

Linguaggi di Programmazione

Sapienza Università di Roma

*libro del corso: integrati con le dispense del professor Cenciarelli
+ consultato appunti di Exyss e aflaag*

aglaia norza

03/02/2026

Indice

| | |
|---|-----------|
| 1 Algebre induttive | 3 |
| 1.1 I numeri naturali | 3 |
| 1.2 Algebre, algebre induttive | 4 |
| 1.3 Omomorfismi, lemma di Lambek | 7 |
| 2 Paradigma funzionale | 10 |
| 2.1 Linguaggi | 10 |
| 2.2 Exp | 10 |
| 2.2.1 Semantica operazionale | 11 |
| 2.3 Valutazioni Eager e Lazy | 13 |
| 2.4 Scoping | 14 |
| 2.4.1 Riassunto delle regole in <i>Exp</i> | 16 |
| 2.5 Fun | 17 |
| 2.5.1 Semantica eager | 17 |
| 2.5.2 Semantica lazy | 19 |
| 2.5.3 Termine ω | 21 |
| 2.5.4 Curryficazione | 21 |
| 3 Lambda calcolo | 23 |
| 3.1 Numeri di Church | 23 |
| 3.2 Cenni di SML | 25 |
| 3.3 Ricorsione nel paradigma funzionale: combinatore di punto fisso | 26 |
| 4 Paradigma imperativo | 28 |
| 4.1 <i>Imp</i> : un semplice linguaggio imperativo | 28 |

| | | |
|----------|---|-----------|
| 4.2 | All: un linguaggio con procedure | 32 |
| 4.2.1 | Semantica Operazionale: dichiarazioni, assegnamenti, references | 33 |
| 4.2.2 | Semantica Operazionale: semantiche di chiamata | 34 |
| 5 | Correttezza dei programmi | 36 |
| 5.1 | Correttezza nei linguaggi imperativi | 36 |
| 5.1.1 | Metodo delle invarianti | 36 |
| 5.2 | Logica di Hoare | 38 |
| 5.2.1 | Correttezza della divisione intera | 40 |
| 5.3 | Correttezza nei linguaggi funzionali | 43 |
| 5.3.1 | Omomorfismi e operatori di ricorsione | 43 |
| 5.3.2 | Logica equazionale | 48 |
| 6 | Sistemi dei tipi | 52 |
| 6.1 | F1: il Lambda Calcolo tipato semplice | 52 |
| 6.1.1 | Espressioni non tipabili | 54 |
| 6.2 | F2: il Lambda Calcolo polimorfo | 55 |
| 6.2.1 | Polimorfismo | 55 |
| 6.2.2 | Sistema F2 | 55 |
| 6.2.3 | Semantica e meccanismi di astrazione | 55 |
| 6.2.4 | Proprietà del Sistema F2 | 59 |
| 6.3 | Fun $_{\tau}$: Il sistema dei tipi di ML | 61 |
| 6.3.1 | Monotipi e Politipi | 61 |
| 6.3.2 | Istanza Generica | 61 |
| 6.3.3 | Regole di Inferenza di Fun $_{\tau}$ | 62 |
| 6.3.4 | Il Let-Polimorfismo | 63 |
| 7 | Isomorfismo di Curry-Howard | 65 |
| 7.0.1 | Corrispondenza tra Regole | 65 |
| 7.0.2 | Tipi e programmi, proposizioni e dimostrazioni | 65 |

Algebrae induttive

1.1. I numeri naturali

Def. 1: Assiomi di Peano

L'insieme \mathbb{N} dei numeri naturali si può definire mediante i cinque **assiomi di Peano**:

- 1) $0 \in \mathbb{N}$
- 2) $n \in \mathbb{N} \implies \text{succ}(n) \in \mathbb{N}$
- 3) $\nexists n \in \mathbb{N} \mid 0 = \text{succ}(n)$
- 4) $\forall n, m \text{ succ}(n) = \text{succ}(m) \implies n = m$ (iniettività)
- 5) $\forall S \subseteq \mathbb{N} (0 \in S \wedge (n \in S \implies \text{succ}(n) \in S) \implies S = \mathbb{N})$ (assioma di induzione)

assioma di induzione

L'assioma di induzione è necessario per evitare di equiparare ai numeri naturali insiemi che, essenzialmente, contengono una struttura come quella di \mathbb{N} , e un “qualcosa in più”. (Se all'interno dell'insieme A che stiamo considerando esiste un altro sottoinsieme proprio che rispetta gli altri assiomi, A non rispetterà il quinto assioma di Peano).

In più, il quinto assioma di Peano ci fornisce essenzialmente una definizione insiemistica di induzione.

Def. 2: Principio di Induzione

L'induzione può essere definita, basandosi sulle “proprietà” invece che sull' insiemistica, come segue:

$$\forall P \quad \frac{P(0), \quad P(n) \implies P(n+1)}{\forall n P(n)}$$

(la notazione equivale a $P(0) \wedge P(n) \wedge (P(0) \wedge (P(n) \implies P(n+1))) \implies \forall n P(n)$)

Possiamo dimostrare che il quinto assioma di Peano è equivalente al principio di induzione (in quanto i concetti di “proprietà” e “sottoinsieme” sono equivalenti).

Infatti, ad ogni proprietà corrisponde un sottoinsieme i cui elementi sono esattamente quelli che soddisfano tale proprietà

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

In questo modo, dire $P(0)$ equivale a dire $0 \in S$, e dire $P(n) \implies P(n+1)$ equivale a dire $n \in S \implies n+1 \in S$. E, allo stesso modo, dire $\forall n P(n)$ equivale a dire $\forall n, n \in S$, ovvero $S = \mathbb{N}$.

Def. 3: Numeri di von Neumann

Un altro modo di descrivere i numeri naturali viene dal matematico **John von Neumann**, che definisce i numeri naturali (“numeri di von Neumann”, \mathcal{N}) in questo modo:

- $0_{\mathcal{N}} = \emptyset$ (ovvero $\{\}$)
- $1_{\mathcal{N}} = \{0_{\mathcal{N}}\}$ (ovvero $\{\{\}\}$)
- $2_{\mathcal{N}} = \{0_{\mathcal{N}}, 1_{\mathcal{N}}\}$ (ovvero $\{\{\}, \{\{\}\}\}$)
- ...

I numeri di von Neumann rispettano gli assiomi di peano! (dalle dispense)

1.2. Algebre, algebre induttive

insieme unità e funzione nullaria

Ci è utile definire l'**insieme unità** $\mathbb{1} = \{*\}$. $\mathbb{1}$ è un insieme formato da un solo elemento (non ci interessa quale).

Un altro concetto che ci servirà è quello di **funzione costante** o **nullaria**. Una funzione nullaria f è tale che:

$$f : \mathbb{1} \rightarrow A \mid f() = a \quad a \in A$$

(chiaramente, essa è sempre iniettiva).

Una funzione nullaria su un insieme A può essere vista come un elemento di A (un qualsiasi insieme A è isomorfo a all’insieme di funzioni $\mathbb{1} \rightarrow A$ (l’insieme di funzioni $\mathbb{1} \rightarrow A$ ha la stessa cardinalità di A), il che ci permette di **trattare gli elementi di un insieme come funzioni**.

Def. 4: Algebra

Una **algebra** è una tupla (A, Γ) , dove:

- A è l’insieme di riferimento (“carrier” o “insieme sottostante”)
- $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_i\}$, è l’insieme di funzioni chiamate “operazioni fondamentali” o “costruttori” dell’algebra

la segnatura dei costruttori è: $\gamma_i : A^{\alpha_i} \times K_i \rightarrow A$.

Tra le algebre, consideriamo anche le algebre eterogenee, che prendono argomenti da insiemi diversi da A .

Def. 5: Chiusura di un insieme rispetto ad un'operazione

Sia $f : A^n \times K \rightarrow A$ un'operazione su A con parametri esterni $K = (K_1 \times \dots \times K_m)$.

Un insieme $S \subseteq A$ si dice **chiuso** rispetto ad f quando:

$$a_1, \dots, a_n \in S \implies f(a_1, \dots, a_n, k_1, \dots, k_n) \in S$$

nota!

Data un'operazione f che prende solo elementi esterni all'insieme S (come per esempio la funzione nullaria $\mathbb{1} \rightarrow A$), un insieme S si dice chiuso rispetto a $f \iff \text{Im}(f) \subseteq S$.

Def. 6: Algebra induttiva

Un'algebra A, Γ si dice **induttiva** quando:

- (1) tutte le $\gamma_i \in \Gamma$ sono iniettive
- (2) $\forall i, j \mid i \neq j, \text{Im}(\gamma_i) \cap \text{Im}(\gamma_j) = \emptyset$, ovvero tutte le γ_i hanno immagini disgiunte
- (3) $\forall S \subseteq A$, se S è chiuso rispetto a tutte le γ_i , allora $S = A$ (ovvero il principio di induzione è rispettato)

La terza condizione pone quindi che A sia la più piccola sotto-algebra di se stessa (ovvero non abbia sotto-algebre diverse da se stessa).

Le tre condizioni garantiscono quindi che:

- ci sia solo un modo per costruire ogni elemento dell'algebra (i, ii)
- non ci siano “elementi inutili” (iii)

Vediamo come possiamo costruire \mathbb{N} come algebra induttiva.

La definizione di algebra induttiva non considera il concetto di “elemento”, quindi, per il primo assioma di Peano, usiamo una *funzione costante* $\mathbb{0}$, con segnatura:

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

Abbiamo quindi una tupla $(\mathbb{N}, \{\text{succ}, \mathbb{0}\})$.

Per dimostrare che questa tupla sia un'algebra induttiva, dobbiamo ora verificare le tre condizioni:

- (1) tutte le γ_i sono induttive:

- $\mathbb{0}$ è necessariamente induttiva
- succ è induttiva per il secondo assioma di Peano

- (2) tutti i costruttori hanno immagini disgiunte:

- grazie al terzo assioma di Peano ($\exists n \in \mathbb{N} \mid 0 = \text{succ}(n)$), sappiamo che succ e $\mathbb{0}$ hanno immagini disgiunte

(3) principio di induzione:

- è verificato dal quinto assioma di Peano ($0 \in S$ corrisponde alla chiusura rispetto a \emptyset e $n \in \mathbb{N} \implies \text{succ}(n) \in \mathbb{N}$ corrisponde alla chiusura rispetto a succ)

alberi binari come algebre induttive

L'insieme degli alberi binari finiti (**B-trees**, leaf, branch), dove:

- **B-trees** = $\{t \mid t \text{ è una foglia, oppure } t = \langle t_1, t_2 \rangle \text{ con } t_1, t_2 \in \text{B-trees}\}$
- **leaf**: $1 \rightarrow \text{B-trees}$ (foglia)
- **branch**: $\text{B-trees} \times \text{B-trees} \rightarrow \text{B-trees} : (t_{sx}, t_{dx}) \rightarrow t$ (costruisce rami in modo che t_{sx} e t_{dx} siano i due sottoalberi di t)

è un'algebra induttiva.

Thm. 1: numero di nodi di un albero binario

Un albero binario con n foglie ha $2n - 1$ nodi

proof

Si può dimostrare per induzione strutturale sui costruttori degli alberi.

- (**caso base**): la proprietà è vera per l'albero formato da una sola foglia costruito con **leaf** (\circ) - esso ha infatti $n = 1$ foglie e $2n - 1 = 1$ nodi.
- (**ipotesi induttiva**): ogni argomento dei costruttori rispetta la proprietà
- dobbiamo quindi verificare che il costruttore **branch**, dati due argomenti che rispettano la proprietà, la mantenga
- (**passo induttivo**): abbiamo $t = \text{branch}(t_1, t_2)$.

Sia $n = n_1 + n_2$ il numero di foglie di t , dove le foglie di t_1 sono n_1 e quelle di t_2 sono n_2 .

Per ipotesi, t_1 ha $2n_1 - 1$ nodi e t_2 ne ha $2n_2 - 1$. Dunque, t avrà $(2n_1 - 1) + (2n_2 - 1) + 1$ nodi, ovvero $2(n_1 + n_2) - 1 = 2n - 1$, qed.

liste finite come algebra induttiva

Dato un insieme A , indichiamo con $A - \text{list}$ l'insieme delle liste finite di elementi di A .

La tupla $(A - \text{list}, \text{empty}, \text{cons})$ è un'algebra induttiva, dove:

- **empty**: $1 \rightarrow A - \text{list}$ è la funzione costante che restituisce la **lista vuota** “ $\langle \rangle$ ”.
- **cons**: $A \times A - \text{list} \rightarrow A - \text{list} : \text{cons}(3, \langle 5, 7 \rangle) = \langle 3, 5, 7 \rangle$ è la funzione che **costruisce una lista** aggiungendo un elemento in testa

Si tratta di un'algebra induttiva (notiamo che i due costruttori hanno immagini chiaramente disgiunte, sono entrambi chiusi per $A - \text{list}$, e c'è un unico modo per costruire ogni lista).

liste infinite

Le liste infinite non possono essere un'algebra induttiva, in quanto contengono una sotto-algebra induttiva (quella delle liste finite).

i booleani come algebra non induttiva

Consideriamo l'algebra (B, not) , dove $B = \{0, 1\}$ e $\text{not}: B \rightarrow B : b \mapsto \neg b$.

Notiamo che not è sicuramente iniettiva, e che, poiché è l'unico costruttore, anche la seconda caratteristica delle algebre induttive è rispettata.

Notiamo però che l'algebra non rispetta il terzo requisito. Se consideriamo infatti $\emptyset \subseteq B$, notiamo che not è chiusa rispetto ad esso.

Infatti, l'implicazione $x \in \emptyset \implies \text{not}(x) \in \emptyset$ risulta vera per falsificazione della premessa (non esistono elementi in \emptyset).

(\emptyset, not) è quindi una sotto-algebra induttiva di B , che però è diversa da essa. L'implicazione della terza condizione $(x \in \emptyset \implies \text{not}(x) \in \emptyset) \implies \emptyset = B$ è falsa, e (B, not) non è quindi un'algebra induttiva.

1.3. Omomorfismi, lemma di Lambek

digressione - teoria delle categorie

Facciamo una piccola parentesi che introduce alcune nozioni di teoria delle categorie (perché è molto interessante).

La teoria delle categorie studia in modo astratto le strutture matematiche. Una categoria \mathcal{C} consiste di:

- una classe $\text{ob}(\mathcal{C})$, i cui elementi sono chiamati **oggetti**
- una classe $\text{mor}(\mathcal{C})$, i cui elementi sono chiamati **morfismi** (o mappe o frecce); ogni morfismo $f : a \rightarrow b$ ha associati un unico oggetto sorgente a e un unico oggetto destinazione b .
- per ogni terna di oggetti $a, b, c \in \mathcal{C}$, è definita una funzione $\text{mor}(b, c) \times \text{mor}(a, b) \rightarrow \text{mor}(a, c)$ chiamata **composizione di morfismi**. La composizione di $f : b \rightarrow c$ con $g : a \rightarrow b$ si indica con $f \circ g : a \rightarrow c$

la composizione deve soddisfare i seguenti assiomi:

(*associatività*): se $f : a \rightarrow b$, $g : b \rightarrow c$ e $h : c \rightarrow d$, allora $h \circ (g \circ f) = (h \circ g) \circ f$

(*identità*): per ogni oggetto x esiste un morfismo $\text{id}_x : x \rightarrow x$ chiamato **morfismo identità**, tale che per ogni morfismo $f : a \rightarrow x$ vale $\text{id}_x \circ f = f$ e per ogni morfismo $g : x \rightarrow b$ si ha $g \circ \text{id}_x = g$.

Quindi, ogni oggetto è associato ad un unico morfismo identità. Questo permette di dare una definizione di categoria basata esclusivamente sulla classe dei morfismi: gli **oggetti vengono identificati con i corrispondenti morfismi identità**.

