"Sapienza" Università di Roma
Ingegneria dell'Informazione,
Informatica e Statistica
Dipartimento di Informatica

Linguaggi di Programmazione

Author
Simone Bianco

27 novembre 2023

Indice

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**
- 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.

1

Struttura e Rappresentazione

1.1 Algebre induttive

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

Proposizione 1: Numeri naturali di Von Neumann

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

$$0_{\mathcal{N}} := \{\}$$

$$1_{\mathcal{N}} := \{\,\{\}\,\}$$
$$2_{\mathcal{N}} := \{\,\{\}, \{\{\}\}\,\}$$
$$3_{\mathcal{N}} := \{\,\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\,\}$$
$$\ldots$$

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

Dimostrazione.

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

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

   Per contro-nominale, dunque, otteniamo che:

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

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

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

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

5. Supponiamo per assurdo che $\exists S \subseteq \mathcal{N} \mid (0_{\mathcal{N}} \in S \land (n \in S \implies \mathrm{succ}_{\mathcal{N}}(n) \in S)) \land S \neq \mathcal{N}$. 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 $n \in S \implies \mathrm{succ}_{\mathcal{N}}(n) \in S$, 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 $\mathrm{succ}_{\mathcal{N}}$ 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 $\mathcal{N} - S$:

   $$n_1 \to n_2 \to \dots \to n_k \to n_1$$

   Notiamo a questo punto che:

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

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

$\square$

Principio 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:

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

Osservazione 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 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:

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

  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 2: Insieme unità

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

Definizione 3: Funzione nullaria

Definiamo una funzione $f : \mathbb{1} \to S$, 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 $\text{zero} : \mathbb{1} \to \mathbb{N} : x \mapsto 0$, indichiamo $\text{zero}(x)$ direttamente come zero

Osservazione 3

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

Definizione 4: Segnatura di una funzione

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

Definizione 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 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 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 γ definita su A esiste un'operazione δ definita su B per cui invertendo B con A all'interno della segnatura di δ si ottiene la segnatura di γ

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 $\text{zero} : \mathbb{1} \to \mathbb{N}$ è equivalente alla segnatura di $\text{zero}_{\mathcal{N}} : \mathbb{1} \to \mathcal{N}$
  - La segnatura di $\text{succ} : \mathbb{N} \to \mathbb{N}$ è equivalente alla segnatura di $\text{succ}_{\mathcal{N}} : \mathcal{N} \to \mathcal{N}$

Definizione 8: Algebra induttiva e Costruttori

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

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

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

Esempi:

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

  L'algebra (B, not) non è un'algebra induttiva poiché:

  $$\exists \varnothing \subseteq B \mid (\forall x \in \varnothing \;\; \mathrm{not}(x) \in \varnothing) \land \varnothing \neq B$$

Proposizione 2: Algebra induttiva dei naturali

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

Dimostrazione.

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

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

  - $\mathrm{zero} \in S \implies \mathrm{succ}(\mathrm{zero}) \in S$
  - $\mathrm{succ}(\mathrm{zero}) \in S \implies \mathrm{succ}(\mathrm{succ}(\mathrm{zero})) \in S$
  - $\ldots$
  - $\mathrm{succ}(...(\mathrm{zero})) \in S \implies x = \mathrm{succ}(\mathrm{succ}(...(\mathrm{zero}))) \in S$

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

$\square$

Osservazione 4

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

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

Definizione 9: Omomorfismo

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

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

Esempio:

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

Definizione 10: Isomorfismo

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

Osservazione 5

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

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

(*dimostrazione omessa*)

Osservazione 6

Data una funzione $f : A \to B$, 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 $\exp : \mathbb{R} \to \mathbb{R}_{>0} : x \mapsto e^x$ è un isomorfismo, poiché exp è un omomorfismo e $\exists \ln : \mathbb{R}_{>0} \to \mathbb{R} \mid \ln(\exp(x)) = x$, dunque f è biettiva

1.1.1 Lemma di Lambek

Proposizione 3: Segnatura equivalente ad un'algebra induttiva

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

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

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

(*dimostrazione omessa*)

Lemma 1: Lemma di Lambek (versione ridotta)

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

Dimostrazione.

- Per la proposizione precedente, si ha che:

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

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

- Consideriamo quindi la funzione $g \circ f : A \to A : x \mapsto g(f(x))$ 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 $\text{id} : A \to A : x \mapsto x$ 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 11: Insieme delle liste finite

Definiamo `List<T>` come l'insieme delle liste finite di elementi di T

Esempio:

- Dato `List<Int>`, si ha che $[3 \to 5 \to 1] \in$ `List<Int>`

Proposizione 4: Algebra induttiva delle liste finite

La tripla $(\texttt{List<T>}, \text{empty}, \text{cons})$, dove:

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

è un'algebra induttiva

Dimostrazione.

1. La funzione empty risulta essere iniettiva poiché nullaria.

   Dati $\ell_1, \ell_2 \in \texttt{List<T>}$ e $x_1, x_2 \in T$, supponiamo che:

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

   Per definizione stessa di cons, si ha che:

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

   $$\implies y_1 = y_2 = x, \ell_1 = \ell_2 = [x_1 \to \ldots \to x_n]$$

   dunque anche cons risulta iniettiva

2. $\text{im}(\text{empty}) \cap \text{im}(\text{cons}) = \{[\ ]\} \cap (\texttt{List<T>} - \{[\ ]\}) = \varnothing$

3. Sia $S \subseteq \texttt{List<T>}$ tale che $\forall x \in T, \ell \in \texttt{List<T>}\ \ \text{cons}(x, \ell) \in S$ e $\text{empty} \in S$.

   Preso $\ell := [x_1 \to x_2 \to \ldots \to x_n] \in \texttt{List<T>}$, possiamo esprimere ℓ come

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

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

□

Osservazione 7

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

Osservazione 8

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

Esempio:

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

$$\text{concat} : \texttt{List<T>} \times \texttt{List<T>} \to \texttt{List<T>}$$

  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 `List<Int>`, abbiamo che:

$$\text{concat}([1 \to 5], [7 \to 2]) = \text{concat}(\text{cons}(1, [5], [7 \to 2])) = \text{cons}(1, \text{concat}([5], [7 \to 2])) =$$
$$\text{cons}(1, \text{concat}(\text{cons}(5, \text{empty}), [7 \to 2])) = \text{cons}(1, \text{cons}(5, \text{concat}(\text{empty}, [7 \to 2]))) =$$
$$\text{cons}(1, \text{cons}(5, [7 \to 2])) = \text{cons}(1, [5 \to 7 \to 2]) = [1 \to 5 \to 7 \to 2]$$

Definizione 12: Insieme degli alberi binari finiti

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

Proposizione 5: Algebra induttiva degli alberi binari finiti

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

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

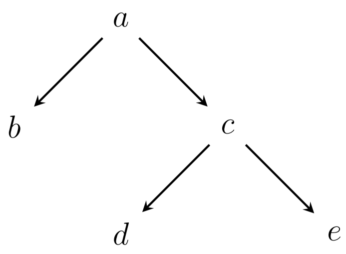
$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ t_{sx} & & t_{dx} \end{tikzcd}$$

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 13: 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: Relazione tra nodi e foglie

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

Dimostrazione per induzione strutturale.

