarrow \mathcal{N}_\lambda : n \mapsto n \cup \{n\}$ , soddisfano gli assiomi di Peano

(*dimostrazione omessa*)

Utilizzando la definizione di Church dei numeri naturali, è possibile definire un modello di calcolo **interamente basato sul lambda calcolo** dove ogni operazione possibile è definibile in termini di lambda astrazioni che lavorano sui numeri di Church (i quali a loro volta sono delle lambda astrazioni).

Di conseguenza, potremmo effettivamente ridurre la grammatica dell'intero linguaggio *Fun* in quella del lambda calcolo.

Procediamo quindi definendo i numeri di Church all'interno del linguaggio *Fun*:

- **zero** :=  $fn\ x \Rightarrow fn\ y \Rightarrow y$  oppure  $fn\ xy \Rightarrow y$
- **one** :=  $fn\ x \Rightarrow fn\ y \Rightarrow xy$  oppure  $fn\ xy \Rightarrow xy$
- **two** :=  $fn\ x \Rightarrow fn\ y \Rightarrow x(xy)$  oppure  $fn\ xy \Rightarrow x(xy)$
- **three** :=  $fn\ x \Rightarrow fn\ y \Rightarrow x(x(xy))$  oppure  $fn\ xy \Rightarrow x(x(xy))$
- ...

Definiamo inoltre una funzione **eval** in grado di convertire un numero di Church nel suo equivalente nei numeri naturali:

$$\mathbf{eval} := fn\ z \Rightarrow z(fn\ x \Rightarrow x + 1)\ 0$$

Ad esempio, l'espressione **eval two** viene valutata come:

$$\begin{aligned} \mathbf{eval\ two} &\xrightarrow{\beta} \\ \{fn\ z \Rightarrow z(fn\ x \Rightarrow x + 1)\ 0\}[fn\ x \Rightarrow fn\ y \Rightarrow x(xy)] &\xrightarrow{\beta} \\ \{[fn\ x \Rightarrow fn\ y \Rightarrow x(xy)](fn\ x \Rightarrow x + 1)\ 0\} &\xrightarrow{\beta} \\ \{[fn\ y \Rightarrow (fn\ x \Rightarrow x + 1)((fn\ x \Rightarrow x + 1)y)]\ 0\} &\xrightarrow{\beta} \\ [(fn\ x \Rightarrow x + 1)\{(fn\ x \Rightarrow x + 1)\ 0\}] &\xrightarrow{\beta} \\ [(fn\ x \Rightarrow x + 1)\ 1] &\xrightarrow{\beta} \\ 2 \end{aligned}$$

A questo punto, definiamo la funzione **succ** che restituisce il successore del numero di Church dato in input:

$$\mathbf{succ} := fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(xy))$$

Ad esempio, l'espressione **succ one** viene valutata come:

$$\begin{aligned} \mathbf{succ\ one} &\xrightarrow{\beta} \\ [fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(xy))](fn\ x \Rightarrow fn\ y \Rightarrow xy) &\xrightarrow{\beta} \\ [fn\ x \Rightarrow fn\ y \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow xy)x(xy)] &\xrightarrow{\beta} \\ [fn\ x \Rightarrow fn\ y \Rightarrow (fn\ y \Rightarrow xy)(xy)] &\xrightarrow{\beta} \\ fn\ x \Rightarrow fn\ y \Rightarrow x(xy) &\xrightarrow{\beta} \\ \mathbf{two} \end{aligned}$$

Successivamente, definiamo le seguenti ulteriori funzioni matematiche:

- La funzione **sum** che somma due numeri di Church:

$$\text{sum} := \lambda z \Rightarrow \lambda w \Rightarrow (\lambda x \Rightarrow \lambda y \Rightarrow zx(wxy))$$

oppure:

$$\text{sum} := \lambda z \Rightarrow \lambda w \Rightarrow z \text{ succ } w$$

- La funzione **prod** che moltiplica due numeri di Church:

$$\text{prod} := \lambda z \Rightarrow \lambda w \Rightarrow (\lambda x \Rightarrow \lambda y \Rightarrow z(wx)y)$$

oppure:

$$\text{prod} := \lambda z \Rightarrow \lambda w \Rightarrow z(\text{sum } w) \text{ zero}$$

- La funzione **power** che eleva un numero di Church ad un altro numero di Church:

$$\text{prod} := \lambda z \Rightarrow \lambda w \Rightarrow wz$$

Oltre ai numeri naturali, il lambda calcolo ci permette di descrivere anche la **logica booleana** di Church, dove i due valori **True** e **False** sono definiti come:

$$\text{True} := \lambda x \Rightarrow \lambda y \Rightarrow x$$

$$\text{False} := \lambda x \Rightarrow \lambda y \Rightarrow y$$

Come per i numeri di Church, definiamo una funzione **evalBool** in grado di convertire un booleano di Church in nel suo equivalente booleano:

$$\text{evalBool} := \lambda z \Rightarrow z \text{ true } \text{false}$$

dove *true* e *false* sono i normali valori booleani

Infine, definiamo i seguenti operatori logici:

- L'operatore **ITE** (abbreviativo di **If-Then-Else**) che dati una condizione  $z$  e due booleani di Church  $u, v$ , valuta  $u$  se  $z$  è *true* oppure valuta  $v$  se  $z$  è *false*:

$$\text{ITE} := \lambda z \Rightarrow \lambda u \Rightarrow \lambda v \Rightarrow z u v$$

- L'operatore **If** che dati una condizione  $z$  ed un booleano di Church  $u$ , valuta  $u$  se  $z$  è *true*:

$$\text{If} := \lambda z \Rightarrow \lambda u \Rightarrow z u \text{ True}$$

- L'operatore **Not** che restituisce il negato di un booleano di Church:

$$\text{Not} := \lambda z \Rightarrow \lambda x \Rightarrow \lambda y \Rightarrow z y x$$

- L'operatore **Or** che restituisce l'or logico tra due booleani di Church:

$$\text{Or} := \lambda z \Rightarrow \lambda w \Rightarrow \text{If}(\text{Not } z)w$$

- L'operatore **And** che restituisce l'and logico tra due booleani di Church:

$$\text{And} := \lambda z \Rightarrow \lambda w \Rightarrow \text{Not}(\text{If } z (\text{Not } w))$$