All'interno della teoria delle categorie, una funzione iniettiva $f : B \rightarrow C$ si chiama **monomorfismo**. Visto che non si possono utilizzare gli elementi per definire l'iniettività, un monomorfismo è descritto come una funzione f tale che:

$$\forall A, \forall h, k : A \rightarrow B, \quad h \circ f = k \circ f \implies h = k$$

(se le funzioni h e k sono identiche ogni volta che vengono composte con f , significa che non ci sono valori in f che sono assunti da più di un elemento di B)

Def. 7: Algebre con la stessa segnatura

Due algebre (A, Γ_A) e (B, Γ_B) hanno la stessa segnatura se, sostituendo A con B in tutte le $\gamma_i \in \Gamma_A$, si ottiene Γ_B .

(La segnatura di un'algebra è data dalle segnature delle sue operazioni).

Def. 8: Omomorfismo

Date due algebre con la stessa segnatura (A, Γ) e $(B, \Delta = \{\delta_1, \dots, \delta_k\})$, un omomorfismo è una funzione $f : A \rightarrow B$ tale che:

$$\forall i \ f(\gamma_i(a_1, \dots, a_k, k_1, \dots, k_m)) = \delta_i(f(a_1), \dots, f(a_k), k_1, \dots, k_m)$$

(con k_1, \dots, k_m parametri esterni)

(definizione algebrica: $\forall a, b \in A$, date \circ operazione di A e \bullet operazione di B , si ha $f(a \circ b) = f(a) \bullet f(b)$)

un omomorfismo “rispetta le operazioni”

- nota: la composizione di due omomorfismi è a sua volta un omomorfismo

Def. 9: Isomorfismo

Un isomorfismo è un omomorfismo biettivo.

(Due algebre sono isomorfe (\cong) quando esiste un isomorfismo tra loro)

Thm. 2: Omomorfismo tra algebre con stessa segnatura

Sia A un'algebra induttiva. Per ogni algebra B (non necessariamente induttiva) con la stessa segnatura, esiste un **unico omoformismo** $A \rightarrow B$.

Thm. 3: Lemma di Lambek

Due algebre induttive A e B con la **stessa segnatura** sono necessariamente **isomorfe**.

proof

- Siccome A è un'algebra induttiva, $\exists!$ omomorfismo $f : A \rightarrow B$.
- Allo stesso modo, $\exists!$ omomorfismo $g : B \rightarrow A$.
- Componendo i due omomorfismi, si ottiene un omomorfismo $g \circ f$ con segnatura $A \rightarrow A$.
- Sappiamo che per ogni algebra esiste l'omomorfismo “identità”.
- Sappiamo anche, per il teorema sopra, che esiste un unico omomorfismo $A \rightarrow A$.

Ne segue necessariamente che $g \circ f = \text{Id}_A$. (lo stesso discorso si applica a $f \circ g = \text{Id}_B$)

- $g \circ f = \text{Id} \iff g = f^{-1}$, quindi g e f sono funzioni invertibili (= biettive) $\implies g, f$ sono isomorfismi $\implies A \cong B$

Paradigma funzionale

2.1. Linguaggi

Definiamo un **linguaggio** L come insieme di stringhe.

Per descrivere la sintassi di linguaggi formali (la grammatica), usiamo la BNF (Backus-Naur Form), con questa sintassi:

`<simbolo> ::= __espressione__`

Esempio: prendiamo come esempio questa grammatica:

$M, N ::= 5 \mid 7 \mid M + N \mid M * N$

Le espressioni che seguono questa grammatica, sono del tipo:

- “5” o “7”
- un’espressione $M + N$ o $M * N$, in cui M e N rispettano a loro volta la grammatica

Introduciamo una funzione $eval : L \rightarrow \mathbb{N}$, che valuta le espressioni del linguaggio:

- $eval(5) = 5$
- $eval(7) = 7$
- $eval(M + N) = eval(M) + eval(N)$
- $eval(M * N) = eval(M) * eval(N)$

Possiamo notare subito che $(L, eval)$ non è un’algebra induttiva. Infatti, una stringa come “ $5 + 7 * 5$ ” potrebbe essere stata generata in due modi diversi: $(5 + 7) * 5$ e $5 + (7 * 5)$.

Possiamo però stipulare che sia induttiva. Ci basta infatti considerare $+$, $*$, 5 e 7 come costruttori dell’algebra. In questo modo, $(5 + 7) * 5$ risulta essere un oggetto diverso da $5 + (7 * 5)$. È quindi possibile dimostrare che $(L, 5, 7, +, *)$ è un’algebra induttiva.

2.2. Exp

Def. 10: Linguaggio Exp

Introduciamo il linguaggio Exp , con grammatica:

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

dove:

- $k \in Val = \{0, 1, \dots\}$ è una costante
- $x \in Var$ è una variabile
- $M + N : Exp \times Exp \rightarrow Exp$ è la somma tra due espressioni
- $let : Var \times Exp \times Exp \rightarrow Exp$ assegna alla variabile x il valore M all'interno di N

esempi:

- $let x = 3 in x + x + 2$ viene valutata come 8
- $let x = 3 in 12$ viene valutata come 12

Questo linguaggio causa però facilmente ambiguità. Per esempio, come valutiamo un'espressione come $let x = 3 in let y = x in let x = 5 in y$?

Per esplicitare la struttura del termine, è necessario legare le occorrenze delle variabili alle dichiarazioni.

Def. 11: Variabili libere, legate, scope

Si dice che un'occorrenza di una variabile x è **libera** in un termine t quando non compare nel corpo di N di nessun sottotermine di t nella forma $let x = M in N$ (quindi, quando non le viene assegnato un valore).

Ogni occorrenza libera di x in un termine N si dice **legata** (bound) alla dichiarazione di x nel termine $let x = M in N$.

Lo scope di una dichiarazione è l'insieme delle occorrenze libere di x in N .

Lo **scope di una variabile** è la porzione di programma all'interno della quale una variabile può essere riferita.

Introduciamo una funzione $free : Exp \rightarrow \mathcal{P}(Var)$, che restituisce l'insieme delle variabili libere di un'espressione:

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

(eliminiamo la x , dalle variabili libere in N perché viene dichiarata dal $let x$, ma non la eliminiamo da M perché potrebbe comparire al suo interno come variabile libera, e M non fa parte dello scope di $let x$ (esempio: in $let x = x in x$, la x è libera perché compare libera in $= x$))

esempio: $free(let x = 7 in x + y) = \{y\}$

2.2.1. Semantica operazionale

Vogliamo introdurre nel linguaggio Exp il concetto di “quanto fa?” (valutazione di un'espressione).

Per farlo, abbiamo bisogno di definire un ambiente all'interno del quale valutare le espressioni (stile operazionale, “structural operational semantics”).

Def. 12: Ambienti

Un **ambiente** è una funzione parziale (funzione non necessariamente definita su tutti gli elementi del dominio) con dominio finito che associa dei valori ad un insieme finito di variabili.

$$E : Var \xrightarrow{fin} Val$$

Scriviamo gli ambienti come insiemi di coppie. Per esempio, l'ambiente E in cui z vale 3 e y vale 9 è indicato con $\{(z, 3), (y, 9)\}$.

Notiamo che, essendo E una funzione parziale, il dominio $dom(E)$ è un sottoinsieme finito di Var .

Def. 13: Insieme di ambienti

Env è definito come l'insieme degli ambienti di Exp .

Gli ambienti si possono **concatenare** in questo modo:

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

Per esempio, $\{(z, 3), (y, 9)\}\{(z, 4)\}(z) = 4$ e $\{(z, 3), (y, 9)\}\{(z, 4)\}(x)$ è indefinito.

Def. 14: Semantica operazionale eager di Exp

La **semantica operazionale** di Exp è una relazione

$$\rightsquigarrow \subseteq Exp \times Env \times Val$$

in cui $(M, E, v) \in \rightsquigarrow \iff$ il programma M , nell'ambiente E , produce il valore v .

Un'asserzione di appartenenza $(M, E, v) \in \rightsquigarrow$ viene chiamata *giudizio operazionale*, e si scrive

$$E \vdash M \rightsquigarrow v$$

Questa relazione è definita dalle seguenti **regole**:

$$E \vdash k \rightsquigarrow k \quad [const]$$

(in ogni ambiente E , una costante k vale k)

$$E \vdash x \rightsquigarrow v \quad \text{se } v = E(x) \quad [var]$$

(una variabile x vale v se la funzione ambiente $E(x)$ le associa il valore v)

$$\frac{E \vdash M \rightsquigarrow v \quad E \vdash N \rightsquigarrow w}{E \vdash M + N \rightsquigarrow v + w} \quad [plus]$$

(se nello stesso ambiente M vale v e N vale w , $M + N$ varrà $v + w$)

$$\frac{E \vdash M \rightsquigarrow v_1 \quad E\{(x, v_1)\} \vdash N \rightsquigarrow v_2}{E \vdash \text{let } x = M \text{ in } N \rightsquigarrow v_2} \quad [let]$$

(essenzialmente, per valutare una *let*:

- si valuta M ($E \vdash M \rightsquigarrow v_1$)
- si “associa” il risultato v_1 a x , concatenando (x, v_1) all’ambiente
- e si valuta N nel nuovo ambiente)

Notiamo che si utilizza la relazione \rightsquigarrow e non una funzione $Exp \times Env \rightarrow Val$, perché si potrebbe avere più di un risultato (per esempio nel caso del multithreading, in cui un diverso ordine di esecuzione di un programma dà output diversi), o anche nessun risultato (per esempio nel caso in cui in Exp compare una variabile x , che però Env non definisce), entrambi casi non accettati dalla definizione di funzione.

precedenza

Introduciamo un concetto di “precedenza” nella valutazione di un’espressione potenzialmente ambigua; un’espressione del tipo:

$$\text{let } x = 3 \text{ in let } x = \text{let } y = 2 \text{ in } x + y \text{ in } x + 7 + x$$

in assenza di parentesi, va valutata “partendo dall’interno”.

Corrisponde quindi a

$$\text{let } \textcolor{red}{x} = 3 \text{ in } \left([\text{let } \textcolor{green}{x} = (\text{let } \textcolor{blue}{y} = 2 \text{ in } \textcolor{purple}{x} + y) \text{ in } \textcolor{green}{x} + 7] + \textcolor{red}{x} \right)$$

E si ha quindi che:

- la $\textcolor{red}{x}$ in $\textcolor{red}{x} + y$ e quella finale ($+ \textcolor{red}{x}$) sono quelle valutate dal *let* iniziale
- il valore della $\textcolor{green}{x}$ in $x + 7$ è invece dato dal risultato di $\text{let } x = (\text{let } y = 2 \text{ in } \textcolor{purple}{x} + y)$
- (il *let* iniziale è del tipo $\text{let } x = M \text{ in } N + L$)

Facciamo un esempio di valutazione di un’espressione:

$$\frac{\begin{array}{c} (x, 3) \vdash 2 \rightarrow 2 \\ \hline (x, 3) \vdash \text{let } y = 2 \text{ in } x + y \rightsquigarrow 5 \end{array} \quad \frac{\begin{array}{c} (x, 3)(y, 2) \vdash x \rightsquigarrow 3 \quad (x, 3)(y, 2) \vdash y \rightsquigarrow 2 \\ \hline (x, 3)(y, 2) \vdash x + y \rightsquigarrow 5 \end{array} \quad [\text{Plus}] \quad \frac{(x, 3)(x, 5) \vdash x + 7 \rightsquigarrow 12}{(x, 3) \vdash \text{let } x = (\text{let } y = 2 \text{ in } x + y) \text{ in } x + 7 \rightsquigarrow 12} \quad [\text{Let}] \quad (x, 3) \vdash x \rightsquigarrow 3 \quad [\text{Plus}] \\ \hline (x, 3) \vdash (\text{let } x = (\text{let } y = 2 \text{ in } x + y) \text{ in } x + 7) + x \rightsquigarrow 15 \quad [\text{Let}] \end{array}}{\text{let } x = 3 \text{ in let } x = (\text{let } y = 2 \text{ in } x + y) \text{ in } x + 7 + x \rightsquigarrow 15}$$

2.3. Valutazioni Eager e Lazy

La valutazione utilizzata fino a questo momento viene definita **eager**, in quanto valuta N immediatamente (anche nel caso in cui non servisse veramente valutarlo).

Se infatti consideriamo un caso del tipo $\text{let } = [\text{espressione lunghissima}] \text{ in } 7$, notiamo immediatamente che la valutazione di N non è necessaria, in quanto l’espressione farà, in ogni caso, 7.

Introduciamo quindi un approccio **lazy**, che consiste nel valutare un termine solo quando (e se) ce n’è veramente bisogno.

La valutazione di N in un termine del tipo $\text{let } x = N \text{ in } M$ viene rimandata, quindi, al momento in cui ad M (eventualmente) servirà il suo valore.

Def. 15: Regole della semantica lazy di Exp

- I termini non valutati subito vengono conservati in un “ambiente pigro” - estendiamo quindi Env in questo modo:

$$Env = Var \xrightarrow{fin} Exp$$

(gli ambienti contengono ora anche i termini non valutati, quindi non possiamo avere come codominio Val)

- la nuova regola per le variabili è:

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

- la nuova regola per il *let* è:

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

Notiamo che però non sempre l’approccio lazy è più veloce: per esempio, per l’espressione $\text{let } x = N \text{ in } (x + x)$, N viene calcolata 3 volte con l’approccio lazy e una sola con quello eager.

Mettiamo i due approcci a confronto sull’espressione

$$\text{let } x = 2 \text{ in let } y = x \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3$$

- approccio eager:**

$$\frac{\begin{array}{c} (x, 2) \vdash x \rightsquigarrow 2 & (x, 2) \vdash 1 \rightsquigarrow 1 \\ \hline (x, 2) \vdash x + 1 \rightsquigarrow 3 \end{array} \quad \frac{\begin{array}{c} (x, 2)(y, 3) \vdash 7 \rightsquigarrow 7 & (x, 2)(y, 3)(x, 7) \vdash y \rightsquigarrow 3 \\ \hline (x, 2)(y, 3) \vdash \text{let } x = 7 \text{ in } y \rightsquigarrow 3 \end{array}}{(x, 2) \vdash \text{let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3} \quad \frac{}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}}$$

- approccio lazy:**

$$\frac{\begin{array}{c} (x, 2)(y, x + 1)(x, 7) \vdash 7 \rightsquigarrow 7 \\ \hline (x, 2)(y, x + 1)(x, 7) \vdash x \rightsquigarrow 7 \quad (x, 2)(y, x + 1)(x, 7) \vdash 1 \rightsquigarrow 1 \\ \hline \frac{\begin{array}{c} (x, 2)(y, x + 1)(x, 7) \vdash x + 1 \rightsquigarrow 8 \\ \hline (x, 2)(y, x + 1)(x, 7) \vdash y \rightsquigarrow 8 \\ \hline (x, 2)(y, x + 1) \text{ let } x = 7 \text{ in } y \rightsquigarrow 8 \\ \hline (x, 2) \vdash \text{let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 8 \end{array}}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 8}}}$$

Notiamo che i due approcci ci danno risultati diversi.

Ciò è causato non dall’approccio valutativo, bensì dallo **scoping** utilizzato. Abbiamo infatti utilizzato quello che viene definito “scoping dinamico”, il che ha causato problemi perché, in Exp , lazy dinamico e eager non sono equivalenti.

2.4. Scoping

Def. 16: Scoping

Lo **scoping** di un linguaggio è l’insieme di regole che determinano la visibilità di una variabile all’inter-

no di un programma (ossia che consentono di associare una variabile a ciascun riferimento (= uso della variabile mediante un identificatore)).

Def. 17: Scoping statico

Quando si usa lo **scoping statico**, i riferimenti ad una variabile sono risolti in base alla **struttura sintattica** del programma (tipicamente in base ad una dichiarazione).