- Definiamo l'operazione

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

  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 $t \in \texttt{BinTree}$, 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 $n = \textbf{leaves}(\text{leaf}) = 1$ foglie. Difatti, si ha che $k = 1 = 2n - 1$

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

Passo induttivo. Se $t \neq \text{leaf}$, allora $\exists t_1, t_2 \in \texttt{BinTree} \mid t = \text{branch}(\text{t}_1, \text{t}_2)$ 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:

$$k = k_1 + k_2 + 1 = 2\text{leaves}(t_1) - 1 + 2\text{leaves}(t_2) - 1 + 1 = 2(\text{leaves}(t_1) + \text{leaves}(t_2)) - 1 =$$

$$= 2(\text{leaves}(\text{branch}(t_1, t_2))) - 1 = 2(\text{leaves}(t)) - 1$$

□

1.3 Sintassi astratta

Definizione 14: Linguaggio

Definiamo come **linguaggio** un insieme di stringhe

Definizione 15: 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:

`<symbol>` ::= `_expression_`

dove:

- `<symbol>` 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
- `<_expression_>` 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:

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

  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 16: 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 $\text{eval} : L \to \mathbb{N}$ in grado di valutare le espressioni del linguaggio:
$$\begin{gathered} \text{eval}(\texttt{"0"}) = 0 \\ \text{eval}(\texttt{"1"}) = 1 \\ \dots \\ \text{eval}(\texttt{"M + N"}) = \text{eval}(\texttt{"M"}) + \text{eval}("N") \\ \text{eval}(\texttt{"M * N"}) = \text{eval}(\texttt{"M"}) + \text{eval}("N") \end{gathered}$$
- Notiamo quindi che la grammatica definisca in modo astratto (ma concretamente tramite eval) le seguenti operazioni:
$$\begin{gathered} 0 : \mathbb{1} \to \mathbb{N} : x \mapsto 0 \\ 1 : \mathbb{1} \to \mathbb{N} : x \mapsto 1 \\ \dots \\ \text{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m + n \\ \text{times} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m \cdot n \end{gathered}$$
- Notiamo però che le operazioni plus e times non risultano essere né iniettive né con immagini disgiunte. Di conseguenza, la funzione 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 2: Algebra induttiva dei termini

Dato un linguaggio L con una sintassi astratta con termini definiti in T, esiste sempre un'algebra induttiva (T, α). 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 17: Il linguaggio Exp

Definiamo come Exp il linguaggio rappresentato dalla seguente grammatica:

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

dove:

- $k \in \{0, 1, \dots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \dots\}$ ossia è una **variabile**
- $+ : Exp \times Exp \to Exp$ la quale **somma le due espressioni**
- $let : Var \times Exp \times Exp \to Exp$ 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.
- $Val = \{0, 1, \dots\}$ è l'**insieme dei valori** in cui un'espressione può essere valutata

Esempi:

- L'espressione $let\ x = 3\ in\ x + 1$ 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 $let\ x = 3\ in\ 7$ viene valutata come 7
- L'espressione $let\ y = 9\ in\ (let\ x = (let\ y = 2\ in\ y + 1)\ in\ x + y)$ 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 18: 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 19: Variabile libera

Data un'espressione $expr \in Exp$, definiamo $x \in expr$ come **libera** se x non ha un valore assegnato durate 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 6: Variabili libere in Exp

Dato il linguaggio Exp, la funzione

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

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

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

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:

$$\text{free}(let\ x = (let\ y = 2\ in\ y+1)\ in\ x+y) =$$

$$= \text{free}(let\ y = 2\ in\ y+1) \cup (\text{free}(x+y) - \{x\}) =$$
$$= \text{free}(let\ y = 2\ in\ y+1) \cup ((\text{free}(x) \cup \text{free}(y)) - \{x\}) =$$
$$= \text{free}(let\ y = 2\ in\ y+1) \cup ((\{x\} \cup \{y\}) - \{x\}) =$$
$$= \text{free}(let\ y = 2\ in\ y+1) \cup \{y\} =$$

$$= (\text{free}(2) \cup (\text{free}(y+1) - \{y\})) \cup \{y\} =$$
$$= ((\text{free}(y)) - \{y\}) \cup \{y\} =$$
$$= \{y\}$$

dunque l'espressione è invalutabile

Definizione 20: 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 21: Concatenazione di ambienti

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

$$\cdot : Env \times Env \to Env$$

dove:

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

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 7: Regola di inferenza

Data la proposizione:

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

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

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

Definizione 22: Semantica operazionale di *Exp*

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

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

definiamo come **giudizio operazionale** 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 8: Regole operazionali di *Exp*

Definiamo come **regole operazionali** le regole di inferenza che dettano le valutazioni effettuate dalla semantica operazionale:

- 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:

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

Osservazione 9: Ambiente iniziale

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

Osservazione 10: Variabili invalutabili

Dato un ambiente $E \in Env$, se $x \notin dom(E)$, 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 $x \notin dom(\varnothing)$, dunque:

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

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

$$\frac{\varnothing \vdash 1 \leadsto 1 \quad \dfrac{\{(x,1)\} \vdash x \leadsto 1 \quad \{(x,1)\} \vdash 4 \leadsto 4}{\{(x,1)\} \vdash x + 1 \leadsto 5}}{\varnothing \vdash let\ x = 1\ in\ x + 4 \leadsto 5}$$

Definizione 23: Albero di derivazione

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

Esempio:

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

$$\frac{\varnothing \vdash 3 \leadsto 3 \quad \dfrac{\{y,3\} \vdash 7 \leadsto 7 \quad \dfrac{\{(y,3),(x,7)\} \vdash x \leadsto 7 \quad \{(y,3),(x,7)\} \vdash y \leadsto 3}{\{(y,3),(x,7)\} \vdash x + y \leadsto 10}}{\{(y,3)\} \vdash let\ x = 7\ in\ x + y \leadsto 10}}{\varnothing \vdash let\ y = 3\ in\ (let\ x = 7\ in\ x + y) \leadsto 10}$$

- 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*:

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

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 24: 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 25: 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 9: 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:

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

- 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 11

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:

$$\dfrac{\dfrac{\dots}{\varnothing \vdash M \leadsto v'} \qquad \dfrac{\{(x,v')\} \vdash x \leadsto v' \quad \{(x,v')\} \vdash x \leadsto v'}{\{(x,v')\} \vdash x + x \leadsto v}}{\varnothing \vdash let\ x = M\ in\ x + x \leadsto v}$$

  dove $v = v' + v'$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\{(x,M)\} \vdash M \leadsto v'}}{\{(x,M)\} \vdash x \leadsto v'} \qquad \dfrac{\dfrac{\dots}{\{(x,M)\} \vdash M \leadsto v'}}{\{(x,M)\} \vdash x \leadsto v'}}{\{(x,M)\} \vdash x + x \leadsto v}}{\varnothing \vdash let\ x = M\ in\ x + x \leadsto v}$$

  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:

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

Prima di tutto, valutiamo tale espressione tramite valutazione eager:

$$\dfrac{\varnothing \vdash 3 \leadsto 3 \qquad \dfrac{\{(x,3)\} \vdash x \leadsto 3 \qquad \dfrac{E \vdash 7 \leadsto 7 \qquad \dfrac{E\{(x,7)\} \vdash y \leadsto 3 \quad E\{(x,7)\} \vdash x \leadsto 7}{E\{(x,7)\} \vdash y + x \leadsto 10}}{\{(x,3),(y,3)\} \vdash let\ x = 7\ in\ y + x \leadsto 10}}{\{(x,3)\} \vdash let\ y = x\ in\ (let\ x = 7\ in\ y + x) \leadsto 10}}{\varnothing \vdash let\ x = 3\ in\ (let\ y = x\ in\ (let\ x = 7\ in\ y + x)) \leadsto 10}$$

