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

Linguaggi di Programmazione

Appunti integrati con il libro

Autore
Simone Bianco

3 ottobre 2025

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*.

1

Algebre e strutture dati induttive

1.1 Assiomi di Peano e Principio di induzione

Definizione 1.1: Assiomi di Peano

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

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

Principio 1.1: 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 ogni $m \in \mathbb{N}$.

In simboli, abbiamo che:

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

Proposizione 1.1: Induzione assiomatizzata

Il quinto assioma di Peano è equivalente al principio di induzione

Dimostrazione.

- Data una proprietà P, definiamo il seguente insieme:

$$S = \{n \in \mathbb{N} \mid P(n) \text{ è vera}\}$$

- Affermare che valga il caso base per P e che assumendo l'ipotesi induttiva si riesca a dimostrare il passo induttivo, equivale a dire che $0 \in S$ e che $n \in S \implies n+1 \in S$
- Di conseguenza, tramite il quinto assioma, ne segue che $S = A$, dunque che P valga per ogni $m \in \mathbb{N}$

□

Osservazione 1.1

Dato $k \in \mathbb{N}$, il principio di induzione può essere utilizzato per dimostrare che una proprietà P valga $\forall n \in \mathbb{N} \mid n \geq k$

Dimostrazione.

- Sia P la proprietà da voler dimostrare $\forall n \in \mathbb{N} \mid n \geq k$
- Definendo la proprietà Q come $Q(m-k) = P(n)$, dimostrare che P valga $\forall n \in \mathbb{N} \mid n \geq k$ equivale a dimostrare che Q valga $\forall n \in \mathbb{N}$, rispettando il quinto assioma di Peano

□

Proposizione 1.2: Numeri naturali di Von Neumann

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

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

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

assieme alla funzione $\text{succ}_{\mathcal{N}} : \mathcal{N} \to \mathcal{N} : n \mapsto n \cup \{n\}$ 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:

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

   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. Dato $S \subseteq \mathcal{N}$, supponiamo che $(0_\mathcal{N} \in S \land (n \in S \implies \mathrm{succ}_\mathcal{N}(n) \in S))$

   Considerato $n \in \mathcal{N}$, possiamo esprimere n come $\mathrm{succ}_\mathcal{N}(\mathrm{succ}_\mathcal{N}(\ldots \mathrm{succ}_\mathcal{N}(0_\mathcal{N}))) = \mathrm{succ}^k_\mathcal{N}(0_\mathcal{N})$ per qualche $k \in \mathbb{N}$. Procediamo quindi per induzione sul numero $k \in \mathbb{N}$ di applicazioni della funzione $\mathrm{succ}_\mathcal{N}$ per ottenere n

   Caso base $(k = 0)$.

   - Se $k = 0$, allora $n = 0_\mathcal{N}$. Poiché per ipotesi si ha che $0_\mathcal{N} \in S$, ne segue che $n \in S$

   Ipotesi induttiva.

   - Dato un $k \in \mathbb{N}$, preso $n = \mathrm{succ}^k_\mathcal{N}(0_\mathcal{N}) \in \mathcal{N}$ si ha che $n \in S$

   Passo induttivo.

   - Dato $n = \mathrm{succ}^{k+1}_\mathcal{N}(0_\mathcal{N}) \in \mathcal{N}$, si ha che:

     $$n = \mathrm{succ}^{k+1}_\mathcal{N}(0_\mathcal{N}) = \mathrm{succ}^{k+1}_\mathcal{N}(\mathrm{succ}^k_\mathcal{N}(0_\mathcal{N}))$$

   - Per ipotesi induttiva, sappiamo che $\mathrm{succ}^k_\mathcal{N}(0_\mathcal{N}) \in S$. Di conseguenza, per ipotesi si ha che:

     $$\mathrm{succ}^k_\mathcal{N}(0_\mathcal{N}) \in S \implies \mathrm{succ}^{k+1}_\mathcal{N}(\mathrm{succ}^k_\mathcal{N}(0_\mathcal{N})) = \mathrm{succ}^{k+1}_\mathcal{N}(0_\mathcal{N}) = n \in S$$

   Di conseguenza, concludiamo che $n \in \mathcal{N} \implies n \in S$ e dunque che $S = \mathrm{N}$

$\square$

1.2 Algebre induttive

Definizione 1.2: Insieme unità

Definiamo come **insieme unità** l'insieme

$$\mathbb{1} = \{()\}$$

ossia l'insieme composto da una zerupla

Definizione 1.3: Funzione nullaria

Dato un insieme A e una funzione f, definiamo una f come **funzione nullaria** (o *funzione costante*) se

$$f : \mathbb{1} \to A : () \mapsto a$$

dove $a \in A$.

Inoltre, per comodità, indichiamo $f(())$ direttamente con f

Esempio:

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

Osservazione 1.2

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

Definizione 1.4: 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 1.5: Algebra

Definiamo come **algebra** (o struttura algebrica) una n-upla $(A, \gamma_1, \ldots, \gamma_n)$ dove A è un insieme non vuoto, detto **dominio**, e $\gamma_1, \ldots, \gamma_n$ sono delle operazioni definite su A, ossia la cui segnatura contiene A

Esempi:

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

Definizione 1.6: Chiusura di un'operazione

Sia A un insieme e sia γ un'operazione definita su A come:

$$\gamma : A \times \ldots \times A \times K_1 \times \ldots K_m \to A$$

Dato un sottoinsieme $S \subseteq A$, diciamo che γ è **chiusa rispetto ad** S se:

$$a_1, \ldots, a_n \in S \implies \gamma(a_1, \ldots, a_n, k_1, \ldots, k_m) \in S$$

Esempi:

- La funzione succ è chiusa per $A \subseteq \mathbb{N}$ solo se $A = \mathbb{N}$
- Dato l'insieme dei booleani $B = \{0, 1\}$, la funzione $\mathrm{not} : B \to B : b \mapsto \overline{b}$ è chiusa su $B \subseteq B$ ma non su $\{0\} \subseteq B$ in quanto dato $0 \in \{0\}$ si ha che $\mathrm{not}(0) \notin \{0\}$

Osservazione 1.3: Chiusura di una funzione nullaria

Dato un insieme A e una funzione nullaria $\gamma : \mathbb{1} \to B : () \mapsto b$, tale funzione risulta chiusa rispetto ad A se e solo se $b \in A$, poiché γ non prende in input alcun parametro di A

Esempio:

- La funzione zero è chiusa su $\{0\} \subseteq \mathbb{N}$ poiché non richiede alcun parametro, ma non è chiusa su $\{1\} \subseteq \mathbb{N}$ in quanto zero $\notin \{1\}$

Definizione 1.7: Algebra induttiva

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
- Dato $S \subseteq A$, se $\gamma_1, \ldots, \gamma_n$ sono chiuse su S allora $S = A$

Nota: la terza condizione è equivalente a dire che:

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

Definizione 1.8: Costruttori di un'algebra induttiva

Data l'algebra induttiva $(A, \gamma_1, \ldots, \gamma_n)$, definiamo $\gamma_1, \ldots, \gamma_n$ come **costruttori** di A.

In particolare, dato γ_i dove $i \in [1, n]$, se $\gamma_i : \mathbb{1} \to A$, ossia è una funzione nullaria, definiamo γ_i come **costruttore base** di A

Esempio di algebra non induttiva:

- Dato l'insieme dei Booleani $B = \{0, 1\}$, consideriamo l'algebra (B, not), dove $\text{not} : B \to B : b \mapsto \overline{b}$
- Per definizione stessa, notiamo che not sia iniettiva e che, essendo l'unica operazione dell'algebra, la sua immagine sia disgiunta da quelle di tutte le altre operazioni
- Consideriamo quindi il sottoinsieme $\varnothing \subsetneq B$. Poiché $\nexists x \in \varnothing$, l'implicazione $x \in \varnothing \implies \text{not}(x) \in \varnothing$ risulta vera a vuoto in quanto l'ipotesi sia falsa.
- Tuttavia, l'implicazione:

$$(x \in \varnothing \implies \text{not}(x) \in \varnothing) \implies \varnothing = B$$

  risulta falsa poiché $\varnothing \neq B$
- Difatti, procedendo analogamente, potremmo dimostrare che $(\varnothing, \text{not})$ sia un'algebra induttiva, concludendo che (B, not) non lo sia in quanto $\varnothing \subsetneq B$

Teorema 1.1: Esistenza di un costruttore base

Se $(A, \gamma_1, \ldots, \gamma_n)$ è un algebra induttiva e $A \neq \varnothing$, allora $\exists i \in [1, n]$ per cui γ_i è un costruttore base di A

Dimostrazione.