Di seguito viene fornito il codice SML per poter lavorare con il modello di calcolo appena definito:

```
(* Numeri di Church *)

val zero = fn x => fn y => y;
val one  = fn x => fn y => x y;
val two  = fn x => fn y => x(x y);
val three = fn x => fn y => x(x(x y));

val eval = fn z => z (fn x => x+1) 0;

val succ = fn z => fn x => fn y => z x (x y);
val sum  = fn z => fn w => fn x => fn y => z x (w x y);
val prod = fn z => fn w => fn x => fn y => z (w x) y;
val power = fn z => fn w => w z;

(* Booleani di Church *)

val True  = fn x => fn y => x;
val False = fn x => fn y => y;

val evalBool = fn z => z true false;

val ITE = fn z => fn u => fn v => z u v;
val If  = fn z => fn u => z u True;

val Not = fn z => fn x => fn y => z y x;
val Or  = fn z => fn w => If (Not z) w;
val And = fn z => fn w => Not (If z (Not w));

(* Esempi *)

eval (sum (power two three) (prod two three));
evalBool (And (ITE True False True) False);
```

# 3

## Paradigma imperativo

### 3.1 *Imp*: un semplice linguaggio imperativo

#### Definizione 3.1.0.1: Il linguaggio *Imp*

Definiamo come *Imp* il linguaggio rappresentato dalle seguenti grammatiche:

$$\begin{aligned} M, N &::= k \mid x \mid M + N \mid M < N \\ P, Q &::= skip \mid P; Q \mid \text{if } M \text{ then } P \text{ else } Q \mid \text{while } M \text{ do } P \mid \\ &\quad \text{var } x = M \text{ in } P \mid x := M \end{aligned}$$

dove:

- La prima grammatica rappresenta l'insieme *Exp* delle **espressioni**
- La seconda grammatica rappresenta l'insieme *Imp* dei **programmi**
- $k \in \{true, false\} \cup \{0, 1, \dots\}$  ossia è una **costante**
- $x \in Var = \{x, y, z, \dots\}$  ossia è una **variabile**
- Il termine *skip* è il programma che **non esegue alcuna operazione**
- Il termine  $P; Q$  esegue prima il programma  $P$  e poi il programma  $Q$
- Il termine *ite* esegue il programma  $P$  se l'espressione  $M$  è vera, altrimenti esegue il programma  $Q$
- Il termine *while* esegue il programma  $P$  finché l'espressione  $M$  è vera
- Il termine *var* **dichiara** la variabile  $x$  e gli **assegna** l'espressione  $M$  all'interno della **valutazione** di  $P$ . Inoltre,  $x$  prende il nome di variabile locale in  $P$
- Il termine  $:=$  **assegna** l'espressione  $M$  alla variabile  $x$  (solo se  $x$  è stata precedentemente dichiarata)

**Esempi:**

- Il programma  $\text{var } x = 0 \text{ in while } x < 10 \text{ do } x := x + 1$  è un termine valido di *Imp*
- Il programma  $\text{var } x = 0 \text{ in while } x < 10 \text{ do } y := x + 1$  non è un termine valido di *Imp*, poiché la variabile  $y$  non è stata dichiarata prima dell'assegnamento

**Definizione 3.1.0.2: Insieme delle locazioni**

Dato il linguaggio *Imp*, definiamo come **insieme delle locazioni**, indicato con *Loc*, l'insieme contenente le locazioni di memoria, ossia gli indirizzi di memoria ai quali sono associati dei valori (sostanzialmente, una locazione è un **puntatore**)

**Definizione 3.1.0.3: Insieme degli ambienti in *Imp***

Dato il linguaggio *Imp*, definiamo come **insieme degli ambienti di *Imp***, indicato con *Env*, il seguente insieme:

$$Env = \{f \mid f : Var \xrightarrow{fin} Loc\}$$

**Definizione 3.1.0.4: Insieme delle memorie in *Imp***

Dato il linguaggio *Imp*, definiamo come **insieme delle memorie di *Imp***, indicato con *Store*, il seguente insieme:

$$Store = \{f \mid f : Loc \xrightarrow{fin} Val\}$$

**Definizione 3.1.0.5: Concatenazione di memorie**

Dato il linguaggio *Imp*, definiamo l'operazione di **concatenazione di memorie**, ossia:

$$\cdot : Store \times Store \rightarrow Store$$

dove:

$$(S_1 S_2)(x) = \begin{cases} S_2(x) & \text{se } x \in \text{dom}(S_1) \\ S_1(x) & \text{altrimenti} \end{cases}$$

**Definizione 3.1.0.6: Semantiche operazionali di *Imp***

Dato il linguaggio *Imp*, definiamo su di esso le seguenti due semantiche:

- La semantica delle espressioni

$$\overset{M}{\rightsquigarrow} \subseteq Env \times Exp \times Store \times Val$$

dove  $(E, M, S, v) \in \overset{M}{\rightsquigarrow}$  viene descritta dalla notazione  $E \vdash M, S \rightsquigarrow v$

- La semantica dei programmi

$$\overset{P}{\rightsquigarrow} \subseteq Env \times Imp \times Store \times Store$$

dove  $(E, P, S, S') \in \overset{P}{\rightsquigarrow}$  viene descritta dalla notazione  $E \vdash P, S \rightsquigarrow S'$

Le regole operazionali di tali semantiche sono definite come:

- **Costanti:**

$$E \vdash k, S \rightsquigarrow k$$

- **Variabili:**

$$S(E(x)) = v \implies E \vdash x, S \rightsquigarrow v$$

- **Somma:**

$$u = v + v' \implies \frac{E \vdash M, S \rightsquigarrow v \quad E \vdash N, S \rightsquigarrow v'}{E \vdash M + N, S \rightsquigarrow u}$$

- **Minorazione:**

$$v < v' \implies \frac{E \vdash M, S \rightsquigarrow v \quad E \vdash N, S \rightsquigarrow v'}{E \vdash M < N, S \rightsquigarrow true}$$

$$v \geq v' \implies \frac{E \vdash M, S \rightsquigarrow v \quad E \vdash N, S \rightsquigarrow v'}{E \vdash M < N, S \rightsquigarrow false}$$

- **Skip:**

$$E \vdash skip, S \rightsquigarrow S$$

- **Esecuzione sequenziale:**

$$\frac{E \vdash P, S \rightsquigarrow S' \quad E \vdash Q, S' \rightsquigarrow S''}{E \vdash P; Q, S \rightsquigarrow S''}$$