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

Valutiamo ora invece tale espressione utilizzando una valutazione lazy:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E\{(x,7)\} \vdash 7 \leadsto 7}{E\{(x,7)\} \vdash x \leadsto 7}}{E\{(x,7)\} \vdash y \leadsto 7} \qquad \dfrac{E\{(x,7)\} \vdash 7 \leadsto 7}{E\{(x,7)\} \vdash x \leadsto 7}}{E\{(x,7)\} \vdash y + x \leadsto 14}}{\{(x,3),(y,x)\} \vdash let\ x = 7\ in\ y + x \leadsto 14}}{\{(x,3)\} \vdash let\ y = x\ in\ (let\ x = 7\ in\ y + x) \leadsto 14}}{\varnothing \vdash let\ x = 3\ in\ (let\ y = x\ in\ (let\ x = 7\ in\ y + x)) \leadsto 14}$$

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 26: 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 27: Scoping dinamico

Definiamo un linguaggio come linguaggio a **scoping statico** 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 $let\ y = x\ in\ \ldots$ (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 10: 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:

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

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

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

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

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

Valutiamo quindi l'espressione precedente utilizzando una semantica lazy statica:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\varnothing \vdash 3 \rightsquigarrow 3}{E \vdash x \rightsquigarrow 3}}{E'' \vdash y \rightsquigarrow 3} \quad \dfrac{E' \vdash 7 \rightsquigarrow 7}{E'' \vdash x \rightsquigarrow 7}}{E'\{(x,(7,E'))\} \vdash y + x \rightsquigarrow 10}}{E\{(y,(x,E))\} \vdash let\ x = 7\ in\ y + x \rightsquigarrow 10}}{\dfrac{\{(x,(3,\varnothing))\} \vdash let\ y = x\ in\ (let\ x = 7\ in\ y + x) \rightsquigarrow 10}{\varnothing \vdash let\ x = 3\ in\ (let\ y = x\ in\ (let\ x = 7\ in\ y + x)) \rightsquigarrow 10}}$$

dove $E := \{(x,(3,\varnothing))\}$, $E' := E\{(y,(x,E))\}$ e $E'' := E'\{(x,(7,E'))\}$. Notiamo quindi che la valutazione nel caso di Exp lazy statico coincida con la valutazione nel caso di Exp eager statico.

Osservazione 12: Linguaggio Exp eager dinamico

All'interno del linguaggio Exp **non vi è distinzione** tra semantica eager statica e eager dinamica, poiché nessuna delle valutazioni dei termini della grammatica di Exp viene influenzata dal tipo di scoping.

Per tanto, all'interno di Exp parliamo direttamente di **semantica eager**

Definizione 28: Equivalenza tra semantiche operazionali

Sia L un linguaggio. Date due semantiche operazionali definite su L, definiamo tali semantiche come **equivalenti** se ogni espressione di L restituisce lo stesso risultato per entrambe le semantiche a seguito della valutazione

Teorema 3: Equivalenze semantiche di Exp

Dato il linguaggio Exp, si ha che:

$$Exp \text{ eager } \equiv Exp \text{ lazy statico } \not\equiv Exp \text{ lazy dinamico}$$

Osservazione 13

In base alla semantica utilizzata, possono generarsi problemi diversi durante le valutazioni

Esempio:

- Consideriamo la seguente espressione $let\ x = x\ in\ x$
- Utilizzando una semantica eager statica o lazy statica, otteniamo che il termine interno del *let* sia invalutabile
- Utilizzando una semantica lazy dinamica, la valutazione entrerà in un loop infinito (si consiglia di provare ad scrivere l'albero di derivazione)

2.4 *Fun*: un linguaggio con funzioni

Definizione 29: Il linguaggio *Fun*

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

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

dove:

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

Esempi:

- L'espressione $(fn\ x \Rightarrow x + 1)\ 7$ viene valutata come 8, poiché la funzione sinistra $fn\ x \Rightarrow x + 1$ viene applicata al termine destro 7 (dunque 7 viene utilizzato come argomento della funzione per il parametro x)
- L'espressione $(fn\ x \Rightarrow x\ 3)\ 7$ 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 $(fn\ x \Rightarrow x\ 3)(fn\ x \Rightarrow x+1)$ viene valutata come 4, poiché l'argomento $fn\ x \Rightarrow x + 1$ viene passato come parametro x della funzione $fn\ x \Rightarrow x\ 3$, per poi valutare l'applicazione $x\ 3$ passando l'argomento 3 come parametro per la funzione contenuta in x (ossia $fn\ x \Rightarrow x + 1$).

  <u>Informalmente</u>, possiamo dire che:

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

Osservazione 14

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 30: Insieme delle funzioni da X ad Y

Dati due insiemi X e Y, indichiamo con $(X \to Y)$ l'insieme di tutte le funzioni da X ad Y:

$$(X \to Y) = \{f \mid f : X \to Y\}$$

dove $|X \to Y| = |Y|^{|X|}$

Teorema 4: Curryficazione

Dati X, Y e Z, la seguente funzione risulta essere biettiva:

$$\text{curry} : (X \times Y \to Z) \to (X \to (Y \to Z)) : f \mapsto h \;\mid\; f(x,y) = h(x)(y)$$

Inoltre, definiamo come **curryficazione** l'applicazione di tale funzione

Dimostrazione.

- La funzione risulta essere iniettiva:

$$\varphi(f) = \varphi(f') \implies \forall x \in X, y \in Y \;\; \varphi(f)(x)(y) = \varphi(f)(x)(y) \implies$$

$$\forall x \in X, y \in Y \;\; h(x)(y) = h'(x)(y) \implies \forall x \in X, y \in Y \;\; f(x,y) = f'(x,y) \implies f = f'$$

- Inoltre, abbiamo che:

$$|X \times Y \to Z| = |Z|^{|X \times Y|} = |Z|^{|X| \cdot |Y|} = (|Z|^{|Y|})^{|X|} =$$

$$|Y \to Z|^{|X|} = |X \to (Y \to Z)|$$

- Di conseguenza, φ risulta essere biettiva

$\square$

Osservazione 15: Curryficazione in *Fun*

Dato il linguaggio *Fun*, definiamo la seguente contrazione sintattica:

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

data dalla curryficazione del primo termine

Esempi:

- La curryficazione 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:

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

Osservazione 16

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

Proposizione 11: 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 12: 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

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

$$Fun \text{ eager dinamico } \not\equiv Fun \text{ eager statico}$$

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:

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

$$\frac{\varnothing \vdash 7 \rightsquigarrow 7 \quad \dfrac{E \vdash fn\ y \Rightarrow M \rightsquigarrow (y, M) \quad E \vdash fn\ z \Rightarrow x \rightsquigarrow (z, x) \quad (*)}{E \vdash (fn\ y \Rightarrow M)(fn\ z \Rightarrow x) \rightsquigarrow 3}}{\varnothing \vdash let\ x = 7\ in\ ((fn\ y \Rightarrow M)(fn\ z \Rightarrow x)) \rightsquigarrow 3}$$