- Supponiamo che $(A, \gamma_1, \ldots, \gamma_n)$ sia un'algebra induttiva
- Considerato il sottoinsieme $\varnothing \neq A$, per ogni $i \in [1, n]$ si ha che:
  - Se γ_i è un costruttore di base, si ha che $\gamma_i \notin \varnothing$, implicando che $(\varnothing, \gamma_1, \ldots, \gamma_n)$ non possa essere un'algebra induttiva
  - Se γ_i non è un costruttore di base, l'implicazione:

$$a_1, \ldots, a_k \in \varnothing \implies \gamma_i(a_1, \ldots, a_k) \in \varnothing$$

    risulta vera a vuoto, implicando che γ_i sia chiusa per $\varnothing$
- Supponiamo quindi per assurdo che $\exists i \in [1, n]$ per cui γ_i è un costruttore base di A
- In tal caso, dato $\varnothing \subsetneq A$, si avrebbe che $\gamma_1, \ldots, \gamma_n$ siano tutte chiuse in S ma che $\varnothing \neq A$, contraddicendo l'ipotesi per cui $(A, \gamma_1, \ldots, \gamma_n)$ sia induttiva
- Di conseguenza, ne segue necessariamente che $\exists i \in [1, n]$ per cui γ_i è un costruttore base di A

$\square$

Proposizione 1.3: Algebra induttiva dei naturali

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

Dimostrazione.

- zero risulta essere iniettiva poiché funzione nullaria, mentre succ risulta essere iniettiva grazie al secondo assioma di Peano
- $\text{im}(\text{zero}) \cap \text{im}(\text{succ}) = \{0\} \cap (\mathbb{N} - \{0\}) = \varnothing$
- Dato $S \subseteq \mathbb{N}$, supponiamo che zero e succ siano chiuse su S, implicando che $\text{zero} = 0 \in S$ e che $n \in \mathbb{N} \implies \text{succ}(n) \in S$. Per il quinto assioma di Peano, segue automaticamente che $S = \mathbb{N}$

□

Teorema 1.2: Algebre induttive finite

Dato un **insieme finito** A, tale insieme è sempre definibile come algebra induttiva

Dimostrazione.

- Dato $A = \{a_1, \dots, a_n\}$, per ogni $i \in [1, n]$, definiamo $\gamma_i : \mathbb{1} \to A : () \mapsto a_i$
- Considerata l'algebra $(A, \gamma_1, \dots, \gamma_n)$, per costruzione stessa abbiamo che:
  - $\gamma_1, \dots, \gamma_n$ sono iniettive poiché sono tutte funzioni nullarie
  - $\forall i \neq j \quad \text{im}(\gamma_i) \cap \text{im}(\gamma_j) = \{a_i\} \cap \{a_j\} = \varnothing$, dunque le immagini sono tutte disgiunte tra loro
  - Dato $S \subseteq A$, si ha che:
    * Se $S = A$, allora per ogni $i \in [1, n]$ si ha che $\gamma_i \in S = A$, soddisfacendo la terza condizione
    * Se $S \subsetneq A$ allora $\exists a_i \in A - S$ per cui si ha che $\gamma_i \notin S$, implicando che γ_i non sia chiusa in S e quindi che la terza condizione sia soddisfatta a vuoto
- Dunque, concludiamo che $(A, \gamma_1, \dots, \gamma_n)$ sia un'algebra induttiva

□

Esempio:

- Dato l'insieme dei Booleani $B = \{0, 1\}$, la tupla $(B, \text{true}, \text{false})$ dove

$$\text{true} : \mathbb{1} \to B : () \mapsto 1$$
$$\text{false} : \mathbb{1} \to B : () \mapsto 0$$

  è un'algebra induttiva

1.2.1 Lemma di Lambek

Definizione 1.9: 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:
$$\exp(x + y) = e^{x+y} = e^x e^y = \exp(x) \cdot \exp(y)$$

Definizione 1.10: 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.

Esempio:

- Date le due algebre $(\mathbb{R}, +)$ e $(\mathbb{R}_{>0}, \cdot)$, sappiamo giò che $\exp : \mathbb{R} \to \mathbb{R}_{>0} : x \mapsto e^x$ sia un omomorfismo
- Poiché $\exists \ln : \mathbb{R}_{>0} \to \mathbb{R} \mid \ln(\exp(x)) = x$, ne segue che exp sia invertibile, dunque che essa sia biettiva
- Di conseguenza, exp risulta essere un isomorfismo, concludendo che $\mathbb{R} \cong \mathbb{R}_{>0}$

Definizione 1.11: Segnatura di un'algebra

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

Inoltre, date due algebre $(A, \gamma_1, \ldots, \gamma_n)$ e $(B, \delta_1, \ldots, \delta_n)$, diciamo che tali algebre hanno **segnature equivalenti** se $\forall i \in [1, n]$ sostituendo A con B all'interno della segnatura di γ_i si ottiene la segnatura di δ_i

Esempio:

- Date le due algebre $(\mathbb{N}, \text{zero}, \text{succ})$ e $(\mathcal{N}, \text{zero}_\mathcal{N}, \text{succ}_\mathcal{N})$, si ha che:
  - 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 $\text{succ}_\mathcal{N} : \mathcal{N} \to \mathcal{N}$

  dunque le segnature di tali algebre sono equivalenti

- Date le due algebre $(\mathbb{N}, \text{zero}, \text{succ})$ e $(B, \text{true}, \text{not})$, dove $B = \{0, 1\}$ si ha che:
  - La segnatura di $\text{zero} : \mathbb{1} \to \mathbb{N}$ è equivalente alla segnatura di $\text{true} : \mathbb{1} \to B$
  - La segnatura di $\text{succ} : \mathbb{N} \to \mathbb{N}$ è equivalente alla segnatura di $\text{not} : B \to B$

  dunque le segnature di tali algebre sono equivalenti

Proposizione 1.4: 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*)

Esempio:

- Poiché le due algebre $(\mathbb{N}, \text{zero}, \text{succ})$ e $(B, \text{true}, \text{not})$, dove $B = \{0, 1\}$, hanno segnature equivalenti ne segue che $\exists!$ omomorfismo $f : \mathbb{N} \to B$ per cui si ha che:

$$f(\text{succ}(n)) = \text{not}(f(n))$$

Lemma 1.1: 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:

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

- 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

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

$\square$

Esempio:

- Date le due algebre induttive $(\mathbb{N}, \mathrm{zero}, \mathrm{succ})$ e $(\mathcal{N}, \mathrm{zero}_{\mathcal{N}}, \mathrm{succ}_{\mathcal{N}})$ 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

1.3 Strutture dati induttive

Definizione 1.12: Insieme delle liste finite

Definiamo $\texttt{List<T>}$ come l'insieme delle liste finite di elementi di T:

$$\texttt{List<T>} = \{\langle a_1, \dots, a_k \rangle \mid a_1, \dots, a_k \in T\}$$

Esempio:

- Dato $\texttt{List<Int>}$, si ha che $\langle 3, 5, 1 \rangle \in \texttt{List<Int>}$

Proposizione 1.5: Algebra induttiva delle liste finite

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

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

è un'algebra induttiva

Dimostrazione.

1. - Poiché empty è una funzione nullaria, ne segue che:

     $$\forall x, y \in \mathbb{1} \;\; f(x) = f(y) \implies \langle\, \rangle = \langle\, \rangle$$

     poiché $x = y = ()$ dato che $\mathbb{1} = \{()\}$

   - Date $(x, \ell), (y, \ell') \in \texttt{T} \times \texttt{List<T>}$, dove $\ell = \langle x_1, \dots, x_k \rangle$ e $\ell' = \langle y_1, \dots, y_k \rangle$, supponiamo che $(x, \ell) \neq (y, \ell')$.