- **If-then-else:**

$$\frac{E \vdash M, S \rightsquigarrow true \quad E \vdash P, S \rightsquigarrow S'}{E \vdash if M then P else Q, S \rightsquigarrow S'}$$

$$\frac{E \vdash M, S \rightsquigarrow false \quad E \vdash Q, S \rightsquigarrow S'}{E \vdash if M then P else Q, S \rightsquigarrow S'}$$



- **While:**

$$\frac{E \vdash M, S \rightsquigarrow true \quad E \vdash P, S \rightsquigarrow S' \quad E \vdash while\ M\ do\ P, S' \rightsquigarrow S''}{E \vdash while\ M\ do\ P, S \rightsquigarrow S''}$$

$$\frac{E \vdash M, S \rightsquigarrow false}{E \vdash while\ M\ do\ P, S \rightsquigarrow S}$$

- **Dichiarazione e assegnamento:**

$$\frac{E \vdash M, S \rightsquigarrow v \quad E\{(x, l)\} \vdash P, S\{(l, v)\} \rightsquigarrow S'}{E \vdash var\ x = M\ in\ P, S \rightsquigarrow S'}$$

dove  $l \notin dom(S)$ , ossia è una nuova locazione di memoria

- **Assegnamento:**

$$E(x) = l \implies \frac{E \vdash M, S \rightsquigarrow v}{E \vdash x := M, S \rightsquigarrow S\{(l, v)\}}$$

#### Osservazione 3.1.0.1

Tramite le definizioni date delle due semantiche e delle loro regole operazionali, notiamo che le espressioni vengono valutate in **valori** mentre i programmi vengono valutati in **memorie**.

Di conseguenza, i programmi **propagano "a ritroso"** (ossia scendendo nell'albero di derivazione) le modifiche alla memoria, mentre le espressioni **propagano "in avanti"** (ossia salendo nell'albero di derivazione) le modifiche all'ambiente

## 3.2 *All*: un linguaggio con procedure

### Definizione 3.2.0.1: Il linguaggio *All*

Definiamo come *All* il linguaggio rappresentato dalle seguenti grammatiche:

$$\begin{aligned}
 V &::= x \mid x[M] \\
 M, N &::= k \mid V \mid M + N \mid M < N \\
 P, Q &::= \text{skip} \mid P; Q \mid \text{if } M \text{ then } P \text{ else } Q \mid \text{while } M \text{ do } P \mid \\
 &\quad \text{var } x = M \text{ in } P \mid \text{arr } x = [M_0, \dots, M_n] \text{ in } P \mid V := M \mid \\
 &\quad \text{proc } y(x) \text{ is } P \text{ in } Q \mid \text{call } y(M)
 \end{aligned}$$

dove:

- La prima grammatica rappresenta l'insieme *LExp* (per *left expressions*) delle **espressioni assegnabili**
- La seconda grammatica rappresenta l'insieme *Exp* delle **espressioni valutabili**
- La terza grammatica rappresenta l'insieme *Imp* dei **programmi**
- $k \in \{\text{true}, \text{false}\} \cup \{0, 1, \dots\}$  ossia è una **costante**
- $x \in \text{Var} = \{x, y, z, \dots\}$  ossia è una **variabile**
- I termini già presenti in *Imp* sono definiti ugualmente
- Il termine *arr* **dichiara** l'array  $x$  e gli **assegna** le espressioni  $M_0, \dots, M_n$  all'interno della **valutazione** di  $P$
- Il termine *proc* dichiara una **procedura**  $y$  (ossia una funzione) con parametro  $x$  richiamabile all'interno di  $P$
- Il termine *call* **richiama** la procedura  $y$  passando  $M$  come argomento di essa (solo se  $y$  è stata precedentemente definita)

### Definizione 3.2.0.2: Insieme delle locazioni contigue

Dato l'insieme *Loc* e il linguaggio *All*, definiamo come **insieme delle locazioni contigue**, indicato con  $\text{Loc}^+$ , l'insieme contenente le sequenze di locazioni contigue di memoria

### Proposizione 3.2.0.1: Ridefinizione di *Env* in *All*

Dato il linguaggio *All*, definiamo come **insieme degli ambienti di *All***, indicato con *Env*, il seguente insieme:

$$\text{Env} = \{f \mid f : \text{Var} \xrightarrow{\text{fin}} \text{Loc}^+ \cup (\text{Var} \times \text{All} \times \text{Env})\}$$

**Definizione 3.2.0.3: Semantiche operazionali di *All***

Dato il linguaggio *All*, definiamo su di esso le seguenti due semantiche:

- La semantica delle espressioni assegnabili

$$\overset{V}{\rightsquigarrow} \subseteq Env \times LExp \times Store \times Loc$$

dove  $(E, M, S, l) \in \overset{V}{\rightsquigarrow}$  viene descritta dalla notazione  $E \vdash V, S \overset{V}{\rightsquigarrow} l$

- La semantica delle espressioni valutabili

$$\overset{M}{\rightsquigarrow} \subseteq Env \times Exp \times Store \times Val$$

dove  $(E, M, S, v) \in \overset{M}{\rightsquigarrow}$  viene descritta dalla notazione  $E \vdash M, S \overset{M}{\rightsquigarrow} v$

- La semantica dei programmi

$$\overset{P}{\rightsquigarrow} \subseteq Env \times All \times Store \times Store$$

dove  $(E, P, S, S') \in \overset{P}{\rightsquigarrow}$  viene descritta dalla notazione  $E \vdash P, S \overset{P}{\rightsquigarrow} S'$

Oltre alle regole operazionali già definite in *Imp*, vengono definite le seguenti regole aggiuntive:

- **Locazione:**

$$E(x) = l \implies E \vdash x, S \overset{V}{\rightsquigarrow} l$$

- **Locazione in array:**

$$E(x) = \langle l_0, \dots, l_n \rangle \wedge m \in [0, n] \implies \frac{E \vdash M, S \overset{M}{\rightsquigarrow} m}{E \vdash x[M], S \overset{V}{\rightsquigarrow} l_m}$$

- **Riferimento:**

$$S(l) = v \implies \frac{E \vdash V, S \overset{V}{\rightsquigarrow} l}{E \vdash V, S \overset{M}{\rightsquigarrow} v}$$

- **Assegnamento:**

$$\frac{E \vdash M, S \overset{M}{\rightsquigarrow} v \quad E \vdash V, S \overset{V}{\rightsquigarrow} l}{E \vdash V := M, S \overset{P}{\rightsquigarrow} S\{(l, v)\}}$$