  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:

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

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

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 13: Linguaggio *Fun* lazy dinamico

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

- L'insieme Env viene ridefinito come:

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

- 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 14: Linguaggio *Fun* lazy statico

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

- L'insieme Env viene ridefinito come:

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

- 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 17

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

$$Fun \text{ lazy dinamico } \not\equiv Fun \text{ lazy statico}$$

Definizione 31: Espressione ω

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

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

In particolare, l'espressione ω è **invalutabile per qualsiasi semantica**

Esempio:

- Analizziamo l'albero di derivazione di ω utilizzando una semantica eager statica:

$$(*) \qquad \varnothing \vdash x \rightsquigarrow (x, xx, \varnothing) \quad \varnothing \vdash x \rightsquigarrow (x, xx, \varnothing) \quad \frac{(*)}{(x, \{(x, xx, \varnothing)\}) \vdash xx \rightsquigarrow v}$$

$$\frac{\varnothing \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \varnothing) \quad \varnothing \vdash fn\ x \Rightarrow xx \rightsquigarrow (x, xx, \varnothing) \quad \dfrac{(*)}{(x, \{(x, xx, \varnothing)\}) \vdash xx \rightsquigarrow v}}{\varnothing \vdash (fn\ x \Rightarrow xx)(fn\ x \Rightarrow xx) \rightsquigarrow v}$$

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

Lemma 3

Dato il linguaggio *Fun*, si ha che:

$$Fun \text{ eager statico } \not\equiv Fun \text{ lazy statico}$$

$$Fun \text{ eager dinamico } \not\equiv Fun \text{ lazy dinamico}$$

Dimostrazione.

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

$\square$

Teorema 5: Equivalenze semantiche di *Fun*

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

Proposizione 15: Variabili libere in *Fun*

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

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

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 $let\ x = M\ in\ N$ corrisponde a `let val x = M in N end`
- L'operatore $fn\ x \Rightarrow M$ 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)
- L'espressione va terminata da un **punto e virgola**
- La semantica utilizzata è **eager statica**

Ad esempio, l'espressione:

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

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 32: 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 λ 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 18: 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 33: 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 19: 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 34: 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 35: Alfa equivalenza

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

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

Esempi:

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

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

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

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

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

dunque ne concludiamo che:

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

Definizione 36: 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 20

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 21

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 ω sarebbe stato valutato (il quale ricordiamo essere invalutabile)

Definizione 37: 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 $x \notin \mathrm{free}(M)$:

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

Esempi:

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

$$\mathrm{free}(\lambda y.y) = \{\mathrm{free}(y) - \{y\}\} = \{y\} - \{y\} = \varnothing \implies x \notin \mathrm{free}(\lambda y.y)$$

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:

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

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 <u>non</u> sono effettivamente la stessa espressione).

Osservazione 22

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

$$fn\ x_1 \ldots x_n \Rightarrow M$$

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 16: Numeri naturali di Church

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

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

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

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

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

$$\ldots$$

dove $\text{succ}_{\mathcal{N}_\lambda} : \mathcal{N}_\lambda \to \mathcal{N}_\lambda : n \mapsto n \cup \{n\}$, 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:

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

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

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

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

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

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

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

Successivamente, definiamo le seguenti ulteriori funzioni matematiche:

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

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

  oppure:

$$\texttt{sum} := fn\ z \Rightarrow fn\ w \Rightarrow z\ \ \texttt{succ}\ \ w$$

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

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow z(wx)y)$$

  oppure:

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow z(\texttt{sum}\ \ w)\ \texttt{zero}$$

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

$$\texttt{prod} := fn\ z \Rightarrow fn\ w \Rightarrow wz$$

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:

$$\texttt{True} := fn\ x \Rightarrow fn\ y \Rightarrow x$$

$$\texttt{False} := fn\ x \Rightarrow fn\ y \Rightarrow y$$

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

$$\texttt{evalBool} := fn\ z \Rightarrow z\ true\ false$$

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*:

$$\texttt{ITE} := fn\ z \Rightarrow fn\ u \Rightarrow fn\ v \Rightarrow z\ u\ v$$

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

$$\texttt{If} := fn\ z \Rightarrow fn\ u \Rightarrow z\ u\ \texttt{True}$$

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

$$\texttt{Not} := fn\ z \Rightarrow fn\ x \Rightarrow fn\ y \Rightarrow z\ y\ x$$

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

$$\texttt{Or} := fn\ z \Rightarrow fn\ w \Rightarrow \texttt{If}(\texttt{Not}\ z)w$$

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

$$\texttt{And} := fn\ z \Rightarrow fn\ w \Rightarrow \texttt{Not}(\texttt{If}\ z\ (\texttt{Not}\ w))$$