     In tal caso, le uniche possibilità sono:

     - Se $x = y$ ma $(x, \ell) = (y, \ell')$, si ha che:

       $$\mathrm{cons}(x, \ell) = \langle x, x_1, \dots, x_k \rangle = \langle y, x_1, \dots, x_k \rangle \neq \langle y, y_1, \dots, y_k \rangle = \mathrm{cons}(y, \ell')$$

– Se $x \neq y$ ma $\ell = \ell'$, si ha che:

$$\text{cons}(x, \ell) = \langle x, x_1, \ldots, x_k \rangle = \langle x, y_1, \ldots, y_k \rangle \neq \langle y, y_1, \ldots, y_k \rangle = \text{cons}(y, \ell')$$

– Se $x \neq y$ e $\ell \neq \ell'$, si ha che:

$$\text{cons}(x, \ell) = \langle x, x_1, \ldots, x_k \rangle \neq \langle y, y_1, \ldots, y_k \rangle = \text{cons}(y, \ell')$$

Di conseguenza, concludiamo che:

$$(x, \ell) \neq (y, \ell') \implies \text{cons}(x, \ell) \neq \text{cons}(y, \ell')$$

e dunque che cons sia iniettiva

2. • Per definizione stessa di empty si ha che $\text{im}(\text{empty}) = \{\langle\ \rangle\}$. Inoltre, per definizione stessa di cons, si ha che $\nexists (x, \ell) \in \texttt{T} \times \texttt{List<T>} \mid \text{cons}(x, \ell) = \text{empty}$, implicando che $\text{empty} \notin \text{im}(\text{cons})$. Dunque, otteniamo che $\text{im}(\text{empty}) \cap \text{im}(\text{cons}) = \varnothing$

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

• Dimostriamo per induzione sulla lunghezza $n \in N$ delle liste che $\ell \in \texttt{List<T>} \implies \ell \in S$

Caso base $(n = 0)$

– Se $n = 0$, allora $\ell \in \texttt{List<T>} \implies \ell = \langle\ \rangle = \in S$

Ipotesi induttiva.

– Per ogni lista $\ell \in \texttt{List<T>}$ di lunghezza $n \in \mathbb{N}$, si ha che $\ell \in S$

Passo induttivo.

– Data una lista $\ell \in \texttt{List<T>}$ di lunghezza $n + 1$, si ha che:

$$\ell = \langle x_1, \ldots, x_n \rangle = \text{cons}(x_1, \langle x_2, \ldots, x_n \rangle)$$

– Di conseguenza, poiché $\langle x_2, \ldots, x_n \rangle$ ha lunghezza n, per ipotesi induttiva si ha che $\langle x_2, \ldots, x_n \rangle \in S$

– A questo punto, per chiusura su S di cons, si ha che:

$$(x_1, \langle x_2, \ldots, x_n \rangle) \in \texttt{T} \times \texttt{List<T>} \implies \ell = \text{cons}(x_1, \langle x_2, \ldots, x_n \rangle) \in S$$

• Dunque, concludiamo che $\texttt{List<T>} \subseteq S$ e dunque che $S = \texttt{List<T>}$

$\square$

Osservazione 1.4

Tramite i costruttori di un'algebra induttiva è possibile definire altre operazioni più complesse per l'algebra

Esempio:

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

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

definita come:

$$\text{concat}(\ell_1, \ell_2) = \begin{cases} \ell_2 & \text{se } \ell_1 = \text{empty} \\ \text{cons}(n, \text{concat}(\ell_3, \ell_2)) & \text{se } \ell_1 = \text{cons}(n, \ell_3) \end{cases}$$

- Ad esempio, in $\texttt{List<Int>}$, abbiamo che:

$$\text{concat}(\langle 1, 5\rangle, \langle 7, 2\rangle) = \text{concat}(\text{cons}(1, \langle 5\rangle), \langle 7, 2\rangle) = \text{cons}(1, \text{concat}(\langle 5\rangle, \langle 7, 2\rangle)) =$$

$$\text{cons}(1, \text{cons}(5, \text{concat}(\langle\ \rangle, \langle 7, 2\rangle))) = \text{cons}(1, \text{cons}(5, \langle 7, 2\rangle)) = \langle 1, 5, 7, 2\rangle$$

Definizione 1.13: Insieme delle liste infinite

Definiamo $\texttt{List<T>}^\infty$ come l'insieme delle liste infinite di elementi di T:

$$\texttt{List<T>}^\infty = \{\langle a_1, a_2, \ldots\rangle \mid a_1, a_2, \ldots \in T\}$$

Proposizione 1.6

L'insieme $\texttt{List<T>}^\infty$ non è **mai definibile** come algebra induttiva

Nota: viene assunto che un'algebra non possa avere infinite operazioni

Dimostrazione.

- Supponiamo per assurdo che $\exists \gamma_1, \ldots, \gamma_n$ tali che $(\texttt{List<T>}^\infty, \gamma_1, \ldots, \gamma_n)$ sia un'algebra induttiva. Poiché $\texttt{List<T>}^\infty \neq \varnothing$, per l'Esistenza di un costruttore base si ha che $\exists i \in [1, n]$ per cui γ_i sia un costruttore base di $\texttt{List<T>}^\infty$
- Sia quindi $\ell = \langle x_1, x_2, \ldots, \rangle \in \texttt{List<T>}^\infty$ tale che $\gamma_i = \ell$ e sia $\ell\text{-}\texttt{List<T>}^\infty$ l'insieme delle liste infinite che estendono ℓ:

$$\ell\text{-}\texttt{List<T>}^\infty = \{\langle y_1, \ldots, y_k, x_1, x_2, \ldots\rangle \mid y_1, \ldots, y_2 \in T\}$$

- Risulta evidente che $\ell\text{-}\texttt{List<T>}^\infty \subseteq \texttt{List<T>}^\infty$ e che $\gamma_1, \ldots, \gamma_n$ siano chiuse per $\ell\text{-}\texttt{List<T>}^\infty$, contraddicendo l'ipotesi per cui $\texttt{List<T>}^\infty$ sia un'algebra induttiva
- Di conseguenza, l'unica possibilità è che $\nexists \gamma_1, \ldots, \gamma_n$

$\square$

Definizione 1.14: Insieme degli alberi binari finiti

Definiamo $\texttt{BinTree}$ come l'insieme degli alberi binari finiti:

$$\texttt{BinTree} = \left\{ t \;\middle|\; \begin{array}{c} t = \circ \text{ ossia è una foglia oppure} \\ t = \langle t_1, t_2 \rangle \text{ dove } t_1, t_2 \in \texttt{BinTree} \end{array} \right\}$$

Proposizione 1.7: Algebra induttiva degli alberi binari finiti

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

- $\text{leaf} : \mathbb{1} \to \texttt{BinTree} : () \mapsto \circ$ 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 t_{sx} e t_{dx} siano i due sottoalberi di t

è un'algebra induttiva

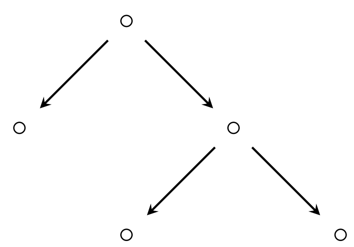
(*dimostrazione omessa*)

Esempio:

- Il seguente albero binario

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

può essere rappresentato graficamente come:



Principio 1.2: 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 1.3: 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 $n = \text{leaves}(\text{leaf}) = 1$ foglie. Difatti, si ha che $k = 1 = 2n - 1$

  Ipotesi induttiva.

  - Ogni argomento t di ogni costruttore possiede $2 \cdot \text{leaves}(t') - 1$ 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:

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

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

□

1.4 Sintassi astratta dei linguaggi

Definizione 1.15: Linguaggio

Definiamo come **linguaggio** un insieme di stringhe

Definizione 1.16: 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:

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

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

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

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

Definizione 1.17: 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:

$$\begin{aligned} \text{eval}(\texttt{"0"}) &= 0 \\ \text{eval}(\texttt{"1"}) &= 1 \\ &\dots \\ \text{eval}(\texttt{"M + N"}) &= \text{eval}(\texttt{"M"}) + \text{eval}(\text{"}N\text{"}) \\ \text{eval}(\texttt{"M * N"}) &= \text{eval}(\texttt{"M"}) + \text{eval}(\text{"}N\text{"}) \end{aligned}$$

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

$$\begin{aligned} 0 : \mathbb{1} \to \mathbb{N} : () \mapsto 0 \\ 1 : \mathbb{1} \to \mathbb{N} : () \mapsto 1 \\ \dots \\ \text{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m + n \\ \text{times} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (m, n) \mapsto m \cdot n \end{aligned}$$

- 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 1.4: 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 2.1: 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:

- $k \in \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \ldots\}$ 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.
- $Val = \{0, 1, \ldots\}$ 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 2.2: Scope di una variabile

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

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

Definizione 2.3: 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 2.1: 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 2.4: Insieme degli ambienti in *Exp*

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

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

Definizione 2.5: 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 2.2: Regola di inferenza

Date delle proposizioni $P_1, \ldots, P_n, C$, indichiamo la seguente proposizione:

$$P_1 \wedge \ldots \wedge P_n \wedge ((P_1 \wedge \ldots \wedge P_n) \implies C)$$

con la seguente notazione alternativa, detta **regola di inferenza**:

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

dove $P_1, \ldots, P_n$ vengono dette **premesse** e C viene detta conclusione

Definizione 2.6: 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 2.3: 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:

$$E \vdash x \leadsto v \quad (\text{se } E(x) = v)$$

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

$$\frac{E \vdash M \leadsto v \quad E \vdash N \leadsto v'}{E \vdash M + N \leadsto u} \quad (\text{se } u = v + v')$$

- 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 2.1: 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 2.2: 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:

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

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

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

Definizione 2.7: 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:

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

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

2.2 Valutazione Eager vs Lazy

Consideriamo la seguente espressione per il linguaggio *Exp*:

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

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** (trad: *pigra*), è costituita da regole operazionali atte al *ritardare* la valutazione dei termini fino a quando non sia strettamente necessario.