- **Dichiarazione array:**

$$\frac{E \vdash M_0, S \overset{M}{\rightsquigarrow} v_0 \quad \dots \quad E \vdash M_n, S \overset{M}{\rightsquigarrow} v_n \quad E\{(x, (l_0, \dots, l_n))\} \vdash P, S\{(l_0, v_0), \dots, (l_n, v_n)\} \overset{P}{\rightsquigarrow} S'}{E \vdash arr \ x = [M_0, \dots, M_n] \ in \ P, S \overset{P}{\rightsquigarrow} S'}$$

dove  $l_0, \dots, l_n \notin dom(S)$ , ossia sono nuove locazioni di memoria

- **Procedura:**

$$\frac{E\{y, (x, P, E)\} \vdash Q, S \xrightarrow{P} S'}{E \vdash \text{proc } y(x) \text{ is } P \text{ in } Q, S \xrightarrow{P} S'}$$

### 3.2.1 Semantiche di *All*

#### Definizione 3.2.1.1: Semantiche di *All*

Per via della separazione tra i concetti di *ambiente* e *memoria*, i valori degli argomenti delle procedure possono essere richiamati tramite tre semantiche operazionali:

- **Call-by-value**, ossia tramite una **semantica eager statica** in cui come argomento viene passato un termine di *Exp valutato* (passaggio per valore)
- **Call-by-reference**, ossia tramite una **semantica eager statica** in cui come argomento viene passato un termine di *LExp valutato* (passaggio per riferimento)
- **Call-by-name**, ossia tramite una **semantica lazy statica** in cui come argomento viene passato un termine di *LExp non ancora valutato* (passaggio per riferimento)

#### Proposizione 3.2.1.1: Linguaggio *All* call-by-value

La semantica call-by-value del linguaggio *All* prevede l'aggiunta della seguente regola operativa:

- **Richiamo by-value:**

$$E(y) = (x, P, E') \implies \frac{E \vdash M, S \xrightarrow{M} v \quad E'\{(x, l)\} \vdash P, S\{(l, v)\} \rightsquigarrow S'}{E \vdash \text{call } y(M), S \xrightarrow{P} S'}$$

dove  $l \notin \text{dom}(S)$ , ossia è una nuova locazione di memoria

#### Proposizione 3.2.1.2: Linguaggio *All* call-by-reference

La semantica call-by-reference del linguaggio *All* prevede l'aggiunta della seguente regola operativa:

- **Richiamo by-reference:**

$$E(y) = (x, P, E') \implies \frac{E \vdash V, S \xrightarrow{V} l \quad E'\{(x, l)\} \vdash P, S \rightsquigarrow S'}{E \vdash \text{call } y(V), S \xrightarrow{P} S'}$$

dove  $l \notin \text{dom}(S)$ , ossia è una nuova locazione di memoria

**Proposizione 3.2.1.3: Linguaggio *All* call-by-name**

La semantica call-by-name del linguaggio *All* prevede la ridefinizione di *Env* come:

$$Env = \{f \mid f : Var \xrightarrow{fin} Loc^+ \cup (Var \times All \times Env) \cup (LExp \times Env)\}$$

e l'aggiunta delle seguenti regole operazionali:

- **Locazione in variabile:**

$$E(x) = (V, E') \implies \frac{E' \vdash V, S \xrightarrow{V} l}{E \vdash x, S \xrightarrow{V} l}$$

- **Richiamo by-name:**

$$E(y) = (x, P, E') \implies \frac{E' \{(x, (V, E))\} \vdash P, S \rightsquigarrow S'}{E \vdash call\ y(V), S \xrightarrow{P} S'}$$

**Lemma 3.2.1.1**

Dato il linguaggio *All*, si ha che:

$$All\ call\text{-}by\text{-}value \not\equiv All\ call\text{-}by\text{-}reference$$

$$All\ call\text{-}by\text{-}value \not\equiv All\ call\text{-}by\text{-}name$$

*Dimostrazione.*

- Consideriamo il programma *proc y(x) is P in Q*
- Nel caso della semantica call-by-value, per la chiamata *call y(M)* verrà sempre creata una nuova locazione per il parametro *x*, portando le operazioni svolte su di *x* stesso ad influenzare tale nuova locazione
- Nel caso delle semantiche call-by-reference e call-by-name, invece, per la chiamata *call y(V)* verrà utilizzata direttamente la locazione associata all'espressione *V*, portando le operazioni svolte su di *x* stesso ad influenzare tale locazione già esistente
- Di conseguenza, risulta evidente come il programma restituisca uno stato diverso in base al tipo di passaggio utilizzato

□

**Esempio:**

- Consideriamo il programma

$$var\ x = 5\ in\ proc\ y(z)\ is\ z := 1\ in\ call\ y(x)$$

- Utilizzando la semantica call-by-value, il suo albero di derivazione corrisponde a:

$$\begin{array}{c}
 \frac{E' \vdash x, \{(l, 5)\} \xrightarrow{V} l \quad \frac{E'' \vdash 1, S \xrightarrow{M} 1 \quad E'' \vdash z, S \xrightarrow{V} \{(l', 1)\}}{E'' \vdash z := 1, S \rightsquigarrow S\{(l', 1)\}}}{E' \vdash x, \{(l, 5)\} \xrightarrow{M} 5} \\
 \frac{\emptyset \vdash 5, \emptyset \rightsquigarrow 5 \quad \frac{E' \vdash \text{call } y(x), \{(l, 5)\} \rightsquigarrow S\{(l', 1)\}}{E \vdash \text{proc } y(z) \text{ is } z := 1 \text{ in call } y(x), \{(l, 5)\} \rightsquigarrow S\{(l', 1)\}}}{\emptyset \vdash \text{var } x = 5 \text{ in proc } y(z) \text{ is } z := 1 \text{ in call } y(x), \emptyset \rightsquigarrow S\{(l', 1)\}}
 \end{array}$$

dove  $E := \{(x, l)\}$ ,  $E' := E\{(y, (z, z := 1, E))\}$ ,  $E'' := E'\{(z, l')\}$  e  $S := \{(l, 5), (l', 5)\}$ , restituendo quindi la memoria  $S\{(l', 1)\} = \{(l, 5), (l', 1)\}$