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);
```

2.6 Ricorsione nei linguaggi funzionali

Definizione 38: Punto fisso

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

Definizione 39: Combinatore di punto fisso

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

$$\mathrm{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Equivalentemente, nel linguaggio *Fun* il combinatore Y corrisponde a:

$$\mathrm{Y} \equiv fn\ f \Rightarrow (fn\ x \Rightarrow f(xx))(fn\ x \Rightarrow f(xx))$$

Teorema 6: Ricorsione nel lambda calcolo

Data una funzione h, l'espressione $\mathrm{Y}h$ applica la funzione h **ricorsivamente**

Dimostrazione:

- Tramite la beta conversione, notiamo facilmente che:

$$\mathrm{Y}h \equiv [\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))]\, h \xrightarrow{\beta}$$

$$(\lambda x.h(xx))(\lambda x.h(xx)) \xrightarrow{\beta}$$

$$h((\lambda x.h(xx))(\lambda x.h(xx))) \equiv h(\mathrm{Y}h)$$

  dunque $\mathrm{Y}h$ è un punto fisso di h

- Di conseguenza, abbiamo che:

$$\mathrm{Y}h \equiv h(\mathrm{Y}h) \equiv h(h(\mathrm{Y}h)) \equiv \dots$$

Osservazione 23

All'interno dell'espressione $\mathrm{Y}h$, il combinatore Y genera <u>solo</u> 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: Ricorsione tramite numeri naturali

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

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

Inoltre, definiamo tale insieme A come un **oggetto su numeri naturali** (da *Natural Numbers Object* in teoria delle categorie)

Dimostrazione.

- Sia unit : $\mathbb{1} \to A$ la funzione nullaria che restituisce sempre a
- L'algebra (A, unit, 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:

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

  dove tramite le proprietà degli omomorfismi abbiamo che:

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

□

Esempio:

- Siano $B = \{true, false\}$ e not : $B \to B : x \mapsto \overline{x}$
- Dato l'elemento $true \in B$, per il lemma precedente si ha che:

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

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

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

Definizione 40: Operatore ρ

Dato il linguaggio Fun, definiamo l'operatore $\rho\ M\ N$ come:

$$(\rho\ M\ N)\ L = \begin{cases} M & \text{se } L = 0 \\ N\ ((\rho\ M\ N)\ n) & \text{se } L = \texttt{succ}\ n \end{cases}$$

In altre parole, se M è un valore di un insieme A e N è una funzione da A in A, l'operatore $\rho\ M\ N$ restituisce l'unica funzione dettata dalla Ricorsione tramite numeri naturali

Esempio:

- Dati i booleani di Church, la valutazione di $(\rho \texttt{ True Not})$ corrisponde alla funzione isEven : $\mathbb{N} \to A$ definita nell'esempio precedente
- Difatti, abbiamo che:

$$(\rho \texttt{ True Not})\ L \equiv \begin{cases} \texttt{True} & \text{se } L = 0 \\ \texttt{Not}((\rho \texttt{ True Not})\ n) & \text{se } L = \texttt{succ}\ n \end{cases}$$

Teorema 7: Unica funzione ricorsiva primitiva

Dato un insieme A, un elemento $a \in A$ e una funzione $h : A \times \mathbb{N} \to A$, si ha che:

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

Inoltre, definiamo f come l'**unica funzione ricorsiva primitiva tramite** h

Dimostrazione.

- Consideriamo l'elemento $(a, 0) \in A \times \mathbb{N}$ e la seguente funzione

$$\hat{h} : A \times \mathbb{N} \to A \times \mathbb{N} : (x, n) \mapsto (h(x, n), \text{succ}(n))$$

- Per il lemma della Ricorsione tramite numeri naturali, si ha che:

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

- Sia $f : \mathbb{N} \to A$ la funzione tale che:

$$f(n) = b \implies \exists m \in A \mid \hat{f}(n) = (b, m)$$

- Per induzione (*dio solo sa come*) si ha che $\forall n \in \mathbb{N}\ \ \hat{f}(n) = (f(n), n)$
- Di conseguenza, dal risultato precedente e dalla definizione stessa di $\hat{f}$, otteniamo che:
  - $(f(0), 0) = \hat{f}(0) = (a, 0) \implies f(0) = a$
  - $(f(\text{succ}(n)), \text{succ}(n)) = \hat{f}(\text{succ}(n)) = (\hat{h}(\hat{f}(n))) = (h(f(n), n), \text{succ}(n)) \implies f(\text{succ}(n)) = h(f(n), n)$
- Supponiamo quindi per assurdo che esista un'altra funzione $g : \mathbb{N} \to A$ diversa da f (dunque $g \neq f$) tale che:

$$g(n) = \begin{cases} a & \text{se } n = 0 \\ h(g(m), m) & \text{se } n = \text{succ}(m) \end{cases}$$

- Poiché $g(0) = a = f(0)$, affinché valga $g \neq f$ ne segue necessariamente che:

$$\exists k \in \mathbb{N} \mid g(\text{succ}(k)) \neq f(\text{succ}(k))$$

- Data la funzione $\hat{g} : \mathbb{N} \to \mathbb{N} \times A : n \mapsto (g(n), n)$, si ha che:

$$\hat{f}(\text{succ}(k)) = (f(\text{succ}(k)), \text{succ}(k)) \neq (g(\text{succ}(k)), \text{succ}(k)) = \hat{g}(\text{succ}(k)) \implies \hat{g} \neq \hat{f}$$

- Inoltre, tramite la definizione stessa di $\hat{g}$ abbiamo che:
  - $\hat{g}(0) = (g(0), 0) = (a, 0)$
  - $\hat{g}(\text{succ}(n)) = (g(\text{succ}(n)), \text{succ}(n)) = (h(g(n), n), \text{succ}(n))$

  contraddicendo la condizione secondo cui $\hat{f}$ sia l'unica funzione da $\mathbb{N}$ ad A godente di tali proprietà

- Di conseguenza, ne segue necessariamente che tale funzione g non esista e dunque che f sia l'unica funzione avente tali proprietà

$\square$

Corollario 1: Unica funzione ricorsiva primitiva curryficata

Dato un insieme A, un elemento $a \in A$ e una funzione $h : A \to (\mathbb{N} \to A)$, tramite la **curryficazione** abbiamo che:

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

Definizione 41: Operatore *rec*

Dato il linguaggio Fun, definiamo l'operatore $rec\ M\ N$ come:

$$(rec\ M\ N)\ L = \begin{cases} M & \text{se } L = 0 \\ N\ ((rec\ M\ N)\ n)\ n & \text{se } L = \texttt{succ}\ n \end{cases}$$

In altre parole, se M è un valore di un insieme A e N è una funzione da A in $(\mathbb{N} \to A)$, l'operatore $\rho\ M\ N$ restituisce l'Unica funzione ricorsiva primitiva

Osservazione 24

Dato il linguaggio Fun, si ha che:

$$rec\ M\ (fn\ x \Rightarrow fn\ y \Rightarrow N\ x) \equiv \rho\ M\ N$$

Esempio:

1. • Vogliamo costruire la funzione `fatt` definita come:

$$\texttt{fatt}\ M \equiv \begin{cases} 1 & \text{se } M = 0 \\ (\texttt{fatt}\ n) * (\texttt{succ}\ n) & \text{se } M = \texttt{succ}\ n \end{cases}$$

   • Notiamo che:

$$(\texttt{succ}\ n) * (\texttt{fatt}\ n) \equiv (fn\ x \Rightarrow fn\ y \Rightarrow x * (\texttt{succ}\ y))(\texttt{fatt}\ n)\ n$$

   • Di conseguenza, posta la funzione:

$$h \equiv (fn\ x \Rightarrow fn\ y \Rightarrow x * (\texttt{succ}\ y))$$

   otteniamo che:

$$\texttt{fatt}\ M \equiv \begin{cases} 1 & \text{se } M = 0 \\ h\ (\texttt{fatt}\ n)\ n & \text{se } M = \texttt{succ}\ n \end{cases}$$

   • Poiché $1 \in \mathbb{N}$ e $h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, dalla definizione di *rec* concludiamo che:

$$\texttt{fatt} \equiv rec\ 1\ h \equiv rec\ 1\ (fn\ x \Rightarrow fn\ y \Rightarrow x * (\texttt{succ}\ y))$$

2. • Vogliamo costruire la funzione `twice` definita come:

$$\texttt{twice}\ M \equiv \begin{cases} 0 & \text{se } M = 0 \\ \texttt{succ}\ (\texttt{succ}\ (\texttt{twice}\ n)) & \text{se } M = \texttt{succ}\ n \end{cases}$$