Definizione 2.8: Valutazione eager

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

Definizione 2.9: Valutazione lazy

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

Proposizione 2.4: 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:

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

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

$$\frac{E \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \quad (\text{se } E(x) = M)$$

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

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

Osservazione 2.3

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 2.10: Scoping statico

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

Definizione 2.11: 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 2.5: Linguaggio Exp lazy statico

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

- L'insieme Env viene ridefinito come:

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

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

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

- 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:

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

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 2.4: 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 2.12: 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 2.1: 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 2.5

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 2.13: Il linguaggio Fun

Definiamo come Fun il linguaggio rappresentato dalla seguente grammatica:

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

dove:

- $k \in \{0, 1, \ldots\}$ ossia è una **costante**
- $x \in Var = \{x, y, z, \ldots\}$ 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
- $Val = \{0, 1, \ldots\} \cup (Var \times Fun)$ 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 2.6

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

Esempio:

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

Definizione 2.14: 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 2.2: 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 2.7: 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 2.8

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

Proposizione 2.6: Linguaggio Fun eager dinamico

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

- L'insieme Env viene ridefinito come:

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

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

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

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

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

Proposizione 2.7: 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 2.1

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:

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

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

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 2.8: Linguaggio *Fun* lazy dinamico

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

- L'insieme Env viene ridefinito come:

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

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

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

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

$$\dfrac{E \vdash M \rightsquigarrow (x, L) \qquad E\{(x, N)\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$$

Proposizione 2.9: Linguaggio *Fun* lazy statico

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

- L'insieme Env viene ridefinito come:

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

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

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

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

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

Osservazione 2.9

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

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

Definizione 2.15: 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:

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

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

- 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 2.2

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.

□

Teorema 2.3: Equivalenze semantiche di *Fun*

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

Proposizione 2.10: 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 2.16: 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 2.10: Curryficazione in lambda calcolo

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

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

Definizione 2.17: Operatore di sostituzione

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

Esempi:

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

Osservazione 2.11: Cattura di variabili

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

Esempio:

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

Definizione 2.18: Alfa conversione

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

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

Esempi:

- Data la lambda astrazione $\lambda x.(xy)$, si ha che:
$$\lambda x.xy \xrightarrow{\alpha} \lambda z.zy$$

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

Definizione 2.19: 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 2.20: Beta conversione

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

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

Osservazione 2.12

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

Esempio:

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

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

Osservazione 2.13

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 2.21: Eta conversione

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

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

Esempi:

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

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

è possibile applicare l'eta conversione:

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

2.5.1 *Fun* vs Lambda calcolo

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

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

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

In particolare, le seguenti due espressioni:

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

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

Osservazione 2.14

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 2.11: 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 2.22: 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 2.23: 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 2.4: 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 2.15

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

Lemma 2.3: Ricorsione tramite numeri naturali

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

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

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

Dimostrazione.

- Sia $\mathrm{unit} : \mathbb{1} \to A$ la funzione nullaria che restituisce sempre a
- L'algebra (A, unit, h) possiede la stessa segnatura dell'algebra induttiva $(\mathbb{N}, \mathrm{zero}, \mathrm{succ})$, 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:

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

□

Esempio:

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

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

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

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

Definizione 2.24: 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 (ρ `True` `Not`) corrisponde alla funzione isEven : $\mathbb{N} \to A$ definita nell'esempio precedente
- Difatti, abbiamo che:

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

Teorema 2.5: 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 2.1: 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 2.25: 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 2.16

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 2.26: 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 3.1: 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 3.2: Insieme delle locazioni

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

Definizione 3.3: Insieme degli ambienti in Imp

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

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

Definizione 3.4: Insieme delle memorie in Imp

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

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

Definizione 3.5: 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 3.6: Semantiche operazionali di Imp

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

- La **semantica delle espressioni**

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

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

- La **semantica dei programmi**

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

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

Le regole operazionali di tali semantiche sono definite come:

- **Costanti**:

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

- **Variabili**:

$$E \vdash x,\ S \rightsquigarrow v \quad (\text{se } S(E(x)) = v)$$

- **Somma**:

$$\frac{E \vdash M,\ S \rightsquigarrow v \quad E \vdash N,\ S \rightsquigarrow v'}{E \vdash M + N,\ S \rightsquigarrow u} \quad (\text{se } u = v + v')$$

- **Minorazione**:

$$\frac{E \vdash M,\ S \rightsquigarrow v \quad E \vdash N,\ S \rightsquigarrow v'}{E \vdash M < N,\ S \rightsquigarrow true} \quad (\text{se } v < v')$$

$$\frac{E \vdash M,\ S \rightsquigarrow v \quad E \vdash N,\ S \rightsquigarrow v'}{E \vdash M < N,\ S \rightsquigarrow false} \quad (\text{se } v \geq v')$$

- **Skip**:

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

- **Esecuzione sequenziale**:

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

- **If-then-else**:

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

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

- **While**:

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

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

- **Dichiarazione e assegnamento**:

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

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

- **Assegnamento**:

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

Osservazione 3.1

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

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

3.2 *All*: un linguaggio con procedure

Definizione 3.7: Il linguaggio *All*

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

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

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 3.8: 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 3.1: Ridefinizione di *Env* in *All*

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

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

Definizione 3.9: 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**:

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

- **Locazione in array**:

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

- **Riferimento**:

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

- **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**:

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

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

- **Procedura**:

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

3.2.1 Semantiche di *All*

Definizione 3.10: Semantiche di *All*

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

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

Proposizione 3.2: Linguaggio *All* call-by-value

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

- **Richiamo by-value**:

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

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

Proposizione 3.3: Linguaggio *All* call-by-reference

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

- **Richiamo by-reference**:

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

Proposizione 3.4: 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**:

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

- **Richiamo by-name**:

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

Lemma 3.1

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

$\square$

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:

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

  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:

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

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

Lemma 3.2

Dato il linguaggio *All*, si ha che:

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

Dimostrazione.

- Consideriamo il programma

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

- 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 3.1: Equivalenze semantiche di *All*

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

4

Correttezza dei programmi

4.1 Correttezza dei programmi imperativi

4.1.1 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 $y = 138$:

$$\begin{array}{r|l} 45 & 138 \\ 90 & 69 \\ 180 & 34 \\ 360 & 17 \\ 720 & 8 \\ 1440 & 4 \\ 2880 & 2 \\ 5760 & 1 \end{array}$$

- 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 4.1: Invariante

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

Proposizione 4.1

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 4.1: Metodo delle invarianti

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

4.1.2 Logica di Hoare

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

Definizione 4.2: 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 4.3: Tripla di Hoare

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

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

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

Definizione 4.4: Formula

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

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

Proposizione 4.2: Regole di inferenza generali

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

- **True**:
$$\{A\}\ P\ \{true\}$$
- **False**:
$$\{false\}\ P\ \{A\}$$
- **Rafforzamento della precondizione (strengthening)**:
$$\frac{A \supset B \quad \{B\}\ P\ \{C\}}{\{A\}\ P\ \{C\}}$$
- **Indebolimento della postcondizione (weakening)**:
$$\frac{\{A\}\ P\ \{B\} \quad B \supset C}{\{A\}\ P\ \{C\}}$$
- **And**:
$$\frac{\{A\}\ P\ \{B_1\} \quad \dots \quad \{A\}\ P\ \{B_n\}}{\{A\}\ P\ \{B_1 \wedge \dots \wedge B_n\}}$$
- **Or**:
$$\frac{\{A_1\}\ P\ \{B\} \quad \dots \quad \{A_n\}\ P\ \{B\}}{\{A_1 \vee \dots \vee A_n\}\ P\ \{B\}}$$

Proposizione 4.3: 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**:
$$\frac{\{A \wedge B\}\ P\ \{C\} \quad \{A \wedge \neg B\}\ Q\ \{C\}}{\{A\}\ if\ B\ then\ P\ else\ Q\ \{C\}}$$
- **While**:
$$\frac{\{A \wedge B\}\ P\ \{A\}}{\{A\}\ while\ B\ do\ P\ \{A \wedge \neg B\}}$$
- **Assegnamento**:
$$\{A[M/x]\}\ x := M\ \{A\}$$
dove $[M/x]$ ricordiamo essere l'operatore di Operatore di sostituzione