- Utilizzando la semantica call-by-reference, il suo albero di derivazione corrisponde a:

$$\begin{array}{c}
 \frac{E' \vdash x, \{(l, 5)\} \xrightarrow{V} l \quad \frac{E'' \vdash 1, \{l, 5\} \xrightarrow{M} 1 \quad E'' \vdash z, \{l, 5\} \xrightarrow{V} \{(l, 1)\}}{E'' \vdash z := 1, \{l, 5\} \rightsquigarrow \{(l, 1)\}}}{E' \vdash \text{call } y(x), \{(l, 5)\} \rightsquigarrow \{(l, 1)\}} \\
 \frac{\emptyset \vdash 5, \emptyset \rightsquigarrow 5 \quad \frac{E' \vdash \text{call } y(x), \{(l, 5)\} \rightsquigarrow \{(l, 1)\}}{E \vdash \text{proc } y(z) \text{ is } z := 1 \text{ in call } y(x), \{(l, 5)\} \rightsquigarrow \{(l, 1)\}}}{\emptyset \vdash \text{var } x = 5 \text{ in proc } y(z) \text{ is } z := 1 \text{ in call } y(x), \emptyset \rightsquigarrow \{(l, 1)\}}
 \end{array}$$

dove  $E := \{(x, l)\}$ ,  $E' := E\{(y, (z, z := 1, E))\}$ ,  $E'' := E'\{(z, l)\}$ , restituendo quindi la memoria  $\{(l, 1)\}$

- Notiamo quindi che nel caso della semantica call-by-value venga assegnata una nuova locazione di memoria aggiuntiva per il parametro  $z$ . Nel caso della semantica call-by-reference, invece, viene utilizzata direttamente la locazione di memoria della variabile  $x$  passata come argomento alla chiamata  $y(x)$ .
- Pertanto, nel primo caso l'assegnamento  $z := 1$  verrà effettuato sulla nuova locazione, mentre nel secondo caso l'assegnamento viene effettuato sulla locazione passata per riferimento

### Lemma 3.2.1.2

Dato il linguaggio *All*, si ha che:

$$All_{\text{call-by-reference}} \neq All_{\text{call-by-name}}$$

*Dimostrazione.*

- Consideriamo il programma

$$\text{var } x = 0 \text{ in arr } z = [3, 7] \text{ in proc } y(w) \text{ is } x := 1; w := 42 \text{ in call } y(z[x])$$

- Nel caso della semantica call-by-reference, durante la chiamata  $y(z[x])$ , l'espressione  $x$  viene valutata immediatamente, implicando che il parametro  $w$  punterà alla locazione di  $z[x] = z[0]$

- Nel caso della semantica call-by-name, invece, la valutazione dell'espressione  $x$  viene rimandata fino all'espressione  $w := 42$ . Tuttavia, prima di tale valutazione si ha che  $x := 1$ , implicando che la valutazione di  $x$  restituirà 1 e dunque che  $w$  punterà alla locazione di  $z[x] = z[1]$
- Di conseguenza, nei due casi verrà restituita una memoria differente

□

**Teorema 3.2.1.1: Equivalenze semantiche di *All***

Dato il linguaggio *All*, **non esistono due semantiche equivalenti**

# 4

## Correttezza dei programmi

### 4.1 Correttezza dei programmi imperativi

#### 4.1.1 Il metodo delle invarianti

Nel Papiro di Rhind (circa 1650 a.C), viene descritto, fra molte altre discussioni matematiche, l'algoritmo usato dagli antichi egizi per svolgere la moltiplicazione.

Dati due numeri  $x, y \in \mathbb{N} - \{0\}$  da moltiplicare, l'algoritmo prosegue come tale:

1. Raddoppio  $x$  e se  $y$  è dispari sottraggo 1 ad esso e lo divido per due, altrimenti se  $y$  è pari lo divido direttamente per due
2. Scrivo i due nuovi valori di  $x$  e  $y$  sotto ai valori precedenti
3. Ripeto il passo 1) finché non ottengo che  $y = 1$
4. Tra tutte le coppie di valori scritte, scarto tutte le coppie in cui  $y$  è pari
5. L'output corrisponde alla somma di tutti i valori di  $x$  delle coppie rimanenti

**Esempio:**

- Applichiamo i primi due step sui numeri  $x = 45$  e  $y = 138$ :

45	138
90	69
180	34
360	17
720	8
1440	4
2880	2
5760	1



- Successivamente, scartiamo tutte le coppie per cui  $y$  è pari:

90	69
360	17
5760	1

- A questo punto, sommiamo i valori di  $x$ :

$$5760 + 360 + 90 = 6210$$

ottenendo il risultato del prodotto  $45 \cdot 138$

Definiamo quindi il codice dell'algoritmo egiziano per la moltiplicazione:

```
int AegyptProduct(int a, int b){
    int x = a;
    int y = b;
    int res = 0;

    while(y > 0){
        if(y % 2 == 0){
            x = x + x;
            y = y/2;
        }
        else{
            res = res + x;
            y = y - 1;
        }
    }

    return res;
}
```

A questo punto, vogliamo dimostrare la **correttezza** di tale algoritmo. Consideriamo quindi la seguente espressione:

$$x \cdot y + \text{res} = a \cdot b$$

Notiamo come tale espressione rimanga **sempre vera** sia prima del while, sia per ogni iterazione del while e sia dopo il while.

Siano quindi  $x'$ ,  $y'$  e  $\text{res}'$  i valori finali assunti dalle tre variabili. Poiché tale espressione è sempre vera e poiché la condizione di uscita del while è  $y' = 0$ , abbiamo che:

$$a \cdot b = x' \cdot y' + \text{res}' = x' \cdot 0 + \text{res}' = \text{res}'$$

dunque la funzione restituisce correttamente il prodotto tra  $a$  e  $b$

#### Definizione 4.1.1.1: Invariante

Definiamo come **invariante** di un oggetto una matematico una sua proprietà che rimane valida a seguito di operazioni o trasformazioni applicate sull'oggetto stesso

**Proposizione 4.1.1.1**

La proprietà:

$$x \cdot y + \text{res} = a \cdot b$$

è un'invariante della funzione `AegyptProduct`

*Dimostrazione.*

- Siano  $x_0, \dots, x_f, y_0, \dots, y_f$  e  $\text{res}_0, \dots, \text{res}_f$  i valori delle variabili  $x, y$  e  $\text{res}$  per ogni iterazione del `while`, dove  $f$  è l'iterazione in cui la condizione del `while` risulta falsa.