   • Notiamo che:

$$\texttt{succ}\ (\texttt{succ}\ (\texttt{twice}\ n)) \equiv (fn\ x \Rightarrow fn\ y \Rightarrow \texttt{succ}\ (\texttt{succ}\ x))\ (\texttt{twice}\ n)\ n$$

   • Di conseguenza, posta la funzione:

$$h \equiv (fn\ x \Rightarrow fn\ y \Rightarrow \texttt{succ}\ (\texttt{succ}\ x))$$

   otteniamo che:

$$\texttt{twice}\ M \equiv \begin{cases} 0 & \text{se } M = 0 \\ h\ (\texttt{twice}\ n)\ n & \text{se } M = \texttt{succ}\ n \end{cases}$$

   • Poiché $1 \in \mathbb{N}$ e $h : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, dalla definizione di *rec* concludiamo che:

$$\texttt{twice} \equiv rec\ 0\ h \equiv rec\ 0\ (fn\ x \Rightarrow fn\ y \Rightarrow \texttt{succ}\ (\texttt{succ}\ x))$$

Definizione 42: Linguaggio Fun_ρ

Definiamo come Fun_ρ il linguaggio rappresentato dalla seguente grammatica:

$$M, N ::= x \mid fn\ x \Rightarrow M \mid M\ N \mid 0 \mid \text{succ}\ M \mid rec\ M\ N$$

3

Paradigma imperativo

3.1 *Imp*: un semplice linguaggio imperativo

Definizione 43: Il linguaggio *Imp*

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

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

dove:

- La prima grammatica rappresenta l'insieme Exp delle **espressioni**
- La seconda grammatica rappresenta l'insieme Imp dei **programmi**
- $k \in \{true, false\} \cup \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \ldots\}$ 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 $var\ x = 0\ in\ while\ x < 10\ do\ x := x + 1$ è un termine valido di Imp
- Il programma $var\ x = 0\ in\ while\ x < 10\ do\ y := x + 1$ non è un termine valido di Imp, poiché la variabile y non è stata dichiarata prima dell'assegnamento

Definizione 44: 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 45: Insieme degli ambienti in Imp

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

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

Definizione 46: Insieme delle memorie in Imp

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

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

Definizione 47: Concatenazione di memorie

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

$$\cdot : Store \times Store \to Store$$

dove:

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

Definizione 48: Semantiche operazionali di *Imp*

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

- La **semantica delle espressioni**

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

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

- La **semantica dei programmi**

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

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

Le regole operazionali di tali semantiche sono definite come:

- **Costanti**:

$$E \vdash k,\ S \leadsto k$$

- **Variabili**:

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

- **Somma**:

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

- **Minorazione**:

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

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

- **Skip**:

$$E \vdash skip,\ S \leadsto S$$

- **Esecuzione sequenziale**:

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

- **If-then-else**:

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

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

- **While**:

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

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

- **Dichiarazione e assegnamento**:

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

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

- **Assegnamento**:

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

Osservazione 25

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 49: Il linguaggio *All*

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

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

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**
- $k \in \{true, false\} \cup \{0, 1, \dots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \dots\}$ 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 50: Insieme delle locazioni contigue

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

Proposizione 17: Ridefinizione di *Env* in *All*

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

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

Definizione 51: 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**:

$$\frac{E\{y,(x,P,E)\} \vdash Q,\ S \overset{P}{\leadsto} S'}{E \vdash \mathit{proc}\ y(x)\ \mathit{is}\ P\ \mathit{in}\ Q,\ S \overset{P}{\leadsto} S'}$$

3.2.1 Semantiche di *All*

Definizione 52: 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 18: Linguaggio *All* call-by-value

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

- **Richiamo by-value**:

$$E(y) = (x, P, E') \implies \frac{E \vdash M,\ S \overset{M}{\leadsto} v \quad E'\{(x,l)\} \vdash P,\ S\{(l,v)\} \leadsto S'}{E \vdash \mathit{call}\ y(M),\ S \overset{P}{\leadsto} S'}$$

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

Proposizione 19: Linguaggio *All* call-by-reference

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

- **Richiamo by-reference**:

$$E(y) = (x, P, E') \implies \frac{E \vdash V,\ S \overset{V}{\leadsto} l \quad E'\{(x,l)\} \vdash P,\ S \leadsto S'}{E \vdash \mathit{call}\ y(V),\ S \overset{P}{\leadsto} S'}$$

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

Proposizione 20: 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**:

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

- **Richiamo by-name**:

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

Lemma 5

Dato il linguaggio *All*, si ha che:

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

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

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:

$$\dfrac{\varnothing \vdash 5,\ \varnothing \leadsto 5 \qquad \dfrac{\dfrac{\dfrac{E' \vdash x,\ \{(l,5)\} \overset{V}{\leadsto} l}{E' \vdash x,\ \{(l,5)\} \overset{M}{\leadsto} 5} \qquad \dfrac{E'' \vdash 1,\ S \overset{M}{\leadsto} 1 \quad E'' \vdash z,\ S \overset{V}{\leadsto} \{(l',1)\}}{E'' \vdash z := 1,\ S \leadsto S\{(l',1)\}}}{E' \vdash call\ y(x),\ \{(l,5)\} \leadsto S\{(l',1)\}}}{E \vdash proc\ y(z)\ is\ z := 1\ in\ call\ y(x),\ \{(l,5)\} \leadsto S\{(l',1)\}}}{\varnothing \vdash var\ x = 5\ in\ proc\ y(z)\ is\ z := 1\ in\ call\ y(x),\ \varnothing \leadsto S\{(l',1)\}}$$

  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:

$$\dfrac{\varnothing \vdash 5,\ \varnothing \leadsto 5 \qquad \dfrac{\dfrac{E' \vdash x,\ \{(l,5)\} \overset{V}{\leadsto} l \qquad \dfrac{E'' \vdash 1,\ \{l,5\} \overset{M}{\leadsto} 1 \quad E'' \vdash z,\ \{l,5\} \overset{V}{\leadsto} \{(l,1)\}}{E'' \vdash z := 1,\ \{l,5\} \leadsto \{(l,1)\}}}{E' \vdash call\ y(x),\ \{(l,5)\} \leadsto \{(l,1)\}}}{E \vdash proc\ y(z)\ is\ z := 1\ in\ call\ y(x),\ \{(l,5)\} \leadsto \{(l,1)\}}}{\varnothing \vdash var\ x = 5\ in\ proc\ y(z)\ is\ z := 1\ in\ call\ y(x),\ \varnothing \leadsto \{(l,1)\}}$$

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

Lemma 6

Dato il linguaggio *All*, si ha che:

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

Dimostrazione.

- Consideriamo il programma

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

- 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 $z[x] = z[1]$

□

Teorema 8: 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 Invarianti di un programma

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:

$$\begin{array}{r|l} 90 & 69 \\ 360 & 17 \\ 5760 & 1 \end{array}$$

- 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 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 53: 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 21

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 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 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 54: Logica di Hoare

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

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

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 $A \lor B$ corrisponde al termine $\neg A \supset B$
- La notazione $A \land B$ corrisponde al termine $\neg(\neg A \supset B)$
- La notazione $A \leq B$ corrisponde al termine $(A < B) \lor (A = B)$
- La notazione $A > B$ corrisponde al termine $\neg((A < B) \lor (A = B))$
- La notazione $A \geq B$ corrisponde al termine $\neg(A < B)$

Definizione 55: 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 56: Formula

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

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

Proposizione 22: 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**:
$$\frac{\{A\}\ P\ \{B_1\} \quad \dots \quad \{A\}\ P\ \{B_n\}}{\{A\}\ P\ \{B_1 \wedge \ldots \wedge B_n\}}$$
- **Or**:
$$\frac{\{A_1\}\ P\ \{B\} \quad \dots \quad \{A_n\}\ P\ \{B\}}{\{A_1 \vee \ldots \vee A_n\}\ P\ \{B\}}$$

Proposizione 23: Regole di inferenza dei programmi

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

- **Skip**:
$$\{A\}\ skip\ \{A\}$$
- **Esecuzione sequenziale**:
$$\frac{\{A\}\ P\ \{B\} \quad \{B\}\ Q\ \{C\}}{\{A\}\ P;\ Q\ \{C\}}$$
- **If-then-else**:
$$\frac{\{A \land B\}\ P\ \{C\} \quad \{A \land \neg B\}\ Q\ \{C\}}{\{A\}\ if\ B\ then\ P\ else\ Q\ \{C\}}$$
- **While**:
$$\frac{\{A \land B\}\ P\ \{A\}}{\{A\}\ while\ B\ do\ P\ \{A \land \neg B\}}$$
- **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 9: Invarianti con logica di Hoare

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

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

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:

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

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

Proposizione 24

Se $x \geq 0$, la proprietà:

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

è un'invariante della funzione `EuclideanDivision`

Dimostrazione.

- Per facilitare la lettura, poniamo:
  - $P \equiv (b := x;\ a := 0;\ \textit{while}\ b \geq y\ \textit{do}\ b := b - y;\ a := a + 1)$
  - $Q \equiv (b := x)$
  - $R \equiv (a := 0)$
  - $S \equiv (\textit{while}\ b \geq y\ \textit{do}\ b := b - y;\ a := a + 1)$
  - $A \equiv (x = ay + b \land b \geq 0 \land b < y)$
- 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:

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

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

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

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

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

- 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:

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

  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:

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

  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:

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

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

- In conclusione, dunque, abbiamo che:

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

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

Similmente alla logica di Hoare per i programmi imperativi, introduciamo un sistema logico equazionale di verifica formale dei programmi.

Definizione 57: Predicato di uguaglianza di Fun_ρ

Dato il linguaggio Fun_ρ, il predicato di uguaglianza $M = N$ permette di verificare formalmente l'equivalenza tra termini di Fun_ρ:

$$M = N \iff M \equiv N$$

Il predicato $M = N$ è definito dalle seguenti regole:

1. **Regola alfa** (α), coincidente con l'alfa equivalenza del lambda calcolo:

$$fn\ x \Rightarrow M = fn\ y \Rightarrow M[y/x]$$

2. **Regola beta** (β), coincidente con la beta conversione del lambda calcolo:

$$(fn\ x \Rightarrow M)\ N = M[N/x]$$

3. **Regola del caso base**:

$$(rec\ M\ \ N)\ 0 = M$$

4. **Regola del passo ricorsivo**:

$$(rec\ M\ \ N)\ (\text{succ}\ L) = N\ ((rec\ M\ \ N)\ L)\ L$$

5. **Regola dell'induzione**:

$$\frac{P(0) \quad P(n) \implies P(\text{succ}\ n)}{\forall m\ P(m)}$$

6. **Regola della congruenza**:

$$\frac{M = N \quad M = L}{N = L}$$

7. **Regola del contesto**:

$$\frac{M = N \quad M' = N'}{M\ N = M'\ N'}$$

8. **Regola xi** (ξ):

$$\frac{M = N}{fn\ x \Rightarrow M = fn\ x \Rightarrow N}$$

Osservazione 26

Il predicato $M = N$ gode delle seguenti proprietà:

1. **Riflessività**:
$$\frac{M}{M = M}$$
2. **Simmetria**:
$$\frac{M = N}{N = M}$$
3. **Transitività:**
$$\frac{M = N \quad N = L}{M = L}$$

di conseguenza, esso stipula una **relazione di equivalenza**

Dimostrazione.

1. Tramite le regole definite su rec e la regola della congruenza, si ha che:
$$\frac{\dfrac{M}{(rec\ M\ N)\ 0 = M \quad (rec\ M\ N)\ 0 = M}}{M = M}$$
2. Tramite la riflessività e la regola di congruenza si ha che:
$$\frac{\dfrac{\dfrac{M = N}{M = N \quad M}}{M = N \quad M = M}}{N = M}$$
3. Tramite la simmetria e la regola di congruenza si ha che:
$$\frac{\dfrac{M = N \quad N = L}{N = M \quad N = L}}{M = L}$$

□

Proposizione 25: Correttezza di `plus`

Data la funzione:

$$\texttt{plus}\ M\ N \equiv \begin{cases} N & \text{se } M = 0 \\ \text{succ}\ (\texttt{plus}\ n\ N) & \text{se } M = \text{succ}\ n \end{cases}$$

si ha che:

$$\texttt{plus} \equiv (fn\ x \Rightarrow fn\ y \Rightarrow (rec\ y\ (fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w))\ x)$$

Dimostrazione.

- Siano:
  - $h \equiv (fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w)$
  - $P(m) := [\texttt{plus}\ m\ N = (fn\ x \Rightarrow fn\ y \Rightarrow (rec\ y\ h)\ x)\ m\ N]$
- Verifichiamo che $P(0)$ sia valido:

$$(fn\ x \Rightarrow fn\ y \Rightarrow (rec\ y\ (fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w))\ x)\ 0\ N =$$

$$(rec\ N\ (fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w))\ 0 =$$
$$N = \texttt{plus}\ 0\ N$$

- Assumendo che $P(n)$ sia valido, verifichiamo che anche $P(\text{succ}\ n)$:

$$(fn\ x \Rightarrow fn\ y \Rightarrow (rec\ y\ h)\ x)\ (\text{succ}\ n)\ N =$$

$$(rec\ N\ h)\ (\text{succ}\ n) =$$
$$(fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w)((rec\ N\ h)\ n)\ n =$$
$$(fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w)(fn\ x \Rightarrow fn\ y \Rightarrow ((rec\ y\ h)\ x)\ n\ N)\ n \overset{P(n)}{=}$$