Ovvero, durante la valutazione viene utilizzato l'**ambiente definito a tempo di interpretazione** (e non di valutazione).

Def. 18: Scoping dinamico

Quando si usa lo **scoping dinamico**, i riferimenti ad una variabile sono risolti in base allo **stato di esecuzione** del programma (per esempio, una dichiarazione estende il suo effetto fino a che non si incontra un'altra dichiarazione di variabile con lo stesso nome).

Quindi, durante la valutazione viene utilizzato l'**ambiente definito a tempo di valutazione** stesso.

Def. 19: Regole della semantica lazy statica di *Exp*

Dobbiamo mantenere, oltre alle espressioni rimaste da valutare, anche gli ambienti in cui valutarle.

$$Env_{LS} = Var \xrightarrow{fin} (Exp \times Env_{LS})$$

- la nuova regola per le variabili è:

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

- la nuova regola per il *let* è:

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

Valutiamo la stessa espressione anche con questo approccio:

$$\begin{array}{c} \dfrac{(x, 2, \emptyset) \vdash x \rightsquigarrow 2 \quad (x, 2, \emptyset) \vdash 1 \rightsquigarrow 1}{(x, 2, \emptyset) \vdash x + 1 \rightsquigarrow 3} \\ \dfrac{}{E(x, 7, E) \vdash y \rightsquigarrow 3} \\ \dfrac{\color{blue}{(x, 2, \emptyset)(y, x + 1, (x, 2, \emptyset)} \vdash \text{let } x = 7 \text{ in } y \rightsquigarrow 3}{\dfrac{(x, 2, \emptyset) \vdash \text{let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}} \end{array}$$

In *Exp* non c'è, invece, differenza tra eager statico e eager dinamico.

Essenzialmente, in *Exp*:

| | statico | dinamico |
|-------|---------------------------|-----------|
| lazy | equiv | non equiv |
| eager | equiv (e uguali tra loro) | |

“commutatività” in Exp

In Exp , si ha:

$$\text{let } x = (\text{let } y = M \text{ in } N) \text{ in } L \not\equiv \text{let } y = M \text{ in let } x = N \text{ in } L$$

- nella prima espressione, y è definita solo all'interno di N
- nella seconda, è definita prima, ed è quindi visibile anche in L
- quindi, le due espressioni sono equivalenti solo se y non compare libera (non ri-definita) in L

2.4.1. Riassunto delle regole in Exp

eager (statico/dinamico)

$$E \vdash k \rightsquigarrow k \quad [const]$$

$$E \vdash x \rightsquigarrow v \quad \text{se } v = E(x) \quad [var]$$

$$\frac{E \vdash M \rightsquigarrow v \quad E \vdash N \rightsquigarrow w}{E \vdash M + N \rightsquigarrow v + w} \quad [plus]$$

$$\frac{E \vdash M \rightsquigarrow v_1 \quad E\{(x, v_1)\} \vdash N \rightsquigarrow v_2}{E \vdash \text{let } x = M \text{ in } N \rightsquigarrow v_2} \quad [let]$$

lazy statico

$$\frac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \quad \text{se } E(x) = (M, E') \quad [var]$$

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

(lazy dinamico)

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

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

2.5. Fun

Introduciamo un nuovo linguaggio, *Fun*, che estende *Exp* con la nozione di **funzione**.

Def. 20: *Fun*

La grammatica di *Fun* è:

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

dove:

- le regole presenti in *Exp* (1-4) rimangono invariate, con gli appropriati cambi di dominio (es. $\text{let} : \text{Var} \times \text{Fun} \times \text{Fun} \rightarrow \text{Fun}$)
- $\text{fn} : \text{Var} \times \text{Fun} \rightarrow \text{Fun}$ è una **funzione** (anonima) con parametro x
 - una funzione $\text{fn } x \Rightarrow M$ si può rappresentare in maniera alternativa attraverso la sua **chiusura**, $(x, M) \in \text{Var} \times \text{Fun}$
- $\cdot : \text{Fun} \times \text{Fun} \rightarrow \text{Fun}$ è l'**applicazione di funzioni**
 - il termine sinistro (che, perché l'espressione abbia semanticamente senso, deve necessariamente essere una funzione) viene applicato al termine destro (quindi $MN = M(N)$)
- l'insieme *Val* non coincide più con quello delle costanti, ma corrisponde a $\{0, 1, 2, \dots\} \cup (\text{Var} \times \text{Fun})$ (ovvero costanti \cup chiusure: una variabile può rappresentare una funzione)

Quindi, per esempio:

- $(\text{fn } x \Rightarrow x + 1) 5 = 6$ (la funzione $x + 1$ è applicata all'argomento 6)
- $(\text{fn } x \Rightarrow x 5)(\text{fn } y \Rightarrow y + 1) = 6$
la funzione prende in input una funzione (in questo caso “successore”), e la applica a 5
- $(\text{fn } x \Rightarrow (\text{fn } y \Rightarrow y x)) 3 (\text{fn } z \Rightarrow z + 1) = 4$
è una funzione che, presa in input un'altra funzione, la applica ad x - le passiamo la funzione “successore”, che, applicata a 3, dà 4.
- un'applicazione del tipo $\text{fn } x \Rightarrow 10 x$ “non ha semantica” (non è valutabile), in quanto 10 non è una funzione e non si può applicare a x

precedenza di *apply*

la precedenza nell'applicazione è a sinistra

$$MNL \equiv (MN)L$$

2.5.1. Semantica eager

Def. 21: **Semantica eager dinamica di *Fun***

- [fn eager dinamico]

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

- [apply eager dinamico]

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

funziona così:

- (1) *valutazione della funzione (M)*: valuto M per capire cosa sto chiamando - (poiché è un *fn*, per la relativa regola) ottengo la chiusura composta dal parametro formale x e dal corpo del codice M' .
- (2) *valutazione dell'argomento (N)*: valuto N nell'ambiente corrente - infatti, essendo una semantica eager, devo risolvere l'espressione dell'argomento e ottenere un valore concreto v prima di entrare nella funzione.
- (3) *passaggio dei parametri (binding)*: aumento l'ambiente creando un'associazione tra il parametro x e il valore appena calcolato v (per passare il parametro alla funzione con il suo valore appena calcolato)
- (4) *esecuzione del corpo (M')*: nel nuovo ambiente valuto il corpo M' . Il risultato v' sarà il risultato finale dell'applicazione.

consideriamo l'espressione: `(fn x => x + 1) (2 * 3)`

- M è `fn x => x + 1` - il parametro è x e il corpo M' è $x + 1$ (quindi $M \rightsquigarrow (x, x + 1)$)
- N è `2 * 3` - lo valuto subito (eager) e ottengo $v = 6$.
- aggiorno l'ambiente: $E' = E(x, 6)$
- valuto M' (ovvero $x + 1$) nell'ambiente E' : sostituisco 6 a x e ottengo $6 + 1 = 7$.

Def. 22: Semantica eager statica di Fun

- [*fn* eager statico]

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

- [*apply* eager statico]

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

funziona così:

- (1) *valutazione della funzione*: valuto M nell'ambiente corrente; il risultato è la chiusura (x, M', E') .
 - x, M' : sono il parametro e il codice della funzione.
 - E' : è l'ambiente definizione della funzione, che era stato salvato dentro la chiusura dalla regola *fn*
- (2) *valutazione dell'argomento (eager)*: valuto N nell'ambiente corrente E (quello in cui avviene la chiamata, non quello della funzione) - ottengo il valore v .
- (3) *ripristino dell'ambiente + binding*: poiché lo scoping è statico, per eseguire la funzione non uso l'ambiente corrente E ; invece:
 - prendo l'ambiente E' dalla chiusura.
 - lo estendo con l'associazione (x, v) .

in qualche modo, “torno indietro nel tempo” a quando la funzione è stata creata, e aggiungo il parametro attuale

- (4) *esecuzione del corpo*: valuto M' usando questo nuovo ambiente ($E'(x, v)$). il risultato v' è il risultato finale.

Lemma 1: Eager dinamico e statico in *Fun*

(Al contrario di *Exp*), si ha che:

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

Eager statico vs eager dinamico

Per capire la differenza tra eager statico e dinamico, dobbiamo far sì che una variabile venga ridefinita all' consideriamo: `let y = 1 in (let f = fn x => x + y in (let y = 100 in f 5))`

- quando definisco `f`, l'ambiente contiene $y = 1$ - in una semantica statica, la chiusura salva questo ambiente: $E' = \dots(y, 1)$.
- quando chiamo `f 5`, sono in un ambiente $E = \dots(y, 100)$
- in una semantica statica, non uso E , ma recupero E' dalla chiusura e calcolo $5 + y$ usando $(y, 1)$
ottengo 6
- in una semantica dinamica, uso l'ambiente corrente con $(y, 100)$
ottengo 105

2.5.2. Semantica lazy

Def. 23: Semantica lazy dinamica di *Fun*

- [fn lazy dinamico]

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

- [apply lazy dinamico]

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

funziona così:

- (1) *valutazione della funzione* (M): valuto M nell'ambiente corrente per ottenere la definizione della funzione: parametro x e corpo M' .
- (2) *nessuna valutazione dell'argomento*: (differenza tra lazy e eager) il linguaggio non calcola quanto vale N adesso - prende l'intera espressione N e la tiene da parte così com'è (la valuterà solo quando (e se) servirà)
- (3) *passaggio dei parametri (call-by-name)*: estendo l'ambiente corrente E . Associo alla variabile x direttamente l'espressione N

$$E_{\text{new}} = E(x, N)$$

(4) *esecuzione del corpo*: valuto M' nel nuovo ambiente.

ogni volta che nel corpo M' verrà usata la variabile x , solo in quel momento verrà valutata l'espressione N (usando l'ambiente attuale (dinamico))

Def. 24: Semantica lazy statica di *Fun*

- [fn lazy statico]

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

- [apply lazy statico]

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

(1) *valutazione della funzione*: valuto M nell'ambiente corrente - ottengo la chiusura (x, M', E') .

- E' è l'ambiente in cui la funzione è stata definita

(2) *creazione del “thunk”*: essendo una dinamica lazy, non calcolo N . ma, essendo anche statica, se dovessi calcolare N , dovrò farlo usando l'ambiente corrente E , non quello della funzione - quindi creo una coppia chiamata “thunk”: (N, E) .

- N : il codice dell'argomento.
- E : l'ambiente di “chiamata” necessario per calcolarlo.

(3) *passaggio dei parametri + binding*: prendo l'ambiente della funzione E' (statico) e lo estendo associando a x il thunk appena creato.

$$E'_{\text{new}} = E'(N, E)$$

(4) *esecuzione del corpo*: valuto M' in questo nuovo ambiente

se nel corpo M' verrà usata x , il sistema prenderà il thunk (N, E) e valuterà N usando l'ambiente E salvato

Lemma 2: Lazy dinamico e statico in *Fun*

(Al contrario di *Exp*), si ha che:

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

Lazy Statico vs Lazy Dinamico

Per capire la differenza tra lazy statico e dinamico, possiamo passare un argomento che contiene una variabile e vedere quale valore viene letto quando l'argomento viene valutato.

Consideriamo questa espressione:

```
let z = 10 in (let f = fn x => (let z = 99 in x) in f z)
```

Qui passiamo z (che vale 10) alla funzione, ma al suo interno c'è una nuova variabile locale z che vale 99. L'argomento x (che è z) viene valutato solo dentro la funzione. Il valore di z sarà diverso tra semantica lazy statica e lazy dinamica.

- (1) **lazy statico** l'argomento si porta dietro il suo ambiente di nascita.

- al momento della chiamata $f z$, siamo nell'ambiente dove $z = 10$
- viene creato un thunk: una coppia $(z, (z, 10))$
- entriamo nella funzione - viene definita una variabile locale $z = 99$
- il corpo richiede x - viene aperto il thunk
- il thunk ignora l'ambiente attuale (dove $z = 99$) e usa il suo ambiente salvato
- risultato: 10

(2) **lazy dinamico** l'argomento viene valutato nell'ambiente attuale.

- al momento della chiamata $f z$, si passa solo il codice dell'argomento: "z" (l'ambiente non viene salvato)
- entriamo nella funzione - viene definita una variabile locale $z = 99$.
- il corpo richiede x - il sistema valuta il codice "z" guardando l'ambiente corrente
- trova che z vale 99
- risultato: 99

2.5.3. Termine ω

Introduciamo un termine interessante - $(fn x \Rightarrow xx)(fn x \Rightarrow xx)$ - e tentiamo di valutarlo con un approccio eager (dinamico)

$$\frac{\emptyset \vdash fn x \Rightarrow xx \rightsquigarrow (x, xx) \quad (x, (x, xx)) \vdash x \rightsquigarrow (x, xx) \quad (x, (x, xx)) \vdash xx \rightsquigarrow}{\emptyset \vdash fn x \Rightarrow xx \rightsquigarrow (x, xx) \quad (x, (x, xx)) \vdash xx \rightsquigarrow}$$

$$\frac{\emptyset \vdash fn x \Rightarrow xx \rightsquigarrow (x, xx) \quad (x, (x, xx)) \vdash xx \rightsquigarrow}{\emptyset \vdash (fn x \Rightarrow xx)(fn x \Rightarrow xx)}$$

Notiamo che il termine va in loop - infatti, si ha che, per valutare $(x, (x, xx)) \vdash xx$, bisogna prima valutare $(x, (x, xx)) \vdash xx$ (se stesso).

Termine ω

Chiamiamo ω il termine appena introdotto:

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

Qui emerge una grande **differenza tra valutazione eager e lazy**: con un approccio eager, un'espressione del tipo $let x = \omega in 7$ va in loop, mentre con una valutazione lazy viene valutata correttamente.

2.5.4. Curryficatione

Def. 25: Curryficatione

La **curryficatione** (o "applicazione parziale") è la tecnica che consiste nel tradurre una funzione che accetta più argomenti in una sequenza di famiglie di funzioni, ciascuna delle quali accetta un singolo argomento.

Partendo da una funzione $f : (X \times Y) \rightarrow Z$ che prende due argomenti, la sua curryficatione tratta il primo argomento come un parametro, e crea una famiglia di funzioni $f_x : Y \rightarrow Z$ tale che, per ogni $x \in X$, c'è esattamente una funzione f_x tale che $\forall y \in Y, f_x(y) = f(x, y)$.

$$\text{curry} : [(X \times Y) \rightarrow Z] \rightarrow [X \rightarrow (Y \rightarrow Z)]$$

$$f \mapsto h : f(x, y) = h(x)(y)$$

Si trasforma quindi una funzione che prende due argomenti in una funzione ad un argomento che ritorna un'altra funzione t.c. $\text{curry}((f))(x)(y) = f(x, y)$.

(Il processo inverso prende il nome di **decurryficazione**).

Curryficazione in *Fun*

La curryficazione ci permette di introdurre una notazione contratta del *fn*:

$$[\text{fn } xy \Rightarrow] \equiv [\text{fn } x \Rightarrow (\text{fn } y \Rightarrow)]$$

Possiamo così introdurre **funzioni a più argomenti** all'interno del linguaggio *Fun*.

3.1. Numeri di Church

Tra i diversi modi di rappresentare i numeri naturali, ci interessa presentare quello di Alonzo Church.

Per Church, il cui mondo è fatto di funzioni, un numero naturale n corrisponde all'applicare n volte una funzione x su un argomento y .

Possiamo, per esempio, rappresentare il numero 2 “di Church” in *Fun* in questo modo:

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

(ovvero, presa una funzione x e un valore y di partenza, si applica due volte la funzione x (prima al valore stesso “ $(x\ y)$ ”, e poi al risultato di questa applicazione - “ $x(“ “)$ ”)))

Def. 26: Numeri di Church in *Fun*

Più in generale, indicando con M^nN il termine $M(M(\dots(MN)\dots))$ (in cui si ripete n volte M), un numero c_n di Church si può rappresentare, con la sintassi di *Fun*, in questo modo:

$$c_n \equiv fn\ x\ y \Rightarrow x^n\ y$$

Ovvero:

- zero := $fn\ xy \Rightarrow y$
- uno := $fn\ xy \Rightarrow xy$
- due := $fn\ xy \Rightarrow x(xy)$
- tre := $fn\ xy \Rightarrow x(x(xy))$
- ...