*Caso base.*

- Prima del ciclo `while`, dunque all'iterazione 0, si ha che:

$$x_0 \cdot y_0 + \text{res}_0 = a \cdot b + 0 = a \cdot b$$

*Ipotesi induttiva.*

- Supponiamo che per l' $i$ -esima iterazione valga che:

$$x_i \cdot y_i + \text{res}_i = a \cdot b$$

*Passo induttivo.*

- Se per l' $i$ -esima iterazione si verifica che  $y_i \equiv 0 \pmod{2}$ , allora:

- $x_{i+1} = 2x_i$
- $y_{i+1} = \frac{y_i}{2}$
- $\text{res}_{i+1} = \text{res}_i$

implicando che:

$$x_{i+1} \cdot y_{i+1} + \text{res}_{i+1} = 2x_i \cdot \frac{y_i}{2} + \text{res}_i = x_i \cdot y_i + \text{res}_i = a \cdot b$$

- Altrimenti, si ha che:

- $x_{i+1} = x_i$
- $y_{i+1} = y_i - 1$
- $\text{res}_{i+1} = \text{res}_i + x_i$

implicando che:

$$x_{i+1} \cdot y_{i+1} + \text{res}_{i+1} = x_i \cdot (y_i - 1) + \text{res}_i + x_i = x_i \cdot y_i + \text{res}_i = a \cdot b$$

□

**Metodo 4.1.1.1: Metodo delle invarianti**

Per dimostrare la correttezza di un algoritmo, è possibile individuare una sua invariante tramite cui dimostrare la sua correttezza, riducendo la dimostrazione della correttezza nella dimostrazione dell'invarianza. Tale invariante viene detta **specifica di correttezza** dell'algoritmo.

**4.1.2 Logica di Hoare**

Avendo trattato i linguaggi imperativi e il metodo delle invarianti, vogliamo definire una grammatica che ci permetta di dimostrare la **correttezza dei programmi imperativi** in modo più diretto.

**Definizione 4.1.2.1: Logica di Hoare**

Definiamo come Logica di Hoare il linguaggio assiomatico rappresentato dalle seguenti grammatiche:

$$\begin{aligned} M, N &::= k \mid x \mid M + N \\ A, B &::= true \mid false \mid A \supset B \mid M < N \mid M = N \\ P, Q &::= skip \mid P; Q \mid if\ B\ then\ P\ else\ Q \mid while\ B\ do\ P \mid x := M \end{aligned}$$

dove:

- La prima grammatica rappresenta l'insieme delle **espressioni numeriche**
- La seconda grammatica rappresenta l'insieme delle **espressioni booleane**
- La terza grammatica rappresenta l'insieme dei **programmi**
- $k \in \{0, 1, \dots\}$  ossia è una **costante**
- $x \in \{x, y, z, \dots\}$  ossia è una **variabile**
- Il simbolo  $\supset$  è l'**implicazione logica** (dunque equivale al simbolo  $\implies$ )

Definiamo inoltre alcune abbreviazioni sintattiche:

- La notazione  $\neg A$  corrisponde al termine  $A \supset false$
- La notazione  $A \vee B$  corrisponde al termine  $\neg A \supset B$
- La notazione  $A \wedge B$  corrisponde al termine  $\neg(\neg A \supset B)$
- La notazione  $A \leq B$  corrisponde al termine  $(A < B) \vee (A = B)$
- La notazione  $A > B$  corrisponde al termine  $\neg((A < B) \vee (A = B))$
- La notazione  $A \geq B$  corrisponde al termine  $\neg(A < B)$

**Definizione 4.1.2.2: Tripla di Hoare**

Data la logica di Hoare, siano  $A$  e  $B$  due espressioni booleane e sia  $P$  un programma. Se **eseguendo**  $P$  in uno stato che **soddisfa**  $A$ , si ottiene uno stato che **soddisfa**  $B$ , definiamo l'espressione

$$\{A\} P \{B\}$$

come **tripla di Hoare**, dove  $A$  viene detta **precondizione** e  $B$  viene detta **postcondizione**

**Definizione 4.1.2.3: Formula**

Data la logica di Hoare, definiamo come **formula** un'espressione appartenente alla seguente grammatica:

$$\varphi ::= A \mid \{A\} P \{B\}$$

**Proposizione 4.1.2.1: Regole di inferenza generali**

Data la logica di Hoare, definiamo le seguenti regole di inferenza:

- **True:**

$$\{A\} P \{true\}$$

- **False:**

$$\{false\} P \{A\}$$

- **Rafforzamento della precondizione (strengthening):**

$$\frac{A \supset B \quad \{B\} P \{C\}}{\{A\} P \{C\}}$$

- **Indebolimento della postcondizione (weakening):**

$$\frac{\{A\} P \{B\} \quad B \supset C}{\{A\} P \{C\}}$$

- **And:**

$$\frac{\{A\} P \{B_1\} \quad \dots \quad \{A\} P \{B_n\}}{\{A\} P \{B_1 \wedge \dots \wedge B_n\}}$$

- **Or:**

$$\frac{\{A_1\} P \{B\} \quad \dots \quad \{A_n\} P \{B\}}{\{A_1 \vee \dots \vee A_n\} P \{B\}}$$

**Proposizione 4.1.2.2: Regole di inferenza dei programmi**

Data la logica di Hoare, definiamo le seguenti regole di inferenza:

- **Skip:**

$$\{A\} \text{ skip } \{A\}$$

- **Esecuzione sequenziale:**

$$\frac{\{A\} P \{B\} \quad \{B\} Q \{C\}}{\{A\} P; Q \{C\}}$$

- **If-then-else:**

$$\frac{\{A \wedge B\} P \{C\} \quad \{A \wedge \neg B\} Q \{C\}}{\{A\} \text{ if } B \text{ then } P \text{ else } Q \{C\}}$$

- **While:**

$$\frac{\{A \wedge B\} P \{A\}}{\{A\} \text{ while } B \text{ do } P \{A \wedge \neg B\}}$$

- **Assegnamento:**

$$\{A[M/x]\} x := M \{A\}$$

dove  $[M/x]$  ricordiamo essere l'operatore di [Sostituzione](#)

**Esempio:**

- Consideriamo la seguente tripla di Hoare:

$$\{x = 1\} x := x + 1 \{x = 2\}$$

- Possiamo utilizzare lo *strengthening* affinché l'assegnamento possa essere valutato correttamente in quanto  $(x = 2)[x + 1/x]$  venga valutata in  $x + 1 = 2$ :

$$\frac{x = 1 \supset x + 1 = 2 \quad \{x + 1 = 2\} x := x + 1 \{x = 2\}}{\{x = 1\} x := x + 1 \{x = 2\}}$$