$$(fn\ w \Rightarrow fn\ z \Rightarrow \text{succ}\ w)(\texttt{plus}\ n\ N)\ n =$$
$$\text{succ}\ (\texttt{plus}\ n\ N) =$$
$$\texttt{plus}\ (\text{succ}\ n)\ N$$

- Di conseguenza, tramite la regola dell'induzione abbiamo che:

$$\frac{P(0) \quad P(n) \implies P(\text{succ}\ n)}{\forall m\ P(m)}$$

  da cui concludiamo che:

$$\texttt{plus} = (fn\ x \Rightarrow fn\ y \Rightarrow (rec\ y\ h)\ x)$$

□

Proposizione 26: Commutatività di plus

Data la funzione plus, si ha che:

$$\texttt{plus}\ M\ N \equiv \texttt{plus}\ N\ M$$

Dimostrazione.

- Sia $P(m) := \forall x\ [\texttt{plus}\ m\ x \equiv \texttt{plus}\ x\ m]$

- Verifichiamo che $P(0)$ sia valido:
  - Sia $Q(x) := [\texttt{plus}\ 0\ x \equiv \texttt{plus}\ x\ 0]$
  - Il predicato $Q(0)$ risulta valido per identità:

$$\texttt{plus}\ 0\ 0 = \texttt{plus}\ 0\ 0$$

  - Assumendo che $Q(y)$ sia valido, verifichiamo che $Q(\text{succ}\ y)$:

$$\texttt{plus}\ (\text{succ}\ y)\ 0 = \text{succ}(\texttt{plus}\ y\ 0) \overset{Q(y)}{=}$$
$$\text{succ}(\texttt{plus}\ 0\ y) = \text{succ}\ y = \texttt{plus}\ 0\ (\text{succ}\ y)$$

  dunque tramite la regola dell'induzione concludiamo che:

$$\cfrac{\cfrac{Q(0) \quad Q(y) \implies Q(\text{succ}\ y)}{\forall x\ Q(x)}}{P(0)}$$

- Assumendo che $P(n)$ sia valido, verifichiamo che anche $P(\text{succ}\ n)$:
  - Sia $R(x) := [\texttt{plus}\ (\text{succ}\ n)\ x \equiv \texttt{plus}\ x\ (\text{succ}\ n)]$
  - Il predicato $R(0)$ risulta valido per identità:

$$\texttt{plus}\ 0\ (\text{succ}\ n) = (\text{succ}\ n) =$$
$$\text{succ}(\texttt{plus}\ 0\ n) = \text{succ}(\texttt{plus}\ n\ 0) = \texttt{plus}\ (\text{succ}\ n)\ 0$$

  - Assumendo che $R(y)$ sia valido, verifichiamo che $R(\text{succ}\ y)$:

$$\texttt{plus}\ (\text{succ}\ y)\ (\text{succ}\ n) = \text{succ}(\texttt{plus}\ y\ (\text{succ}\ n)) \overset{R(y)}{=}$$
$$\text{succ}(\texttt{plus}\ (\text{succ}\ n)\ y) = \text{succ}(\text{succ}(\texttt{plus}\ n\ y)) \overset{P(n)}{=}$$
$$\text{succ}(\text{succ}(\texttt{plus}\ y\ n)) = \text{succ}(\texttt{plus}\ (\text{succ}\ y)\ n) \overset{R(y)}{=}$$
$$\text{succ}(\texttt{plus}\ n\ (\text{succ}\ y)) = \texttt{plus}\ (\text{succ}\ n)\ (\text{succ}\ y)$$

  dunque tramite la regola dell'induzione concludiamo che:

$$\cfrac{\cfrac{R(0) \quad R(y) \implies R(\text{succ}\ y)}{\forall x\ R(x)}}{P(\text{succ}\ n)}$$

- Infine, otteniamo che:

$$\cfrac{P(0) \quad P(n) \implies P(\text{succ}\ n)}{\forall m\ P(m)}$$

  da cui concludiamo che:

$$\texttt{plus}\ M\ N \equiv \texttt{plus}\ N\ M$$

$\square$

5

Sistema dei Tipi

Un **sistema dei tipi** è un sistema logico composto da un insieme di regole che assegna una proprietà detta **tipo** ad ogni **termine** di un linguaggio, dettando le operazioni che possono essere effettuate su di essi (es: il tipo di una variabile determina quali valori possano essere assegnati a tale variabile)

Lo scopo principale dei sistemi dei tipi nei linguaggi di programmazione è la riduzione della possibile presenza di bug o errori di computazione tramite il controllo della presenza di **errori di tipo** (es: impedendo che un intero possa essere sommato ad un booleano).

5.1 Lambda calcolo tipato

Definizione 58: Lambda calcolo tipato

Definiamo come **lambda calcolo tipato** il sistema dei tipi rappresentato dalla seguente grammatica:

$$\begin{array}{rcl} A, B & ::= & K \mid A \to B \\ M, N & ::= & k \mid x \mid \lambda x : A.M \mid M\ N \end{array}$$

dove:

- La prima grammatica rappresenta l'insieme dei **tipi**, indicato con $Types$
- La seconda grammatica rappresenta l'insieme dei **termini**, indicato con $Terms$
- $k \in \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in \{x, y, z, \ldots\}$ ossia è una **variabile**
- $K \in \{\texttt{Int}, \texttt{Bool}, \texttt{String}, \ldots\}$ ossia è un **tipo base**

Definizione 59: Insieme dei contesti

Dato il lambda calcolo tipato, definiamo come **insieme dei contesti**, indicato con Ctx, l'insieme delle funzioni parziali che associano ogni variabile al proprio tipo:

$$Ctx = \{f \mid f : Var \overset{fin}{\rightarrow} Val\}$$

Inoltre, dato un contesto $\Gamma \in Ctx$, se $\Gamma(x) = A$ usiamo la notazione infissa $x : A$

Definizione 60: Concatenazione di contesti

Dato il lambda calcolo tipato, definiamo l'operazione di **concatenazione di contesti**, ossia:

$$\cdot : Ctx \times Ctx \rightarrow Ctx$$

dove:

$$(\Gamma_1\Gamma_2)(x) = \begin{cases} \Gamma_2(x) & \text{se } x \in dom(\Gamma_2) \\ \Gamma_1(x) & \text{altrimenti} \end{cases}$$

Definizione 61: Semantica dei tipi

Data la seguente relazione detta **semantica dei tipi**, ossia:

$$: \subseteq Ctx \times Terms \times Types$$

definiamo come **asserzione di tipo** la tripla $(\Gamma, M, v) \in :$ descritta dalla notazione

$$\Gamma \vdash M : A$$

la quale viene letta come "nel contesto Γ, M è un termine legale di tipo A".

All'interno del **lambda calcolo tipato**, i termini vengono valutati tramite le seguenti regole di inferenza:

- **Costanti**:
$$\Gamma \vdash k : K$$
- **Variabili**:
$$\Gamma(x) = A \implies \Gamma \vdash x : A$$
- **Funzione**:
$$\frac{\Gamma,\ x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \rightarrow B}$$
- **Applicazione**:
$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Esempio:

- Consideriamo il seguente termine

$$[\lambda x : (\texttt{Int} \to \texttt{Bool}).x\ \{[\lambda y : \texttt{String}.7]\ \texttt{"ciao"}\}][\lambda z : \texttt{Int}.\texttt{true}]$$

- La sua valutazione corrisponde a:

$$(*) \quad \dfrac{\Gamma \vdash x : \texttt{Int} \to \texttt{Bool} \qquad \dfrac{\dfrac{\Gamma,\ y : \texttt{String} \vdash 7 : \texttt{Int}}{\Gamma \vdash \lambda y : \texttt{String}.7 : \texttt{String} \to \texttt{Int}} \qquad \Gamma \vdash \texttt{"ciao"} : \texttt{String}}{\Gamma \vdash [\lambda y : \texttt{String}.7]\ \texttt{"ciao"} : \texttt{Int}}}{\dfrac{\Gamma \vdash x\ \{[\lambda y : \texttt{String}.7]\ \texttt{"ciao"}\} : \texttt{Bool}}{\varnothing \vdash [\lambda x : (\texttt{Int} \to \texttt{Bool}).x\ \{[\lambda y : \texttt{String}.7]\ \texttt{"ciao"}\}] : (\texttt{Int} \to \texttt{Bool}) \to \texttt{Bool}}}$$

$$\dfrac{(*) \qquad \dfrac{z : \texttt{Int} \vdash \texttt{true} : \texttt{Bool}}{\varnothing \vdash \lambda z : \texttt{Int}.\texttt{true} : \texttt{Int} \to \texttt{Bool}}}{\varnothing \vdash [\lambda x : (\texttt{Int} \to \texttt{Bool}).x\ \{[\lambda y : \texttt{String}.7]\ \texttt{"ciao"}\}][\lambda z : \texttt{Int}.\texttt{true}] : \texttt{Bool}}$$

dove $\Gamma = x : \texttt{Int} \to \texttt{Bool}$

Osservazione 27: Termini non tipabili

All'interno del lambda calcolo tipato, alcuni termini **non sono tipabili** e pertanto essi sono **illegali**

Esempio:

- Consideriamo il termine $\lambda x : A.xx$
- La sua valutazione corrisponde a:

$$\dfrac{\dfrac{x : A \vdash x : A \to A \qquad x : A \vdash x : A}{x : A \vdash xx : A}}{\varnothing \vdash \lambda x : A.xx : A}$$

- Di conseguenza, ci troviamo in una situazione in cui il tipo della variabile x venga valutato sia in A sia in $A \to A$
- Essendo ciò impossibile, ne segue che il termine $\lambda x : A.xx$ sia non tipabile e dunque che esso sia illegale