Possiamo rappresentare anche altri concetti essenziali come la funzione “successore”, la somma e il prodotto tramite numeri di Church.

Def. 27: *succ* di Church

La funzione **successore di Church**, seguendo lo stesso ragionamento, dovrà ricevere un numero di Church z in ingresso, e restituire il numero di Church che applica x a y , $z + 1$ volte:

$$succ \equiv fn\ z\ x\ y \Rightarrow x\ (z\ x\ y)$$

$$\text{succ} \equiv fn z \Rightarrow fn x \Rightarrow fn y \Rightarrow x (z x y)$$

essenzialmente, dato z numero di Church di cui calcolare il successore (che vuole quindi come parametri x funzione e y valore di partenza), si applica una volta in più x .

La funzione si può scrivere anche, equivalentemente, in questo modo:

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

“anticipando” essenzialmente il $+1$ (prima si applica x una volta “in più”, e poi le altre z volte)

Facciamo un esempio concreto:

$$\begin{aligned}\text{succ } c_1 &= fn x y \Rightarrow x (c_1 x y) \\ &= fn x y \Rightarrow x ((fn x y \Rightarrow xy) x y) \\ &= fn x y \Rightarrow x (x y)\end{aligned}$$

che corrisponde al due di Church !

Def. 28: Dechurchificazione

Perché questo abbia più senso, ci è utile poter ricondurre i numeri di Church all’algebra dei numeri naturali.

Possiamo, con questo scopo, definire una funzione *dechurch* (o “*eval*”), che, dato un numero di Church c_n , ci restituisce l’intero corrispondente n .

$$\text{dechurch}(M) = M (fn x \Rightarrow x + 1) 0$$

Quello che stiamo facendo, essenzialmente, è passare al numero di Church in input la funzione “successore” dei numeri naturali, e il numero 0. Il numero di Church applicherà quindi n volte *succ()* a partire da 0, ritornandoci n .

Facciamo un esempio. Dato $c_2 = fn x y \Rightarrow x (x y)$, calcoliamo *dechurch*(c_2) in questo modo:

$$\begin{aligned}\text{dechurch}(c_2) &= (fn x \Rightarrow fn y \Rightarrow x (x y)) (fn x \Rightarrow x + 1) 0 \\ &\text{sostituisco } x : \\ &= (fn y \Rightarrow (fn x \Rightarrow x + 1)((fn x \Rightarrow x + 1) y)) 0 \\ &\text{sostituisco } y : \\ &= (fn x \Rightarrow x + 1)((fn x \Rightarrow x + 1) 0) \\ &\text{applico:} \\ &= (fn x \Rightarrow x + 1)(0 + 1) \\ &= (fn x \Rightarrow x + 1) 1 \\ &= 1 + 1 \\ &= 2\end{aligned}$$

Thm. 4

Si ha:

$$\text{dechurch}(\text{church}(M)) \rightsquigarrow k \iff M \rightsquigarrow k$$

Def. 29: Somma di Church

Seguendo lo stesso ragionamento usato per calcolare *succ* di Church, la **somma di Church** tra z e w è quella funzione che applica $x w$ volte a y , e passa il risultato a z , (che la applicherà altre z volte).

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

(passo a z , numero di Church che vuole quindi una funzione e un valore di partenza, come funzione x e come valore di partenza l'applicazione di x a partire da y , w volte)

Def. 30: Prodotto di Church

Allo stesso modo, possiamo calcolare il **prodotto di Church**.

Sappiamo che $u \times v = u + u + u + \dots + u$.
 v volte

Sappiamo che un numero di Church ha come parametri una funzione e un valore da cui iniziare. Possiamo quindi “definire” una funzione plus_u che somma u al suo input: $\text{fn } x \Rightarrow (\text{plus } x \text{ } v)$ (in cui plus è la somma di Church precedentemente definita).

Ora, ci basta fornire al numero v come parametri questa nuova funzione e c_0 come valore di inizio, perché questa, essenzialmente, sommi $u \times v$ volte a partire da zero.

$$\begin{aligned} \text{times} &\equiv \text{fn } v \Rightarrow \text{fn } u \Rightarrow v \text{ } (\text{fn } x \Rightarrow (\text{plus } x \text{ } u)) \text{ } c_0 \\ &\equiv \text{fn } v \text{ } u \Rightarrow v \text{ } (\text{plus}_u) \text{ } c_0 \end{aligned}$$

3.2. Cenni di SML

Standard ML (SML) è un linguaggio di programmazione funzionale di alto livello, appartenente alla famiglia di linguaggi *ML (Meta Language)*. È stato standardizzato negli anni '90 e si distingue per il suo forte sistema di tipi statico con inferenza automatica. Il linguaggio discende direttamente da *ML*, sviluppato da Robin Milner nel 1973 come linguaggio di metaprogrammazione per dimostrazioni automatiche.

L'SML di cui tratta questo corso è SML/NJ, lo Standard ML of New Jersey.

Def. 31: Sintassi base di SML/NJ

Definiamo i costrutti base del linguaggio *Fun* in SML.

- definizione di una variabile:

```
val x = 10;
val y = 10 + x;
```

- *let*

```
val x = let val a = 3 in a + 5 end;
```

- *fn*

```
val succ = fn x => x+1;
fun succ x = x+1;
```

- *apply*

```
val due = succ 1;
val tre = succ (succ 1);
```

in SML, la precedenza nell'applicazione è a sinistra: $x\ x\ y \equiv (x\ x)\ y$

Possiamo ora definire i numeri di Church e le loro operazioni.

numeri di Church:

```
val zero = fn x => fn y => y;
fun zero x y = y;

val uno = fn f => fn x => f x;
fun uno x y = x y;

val due = fn f => fn x => f (f x);
fun due x y = x (x y);
```

Operazioni:

```
val succ = fn w => (fn x => fn y => x(w x y));
val plus = fn u => fn v => (u succ v);
val plus u v x y = v x(u x y);

val times = fn u => fn v => (u (fn z => (plus z v)) zero);
```

3.3. Ricorsione nel paradigma funzionale: combinatore di punto fisso

Def. 32: Punto fisso di una funzione

Un punto fisso per una funzione $f : A \rightarrow A$ (endofunzione) è un elemento $x \in A$ t.c. $f(x) = x$.

Esempio 1: In un anello $(A, +, \cdot)$, l'elemento neutro della somma 0_A è un punto fisso per l'operazione di opposto (l'oppuesto dell'elemento neutro è se stesso).

Esempio 2: data la funzione F definita in questo modo:

$F\ f \equiv \text{if } (n>0) \text{ then } 1 \text{ else } n*f(n-1)$

(funzione che prende in input un'altra funzione g e ritorna la funzione $\text{if } (n>0) \text{ then } 1 \text{ else } n*g(n-1)$)
(quindi, se le passiamo in input la funzione "72", abbiamo " $F\ 72 \equiv \text{if } (n>0) \text{ then } 1 \text{ else } n*72(n-1)$ ")

Qual è il suo punto fisso? Notiamo che è la funzione fattoriale.

Infatti, $F\ \text{fact} \equiv \text{if } (n>0) \text{ then } 1 \text{ else } n*\text{fact}(n-1) \equiv \text{fact}$ stessa!

Def. 33: Combinatore di punto fisso Y

("combinatore paradossale Y" di Haskell Curry)

Definiamo come **combinatore di punto fisso** una funzione che, data una funzione F , trova il suo punto fisso YF .

Quindi una funzione per cui si ha che $F(YF) = YF$.

Nel *lambda calcolo*,

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

Applichiamo Y ad una funzione F :

$$\begin{aligned} YF &\equiv fn\ f \Rightarrow ((fn\ x \Rightarrow f(xx))(fn\ x \Rightarrow f(xx)))F \\ &\quad \text{sostituiamo il parametro } F \\ &\xrightarrow{\beta} (fn\ x \Rightarrow F(xx))(fn\ x \Rightarrow F(xx)) \\ &\quad \text{sostituiamo il parametro } (fn\ x \Rightarrow F(xx)) \\ &\xrightarrow{\beta} F((fn\ x \Rightarrow F(xx))(fn\ x \Rightarrow F(xx))) \\ &\equiv F(YF) \end{aligned}$$

Otteniamo quindi che $YF \equiv F(YF)$.

Possiamo continuare all'infinito: $YF = F(YF) = F(F(YF)) = \dots$: abbiamo introdotto la **ricorsione** nel lambda calcolo.

Thm. 5: Ricorsione nel lambda calcolo

Grazie al combinatore di punto fisso Y , possiamo ottenere la ricorsione nel lambda calcolo. Infatti, data una funzione F , l'espressione YF applica F ricorsivamente.

Paradigma imperativo

Ci è utile conoscere la differenza tra **call by reference** e **call by value**:

- call by reference: viene passato alla funzione un riferimento (l'indirizzo) alla variabile originale; qualsiasi modifica interna modifica direttamente il valore originale.
- call by value: viene passata alla funzione una copia del valore della variabile; qualsiasi modifica interna è fatta solo sulla copia e non influisce sul valore originale.

4.1. *Imp*: un semplice linguaggio imperativo

Def. 34: **Grammatica del linguaggio *Imp***

$$\begin{aligned}
 k ::= & 0 \mid 1 \mid \dots \mid \text{true} \mid \text{false} \\
 M, N ::= & k \mid x \mid M + N \mid M < N \\
 p, q ::= & \text{skip} \mid p; q \mid \text{if } M \text{ then } p \text{ else } q \mid \text{while } M \text{ do } p \mid \\
 & \text{var } x = M \text{ in } p \mid x := M
 \end{aligned}$$

dove:

- $M, N \in \text{Exp}$
- $p, q \in \text{Imp}$
- ***skip*** è il **programma che non fa niente**
- $p; q$ indica che verrà eseguito prima p , e poi q
- ***if – then – else*** esegue p se l'espressione M è vera e q altrimenti
- ***while*** esegue p finché M è vera
- ***var*** rappresenta l'**inizializzazione** di una nuova variabile all'interno di p (variabile locale in p)
- $::=$ rappresenta l'**assegnamento** di un valore ad una variabile già definita

Introduciamo il concetto di **locazioni** per prepararci al linguaggio *All*. Le locazioni, definite dall'insieme Loc , sono le locazioni di memoria, ovvero gli indirizzi dove si trovano i valori (essenzialmente come i puntatori).

Domini semanticci

Definiamo Env , l'insieme degli ambienti, in questo modo:

$$Env = Var \xrightarrow{fin} Loc$$

(un ambiente associa alle variabili, invece dei valori, le loro locazioni di memoria)

Introduciamo anche un insieme $Store$ delle memorie:

$$Store = Loc \xrightarrow{fin} Val$$

che “estrae” i valori dalle loro locazioni di memoria.

Allo stesso modo in cui si possono concatenare ambienti, si possono concatenare anche memorie:

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

Def. 35: Semantiche operazionali

Imp definisce due relazioni di valutazione:

- una per le **espressioni** M (che producono valori senza avere side-effects):

$$\xrightarrow{M} \subseteq Env \times Exp \times Store \times Val$$

$$E \vdash M, S \xrightarrow{M} v$$

- e una per i **programmi** p (che non producono valori ma cambiano la memoria)

$$\xrightarrow{p} Env \times Imp \times Store \times Store$$

$$E \vdash p, S \xrightarrow{p} S'$$

(omitteremo gli indici M e p perché deducibili dal contesto)

- Costante:

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

- Variabile:

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

Per valutare x , guardiamo l'ambiente E per trovare il suo indirizzo l , e poi lo store S per trovare il valore v contenuto in l .

- Somma:

$$(\text{se } v_1 + v_2 = v) \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M + N, S \rightsquigarrow v}$$

- Minore (vero):

$$(\text{se } v_1 < v_2) \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M < N, S \rightsquigarrow \text{true}}$$

- Minore (falso):

$$(\text{se } v_1 \geq v_2) \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M < N, S \rightsquigarrow \text{false}}$$

- Skip:

$$E \vdash \text{skip}, S \rightsquigarrow S$$

- Sequenza:

$$\frac{E \vdash p, S \rightsquigarrow S' \quad E \vdash q, S' \rightsquigarrow S''}{E \vdash p; q, S \rightsquigarrow S''}$$

Il comando p modifica S in S' . Il comando q modifica il nuovo stato S' in S'' .

- If (condizione vera):

$$\frac{E \vdash M, S \rightsquigarrow \text{true} \quad E \vdash p, S \rightsquigarrow S'}{E \vdash \text{if } M \text{ then } p \text{ else } q, S \rightsquigarrow S'}$$

- If (condizione falsa):

$$\frac{E \vdash M, S \rightsquigarrow \text{false} \quad E \vdash q, S \rightsquigarrow S'}{E \vdash \text{if } M \text{ then } p \text{ else } q, S \rightsquigarrow S'}$$

- While (condizione vera):

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

Caso induttivo (true): se la guardia è vera: si esegue il corpo p che produce uno stato intermedio S' , e si ri-esegue l'intero ciclo partendo da S' . Il risultato finale S'' è il risultato del ciclo.

- While (condizione falsa):

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

Caso base (false): Se la guardia è falsa, il ciclo termina e lo stato non cambia.

- Inizializzazione:

$$(\text{con } l \text{ nuova}) \frac{E \vdash M, S \rightsquigarrow v \quad E(x, l) \vdash p, S(l, v) \rightsquigarrow S'}{E \vdash \text{var } x := M \text{ in } p, S \rightsquigarrow S'}$$

- (1) si calcola il valore iniziale v .
- (2) si trova una locazione l libera (allocazione).
- (3) si estende l'ambiente ($E(x, l)$) e si aggiorna lo store ($S(l, v)$) (inizializzazione).
- (4) si esegue p in questo nuovo contesto.

NB: All'uscita da p , le modifiche allo store (S') rimangono, ma la variabile x scompare dallo scope perché si torna all'ambiente E originale.

- Assegnamento:

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

Non si modifica l'ambiente E . Si cerca la locazione già esistente associata a x e si sovrascrive il valore nello store.

Notiamo che questa semantica non fa “garbage collection” (all'uscita di un blocco var, la locazione l non viene rimossa dallo Store S' , anche se non è più accessibile dall'ambiente E)

4.2. All: un linguaggio con procedure

All rappresenta il nucleo di un linguaggio “Algol-like”.

All estende $Exp \in Imp$ con un costrutto sintattico per indicizzare array. Inoltre, distingue tra espressioni “assegnabili” ($L - Exp$, ammesse alla sinistra di un assegnamento), e non.

Def. 36: Grammatica del linguaggio All

$$k ::= 0 \mid 1 \mid \dots \mid true \mid false$$

$$(L - Exp \ni) V ::= x \mid x[M]$$

$$M, N ::= k \mid V \mid M + N \mid M < N$$

$$p, q ::= skip \mid p; q \mid if\ M\ then\ p\ else\ q \mid while\ M\ do\ p \mid$$

$$var\ x = M\ in\ p \mid arr\ x = [M_0, \dots, M_n]\ in\ p \mid V := M \mid$$

$$proc\ y(x)\ is\ p\ in\ q \mid call\ y(M)$$

dove:

- V rappresenta quindi le $L - Exp$
- $M, N \in Exp$
- $p, q \in Imp$ (programmi)
- arr dichiara un array x e gli assegna le espressioni M_0, \dots, M_n in p
- $proc$ dichiara una procedura (funzione) y con parametro x in p
- $call$ chiama la procedura y passandole come argomento M

Gli array in All sono associati a sequenze finite e non vuote di locazioni (che indichiamo con Loc^+ (con $+$ “chiusura positiva” della stella di Kleene, ovvero insieme di stringhe di lunghezza finita sull’alfabeto, esclusa la stringa vuota))

Domini semanticici

A causa dell’introduzione degli array, una variabile non è più associata a una singola locazione, ma a una sequenza finita e non vuota di locazioni (Loc^+).