**Teorema 4.1.2.1: Invarianti con logica di Hoare**

Dato il programma *while B do Q* ed una proprietà *A*, si ha che:

$$A \text{ invariante} \iff \{A\} \text{ while } B \text{ do } Q \{A \wedge \neg B\}$$

A questo punto, consideriamo la seguente funzione:

```
int EuclideanDivision(int x, int y){
    int b = x;
    int a = 0;

    while(b >= y){
        b = b - y;
        a = a + 1;
    }

    return {a, b};
}
```

Vogliamo dimostrare che tale funzione calcoli effettivamente la divisione con resto euclidea, ossia che i valori  $a$  e  $b$  restituiti siano tali che  $x = ay + b$  e  $0 \leq b < y$  (ossia che  $a$  sia il quoziente della divisione e che  $b$  sia il resto di quest'ultima)

Prima di tutto convertiamo il codice in un programma espresso dalla logica di Hoare:

$$b := x; a := 0; \text{ while } b \geq y \text{ do } b := b - y; a := a + 1$$

A questo punto, cerchiamo di dimostrare che tali proprietà siano un'invariante del programma utilizzando le regole operazionali fornite dalla logica di Hoare.

#### Proposizione 4.1.2.3

Se  $x \geq 0$ , la proprietà:

$$x = ay + b \wedge b \geq 0 \wedge b < y$$

è un'invariante della funzione `EuclideanDivision`

*Dimostrazione.*

- Per facilitare la lettura, poniamo:
  - $P \equiv (b := x; a := 0; \text{ while } b \geq y \text{ do } b := b - y; a := a + 1)$
  - $Q \equiv (b := x)$
  - $R \equiv (a := 0)$
  - $S \equiv (\text{ while } b \geq y \text{ do } b := b - y; a := a + 1)$
  - $A \equiv (x = ay + b \wedge b \geq 0 \wedge b < y)$
- Affinché la proprietà  $A$  sia un'invariante di  $P$ , è necessario che trovare una precondizione  $B$  tale che la seguente tripla di Hoare sia valida:

$$\{B\} b := x; a := 0; \text{ while } b \geq y \text{ do } b := b - y; a := a + 1 \{A\}$$

- Tramite le regole operazionali della logica di Hoare, abbiamo che:

$$\frac{\{B\} Q \{C\} \quad \{C\} R \{D\} \quad \{D\} S \{A\}}{\{B\} b := x; a := 0; \text{while } b \geq y \text{ do } b := b - y; a := a + 1 \{A\}}$$

- Consideriamo quindi la tripla  $\{D\} S \{A\}$ . La sua valutazione è data da:

$$\frac{\frac{\{D \wedge b \geq y\} b := b - y \{F\} \quad \{F\} a := a + 1 \{D\}}{\{D \wedge b \geq y\} b := b - y; a := a + 1 \{D\}}}{\{D\} \text{while } b \geq y \text{ do } b := b - y; a := a + 1 \{A\}}$$

- Affinché tale tripla sia valida,  $D$  deve necessariamente essere una condizione tale che

$$x = ay + b \wedge b \geq 0 \wedge b < y \equiv A \equiv D \wedge \neg(b \geq y) \equiv D \wedge b < y$$

dunque otteniamo che  $D \equiv x = ay + b \wedge b \geq 0$

- Una volta trovato  $D$ , consideriamo la tripla  $\{F\} a := a + 1 \{D\}$ . Tramite la regola dell'assegnamento, notiamo facilmente che:

$$\{F\} a := a + 1 \{D\} \implies \{D[a + 1/a]\} a := a + 1 \{D\}$$

dunque otteniamo che  $F \equiv D[a + 1/a] \equiv x = (a + 1)y + b \wedge b \geq 0$

- Una volta trovato  $F$ , consideriamo la tripla  $\{D \wedge b \geq y\} b := b - y \{F\}$ , dove:

$$\{D \wedge b \geq y\} b := b - y \{F\} \iff$$

$$\{x = ay + b \wedge b \geq 0 \wedge b \geq y\} b := b - y \{x = (a + 1)y + b \wedge b \geq 0\}$$

A questo punto, notiamo che:

$$x = ay + b \wedge b \geq 0 \wedge b \geq y \implies x = (a + 1)y + b - y \wedge b - y \geq y$$

Di conseguenza, posto  $E \equiv x = (a + 1)y + b - y \wedge b - y \geq y$ , tramite lo *strengthening*, abbiamo che:

$$\frac{D \wedge b \geq y \supset E \quad \{E\} b := b - y \{x = (a + 1)y + b \wedge b \geq 0\}}{\{x = ay + b \wedge b \geq 0 \wedge b \geq y\} b := b - y \{x = (a + 1)y + b \wedge b \geq 0\}}$$

inoltre, abbiamo che  $E \equiv F[b - y/b] \equiv (D[a + 1/a])[b - y/b]$

- Ricapitolando, dunque, affinché tale tripla sia valida, abbiamo che:

$$(*) \quad \frac{\frac{D \wedge b \geq y \supset E \quad \{E\} \ b = b - y \ \{D[a + 1/a]\}}{\{D \wedge b \geq y\} \ b := b - y \ \{D[a + 1/a]\}} \quad \{D[a + 1/a]\} \ a := a + 1 \ \{D\}}{\frac{\{D \wedge b \geq y\} \ b := b - y; \ a := a + 1 \ \{D\}}{\{D\} \ \text{while } b \geq y \ \text{do } b := b - y; \ a := a + 1 \ \{A\}}}$$

dove:

- $D \equiv x = ay + b \wedge b \geq 0$
- $E \equiv (D[a + 1/a])[b - y/b]$

- Successivamente, consideriamo la tripla  $\{C\} \ R \ \{D\}$ , dove:

$$\{C\} \ R \ \{D\} \iff \{C\} \ a := 0 \ \{D\} \implies \{D[0/a]\} \ a := 0 \ \{D\}$$

dunque otteniamo che  $C \equiv D[0/a] \equiv x = 0 \cdot y + b \wedge b \geq 0$

- Infine, consideriamo la tripla  $\{B\} \ Q \ \{C\}$ , dove:

$$\{B\} \ Q \ \{C\} \iff \{B\} \ b := y \ \{x = 0 \cdot y + b \wedge b \geq 0\}$$