Esempio:

- Consideriamo la seguente tripla di Hoare:
$$\{x = 1\}\ x := x + 1\ \{x = 2\}$$
- Possiamo utilizzare lo *strengthening* affinché l'assegnamento possa essere valutato correttamente in quanto $(x = 2)[x + 1/x]$ venga valutata in $x + 1 = 2$:

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

Teorema 4.1: Invarianti con logica di Hoare

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

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

A questo punto, consideriamo la seguente funzione:

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

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

    return {a, b};
}
```

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

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

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

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

Proposizione 4.4

Se $x \geq 0$, la proprietà:

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

è un'invariante della funzione `EuclideanDivision`

Dimostrazione.

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

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

- 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:

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

- 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:

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

  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:

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

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

- In conclusione, dunque, abbiamo che:

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

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 4.5: 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**:
$$\frac{M = M' \quad N = N'}{M\ N = M'\ N'}$$
8. **Regola xi** (ξ):
$$\frac{M = N}{fn\ x \Rightarrow M = fn\ x \Rightarrow N}$$

Osservazione 4.1

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 4.5: 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 4.6: 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 semplice

Definizione 5.1: Lambda calcolo tipato semplice

Definiamo come **lambda calcolo tipato semplice** 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 5.2: Insieme dei contesti

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

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

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

Definizione 5.3: Concatenazione di contesti

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

$$\cdot : Ctx \times Ctx \to Ctx$$

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 5.4: 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 semplice**, i termini vengono valutati tramite le seguenti regole di inferenza:

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

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:

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

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

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

Proposizione 5.1: Circolarità non tipabile

Dato il lambda calcolo tipato semplice, non esiste un tipo A tale che $A \equiv A \to B$ per qualche tipo B.

Per tanto, qualsiasi termine che dovrebbe assumere tale tipo, detto **circolare**, risulta **non tipabile**.

Dimostrazione.

- Supponiamo per assurdo che esista un tipo A tale che $A \equiv A \to B$ per qualche tipo B.
- Ciò risulta possibile solo se $A \equiv B \to B \to B \to \ldots$, implicando che A appartenga all'insieme dei tipi infiniti, ossia che $A \in \mathit{Types}_\infty$
- Essendo tuttavia il tipo A generato dalla grammatica $B, C ::= K \ | \ B \to C$ del lambda calcolo tipato, ne seguirebbe che tale grammatica generi l'algebra $(\mathit{Types}_\infty, K, \to)$ e che essa sia induttiva, creando una contraddizione in quanto anche $(\mathit{Types}, K, \to)$ sia un'algebra induttiva e $\mathit{Types} \subsetneq \mathit{Types}_\infty$
- Di conseguenza, l'unica possibilità è che tale tipo A non possa esistere

$\square$

Esempio:

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

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

- Di conseguenza, il tipo del termine $\lambda x : A.xx$ corrisponde sia a A che a $A \to B$, implicando che $A \equiv A \to B$ e dunque che tale termine sia non tipabile

Osservazione 5.1: Precedenza delle parentesi nei tipi

All'interno del lambda calcolo tipato semplice si ha che:

$$A \to B \to C \to D \to \ldots \equiv A \to (B \to (C \to (D \to \ldots)))$$

5.2 Lambda calcolo polimorfo

Il lambda calcolo tipato semplice viene generalmente considerato un *sistema dei tipi di primo ordine*. Difatti, la sua formalizzazione non risulta essere sufficientemente astratta per poter permettere di tipare termini **generici**, ossia termini in grado di lavorare indipendentemente dal tipo assunto dalle variabili al suo interno.

Consideriamo ad esempio il seguente termine:

$$\lambda x.((x\ 5)((x\ \texttt{true})\ \texttt{false}))$$

Utilizzando le regole di inferenza dettate dal lambda calcolo tipato semplice, notiamo facilmente che tale termine sia illegale in quanto il parametro x dovrebbe rispettare tutte le seguenti condizioni:

- Deve essere una funzione avente un intero come input
- Deve essere una funzione avente un booleano come input
- Deve essere una funzione che restituisce una funzione $f : \texttt{Bool} \to \texttt{Bool}$

All'interno dei *sistemi dei tipi del secondo ordine*, invece, le variabili possono assumere **tipi polimorfi**, ossia quantificati universalmente. Ad esempio, supponiamo all'interno del nostro sistema dei tipi valga il seguente tipo $\forall X.(X \to (\texttt{Bool} \to \texttt{Bool}))$. Risulta evidente che tale tipo sia perfettamente valido per il termine proposto precedentemente, rendendolo un termine legale all'interno del nostro sistema dei tipi.

A differenza dei sistemi del primo ordine, dunque, i sistemi del secondo ordine introducono **variabili per i tipi** oltre alle normali variabili per i termini.

Definizione 5.5: Lambda calcolo polimorfo

Definiamo come **lambda calcolo polimorfo (o *System F*)** il sistema dei tipi che estende il lambda calcolo tipato semplice, rappresentato dalla seguente grammatica:

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

dove:

- $X \in \{X, Y, Z, \ldots\}$ ossia è una **variabile di tipo** (o *tipo generico*)
- $x \in \{x, y, z, \ldots\}$ ossia è una **variabile di termine**
- Nel tipo $\forall X.A$ e nel termine $\Lambda X.M$, gli **operatori di generalizzazione** $\forall X$ (*per ogni* X) e ΛX (*per un'arbitraria* X) legano le occorrenze libere di X rispettivamente nel tipo A e nel termine M

Oltre alle regole di inferenza già previste dal lambda calcolo tipato semplice, nel System F vengono aggiunte le seguenti due regole:

- **Generalizzazione dei tipi**:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda X.M : \forall X.A} \quad (\text{se } X \notin \text{free}(\Gamma))$$

- **Specializzazione del tipo**:

$$\frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash M\ B : A[B/X]}$$

Esempio:

- Valutiamo il tipo dell'*identità polimorfa*, ossia $\Lambda X.\lambda x : X.x$:

$$\cfrac{\cfrac{x : X \vdash x : X}{\varnothing \vdash \lambda x : X.x : X \to X}}{\varnothing \vdash \Lambda X.\lambda x : X.x : \forall X.(X \to X)}$$