Definiamo quindi Env , l’insieme degli ambienti, in questo modo:

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

(array, variabile, corpo della funzione e ambiente in cui valutarla)

Def. 37: Semantiche operazionali

In All introduciamo una nuova relazione di valutazione specifica per le **L-Value** (che restituiscono una locazione l), distinta da quella per le **R-Value** (che restituiscono un valore v).

- relazione di valutazione per le **espressioni assegnabili**:

$$\xrightarrow{V} \subseteq Env \times L - Exp \times Store \times Loc$$

$$E \vdash V, S \xrightarrow{V} l$$

- relazioni (già esistenti) per le **espressioni** M :

$$\xrightarrow{M} \subseteq Env \times Exp \times Store \times Val$$

$$E \vdash M, S \xrightarrow{M} v$$

- e per i **programmi** p :

$$\xrightarrow{p} \subseteq Env \times Imp \times Store \times Store$$

$$E \vdash p, S \xrightarrow{p} S'$$

4.2.1. Semantica Operazionale: dichiarazioni, assegnamenti, references

- **Accesso a variabile e array** ([loc1], [loc2]):

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

$$\frac{E \vdash M, S \xrightarrow{M} m}{E \vdash x[M], S \xrightarrow{V} l_m} (\text{se } E(x) = \langle l_0, \dots, l_n \rangle \text{ e } 0 \leq m \leq n)$$

Per ottenere il valore di una variabile, ora sono necessari due passi: risolvere la locazione e poi leggere lo store.

- **Dereferenziazione** ([ref]):

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

Si usa la regola di accesso (relazione \xrightarrow{V}) per ottenere la locazione di V , e ne si legge il valore con la relazione \xrightarrow{M} .

- **Assegnamento** ([assign]):

$$\frac{E \vdash M, S \xrightarrow{M} v \quad E \vdash V, S \xrightarrow{V} l}{E \vdash V := M, S \xrightarrow{p} S(l, v)}$$

Per assegnare un nuovo valore a V , si valuta il nuovo valore M , si usa la relazione \xrightarrow{M} per accedere alla sua locazione, e si dà un nuovo valore alla locazione nello store.

La regola distingue tra il calcolo del *R-value* (l'espressione M valutata a v) e del *L-value* (l'espressione assegnabile V valutata alla locazione l).

- **Dichiarazione di Array** ([arr]):

$$\frac{E \vdash M_0, S \xrightarrow{v} v_0 \dots \quad E \vdash M_n, S \xrightarrow{v} v_n \quad E(x, \langle l_0, \dots, l_n \rangle) \vdash p, S((l_0, v_0) \dots (l_n, v_n)) \xrightarrow{v} S'}{E \vdash \text{arr } x = [M_0, \dots, M_n] \text{ in } p, S \xrightarrow{v} S'} (*)$$

(*) dove l_i è nuova, per $i = 0 \dots n$

Questa regola gestisce l'allocazione di blocchi contigui di memoria.

- (1) tutte le espressioni iniziali M_i vengono valutate.
- (2) vengono reperite $n + 1$ locazioni libere (l_i).
- (3) la variabile x viene associata nell'ambiente all'intera sequenza di locazioni $\langle l_0, \dots, l_n \rangle$.
- (4) il programma p viene eseguito con lo store in cui alle nuove locazioni corrispondono i valori v_i

■ **Dichiarazione di Procedura ([proc]):**

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

Si valuta il programma q nell'ambiente in cui a y viene associata la *chiusura* formata da parametro x e corpo del programma p - si salva anche l'ambiente corrente (di definizione della procedura), garantendo quindi lo **scoping statico**.

4.2.2. Semantica Operazionale: semantiche di chiamata

Esistono tre modalità principali per il passaggio dei parametri in *All*, distinte in base a *cosa* viene passato (valore vs locazione) e *quando* viene valutato (eager vs lazy).

Call-by-Value (Passaggio per Valore)

È una semantica **eager statica**. L'argomento viene valutato a un valore v *prima* dell'esecuzione del corpo della procedura. Viene allocata una **nuova locazione** per il parametro formale.

Def. 38: **Regola Call-by-Value**

$$\frac{E \vdash M, S \xrightarrow{M} v \quad E'(x, l) \vdash p, S(l, v) \xrightarrow{p} S'}{E \vdash \text{call } y(M), S \xrightarrow{p} S'} (*)$$

(*) dove $E(y) = (x, p, E')$ e l è una locazione **nuova** ($l \notin \text{dom}(S)$).

Effetto: Modifiche alla variabile x dentro la procedura non influenzano l'argomento originale (lavorano su una copia).

Call-by-Reference (Passaggio per Riferimento)

È una semantica **eager statica**, ma l'argomento deve essere una L-Exp (V). Viene passata la **locazione** dell'argomento già esistente valutato.

Def. 39: **Regola Call-by-Reference**

L'argomento della chiamata **deve** essere di tipo assegnabile (V).

$$\frac{E \vdash V, S \xrightarrow{V} l \quad E'(x, l) \vdash p, S \xrightarrow{p} S'}{E \vdash \text{call } y(V), S \xrightarrow{p} S'} (*)$$

(*) dove $E(y) = (x, p, E')$.

Effetto: Il parametro formale x diventa un *alias* dell'argomento V . Le modifiche a x sono visibili esternamente.

Call-by-Name (Passaggio per Nome)

È una semantica **lazy statica**. L'argomento non viene valutato subito. Viene passata l'espressione non valutata insieme al suo ambiente. L'argomento viene valutato solo quando (e ogni volta che) il parametro viene usato.

Estensione Ambiente per CBN

L'ambiente deve poter contenere espressioni non valutate:

$$Env = Var \xrightarrow{fin} Loc^+ \cup (Var \times Imp \times Env) \cup (\mathbf{LExp} \times \mathbf{Env})$$

Questa semantica richiede due modifiche alle regole operazionali:

- (1) **Regola di Chiamata:** associa al parametro la chiusura dell'argomento.
- (2) **Regola di Accesso ([loc3]):** quando si usa la variabile, si valuta l'espressione salvata.

Def. 40: Regole Call-by-Name

- nuova regola di accesso [loc3]:

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

il sistema guarda nell'ambiente E per vedere cos'è x - non è associato a una locazione fisica diretta l , ma a una coppia (V, E') (in cui V è l'espressione originale passata come argomento dal chiamante, e E' l'ambiente del chiamante (dove V ha senso)), quindi, per ottenere la locazione di x deve trovare quella di V nell'ambiente E'

- [call-by-name]

$$\frac{E'(x, (V, E)) \vdash p, S \xrightarrow{p} S'}{E \vdash \text{call } y(V), S \xrightarrow{p} S'} \text{ (se } E(y) = (x, p, E')\text{)}$$

Lemma 3: (Non-)equivalenza tra le semantiche

In *All*, non esistono due semantiche equivalenti.

5.1. Correttezza nei linguaggi imperativi

5.1.1. Metodo delle invarianti

La correttezza dei programmi viene spesso definita in relazione al contenuto di determinate variabili (per esempio, “il contenuto finale di `result` è $x+y$ ”). Questo risulta facile da tracciare se il programma consta di sole assegnazioni, ma più difficile se per esempio contiene loops.

Una possibile soluzione a questo problema è l'utilizzo delle **invarianti**.

Def. 41: Invariante

Un predicato è detto **invariante** per una sequenza di operazioni quando il esso risulta vero prima e dopo l'esecuzione della sequenza.

Una **loop invariant** è un'invariante (quindi una proprietà, un predicato) per cui si può dimostrare che, data per assunta la sua verità prima dell'inizio del loop, essa rimarrà vera al termine di un'iterazione. Se si può dimostrare che è vera prima di entrare nel loop e rimane vera al termine della prima iterazione, si può dedurre che rimarrà vera anche una volta terminato il loop.

(Si tratta di una semplice dimostrazione per induzione: data P invariante, dobbiamo dimostrare $Q(n) = "P$ rimane vera dopo n iterazioni". $Q(1)$ è il caso base ed è vera per ipotesi; assumiamo per ipotesi induttiva che $Q(n-1)$ valga. Visto che sappiamo che il contenuto di un'iterazione non cambia la verità di P , possiamo dedurre che anche $Q(n)$ valga (e quindi che Q valga $\forall n$))

5.1.1.1. Correttezza della moltiplicazione egizia

Il papiro di Rhind (~ 1650 a.C.) descrive l'algoritmo usato dagli antichi egizi per svolgere la moltiplicazione.

L'algoritmo della moltiplicazione egizia tra x e y in questo modo:

- (1) Raddoppio x .
- (2) Se y è pari, lo divido per due; altrimenti, sottraggo uno e lo divido per due.
- (3) Ripeto i passi (1) e (2) fino a ottenere $y = 1$
- (4) Mantengo solo le coppie in cui y è dispari, e sommo tra loro tutti i valori x di quelle coppie.
- (5) Il risultato corrisponderà a $x \times y$.

Proviamo con 45×138 . Moltiplico e divido per due seguendo l'algoritmo:

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

Mantengo solo le coppie con y dispari e sommo le x corrispondenti:

| | |
|------|----|
| 90 | 69 |
| 360 | 17 |
| 5760 | 1 |

$$90 + 360 + 5760 = 6210 = 45 \times 138$$

Definiamo l'algoritmo sotto forma di codice Java:

```
public class Aegypt {
    public static void main(String[] args) {
        int a = 45;
        int b = 138;
        int x = a, y = b, res = 0;

        while (y >= 1) {
            if (y % 2 == 0) {
                x = x + x;
                y = y/2;
            }
            else {
                res = res + x;
                y = y - 1;
            }
        }
        System.out.println(a + " times " + b + " is " + res);
    }
}
```

L'invariante di questo programma è:

$$y \geq 0 \quad \& \quad a * b = x * y + res$$

Infatti:

- vale pre-loop:
abbiamo $x=a$, $y=b$, $res=0$, quindi (sostituendo) $a * b = a * b + 0$ (e $y=b=45 \geq 0$)
- vale anche dopo una prima iterazione; abbiamo due casi:
 - (1) $y \% 2 == 0$ - entro nell'if:
 res non cambia, e, se avevo $y \geq 0$ prima, sicuramente anche $y/2 \geq 0$
 - (2) $y \% 2 != 0$ - non entro nell'if:

- il “caso peggiore” per $y \geq 0$ è $y = 1$ (ultima iterazione del loop) - anche in quel caso, $y = 1 - 1 = 0$, e la proprietà $y \geq 0$ rimane vera.
- per $a * b = x * y + res$, invece: avendo $y=1$, stiamo moltiplicando x per un valore più piccolo di 1 (quindi stiamo essenzialmente sottraendo una x); questa sottrazione viene però compensata da $res += x$ (quello che viene tolto da $x * y$ viene riaggiunto a res).

Abbiamo quindi che $y \geq 0 \ \&\ a * b = x * y + res$ vale prima del loop, e dopo una qualsiasi iterazione del loop. Possiamo quindi concludere che è un’invariante per questo programma !

5.2. Logica di Hoare

(introdotta nel 1969 da C.A.R. Hoare (inventore del quicksort !))

La **logica di Hoare** è un sistema formale che rientra tra le semantiche assiomatiche e permette di valutare la correttezza di programmi utilizzando formalismi matematici.

Def. 42: Semantica della logica di Hoare

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

dove:

- M, N sono **espressioni numeriche**
- A, B sono **espressioni booleane**
- p, q sono **programmi**
- introduciamo il simbolo \supset per indicare l'**implicazione logica** \Rightarrow (Peano–Russell notation)

(usiamo una versione minimalista delle operazioni booleane - gli altri simboli si possono derivare (per esempio, $\neg A \equiv A \supset false$))

Def. 43: Tripla di Hoare

Siano A e B due espressioni booleane e p un programma.

La tripla:

$$\{A\} p \{B\}$$

significa “se A è soddisfatto prima dell’esecuzione di p , allora B è soddisfatto dopo la terminazione di p , se questa avviene.”

(essenzialmente, se eseguo p in uno “stato” che soddisfa A , ottengo uno “stato” che soddisfa B .)

A viene chiamata *precondizione*, e B *postcondizione*.

- nota bene ! questa interpretazione della tripla di Hoare si chiama **correttezza parziale**: la postcondizione vale infatti **a condizione che p termini**.

Quindi, definiamo come **formula** un'espressione che appartiene alla grammatica:

$$\varphi ::= \{A\} p \{B\}$$

Def. 44: Regole di inferenza generali

- regola del **true**:

$$\frac{}{\{P\} C \{true\}} (true)$$

se si esegue un comando C partendo da P , e il comando termina, si finisce in uno stato “qualsiasi”
- poiché *true* non richiede nessuna proprietà specifica, la tripla è sempre valida.

- regola del **false**:

$$\frac{}{\{false\} C \{P\}} (false)$$

se la precondizione è *false* (cioè uno stato impossibile), allora la tripla è valida “a vuoto”

- *Regole di “adattamento”*: si usano quando le regole di un comando danno una tripla che non è esattamente quella che si vuole dimostrare - fanno da ponte tra la tripla ottenuta e quella desiderata

- regola dello **strengthening**, ovvero rafforzamento della precondizione:

$$\frac{P \supset Q \quad \{Q\} C \{R\}}{\{P\} C \{R\}} (str)$$

(se il codice funziona per un caso generale (Q), funzionerà sicuramente anche per un caso più specifico (P))

- regola dello **weakening**, ovvero indebolimento della postcondizione:

$$\frac{\{P\} C \{Q\} \quad Q \supset R}{\{P\} C \{R\}} (weak)$$

se il codice garantisce una proprietà forte (Q), allora garantisce implicitamente anche una proprietà più debole (R)

- regola dell'**and**:

$$\frac{\{P\} C \{Q_0\} \quad \dots \quad \{P\} C \{Q_n\}}{\{P\} C \{Q_0 \wedge \dots \wedge Q_n\}} (and)$$

- regola dell'**or**:

$$\frac{\{P_0\} C \{Q\} \quad \dots \quad \{P_n\} C \{Q\}}{\{P_0 \vee \dots \vee P_n\} C \{Q\}} (or)$$

queste due regole si utilizzano per dividere dimostrazioni complesse in sotto-dimostrazioni più semplici (divide et impera)

Def. 45: Regole di inferenza per i programmi

- regola dello **skip**:

$$\frac{}{\{P\} skip \{P\}} (skip)$$

Il comando `skip` non fa nulla: lo stato della memoria non cambia. Quindi, qualsiasi proprietà P che era vera prima, rimane vera dopo.

■ regola dell'**assign**:

$$\frac{}{\{[E/x]P\} x := E \{P\}} (\text{assign})$$

Per garantire che la proprietà P sia vera dopo l'assegnamento $x := E$, deve essere vero prima di esso che la proprietà P valga sostituendo x con E (si ragiona da destra verso sinistra).

- se voglio ottenere la postcondizione $\{x > 10\}$ con il comando $x := y + 1$.
- cosa deve essere vero prima? prendo $x > 10$ e sostituisco x con $y + 1$.
- la precondizione sarà $\{y + 1 > 10\}$, ovvero $\{y > 9\}$.

■ regola dell'**if-then-else**:

$$\frac{\{P \wedge Q\} C_1 \{R\} \quad \{P \wedge \neg Q\} C_2 \{R\}}{\{P\} \text{ if } Q \text{ then } C_1 \text{ else } C_2 \{R\}} (\text{if})$$

Per dimostrare che l'intero `if` porta alla postcondizione, dobbiamo dimostrare due sotto-condizioni:

- (1) se vale la precondizione P e la guardia Q è vera, eseguendo C_1 arrivo a R .
- (2) se vale la precondizione P e la guardia Q è falsa ($\neg Q$), eseguendo C_2 arrivo comunque a R .