Poiché:

$$x \geq 0 \implies x = 0 \cdot y + x \wedge x \geq 0$$

posto  $G \equiv (D[0/a])[y/b] \equiv C[y/b] \equiv x = 0 \cdot y + x \wedge x \geq 0$ , tramite lo *strengthening* otteniamo facilmente che:

$$\frac{x \geq 0 \supset G \quad \{G\} \ b = x \ \{x = 0 \cdot y + b \wedge b \geq 0\}}{\{x \geq 0\} \ b := y \ \{x = 0 \cdot y + b \wedge b \geq 0\}}$$

dunque concludiamo che  $B \equiv x \geq 0$

- In conclusione, dunque, abbiamo che:

$$\frac{\frac{x \geq 0 \supset (D[0/a])[y/b] \quad \{(D[0/a])[y/b]\} \ b := x \ \{D[0/a]\}}{\{x \geq 0\} \ b := y \ \{D[0/a]\}} \quad \{D[0/a]\} \ a := 0 \ \{D\}}{\{x \geq 0\} \ b := x; \ a := 0; \ \text{while } b \geq y \ \text{do } b := b - y; \ a := a + 1 \ \{A\}} \quad (*)$$

dove:

- $A \equiv x = ay + b \wedge b \geq 0 \wedge b < y$
- $D \equiv x = ay + b \wedge b \geq 0$

□



## 4.2 Correttezza dei programmi funzionali

### 4.2.1 Ricorsione nei programmi funzionali

#### Definizione 4.2.1.1: Punto fisso

Data una funzione  $f : X \rightarrow X$  e un elemento  $x \in X$ , definiamo  $x$  come **punto fisso** di  $f$  se  $f(x) = x$

#### Definizione 4.2.1.2: Combinatore di punto fisso

All'interno del lambda calcolo, definiamo come **combinatore di punto fisso** (o *combinatore*  $Y$ ) la seguente funzione:

$$Y \equiv \lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$$

Equivalentemente, nel linguaggio *Fun* il combinator  $Y$  corrisponde a:

$$Y \equiv fn\ f \Rightarrow (fn\ x \Rightarrow f(xx)) (fn\ x \Rightarrow f(xx))$$

#### Teorema 4.2.1.1: Ricorsione nel lambda calcolo

Data una funzione  $h$ , l'espressione  $Yh$  applica la funzione  $h$  **ricorsivamente**

*Dimostrazione:*

- Tramite la conversione beta, notiamo facilmente che:

$$\begin{aligned} Yh &\equiv [\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))] h \xrightarrow{\beta} \\ &\quad (\lambda x. h(xx)) (\lambda x. h(xx)) \xrightarrow{\beta} \\ &\quad h((\lambda x. h(xx)) (\lambda x. h(xx))) \equiv h(Yh) \end{aligned}$$

dunque  $Yh$  è un punto fisso di  $h$

- Di conseguenza, abbiamo che:

$$Yh \equiv h(Yh) \equiv h(h(Yh)) \equiv \dots$$

#### Osservazione 4.2.1.1

All'interno dell'espressione  $Yh$ , il combinator  $Y$  genera solo la ricorsione. Di conseguenza, all'interno di  $h$  deve essere (in qualche modo) definito un caso base che possa fermare la ricorsione, poiché altrimenti si otterrebbe una valutazione infinita

**Lemma 4.2.1.1: Ricorsione debole terminante**

Dato un insieme  $A$ , un elemento  $a \in A$  e una funzione  $h : A \rightarrow A$ , si ha che:

$$\exists! \text{ omomorfismo } f : \mathbb{N} \rightarrow A \mid f(n) = \begin{cases} a & \text{se } n = 0 \\ h(f(m)) & \text{se } n = \text{succ}(m) \end{cases}$$

Di conseguenza, esiste un'unica funzione ricorsiva  $f$  definita tramite  $\mathbb{N}$  che applichi ricorsivamente  $h$  e che **termini**

*Dimostrazione.*

- Sia  $g : \mathbb{1} \rightarrow A$  la funzione nullaria che restituisce sempre  $a$
- L'algebra  $(A, g, h)$  possiede la stessa segnatura dell'algebra induttiva  $(\mathbb{N}, \text{zero}, \text{succ})$ , dunque per la [Segnatura equivalente ad un'algebra induttiva](#) ne segue che:

$$\exists! \text{ omomorfismo } f : \mathbb{N} \rightarrow A$$

dove tramite le proprietà degli omomorfismi abbiamo che:

- $f(0) = f(\text{zero}(x)) = g(f(x)) = a$
- $f(\text{succ}(m)) = h(f(m))$

□

**Esempio:**

- Siano  $B = \{\text{true}, \text{false}\}$  e  $\text{not} : B \rightarrow B : x \mapsto \bar{x}$
- Dato l'elemento  $\text{true} \in B$ , per la proposizione precedente si ha che:

$$\exists! \text{ omomorf. isEven} : \mathbb{N} \rightarrow A \mid \text{isEven}(n) = \begin{cases} \text{true} & \text{se } n = 0 \\ \text{not}(\text{isEven}(m)) & \text{se } n = \text{succ}(m) \end{cases}$$

- Analogamente, dato l'elemento  $\text{false} \in B$  si ha che:

$$\exists! \text{ omomorf. isOdd} : \mathbb{N} \rightarrow A \mid \text{isOdd}(n) = \begin{cases} \text{false} & \text{se } n = 0 \\ \text{not}(\text{isOdd}(m)) & \text{se } n = \text{succ}(m) \end{cases}$$

**Definizione 4.2.1.3: Operatore  $\rho$** 

Dato il linguaggio  $Fun$ , se l'espressione  $M$  è valutabile in un elemento  $a \in A$  e l'espressione  $N$  è valutabile in una funzione  $h : A \rightarrow A$ , il seguente operatore:

$$(\rho M N)$$

viene valutato nell'unico omomorfismo  $f : \mathbb{N} \rightarrow A$  generato dalla [Ricorsione debole terminante](#)

**Esempio:**

- Dati i booleani di Church, la valutazione di  $(\rho \text{ True Not})$  corrisponde all'unico omomorfismo **isEven**, dunque si ha che:

$$(\rho \text{ True Not})\ 0 \equiv \text{True}$$

$$(\rho \text{ True Not})(\text{succ } n) \equiv \text{Not}((\rho \text{ True Not})\ n)$$