- Tale funzione non è tuttavia ancora applicabile, poiché il suo tipo non è un *tipo funzione* (ossia nella forma $A \to B$), ma bensì un *tipo polimorfo* (ossia nella forma $\forall X.A$). Difatti, è necessario prima specializzarne il suo tipo affinché essa possa essere applicata correttamente
- Specializzarne il tipo `Int` su tale funzione, otteniamo che:

$$\cfrac{\cfrac{\cfrac{x : X \vdash x : X}{\varnothing \vdash \lambda x : X.x : X \to X}}{\varnothing \vdash \Lambda X.\lambda x : X.x : \forall X.(X \to X)}}{\varnothing \vdash (\Lambda X.\lambda x : X.x)\ \texttt{Int} : \texttt{Int} \to \texttt{Int}}$$

  ottenendo così un termine operativamente equivalente all'identità sugli interi, ossia $\lambda x : \texttt{Int}.x$

Proposizione 5.2: Variabili libere in System F

Dato il System F, la funzione:

$$\text{free} : Types \cup Ctx \to \mathcal{P}(Var)$$

restituisce l'insieme di tutte le **variabili di tipo libere**, dove:

$$\begin{cases} \text{free}(K) = \varnothing \\ \text{free}(X) = \{X\} \\ \text{free}(A \to B) = \text{free}(A) \cup \text{free}(B) \\ \text{free}(\forall X.A) = \text{free}(A) - \{X\} \\ \text{free}(\Gamma) = \bigcup\limits_{x:A \in \Gamma} \text{free}(A) \end{cases}$$

Esempio:

- Dato il contesto $\Gamma = x : X$, si ha che $\text{free}(\Gamma) = \{X\}$
- Dato il contesto $\Gamma = x : \forall X.X$, si ha che $\text{free}(\Gamma) = \varnothing$
- Dato il contesto $\Gamma = x : X \to (\forall Y.Y)$, si ha che $\text{free}(\Gamma) = \{X\}$

Osservazione 5.2: Cattura di variabili di tipo

La condizione laterale $X \notin \text{free}(\Gamma)$ nella regola della generalizzazione dei tipi risulta <u>fondamentale</u> al fine di impedire la creazione di termini di tipo insensato

Esempio:

- Sia $M \equiv \Lambda X.\lambda x : X.(\Lambda X.x)\ \texttt{Int}$. Ignorando la condizione laterale della regola della generalizzazione, valutiamo il tipo di M:

$$\dfrac{\dfrac{\dfrac{\dfrac{\color{red}{x : X} \vdash x : X}{\color{red}{x : X} \vdash \Lambda X.x : \forall X.X}}{x : X \vdash (\Lambda X.x)\ \texttt{Int} : \texttt{Int}}}{\varnothing \vdash \lambda x : X.(\Lambda X.x)\ \texttt{Int} : X \to \texttt{Int}}}{\varnothing \vdash \Lambda X.\lambda x : X.(\Lambda X.x)\ \texttt{Int} : \forall X.(X \to \texttt{Int})}$$

  (*in rosso viene evidenziato dove si verifica che* $X \in \text{free}(\Gamma)$)

- A questo punto, consideriamo la valutazione del termine $M\ (\texttt{Int} \to \texttt{Int})(M\ \texttt{Int})$:

$$\dfrac{\dfrac{\varnothing \vdash M : \forall X.(X \to \texttt{Int})}{\varnothing \vdash M\ (\texttt{Int} \to \texttt{Int}) : (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}} \quad \dfrac{\varnothing \vdash M : \forall X.(X \to \texttt{Int})}{\varnothing \vdash M\ \texttt{Int} : \texttt{Int} \to \texttt{Int}}}{\varnothing \vdash M\ (\texttt{Int} \to \texttt{Int})(M\ \texttt{Int}) : \texttt{Int}}$$

- In tal modo, quindi, il termine $M\ (\texttt{Int} \to \texttt{Int})(M\ \texttt{Int})$ corrisponderebbe ad un intero, nonostante al suo interno non sia presente alcun numero intero

5.3 Sistema dei tipi di Hindley-Milner

Il **sistema dei tipi di Hindley-Milner** (o *sistema HM*), in termini di potenza polimorfismo, è un sistema situato nel mezzo tra il lambda calcolo tipato semplice e il lambda calcolo polimorfo. Tale sistema viene utilizzato anche dal linguaggio **Standard ML**.

A differenza del System F, il sistema HM **restringe** i tipi possibili all'interno del linguaggio, risultando quindi più debole. Ad esempio, il tipo $(\forall X.X) \to (\forall Y.Y)$ risulta tipabile nel System F, ma non nel sistema HM.

Tuttavia, tale restrizione permette al sistema dei tipi di **eliminare la regola di specializzazione** in quanto sarà il sistema stesso a specializzare i tipi. In tal modo, l'uso di tale sistema dei tipi risulta più **semplice** e privo di necessità da parte dell'utilizzatore di conoscere a priori i tipi specializzati.

Consideriamo ad esempio la **funzione identità in SML**, ossia `fn x => x`. L'interprete di SML valuta il tipo di tale funzione come `'a -> 'a`, dove `'a` indica un **tipo generico**. In altre parole, il tipo di tale funzione viene valutato in $\forall A.A \to A$. Passando il termine `5` a tale funzione, ossia considerando il termine `(fn x => x) 5`, l'interprete di SML è in grado di specializzare automaticamente il tipo della funzione, per poi applicarla al termine `5`. Nel System F, invece, tale risultato sarebbe ottenibile solo tramite il termine $((\Lambda X.\lambda x : X.x)\ \texttt{Int})\ 5$.

In particolare, il sistema HM risulta definito sul linguaggio *Fun* visto nei capitoli precedenti.

Definizione 5.6: Sistema dei tipi di Hidley-Milner

Definiamo come **sistema dei tipi di Hidley-Milner** (o sistema HM) il sistema dei tipi del linguaggio *Fun*, rappresentato dalla seguente grammatica:

$$\begin{array}{rcl} \tau & ::= & K \mid X \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall X.\sigma \\ M, N & ::= & k \mid x \mid fn\ x \Rightarrow M \mid let\ x = M\ in\ N \end{array}$$

dove:

- La prima grammatica rappresenta l'insieme dei **tipi primitivi**
- La prima grammatica rappresenta l'insieme degli **schemi di tipo**

Osservazione 5.3

Dato il sistema sistema HM, definiamo la seguente contrazione sintattica:

$$\forall X_1, X_2, \ldots, X_n.\sigma \equiv \forall X_1.\forall X_2.\ldots.\forall X_n.\sigma$$

Definizione 5.7: Istanza generica

Siano $\sigma = \forall X_1, \dots X_n.\tau$ e $\sigma' = \forall Y_1, \dots Y_m.\tau'$ due schemi di tipo. Definiamo σ' come **istanza generica di** σ, indicato con $\sigma' \sqsubset \sigma$ se:

- $\tau' = \tau[\tau_1, \tau_2, \dots, \tau_n / X_1, X_2, \dots, X_n]$
- $\forall j \in [0, m] \;\; Y_j \notin \text{free}(\sigma)$

Nota: le sostituzioni possono essere effettuate solo sulle variabili quantificate (ossia $X_1, \dots, X_n$). In altre parole, è necessario che $\forall i \in [0, n] \;\; X_i \notin \text{free}(\sigma)$

Esempi:

1. Dati gli schemi $\sigma = X$ e $\sigma' = \forall Y.X$, poiché $Y \notin \text{free}(\sigma)$ si ha che $\sigma' \sqsubset \sigma$
2. Dati gli schemi $\sigma = \forall X, Y.X \to Y$ e $\sigma' = \texttt{Int} \to \texttt{Bool}$, si ha che:

   $$\texttt{Int} \to \texttt{Bool} = (X \to Y)[\texttt{Int}, \texttt{Bool} / X, Y]$$

   dunque $\sigma' \sqsubset \sigma$
3. Dati gli schemi $\sigma = \forall X, Y.X \to Y$ e $\sigma' = \forall Y.\texttt{Int} \to Y$, si ha che:

   $$\texttt{Int} \to Y = (X \to Y)[\texttt{Int}, Y / X, Y]$$

   e inoltre $Y \notin \text{free}(\sigma)$, dunque $\sigma' \sqsubset \sigma$
4. Dati gli schemi $\sigma = X \to Y$ e $\sigma' = \texttt{Int} \to Y$, si ha che:

   $$\texttt{Int} \to Y = (X \to Y)[\texttt{Int} / X]$$

   ma la sostituzione è inapplicabile in quanto $X \in free(\sigma)$, dunque $\sigma' \not\sqsubset \sigma$
5. Dati gli schemi $\sigma = X$ e $\sigma' = \forall X.X$, si ha che:

   $$X = X[X/X]$$

   ma $X \in free(\sigma)$ implica sia che la sostituzione sia inapplicabile sia che la generalizzazione tramite $\forall X$ sia inapplicabile, dunque $\sigma' \not\sqsubset \sigma$

A questo punto, definiamo le regole di inferenza del sistema HM:

- **Costanti**:

  $$\Gamma \vdash k : K$$

- **Variabili**:

  $$\Gamma \vdash x : \sigma \quad (\text{se } \Gamma(x) = \sigma)$$