Se entrambi i rami convergono su R , allora l'intero costrutto è corretto.

■ regola del **while**:

$$\frac{\{P \wedge Q\} C \{P\}}{\{P\} \text{ while } Q \text{ do } C \{P \wedge \neg Q\}} (\text{while})$$

Non possiamo sapere quante volte sarà eseguito il corpo di un ciclo `while` - per risolverlo usiamo quindi l'invariante di ciclo P .

- premessa (corpo del ciclo): Dobbiamo dimostrare che se l'invariante P è vero all'inizio del giro, e la guardia Q è vera (quindi si entra nel ciclo), allora dopo aver eseguito C , l'invariante P è ancora vera.
- conclusione: se la premessa regge, allora quando il ciclo finisce (e se finisce), sapremo due cose:
 - (1) l'invariante P è ancora vero (perché il corpo lo preserva sempre).
 - (2) la guardia Q è falsa (altrimenti il ciclo non sarebbe finito).

La postcondizione è quindi $P \wedge \neg Q$.

■ regola della **concatenazione**:

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}} (\text{comp})$$

Per unire due comandi in sequenza, serve uno "stato intermedio" Q - la postcondizione del primo comando (C_1) deve combaciare esattamente con la precondizione del secondo comando (C_2) (come una catena: $P \rightarrow [C_1] \rightarrow Q \rightarrow [C_2] \rightarrow R$)

5.2.1. Correttezza della divisione intera

Il seguente programma calcola la divisione intera di $x \div y$.

```
b := x;
a := 0;
while (b >= y) do
  b = b-y;
  a = a+1;
```

(il resto si trova in b)

Vogliamo trovare la **tripla di Hoare** per questo programma, e dimostrare la sua correttezza.

Notiamo che una tripla adatta sarebbe:

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

(Analizziamola:

- il programma va in loop quando $y \leq 0 \leq x$, ma controllare $y \geq 0$ non ci interessa perché ne stiamo dimostrando la *correttezza parziale*. Ci basta quindi $x \geq 0$, per assicurarci che si entri nel *while*.
- $ay + b = x$ indica la corretta esecuzione della divisione
- $b \geq 0$ è necessario, in quanto il resto non può essere negativo (non possiamo dividere una o più volte di troppo)
- $b < y$ ci assicura che la divisione sia finita (che non avremmo potuto dividere una volta in più)

)

Notiamo che anche un programma come

```
a := 0;
b := 0;
y := 1;
x := 0;
```

rispetterebbe la tripla di Hoare (senza però eseguire la divisione intera tra x e y).

Per impedire una situazione come questa dovremmo usare il simbolo ' per riferirci, nella postcondizione, ai valori che le variabili avevano ad inizio programma.

La tripla diventerebbe quindi $ay' + b = x' \wedge b \geq 0 \wedge b < y$. (Alternativamente, potremmo far sì che i programmi non possano modificare i valori di input).

In ogni caso, prenderemo come buona la specifica senza apici.

Passiamo ora a dimostrare la correttezza della tripla tramite le regole di inferenza della logica di Hoare.

Dividiamo il programma in sezioni per semplificargli la dimostrazione.

- $b := x$;

Dobbiamo avere $b = x \wedge b \geq 0$ (per la precondizione della tripla di Hoare).

$$\frac{\text{??} \quad b := x \quad \{b = x \wedge b \geq 0\}}{\{x \geq 0\} b := x \{b = x \wedge b \geq 0\}}$$

Qual è la più debole precondizione che, dato $b := x$, ci permette di affermare che sicuramente $\{b = x \wedge b \geq 0\}$ sarà soddisfatta?

La weakest precondition ci viene data dalla regola dell'**assign**, $\{[M/x]A\} x := M \{A\}$. (la precondizione è la postcondizione, in cui si sostituisce x con M)

$$\frac{\{x = x \wedge x \geq 0\} \quad b := x \quad \{b = x \wedge b \geq 0\}}{\{x \geq 0\} b := x \{b = x \wedge b \geq 0\}}$$

Notiamo però che la precondizione della regola dell'assign non corrisponde a quella della tripla di Hoare che stiamo cercando di dimostrare. Ci serve un altro passaggio per determinare la sua correttezza. Usiamo la regola dello **strengthening** per arrivare dalla premessa che vogliamo dimostrare a quella che abbiamo.

$$\frac{(x \geq 0) \supset (x = x \wedge x \geq 0) \quad \{x = x \wedge x \geq 0\} \ b := x \ \{b = x \wedge b \geq 0\}}{\{x \geq 0\} b := x \{b = x \wedge b \geq 0\}}$$

Non ci serve dimostrare la correttezza dell'implicazione $(x \geq 0) \supset (x = x \wedge x \geq 0)$, in quanto presumiamo che nella nostra logica le verità dell'aritmetica (come $x = x$) e i teoremi della logica classica (come $a \wedge b \supset a$) siano già dimostrati.

Ci interessa ora continuare la composizione della dimostrazione passando alla prossima sezione del programma, e utilizzando la post-condizione appena dimostrata come pre-condizione.

■ `a := 0`

Usiamo lo stesso ragionamento visto sopra (assign + strengthening).

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

■ `b := x; a := 0;`

Ora dobbiamo mettere insieme le due dimostrazioni. Visto che la pre-condizione della seconda corrisponde alla post-condizione della prima, possiamo usare la regola della **concatenazione**:

$$\frac{\{x \geq 0\} b := x \{b = x \wedge b \geq 0\} \quad \{b = x \wedge b \geq 0\} a := 0 \{ay + b = x \wedge b \geq 0\}}{\{x \geq 0\} b := x; a := 0 \{ay + b = x \wedge b \geq 0\}}$$

Sappiamo quindi che, prima di entrare nel `while`, la condizione $\{ay + b = x \wedge b \geq 0\}$, che da ora in poi chiameremo A , è soddisfatta.

■ `while (b >= y) do b := b - y; a := a + 1;`

Dobbiamo quindi applicare la regola del **while** $\left(\frac{\{P \wedge Q\} C \{P\}}{\{P\} \text{while } Q \text{ do } C \{P \wedge \neg Q\}} (\text{while}) \right)$

La situazione è quindi:

$$\frac{\{A \wedge b \geq y\} \ b := b - y; a := a + 1; \ \{A\}}{\{ay + b = x \wedge b \geq 0\} \ \text{while } (b \geq y) \text{ do } b := b - y; a := a + 1 \ \{A \wedge \neg(b < y)\}}$$

Per sviluppare la dimostrazione di $\{A \wedge b \geq y\} \ b := b - y; a := a + 1; \ \{A\}$, possiamo usare una strategia analoga a quella vista sopra.

Dobbiamo trovare una “condizione ponte” per poter utilizzare la regola della concatenazione. Ci conviene partire dalla destra e applicare la regola dell’assegnamento per trovare la weakest precondition.

Otteniamo quindi:

$$\{(a+1)y + b = x \wedge b >= 0 \wedge b < y\} \quad a := a + 1; \quad \{A\}$$

Usiamo la pre-condizione appena trovata come post-condizione per applicare la regola dell'assign su $b := b - y$:

$$\{(a+1)y + (b-y) = x \wedge b - y >= 0 \wedge b < y\} \quad b := b - y; \quad \{(a+1)y + b = x \wedge b \geq 0\}$$

Abbiamo in questo modo trovato una condizione che faccia da “ponte” tra i due assegnamenti e ci permetta di utilizzare la regola della concatenazione.

Ma notiamo che, anche in questo caso, la nostra premessa $\{(a+1)y + (b-y) = x \wedge b - y >= 0 \wedge b < y\}$ non coincide con $A \wedge b \geq y$, pre-condizione da cui partiamo per la regola del while. Applichiamo quindi anche in questo caso lo strengthening:

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

(ovvero $(ay + b = x \wedge b \geq y) \supset (ay + b = x \wedge b - y \geq 0)$, evidentemente vero nell’aritmetica).

(Il tutto, in un unico albero di inferenza:

$$\frac{\frac{\frac{\frac{\{(a+1)y + (b-y) = x \wedge (b-y) \geq 0\} \quad b := b - y \quad \{(a+1)y + b = x \wedge b \geq 0\}}{\text{[assign]}} \quad \frac{\{(a+1)y + b = x \wedge b \geq 0\} \quad a := a + 1 \quad \{A\}}{\text{[assign]}}}{\text{[comp]}}}{\frac{\{(a+1)y + (b-y) = x \wedge b - y \geq 0\} \quad b := b - y; \quad a := a + 1 \quad \{A\}}{\text{[math(str implicito)]}}}{\frac{\{ay + b = x \wedge b \geq y\} \quad b := b - y; \quad a := a + 1 \quad \{A\}}{\text{[while]}}}}{\{A\} \text{ while } (b \geq y) \text{ do } b := b - y; \quad a := a + 1 \quad \{A \wedge \neg(b \geq y)\}}$$

)

5.3. Correttezza nei linguaggi funzionali

Per dimostrare la correttezza dei programmi funzionali, adottiamo una **logica equazionale**.

5.3.1. Omomorfismi e operatori di ricorsione

Prima di definire le regole di questa logica, ci è utile re-introdurre una proposizione trattata all’inizio del corso (“omomorfismo tra algebre con la stessa segnatura”, p.8).

Prop. 1: Omomorfismi e numeri naturali

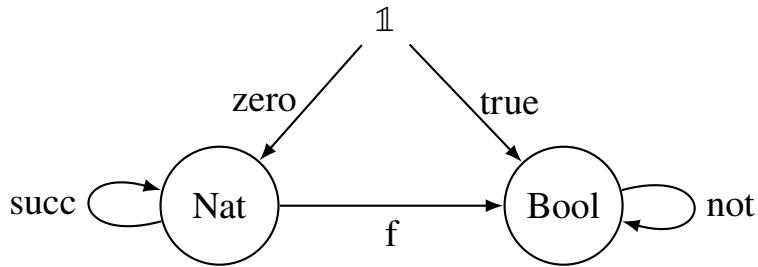
Dati un insieme A , un elemento $a \in A$ ed una funzione $h : A \rightarrow A$, esiste **una sola** $f : \mathbb{N} \rightarrow A$ (omomorfismo) tale che $f(0) = a$ e $f(\text{succ } n) = h(f(n))$

Notiamo che l’algebra fornita non deve essere necessariamente induttiva (è \mathbb{N} ad esserlo).

Prendiamo per esempio $Bool$, l’algebra (non induttiva) dei booleani (definita a p. 7)

Le due algebre \mathbb{N} e $Bool$ hanno la stessa segnatura. Per il teorema, esiste quindi un unico omomorfismo (chiamiamolo f) tra di loro, tale che:

- $f(\text{zero}) = \text{true}$
- $f(\text{succ } x) = \text{not}(f(x))$



Notiamo che questa funzione corrisponde perfettamente alla funzione `is_even` precedentemente definita in SML in questo modo:

```

fun is_even 0 = true
| is_even(succ n) = not(is_even n);
  
```

`is_even` è quindi l'unico omomorfismo tra \mathbb{N} e *Bool*.

Def. 46: Operatore ρ

Indichiamo con ρ la funzione che, dati a e h (“costruttori”) come nella proposizione 1, ci restituisce l’omorfismo f .

Si ha dunque:

- $f = \rho a h$
- $\rho a h \text{ zero} = a$
- $\rho a h (\text{succ } n) = h(\rho a h n)$.

Applicando ricorsivamente queste equazioni, si ottiene:

$$\rho a h n = \rho a h \underbrace{(\text{succ}(\dots(\text{succ } 0)\dots))}_{n \text{ volte}} = h(\dots(h a)\dots)$$

Ovvero, $\rho a h n$ itera n volte h a partire da a . Viene per questo chiamato *iteratore*.

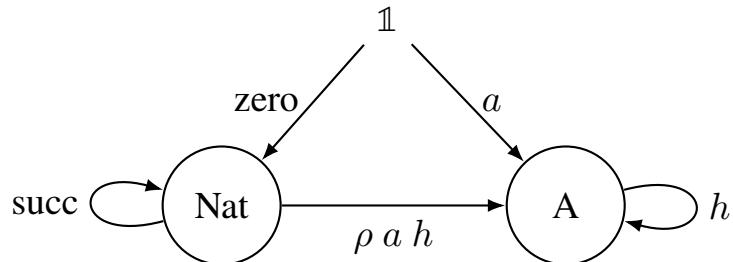
Possiamo anche usare ρ per definire i numeri di Church:

$$c_n h a = \rho a h n$$

(visto che un numero di Church è definito come un applicare n volte h a partire da a)

Quindi, essenzialmente, ρ prende gli equivalenti di “zero” e “succ” e restituisce l’omomorfismo da \mathbb{N} all’insieme che è definito dai costruttori forniti.

Si ha quindi questa situazione:



Definiamo quindi un primo linguaggio funzionale che contenga quanto introdotto.

Def. 47: Fun_ρ

La grammatica di questo linguaggio è:

$$M, N ::= \text{zero} \mid x \mid \text{fn } x \Rightarrow M \mid MN \mid \text{succ}(M) \mid \rho(M, N)$$

e deve essere vero che:

- $\rho(M, N)\text{zero} = M$

(l'omomorfismo, applicato su *zero*, deve portare al costruttore unario dell'altra algebra, ovvero *M* (come l'elemento neutro))

- $\rho(M, N)(\text{succ } L) = N(\rho(M, N)L)$

(l'omomorfismo, applicato su *succ L*, deve portare all'operazione di *A* applicata su *L*, ma nell'altro insieme, ovvero $\rho(M, N)L$)

Possiamo, per esempio, usare i costrutti di questo linguaggio per definire *is_even* tramite ρ .

$$\text{is_even} \equiv \rho \text{ true not}$$

Notiamo però che questo linguaggio non è ancora perfetto.

Consideriamo per esempio l'algebra induttiva delle liste finite di numeri naturali definita dai costruttori *empty* : $1 \rightarrow N - \text{List}$ e *cons* : $(N - \text{List} \times \mathbb{N}) \rightarrow N - \text{List}$ (definita a come a p.6).

Possiamo definirla in SML in questo modo:

```
datatype N-list = empty of Unit
  | cons of (N-list * N);
```

Introduciamo anche una funzione *nicelist*:

```
fun nicelist(0) = empty()
  | nicelist(succ n) = cons(nicelist n, succ n);
```

Per capire meglio come funzionino i costruttori ricorsivi, ci può essere utile simulare il costruttore *cons*.

Vediamo cosa succede se costruiamo *nicelist(succ 2)*

nicelist(succ 2) = cons(nicelist 2, succ 2)

$$\langle 3, \underbrace{}_{\text{nicelist 2}} \rangle = \langle 3, 2, 1 \rangle$$

$$\langle 2, \underbrace{}_{\text{nicelist 1}} \rangle$$

$$\langle 1, \underbrace{}_{\text{nicelist 0}} \rangle$$

$$\text{nicelist 0} = \emptyset$$

Contrariamente a quanto avveniva prima, per calcolare *nicelist(succ n)* non ci basta *nicelist(n)*, ma ci serve anche *n* stesso (l'equivalente di *succ* non è più una endofunzione, prende anche da un altro dominio).

Dobbiamo quindi introdurre una nuova versione della Proposizione 1.

Prop. 2: Omomorfismi e numeri naturali (v. 2)

Dati un insieme A , un elemento $a \in A$ e una funzione $h : A \rightarrow (\mathbb{N} \rightarrow A)$, esiste una sola funzione $f : \mathbb{N} \rightarrow A$ tale che $f(0) = a$ e $f(succ n) = h(f(n), n)$.

Definiamo quindi un equivalente di ρ che restituisca questo omomorfismo. Lo chiamiamo *rec*.

Def. 48: Operatore *rec*

L'operatore *rec*, dato un elemento $a \in A$ e una funzione $h : A \rightarrow (\mathbb{N} \rightarrow A)$, restituisce l'unico omomorfismo $\mathbb{N} \rightarrow A$.

L'operatore *rec* appartiene al *System-T* di Gödel.

Introduciamo quindi un linguaggio funzionale che contenga *rec*.

Def. 49: Fun_{rec}

La grammatica di questo linguaggio è:

$$M, N ::= zero \mid x \mid fn\ x \Rightarrow M \mid MN \mid succ(M) \mid rec(M, N)$$

dove:

- $rec\ MN\ zero = M$
- $rec\ MN\ succ(L) = N((rec\ M, N)L)L = h(f(n), n)$

Costruire funzioni con *rec*

Possiamo definire funzioni a noi note tramite *rec*:

$$is_even \equiv rec(true, fn\ yz \Rightarrow not\ y)$$

$$fact \equiv rec(1, fn\ yz \Rightarrow y * (succ\ z))$$

$$plus \equiv fn\ xy \Rightarrow rec(y, fn\ wz \Rightarrow succ\ w)x$$

Per costruire una funzione $f(n)$ tramite $rec(M, N)$, si può seguire uno schema di questo tipo:

- Caso Base (M):
ci si chiede quanto valga la funzione quando $n = 0$

$$M = f(0)$$

- Passo Induttivo (N):
la funzione N descrive come passare dal passo n al passo $n+1$. Essa accetta sempre due argomenti:
 $fn\ y\ z \Rightarrow \dots$
 - y : è il *risultato già calcolato* al passo precedente (ovvero $f(n)$);
 - z : è il “contatore” al passo precedente (ovvero n).

Il corpo della funzione deve rispondere alla domanda: “Come ottengo il risultato per $n+1$ avendo a disposizione il risultato di n e il numero n ?“

N.B.

Non bisogna necessariamente utilizzare entrambi i parametri di fn - se il risultato $f(n + 1)$ dipende solo da $f(n)$ (e non da n), si può semplicemente ignorare il secondo parametro (rec si può comportare come ρ)

Esempio: costruiamo la funzione fattoriale $fact(x) = x!$ utilizzando il metodo appena descritto.

- Caso Base ($x = 0$)

sappiamo che $0! = 1$.

$$M = 1$$

- Passo Induttivo ($n + 1$)

vogliamo calcolare $fact(n + 1)$, avendo a disposizione

- y : il risultato precedente ($n!$)
- z : il contatore precedente (n)

La formula matematica è:

$$(n + 1)! = \underbrace{(n + 1)}_{\text{numero corrente}} \cdot \underbrace{n!}_{\text{risultato prec.}}$$

Quindi la funzione N è:

$$N \equiv fn y z \Rightarrow y * succ(z)$$

- Definizione con rec :

assemblando M e N :

$$fact \equiv rec(1, fn y z \Rightarrow y * succ(z))$$

Simulazione rec

Per capire ancora meglio come funzioni rec , facciamo una simulazione su $plus$.

Proviamo a calcolare $plus(2, 3)$, dove $x = 2$ e $y = 3$.

Sia $R = rec(3, fn w z \Rightarrow succ(w))$ (l'omomorfismo).

Dunque $plus(x, y) = R(x)$.

$$\text{Abbiamo } R(x) = \begin{cases} y & \text{se } x = zero \\ succ(R(L)) & \text{se } x = succ(L) \end{cases}$$

Simuliamo quindi una possibile esecuzione:

$$\begin{aligned} plus(2, 3) &\equiv R(2) \\ &\equiv R(succ(1)) && 2 = succ(1) \\ &\equiv succ(R(1)) && x = 2 = succ(1) \text{ quindi } R(x) = succ(R(1)) \\ &\equiv succ(R(succ(0))) && 1 = succ(0) \\ &\equiv succ(succ(R(0))) && \text{stesso passo ricorsivo (succ}(R(0))) \\ &\equiv succ(succ(3)) && \text{caso base } M: rec M N zero = M = 3. \\ &\equiv succ(4) \\ &\equiv 5 \end{aligned}$$

5.3.2. Logica equazionale

Introduciamo ora le regole di un sistema logico per la verifica formale dei programmi di Fun_{rec} . Si tratta di un sistema equazionale, che ha come unico predicato l'uguaglianza: $M = N$.

Def. 50: Regole di inferenza per i programmi

- regola di α -equivalenza:

$$\frac{}{fn\ x \Rightarrow M = fn\ y \Rightarrow [y/x]M} [\alpha]$$

- regola β (come nel lambda calcolo):

$$\frac{}{(fn\ x \Rightarrow M)N = [N/x]M} [\beta]$$

- regola di ricorsione (caso base):

$$\frac{}{rec(M, N)0 = M} [rec_0]$$

- regola di ricorsione (passo induttivo):

$$\frac{}{rec(M, N)(succ\ L) = N(rec(M, N)L)L} [rec_{succ}]$$

- principio di induzione:

$$\frac{P(0) \quad P(x) \Rightarrow P(succ\ x)}{\forall n.P(n)} [Ind]$$

- regola di congruenza:

$$\frac{M = N \quad M = N'}{N = N'} [Cong]$$

- regola del contesto:

$$\frac{M = M' \quad N = N'}{MN = M'N'} [Cont]$$

- regola di λ -astrazione:

$$\frac{M = N}{fn\ x \Rightarrow M = fn\ x \Rightarrow N} [\lambda_{Astr}/\xi]$$

Nota bene: quando applichiamo l'operatore di sostituzione, dobbiamo fare attenzione. Per esempio, se applicassimo $[y/x]fn\ y \Rightarrow x$ senza fare attenzione, otterremmo $fn\ y \Rightarrow y$, che ci permetterebbe di dimostrare affermazioni false.

Se per esempio abbiamo

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

sostituendo “bovinamente”, l’ultima y , che è legata al fn più esterno, entra nello scope del fn più interno (si dice che “viene catturata”).

Per evitarlo, ci è utile la $\alpha - regola$. Possiamo assumere che l’operatore di sostituzione rinomini eventuali variabili legate per evitarne la cattura.

Per esempio, visto che $fn\ y \Rightarrow x$ è α -equivalente a $fn\ z \Rightarrow x$, otteniamo:

$$(fn\ x \Rightarrow fn\ y \Rightarrow x)y = [y/x]fn\ y \Rightarrow x = [y/x]fn\ z \Rightarrow x = fn\ z \Rightarrow y$$

Il predicato di uguaglianza $M = N$ rappresenta una **relazione di equivalenza**, in quanto gode delle tre proprietà

di:

- riflessività:

$$\frac{M}{M = M}$$

ottenuta da:

$$\frac{\frac{M}{(rec\ M\ N)\ 0 = M} \quad (rec\ M\ N)\ 0 = M}{M = M}$$

- simmetria:

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

ottenuta da:

$$\frac{\frac{\frac{M = N}{M = N} \quad M}{[rifl] \frac{M = N \quad M}{M = M}} \quad [cong] \frac{M = N \quad M = M}{N = M}}{N = M}$$

- transitività:

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

ottenuta da:

$$\frac{\frac{[simm] \frac{M = N \quad N = L}{N = M \quad N = L}}{[cong] \frac{N = M \quad N = L}{M = L}}}{M = L}$$

5.3.2.1. Correttezza di plus

Def. 51: plus

La funzione *plus*

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

può essere definita tramite *rec* in questo modo:

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

Vogliamo dimostrare che l'operatore *plus* gode della proprietà commutativa.

Lemma 4: commutatività

Si ha che $\text{plus } M\ N \equiv \text{plus } N\ M$

Procediamo per induzione su n per dimostrare $P(n) : \forall m. \text{plus } m\ n = \text{plus } n\ m$

- (1) **C.B.** ($n = 0$):

Sia $Q(m) = \text{plus } 0\ m = \text{plus } m\ 0$.

- $Q(0)$ è valido per identità:

$$\text{plus } 0\ 0 = \text{plus } 0\ 0$$

- assumiamo $Q(y)$, e verifichiamo $Q(\text{succ } y)$

$$\begin{aligned} \text{plus}(\text{succ } y) 0 &= \text{succ}(\text{plus } y 0) && \text{def} \\ &= \text{succ}(\text{plus } 0 y) && \text{ip. ind.} \\ &= \text{succ } y \\ &= \text{plus } 0 (\text{succ } y) \end{aligned}$$

Quindi, per la regola di induzione:

$$\frac{Q(0) \quad Q(y) \Rightarrow Q(\text{succ } y)}{\frac{\forall x Q(x)}{P(0)}}$$

(2) P.I.:

Assumiamo $P(n) : \text{plus } n m \equiv \text{plus } m n$.

Dobbiamo dimostrare $P(\text{succ } n) : \text{plus}(\text{succ } n) m \equiv \text{plus } m (\text{succ } n)$.

Definiamo il predicato ausiliario $R(x) := [\text{plus } (\text{succ } n) x = \text{plus } x (\text{succ } n)]$.

Per induzione su x :

- Base $R(0)$:**

$$\begin{aligned} \text{plus } 0 (\text{succ } n) &= \text{succ } n && \text{(def)} \\ &= \text{succ } (\text{plus } 0 n) && \text{(def)} \\ &= \text{succ } (\text{plus } n 0) && (P(n) \text{ con } m = 0) \\ &= \text{plus } (\text{succ } n) 0 && \text{(def inversa)} \end{aligned}$$

- Passo $R(y) \Rightarrow R(\text{succ } y)$:**

Assumiamo $R(y) : \text{plus } (\text{succ } n) y = \text{plus } y (\text{succ } n)$.

Sviluppiamo il lato sinistro:

$$\begin{aligned} \text{plus } (\text{succ } y) (\text{succ } n) &= \text{succ } (\text{plus } y (\text{succ } n)) && \text{(def)} \\ &= \text{succ } (\text{plus } (\text{succ } n) y) && (R(y)) \\ &= \text{succ } (\text{succ } (\text{plus } n y)) && \text{(def)} \\ &= \text{succ } (\text{succ } (\text{plus } y n)) && (P(n) \text{ con } m = y) \\ &= \text{succ } (\text{plus } (\text{succ } y) n) && \text{(def inversa)} \\ &= \text{succ } (\text{plus } n (\text{succ } y)) && (P(n) \text{ con } m = \text{succ } y) \\ &= \text{plus } (\text{succ } n) (\text{succ } y) && \text{(def inversa)} \end{aligned}$$

Per la regola di induzione:

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

Abbiamo quindi dimostrato $P(\text{succ } n)$.

Per il principio di induzione, otteniamo quindi:

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

Abbiamo quindi che $\text{plus } M N \equiv \text{plus } N M$. □

5.3.2.2. Correttezza di twice

Def. 52: `twice`

La funzione `twice`

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

può essere definita tramite `rec` in questo modo:

$$\text{twice} \equiv \text{rec}(0, fn\ y\ z \Rightarrow \text{succ}(\text{succ } z))$$

Lemma 5: `twice` e `plus`

Si ha:

$$\text{twice } n \equiv \text{plus } n\ n$$

(1) **C.B.** ($n = 0$):

$$\underbrace{\text{twice } 0}_{=0} \equiv \underbrace{\text{plus } 0\ 0}_{=0}$$

(2) **P.I.** Supponiamo $\text{twice } n \equiv \text{plus } n\ n$.

Dimostriamo

$$\text{twice}(\text{succ } n) \equiv \text{plus}(\text{succ } n)(\text{succ } n)$$

$$\begin{aligned} \text{twice } (\text{succ } n) &= \text{succ}(\text{succ}(\text{twice } n)) && \text{(def)} \\ &= \text{succ}(\text{succ}(\text{plus } n\ n)) && \text{(I.I.)} \\ &= \text{succ}(\text{plus } (\text{succ } n)\ n) && \text{(def plus)} \\ &= \text{succ}(\text{plus } n\ (\text{succ } n)) && \text{(commutatività)} \\ &= \text{plus } (\text{succ } n)\ (\text{succ } n) && \text{(def. plus)} \end{aligned}$$

□

Un **sistema dei tipi** per un linguaggio di programmazione è un insieme di regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti del linguaggio.

Un linguaggio si dice *tipato* se per esso è definito un tale sistema, e *non tipato* altrimenti. Un linguaggio si dice *fortemente tipato* se il tipo di tutte le variabili è determinato a tempo di compilazione, e *dinamicamente tipato* se è determinato a tempo di esecuzione.

Lo scopo di un sistema dei tipi è evitare che durante l'esecuzione del programma occorrono errori legati ai tipi di dato.

La definizione di una **teoria dei tipi** (ovvero un sistema di tipi di dato sotto forma di regole formali di deduzione) permette, se si definisce anche una semantica formale, di dimostrare matematicamente proprietà dinamiche di un linguaggio.

6.1. F1: il Lambda Calcolo tipato semplice

F1 è un sistema di *primo ordine*, ovvero senza parametrizzazione o astrazione sui tipi.

Def. 53: **Grammatica di *F1***

$$\text{Types} \ni A, B ::= K \mid A \rightarrow B$$

$$\text{Terms} \ni M, N ::= k \mid x \mid fn\ x : A \Rightarrow B \mid M\ N$$

dove:

- K indica un generico **tipo costante** o tipo base (*int*, *bool*...)
- Il costruttore di **tipo funzionale** $A \rightarrow B$ indica l'insieme delle funzioni che accettano un argomento di tipo A (dominio) e restituiscono un risultato di tipo B (codominio).
 - è associativo a destra: $A \rightarrow B \rightarrow C$ corrisponde a $A \rightarrow (B \rightarrow C)$
- $Terms$ è l'insieme dei “termini grezzi” del linguaggio (grezzi perché non necessariamente tipabili)
- k indica una costante di tipo K
- $x \in Var$ indica una variabile

Def. 54: Contesti

Nel lambda calcolo tipato semplice, definiamo come **insieme dei contesti** l'insieme delle funzioni parziali che associano ogni variabile al suo tipo:

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

Una generica metavariabile Γ indica un generico **contesto dei tipi** (quindi una mappa), esprimibile tramite una lista ordinata $x_1 : \Gamma(x_1) = A_1, x_2 : A_2, \dots, x_n : A_n$ di variabili x_i distinte, associate al loro tipo A_i secondo Γ .

I contesti si possono concatenare analogamente agli ambienti e alle memorie.

Def. 55: Asserzione di tipi (judgements)

Il sistema $F1$ permette di dedurre **asserzioni di tipo**, ovvero clausole del tipo:

$$\Gamma \vdash M : A$$

(si legge: “nel contesto Γ , il termine M ha tipo A ”)

all'interno della *semantica di tipi*, indicata con il simbolo “ $:$ ”, definita come:

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

Def. 56: Regole di inferenza

- **Costanti:**

$$\overline{\Gamma \vdash k : K}$$

- **Variabili:**

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

- **Funzione:**

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash fn x : A \Rightarrow M : A \rightarrow B}$$

per dimostrare che una funzione accetta A e restituisce B (ha tipo $A \rightarrow B$), devo analizzare il suo corpo M - poiché M contiene la variabile x , “suppongo” che x sia di tipo A (aggiungo $x : A$ al contesto) e, se sotto questa ipotesi riesco a dimostrare che il corpo M produce un B , allora la funzione ha quel tipo.