- **Generalizzazione dei tipi**:

  $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall X.\sigma} \quad (\text{se } X \notin \text{free}(\Gamma))$$

- **Specializzazione dei tipi**:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma'} \quad (\text{se } \sigma' \sqsubset \sigma)$$

- **Let-in**:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash N : \sigma'}{\Gamma \vdash let\ x = M\ in\ N : \sigma'}$$

- **Funzione**:

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash fn\ x \Rightarrow M : \tau \to \tau'}$$

- **Applicazione**:

$$\frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \tau'}$$

Osservazione 5.4: Termini polimorfi

Tramite la regola della specializzazione dei tipi del sistema HM, **un termine può avere più tipi**

Esempi:

1. - Consideriamo il seguente termine:

$$let\ x = (fn\ y \Rightarrow y)\ in\ x(fn\ z \Rightarrow z)(x\ 5)$$

- La valutazione del suo tipo corrisponde a:

$$(*) \qquad \frac{\dfrac{\Gamma \vdash x : \forall Y.Y \to Y}{\Gamma \vdash x : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})} \quad \dfrac{\dfrac{\dfrac{\Gamma, z : Z \vdash z : Z}{\Gamma \vdash fn\ z \Rightarrow z : Z \to Z}}{\Gamma \vdash fn\ z \Rightarrow z : \forall Z.Z \to Z}}{\Gamma \vdash fn\ z \Rightarrow z : \texttt{Int} \to \texttt{Int}}}{\Gamma \vdash x(fn\ z \Rightarrow z) : \texttt{Int} \to \texttt{Int}}$$

$$\frac{\dfrac{\dfrac{y : Y \vdash y : Y}{\varnothing \vdash fn\ y \Rightarrow y : Y \to Y}}{\varnothing \vdash fn\ y \Rightarrow y : \forall Y.Y \to Y} \quad \dfrac{(*) \quad \dfrac{\dfrac{\Gamma \vdash x : \forall Y.Y \to Y}{\Gamma \vdash x : \texttt{Int} \to \texttt{Int}} \quad \Gamma \vdash 5 : \texttt{Int}}{\Gamma \vdash x\ 5 : \texttt{Int}}}{\Gamma \vdash x(fn\ z \Rightarrow z)(x\ 5) : \texttt{Int}}}{\varnothing \vdash let\ x = (fn\ y \Rightarrow y)\ in\ x(fn\ z \Rightarrow z)(x\ 5) : \texttt{Int}}$$

dove $\Gamma = x : \forall Y.Y \to Y$

2.
   - Consideriamo il seguente termine:

$$fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x)$$

   - La valutazione del suo tipo corrisponde a:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x : X \to Y \qquad \dfrac{\Gamma \vdash y : (X \to Y) \to X \quad \Gamma \vdash x : X \to Y}{\Gamma \vdash y\ x : X}}{\Gamma \vdash x\ (y\ x) : Y}}{x : X \to Y \vdash fn\ y \Rightarrow x\ (y\ x) : ((X \to Y) \to X) \to Y}}{\varnothing \vdash fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x) : (X \to Y) \to ((X \to Y) \to X) \to Y}}{\varnothing \vdash fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x) : (X \to Y) \to ((X \to Y) \to X) \to Y}}{\varnothing \vdash fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x) : \forall X, Y.(X \to Y) \to ((X \to Y) \to X) \to Y}$$

   dove $\Gamma = x : X \to Y, y : (X \to Y) \to X$

3.
   - Consideriamo il seguente termine:

$$fn\ x \Rightarrow fn\ y \Rightarrow y\ x\ y$$

   - La valutazione del suo tipo corrisponde a:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : X, y : Y \vdash y : X \to (Y \to Z) \quad x : X, y : Y \vdash x : X}{x : X, y : Y \vdash y\ x : Y \to Z} \qquad x : X, y : Y \vdash y : Y}{x : X, y : Y \vdash y\ x\ y : Z}}{x : X \vdash fn\ y \Rightarrow y\ x\ y : Y \to Z}}{\varnothing \vdash fn\ x \Rightarrow fn\ y \Rightarrow y\ x\ y : X \to Y \to Z}}{\varnothing \vdash fn\ x \Rightarrow fn\ y \Rightarrow y\ x\ y : \forall X, Y, Z.X \to Y \to Z}$$

   - Tuttavia, tale valutazione risulta possibile solo se $Y \equiv X \to (Y \to Z)$, dando vita ad una circolarità
   - Di conseguenza, il termine $fn\ x \Rightarrow fn\ y \Rightarrow y\ x\ y$ non è tipabile

5.3.1 Algoritmo $\mathcal{W}$

Definizione 5.8: Sostituzione

Definiamo come **sostituzione** una funzione parziale V che mappa variabili di tipo a dei tipi primitivi

$$V : TypeVar \xrightarrow{fin} PrimTypes$$

Proposizione 5.3

Per induzione strutturale su $Types$, data una sostituzione V si ha che:

$$V(X \to Y) = V(X) \to V(Y)$$

(*dimostrazione omessa*)

Definizione 5.9: Unificatore

Definiamo V **unificatore di due tipi primitivi** τ_1 **e** τ_2 se V è una sostituzione tale che $V(\tau_1) = V(\tau_2)$

Esempi:

1. - Consideriamo i seguenti due tipi:

   $$\tau_1 : A \to B \to C$$

   $$\tau_2 : (C \to E) \to D$$

   - Consideriamo quindi una sostituzione V definita come:

   $$V : \begin{array}{l} A \mapsto C \to E \\ D \mapsto B \to C \end{array}$$

   - Notiamo che::

   $$V(\tau_1) = (C \to E) \to B \to C = V(\tau_2)$$

   dunque V unifica τ_1 e τ_2

2. - Consideriamo i seguenti due tipi:

   $$\tau_1 : A \to B \to D$$

   $$\tau_2 : (B \to C) \to A$$

   - Consideriamo quindi una sostituzione V definita come:

   $$V : \begin{array}{l} A \mapsto B \to D \\ C \mapsto D \end{array}$$

- Notiamo che::
$$V(\tau_1) = (B \to D) \to B \to D = V(\tau_2)$$
dunque V unifica τ_1 e τ_2

3. - Consideriamo i seguenti due tipi:
$$\tau_1 : A \to (B \to D) \to E \to E$$
$$\tau_2 : (B \to C) \to A \to B$$
   - Consideriamo quindi una sostituzione V definita come:
$$V : \begin{array}{l} A \mapsto B \to D \\ C \mapsto D \\ E \mapsto E \to E \end{array}$$
   - Notiamo che::
$$V(\tau_1) = ((E \to E) \to C) \to ((E \to E) \to C) \to E \to E = V(\tau_2)$$
dunque V unifica τ_1 e τ_2

Osservazione 5.5: Tipi non unificabili

Esistono alcuni tipi non unificabili tra loro

Esempi:

1. - Consideriamo i seguenti due tipi:
$$\tau_1 : A \to A$$
$$\tau_2 : \texttt{Int} \to \texttt{Bool}$$
   - Affinché possa esistere un unificatore di τ_1 e τ_2, sarebbe necessario che $A \mapsto \texttt{Int}$ e $A \mapsto \texttt{Bool}$, implicando che $\texttt{Int} \equiv \texttt{Bool}$, il che è impossibile
2. - Consideriamo i seguenti due tipi:
$$\tau_1 : A \to A \to A$$
$$\tau_2 : B \to B$$
   - Affinché possa esistere un unificatore di τ_1 e τ_2, sarebbe necessario che $B \mapsto A$ e $B \mapsto A \to A$, implicando che $A \equiv A \to A$, il che è impossibile

Definizione 5.10: Composizione tra sostituzioni

Date due sostituzioni V e V', si ha che:

$$V \circ V'(\tau) = V(V'(\tau))$$

Teorema 5.1: Teorema di unificazione di Robinson

Esiste un algoritmo $\mathcal{U}$ che dati due tipi primitivi τ_1 e τ_2 **restituisce una sostituzione** V o **fallisce**.

Inoltre, si ha che:

- Se $\mathcal{U}(\tau_1, \tau_2) = V$, ossia non fallisce, allora:
    - $V(X) \neq X \implies X$ compare in τ_1 o in τ_2
    - V **unifica** τ_1 e τ_2
- Se esiste un unificatore W di τ_1 e τ_2, allora:
    - $\mathcal{U}(\tau_1, \tau_2) = V$, ossia non $\mathcal{U}$ **non può fallire**
    - $W = Z \circ V$ per qualche sostituzione Z, ossia V è la **sostituzione più "generica" possibile**

(*dimostrazione omessa*)

Esempio:

- Consideriamo i seguenti due tipi:

$$\tau_1 : X \to Y$$

$$\tau_2 : Z \to Y$$

- Consideriamo quindi le seguenti due sostituzioni:

$$V : \; X \mapsto Z \qquad\qquad W : \begin{array}{l} X \mapsto Z \\ Y \mapsto A \to B \end{array}$$

- Entrambe le due sostituzioni risultano essere due unificatori:

$$V(\tau_1) = Z \to Y = V(\tau_2)$$

$$W(\tau_1) = Z \to (A \to B) = W(\tau_2)$$

- Tuttavia, data la seguente sostituzione:

$$U : \; Y \mapsto A \to B$$

abbiamo che $W = U \circ V$, implicando quindi che V sia "più generica" di W

Definizione 5.11: Generalizzazione massima

Data una variabile x e un contesto Γ, definiamo la **generalizzazione massima di** x, indicata con $\overline{\Gamma}(x)$, come:

$$\overline{\Gamma}(x) = \forall X_1, \ldots, X_n.\tau$$

dove $\Gamma(x) = \tau$ e $TypeVar - \mathrm{free}(\Gamma) = \{X_1, \ldots, X_n\}$

Teorema 5.2: Algoritmo $\mathcal{W}$

Esiste un algoritmo $\mathcal{W}$ che dato un contesto Γ e un termine M **restituisce una tupla** (V, τ) **o fallisce**, dove V è una sostituzione e τ è un tipo.

Inoltre, se $\mathcal{W}(\Gamma, M) = (V, \tau)$, ossia non fallisce, allora:

- Se $M \equiv x$ e $\Gamma(x) = \forall X_1, \ldots, X_n.\tau'$, allora:
  - $V = \mathrm{id}$, ossia è l'identità
  - $\tau = \tau'[Y_1, \ldots, Y_n/X_1, \ldots, X_n]$ dove $Y_1, \ldots, Y_n \in \mathrm{free}(\Gamma(x))$
- Se $M \equiv M_1\ M_2$ e inoltre:
  - $\mathcal{W}(\Gamma, M_1) = (V_1, \tau_1)$
  - $\mathcal{W}(V_1(\Gamma), M_2) = (V_2, \tau_2)$
  - $\mathcal{U}(V_2(\tau_1), \tau_2 \to Y) = W$, dove Y è una nuova variabile

  allora $V = W \circ V_2 \circ V_1$ e $\tau = W(Y)$
- Se $M \equiv fn\ x \Rightarrow N$ e inoltre:
  - $\mathcal{W}((\Gamma, x : X), N) = (V_1, \tau_1)$ dove X è una nuova variabile

  allora $V = V_1$ e $\tau = V_1(X) \to \tau_1$
- Se $M \equiv let\ x = N\ in\ L$ e inoltre:
  - $\mathcal{W}((V_1(\Gamma), x : \tau'), L) = (V_2, \tau_2)$ dove $\overline{V_1(\Gamma)}(x) = \tau'$

  allora $V = V_2 \circ V_1$ e $\tau = \tau_2$

Nota: L'algoritmo $\mathcal{W}$ fallisce solo se $\mathcal{U}$ fallisce

Definizione 5.12: Schema principale

Dato un contesto Γ e un termine M, definiamo uno schema di tipi σ come **principale per** Γ **e** M se:

- $\Gamma \vdash M : \sigma$
- $\forall \sigma'\ \ \Gamma \vdash M : \sigma' \implies \sigma' \sqsubset \sigma$

Teorema 5.3: Correttezza e completezza dell'algoritmo $\mathcal{W}$

Se $\mathcal{W}(\Gamma, M) = (V, \tau)$, allora:

- $V(\Gamma) \vdash M : \tau$
- $\overline{V(\Gamma)}(\tau)$ è principale per $V(\Gamma)$ e M

Esempio:

- Consideriamo il termine $fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x)$
- Applicando l'algoritmo $\mathcal{W}$ abbiamo che:
  - $\mathcal{W}(\varnothing, fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x)) = (V_1, \tau_1)$, dove:
    * $V_1 = V_2$
    * $\tau_1 = V_2(A) \to \tau_2$
  - $\mathcal{W}(x : A, fn\ y \Rightarrow x\ (y\ x)) = (V_2, \tau_2)$, dove:
    * $V_2 = V_3$
    * $\tau_2 = V_3(B) \to \tau_3$
  - $\mathcal{W}((x : A, y : B), x\ (y\ x)) = (V_3, \tau_3)$, dove:
    * $\mathcal{W}((x : A, y : B), x) = (\mathrm{id}, A)$
    * $\mathcal{W}(\mathrm{id}(x : A, y : B), y\ x) = \mathcal{W}((x : A, y : B), y\ x) = (V_4, \tau_4)$, dove:
      · $\mathcal{W}((x : A, y : B), y) = (\mathrm{id}, B)$
      · $\mathcal{W}(\mathrm{id}(x : A, y : B), x) = \mathcal{W}((x : A, y : B), x) = (\mathrm{id}, A)$
      · $\mathcal{U}(\mathrm{id}(B), A \to C) = \mathcal{U}(B, A \to C) = W$, con $W : B \mapsto A \to C$
      · $V_4 = W \circ \mathrm{id} \circ \mathrm{id} = W$
      · $\tau_4 = W(C) = C$
    * $\mathcal{U}(V_4(A), \tau_4 \to D) = \mathcal{U}(A, C \to D) = U$, con $U : A \mapsto C \to D$
    * $V_3 = U \circ V_4 \circ \mathrm{id} = U \circ W$, con $U \circ W : A \mapsto C \to D, B \mapsto (C \to D) \to C$
    * $\tau_3 = U(D) = D$
- Di conseguenza, abbiamo che:

$$\mathcal{W}(\varnothing, fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x)) = (V_1, \tau_1) =$$

$$(V_2, V_2(A) \to \tau_2) = (V_3, V_3(A) \to V_3(B) \to \tau_3) =$$
$$(U \circ W, (C \to D) \to ((C \to D) \to C) \to D)$$

- Dunque, concludiamo che un tipo principale di $fn\ x \Rightarrow fn\ y \Rightarrow x\ (y\ x)$ sia:

$$\forall C, D.(C \to D) \to ((C \to D) \to C) \to D$$

5.4 Isomorfismo di Curry-Howard

Consideriamo la regola di inferenza del **tipo di un'applicazione** descritta nelle sezioni precedenti:

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Consideriamo ora una regola fondamentale all'interno della logica del primo ordine, ossia la regola dell'**eliminazione dell'implicazione** (o *modus ponens*), descritta come:

$$\frac{A \implies B \quad A}{B}$$

ossia che *"se è vero che A implica B e vale A, allora è vale anche B"*.

In forma più generica, dato un **insieme di proposizioni vere** Γ e il connettivo logico $\vdash$, letto come *"deduce"*, possiamo affermare che:

$$\frac{\Gamma \vdash A \implies B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

ossia che *"se da Γ si deduce che A implica B e A siano veri, allora si deduce anche che B sia vero"*.

Notiamo quindi che vi sia una **corrispondenza diretta** tra il *modus ponens* e la valutazione del tipo di un'applicazione.

Analogamente, possiamo individuare una corrispondenza diretta tra un'altra regola logica, ossia la regola dell'**introduzione dell'implicazione**, e la valutazione del **tipo di una funzione**:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \implies B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash fn\ x \Rightarrow M : A \to B}$$

Consideriamo ora il seguente termine e il suo tipo principale:

$$fn\ x \Rightarrow fn\ y \Rightarrow x : \forall X, Y.X \to Y \to X$$

In senso improprio, possiamo dire che tale tipo afferma che textit"indipendentemente dal tipo Y, verrà restituito un elemento di tipo X".

Notiamo che tale tipo corrisponda esattamente al **primo assioma logico**, ossia:

$$A \implies (B \implies A)$$

il quale descrive che *"Se A è vero allora ne segue che A sia vero indipendentemente dalla veridicità di B*

Tali corrispondenze descritte precedentemente risultano essere frutto di una **corrispondenza diretta tra tipi e proposizioni** dettata dall'**isomorfismo di Curry-Howard**.

Teorema 5.4: Isomorfismo di Curry-Howard

L'**isomorfismo di Curry-Howard** è una corrispondenza tra la **logica** e il **sistema dei tipi**. In particolare, tale corrispondenza si divide in:

- Una **corrispondenza tra formule logiche e tipi**:

Logica	**Sistema dei tipi**
$A \implies B$	$A \to B$
$A \land B$	$A \times B$
$A \lor B$	$A + B$
$True$	$\mathbb{1}$
$False$	$\varnothing$

  dove il tipo $A+B$ è l'unione disgiunta tra il tipo A e il tipo B e $\varnothing$ è il tipo vuoto

- Una **corrispondenza tra deduzione naturale e il lambda calcolo tipato**:

Logica	**Sistema dei tipi**
Ipotesi	Variabili libere
Eliminazione dell'implicazione	Applicazione
Introduzione dell'implicazione	Astrazione

Proposizione 5.4: Dimostrazioni come programmi

Dati un tipo A e la sua proposizione corrispondente φ_A, si ha che:

$$\text{Esiste un termine di tipo principale } A \iff \varphi_A \text{ è una tautologia}$$

Esempi:

1. - Consideriamo il seguente tipo $(A \to B) \to (A \to (C \to B))$
   - Notiamo che la proposizione ad esso equivalente sia una tautologia:

     $$(A \implies B) \implies (A \implies (C \implies B)) = \neg(\neg A \lor B) \lor \neg A \lor \neg C \lor B =$$

     $$(A \land \neg B) \lor \neg A \lor \neg C \lor B = A \lor \neg A \lor \neg C \lor B = 1$$

     di conseguenza, esiste necessariamente un termine avente tale tipo come principale.
   - In particolare, il termine $fn\ x \Rightarrow fn\ y \Rightarrow fn\ z \Rightarrow xy$ possiede tale tipo come principale
2. - Consideriamo il seguente tipo $(A \to B) \to C$
   - Notiamo che la proposizione ad esso equivalente non sia una tautologia:

     $$(A \implies B) \implies C = \neg(\neg A \lor B) \lor C = (A \land \neg B) \lor C$$

     di conseguenza, non esiste un termine avente tale tipo come principale.