- **Applicazione:**

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

Esempio:

Consideriamo il termine

$$(fn\ x : int \rightarrow bool \Rightarrow x\ 5) \ (fn\ y : int \Rightarrow true)$$

Deriviamo il suo tipo in questo modo:

$$\frac{\text{[apply]} \frac{x : int \rightarrow bool \vdash x : \text{int} \rightarrow \text{bool} \quad x : int \rightarrow bool \vdash 5 : \text{int}}{\frac{[fn] \frac{\emptyset \vdash fn\ x : int \rightarrow bool \Rightarrow x\ 5 : (\text{int} \rightarrow \text{bool}) \rightarrow \text{bool}}{(fn\ x : int \rightarrow bool \Rightarrow x\ 5) \ (fn\ y : int \Rightarrow true) : \text{bool}}}{y : int \vdash true : \text{bool}} \ [fn]} \ [fn]}$$

Facciamo un altro esempio:

$$\emptyset \vdash fn\ f : int \rightarrow int \Rightarrow (fn\ x : int \Rightarrow f\ x)$$

Deriviamo il suo tipo in questo modo:

$$\frac{\frac{\frac{f : int \rightarrow int, x : int \vdash f : \text{int} \rightarrow \text{int} \quad f : int \rightarrow int, x : int \vdash x : \text{int}}{f : int \rightarrow int, x : int \vdash f\ x : \text{int}} \text{[apply]}}{f : int \rightarrow int \vdash fn\ x : int \Rightarrow f\ x : \text{int} \rightarrow \text{int}} \text{[fn]}}{\emptyset \vdash fn\ f : int \rightarrow int \Rightarrow (fn\ x : int \Rightarrow f\ x) : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})} \text{[fn]}$$

6.1.1. Espressioni non tipabili

Non tutte le espressioni sono però tipabili in $F1$

Prendiamo per esempio l'espressione

$$fn\ x \Rightarrow xx$$

Questa si deriverebbe con albero:

$$\frac{x : A \vdash x : A \rightarrow B \quad x : A \vdash x : A}{\frac{x : A \vdash xx : B}{\emptyset \vdash fn\ x : A \Rightarrow xx : ??}}$$

che richiede che x sia allo stesso momento di tipo A e $A \rightarrow B$, ovvero che $A \equiv A \rightarrow B$.

Un tale tipo (detto “circolare”) non esiste, e il termine non è perciò tipabile in $F1$.

6.2. F2: il Lambda Calcolo polimorfo

Il sistema $F2$ (o System F) estende il lambda calcolo tipato introducendo il polimorfismo parametrico esplicito.

6.2.1. Polimorfismo

Def. 57: Polimorfismo

Il polimorfismo è la possibilità per un'espressione di assumere molteplici tipi in base al contesto considerato.

Il polimorfismo è introdotto dai sistemi di tipo di *secondo ordine* attraverso **variabili di tipo**, che permettono di descrivere in maniera generica i termini.

Consideriamo per esempio il termine:

$$fn\ x \Rightarrow ((x\ 5)((x\ true)false))$$

Notiamo che questo termine non è tipabile in $F1$, in quanto x dovrebbe essere allo stesso tempo:

- una funzione che prende un intero in input
- una funzione che prende un booleano in input
- una funzione che restituisce un'altra funzione $Bool \rightarrow Bool$

All'interno di un sistema dei tipi di secondo ordine, che permette di quantificare universalmente sui tipi, invece, il termine risulta tipabile senza problemi con il tipo $\forall X.(X \rightarrow (Bool \rightarrow Bool))$.

6.2.2. Sistema F2

Def. 58: Grammatica di $F2$

La sintassi prevede due livelli distinti: termini e tipi.

$$M, N ::= k \mid x \mid fn\ x : A \Rightarrow M \mid MN \mid \Lambda X.M \mid MB$$

$$A, B ::= K \mid A \rightarrow B \mid X \mid \forall X.A$$

6.2.3. Semantica e meccanismi di astrazione

Per comprendere a fondo il funzionamento di $F2$, è fondamentale distinguere i due livelli di astrazione introdotti dai sistemi del secondo ordine. Mentre in $F1$ i termini vengono tipati in un contesto che assegna tipi alle variabili, in $F2$ le variabili agiscono da segnaposto non solo per i termini, ma anche per i tipi. Termini e tipi operano quindi su livelli distinti e i simboli \forall e Λ ne gestiscono l'interazione.

6.2.3.1. Il quantificatore universale ($\forall X.A$)

Il simbolo \forall non denota un singolo tipo, bensì una **famiglia di tipi**. Asserire che un termine possiede il tipo $\forall X.A$ equivale a una quantificazione logica universale: significa garantire che la struttura A è valida per *qualsiasi* tipo concreto venga sostituito alla variabile X .

Da un punto di vista semantico, questo implica una proprietà di uniformità (o parametricità): il comportamento del termine non deve dipendere dalle specificità del tipo X . Facciamo alcuni esempi per vedere come X viene usata nel corpo A per capire la relazione tra il tipo generico e il risultato:

- $\forall X.X \rightarrow X$

output dipendente da X

il tipo di ritorno varia in base all'istanza. Se istanziamo X con *int*, la funzione restituisce un *int*. Se lo istanziamo con *bool*, restituisce un *bool* (l'output "segue" il tipo di input)

- $\forall X.X \rightarrow \text{bool}$

output fisso, input variabile

in questo caso, la funzione accetta un argomento di tipo qualsiasi (X), ma restituisce sempre un booleano (ad esempio, una funzione che scarta l'argomento e restituisce sempre *true*: $fn\ x \Rightarrow \text{true}$)

- $\forall X.\text{int}$

completa indipendenza

qui la variabile X non appare nel tipo del risultato - parametrizziamo su un X , e possiamo passare un tipo solo per soddisfare il quantificatore \forall , ma il termine lo ignora e restituisce un valore fisso (esempio: $M = \Lambda X.42$ (vale 42))

6.2.3.2. L'astrazione sui tipi ($\Lambda X.M$)

Il simbolo Λ è l'operatore costruttivo che realizza la quantificazione universale. Utilizziamo la sintassi $\Lambda X.M$ per **dichiarare la variabile** di tipo X all'interno dell'espressione M .

Possiamo interpretare questo costrutto in due modi complementari:

- *come definizione (definition)*: è il meccanismo che produce un termine di tipo $\forall X.A$. Il corpo M è definito in modo parametrico rispetto a X e il calcolo è sospeso fintanto che non viene fornito un tipo concreto.
- *come funzione (Type \rightarrow Term)*: mentre l'astrazione classica (*fn*) mappa valori in valori, l'astrazione Λ definisce una funzione che *mappa tipi in termini*.

6.2.3.3. L'Istanziazione dei tipi (Specialization)

Per utilizzare un'espressione polimorfa definita tramite Λ , è necessario prima **istanziarne** (o specializzarne) il tipo. La sintassi dell'istanziazione è MB (dove M è il termine polimorfo e B è il tipo specifico).

Questo passaggio è cruciale per l'esecuzione:

- (1) elimina il quantificatore \forall dal tipo
- (2) sostituisce tutte le occorrenze della variabile di tipo X con il tipo concreto A nel corpo del termine ($M[A/X]$)
- (3) produce un termine specializzato pronto per essere applicato a un valore

Esempio: L'Identità Polimorfa

L'oggetto matematico "identità polimorfa" si definisce così:

$$ID_{poly} = \Lambda X.(fn\ x : X \Rightarrow x)$$

Il suo tipo è $\forall X.(X \rightarrow X)$.

Se vogliamo usare questa funzione per restituire l'intero 5, dobbiamo procedere in due passaggi:

- (1) *Istanziazione del tipo (Type Application)*

Passiamo il tipo int alla Λ . La X diventa int .

$$(\Lambda X. fn\ x : X \Rightarrow x) \ int \longrightarrow_{\beta} (fn\ x : int \Rightarrow x)$$

Ora abbiamo ottenuto una normale funzione identità sugli interi (come in $F1$).

(2) Applicazione del termine (Term Application)

Ora possiamo passare il valore 5 alla funzione fn .

$$(fn\ x : int \Rightarrow x) \ 5 \longrightarrow_{\beta} 5$$

L'espressione completa è quindi:

$$\underbrace{((\Lambda X. fn\ x : X \Rightarrow x) \ int)}_{\text{produce una funzione } int \rightarrow int} \ 5$$

Il comportamento dell'identità è indipendente dal tipo: se proviamo con i booleani, la struttura del calcolo rimarrà identica:

$$\underbrace{((\Lambda X. fn\ x : X \Rightarrow x) \ bool)}_{\text{diventa } bool \rightarrow bool} \ true \longrightarrow_{\beta} true$$

In entrambi i casi, l'astrazione sul tipo ΛX è stata eliminata senza che la funzione dovesse "controllare" cosa fosse X . L'algoritmo ($x \Rightarrow x$) agisce ciecamente sul valore, sia esso un 5 o un $true$.

Def. 59: Variabili libere

La funzione:

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

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

$$\begin{cases} \text{free}(K) = \emptyset \\ \text{free}(X) = \{X\} \\ \text{free}(A \rightarrow B) = \text{free}(A) \cup \text{free}(B) \\ \text{free}(\forall X.A) = \text{free}(A) - \{X\} \\ \text{free}(\Gamma) = \bigcup_{x:A \in \Gamma} \text{free}(A) \end{cases}$$

essenzialmente $X \in \text{free}(\Gamma)$ se non compare in Γ oppure se compare, ma non "legato"; per esempio:

- se nell'ambiente ho $x : X$, quell' X è libero: si riferisce ad un tipo specifico esistente nell'ambiente esterno, ma che non conosciamo: non è definito o specificato
- se ho invece $y : \forall X.X \rightarrow X$, X non è libero: è vincolato da $\forall X.X \rightarrow X$

Def. 60: Regole di inferenza

Oltre a quelle di $F1$, $F2$ introduce le seguenti regole:

- **Generalizzazione dei tipi:**

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

– “top-down”

la regola dice che se il termine M è ben tipato assumendo che X sia un tipo qualsiasi (una variabile libera non vincolata nel contesto Γ), allora la logica di M è valida universalmente.

possiamo quindi astrarre X : usiamo ΛX per trasformare quella variabile libera in un parametro di ingresso

$\Lambda X.M$ non è più un termine che “usa” un tipo X , ma una funzione che “aspetta” un tipo per essere eseguita.

Immaginiamo un termine $M = fn\ y : X \rightarrow y$.

Qui M ha senso solo se sappiamo cos’è X . Applicando la regola $(\Lambda X.M)$, otteniamo $\Lambda X.fn\ y : X \Rightarrow y$: ora il termine è autosufficiente: non dipende più da una definizione esterna di X , perché la definisce lui stesso come parametro.

– “bottom-up” (verifica / type inference)

in una derivazione, per inferire da ΛX il tipo completo $\forall X.A$:

- (1) “spogliamo” il termine della sua astrazione ΛX .
- (2) analizziamo il corpo M trattando la variabile X come se fosse un tipo base disponibile (una costante temporanea)
- (3) troviamo che il corpo M ha un certo tipo A
- (4) il tipo finale del termine originale è $\forall X.$ seguito dal tipo A appena trovato.

■ **Specializzazione del tipo:**

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

permette di istanziare un termine polimorfo con un tipo concreto B (per poterlo fare, dobbiamo dimostrare che M sia effettivamente polimorfo)

Conseguenze della violazione della condizione laterale

La regola $X \notin \text{free}(\Gamma)$ è fondamentale per garantire la coerenza del sistema.

Impedisce di trattare una variabile che ha un tipo specifico (vincolato dal contesto esterno) come se fosse generica.

Supponiamo di avere nel contesto $\Gamma = \{x : X\}$. X è libero, e potrebbe essere qualsiasi cosa (es. `bool` e x un valore concreto (es. `true`)).

Possiamo analizzare le conseguenze di una violazione della regola su due livelli.

■ *a livello “pratico”: rottura della type safety*

se potessimo generalizzare illegalmente su X :

$$\frac{x : X \vdash x : X}{x : X \vdash \Lambda X.x : \forall X.X}$$

Avremmo creato una funzione che promette di convertire x in qualsiasi tipo. Basterebbe specializzarla per esempio con `int`:

$$(\Lambda X.x) \ int : int$$

Il type system crede di avere un intero, ma a runtime il valore è ancora `true`.

■ *livello logico: incoerenza del sistema*

formalmente, questa violazione permette di costruire termini che violano la natura costruttiva del calcolo (es. creare un intero senza avere nessun numero).

Costruiamo il termine $M = \Lambda X. fn\ x : X \Rightarrow (\Lambda X. x) int$.

$$\frac{\frac{\frac{x : X \vdash x : X}{x : X \vdash \Lambda X. x : \forall X. X}}{x : X \vdash (\Lambda X. x) int : int}}{\frac{\emptyset \vdash fn\ x : X \Rightarrow (\Lambda X. x) int : X \rightarrow int}{\emptyset \vdash \Lambda X. fn\ x : X \Rightarrow (\Lambda X. x) int : \forall X. (X \rightarrow int)}}$$

Se ora applichiamo questo termine a se stesso:

$$\frac{\frac{\emptyset \vdash M : \forall X. (X \rightarrow int)}{\emptyset \vdash M (int \rightarrow int) : (int \rightarrow int) \rightarrow int} \quad \frac{\emptyset \vdash M : \forall X. (X \rightarrow int)}{\emptyset \vdash M int : int \rightarrow int}}{\emptyset \vdash M (int \rightarrow int)(M int) : int}$$

Otteniamo un termine che il sistema classifica come intero, nonostante al suo interno non compaia nessuna costante numerica.

Abbiamo dimostrato l'esistenza di un oggetto che non esiste ("costruito un intero di aria fritta"), rendendo la logica del sistema incoerente.

6.2.4. Proprietà del Sistema F2

- **Inferenza indecidibile:** non esiste un algoritmo che, dato un termine non annotato, possa ricostruirne il tipo in F2 (l'utente deve scrivere esplicitamente i tipi)
- **Mancanza di punto fisso:** in F2 puro non è possibile tipare il combinatore di punto fisso (Y), quindi la ricorsione generale non è supportata direttamente.
- Ogni termine di F2 **termina**.

Esempi di inferenza

Deriviamo il tipo del termine: $\Lambda A. \Lambda B. fn\ x : A \Rightarrow fn\ y : B \Rightarrow x$

$$\frac{\frac{\frac{x : A, y : B \vdash x : A}{x : A \vdash fn\ y : B \Rightarrow x : B \rightarrow A} [fn]}{\emptyset \vdash fn\ x : A \Rightarrow fn\ y : B \Rightarrow x : A \rightarrow (B \rightarrow A)} [fn]}{\emptyset \vdash \Lambda B. fn\ x : A \Rightarrow fn\ y : B \Rightarrow x : \forall B. (A \rightarrow B \rightarrow A)} [\text{gen}]}$$

$$\emptyset \vdash \Lambda A. \Lambda B. fn\ x : A \Rightarrow fn\ y : B \Rightarrow x : \forall A. \forall B. A \rightarrow B \rightarrow A$$

Ora, il termine $((\Lambda A. \Lambda B. fn\ x : A \Rightarrow fn\ y : B \Rightarrow x) Nat) Bool$

$$\begin{array}{c}
 \frac{x : A, y : B \vdash x : \textcolor{red}{A}}{x : A \vdash fn\ y : B \Rightarrow x : \textcolor{red}{B \rightarrow A}} \text{ [fn]} \\
 \frac{\emptyset \vdash fn\ x : A \Rightarrow fn\ y : B \Rightarrow x : \textcolor{red}{A \rightarrow (B \rightarrow A)}}{\emptyset \vdash \Lambda B. fn\ x : A \Rightarrow \dots : \forall B. (\textcolor{red}{A \rightarrow B \rightarrow A})} \text{ [gen]} \\
 \frac{\emptyset \vdash \Lambda A. \Lambda B. fn\ x : A \Rightarrow \dots : \forall A. \forall B. \textcolor{red}{A \rightarrow B \rightarrow A}}{\emptyset \vdash (\Lambda A. \Lambda B. \dots) Nat : \forall B. (\textcolor{red}{Nat \rightarrow B \rightarrow Nat})} \text{ [gen]} \\
 \frac{}{\emptyset \vdash ((\Lambda A. \Lambda B. fn\ x : A \Rightarrow fn\ y : B \Rightarrow x) Nat) Bool : \textcolor{red}{Nat \rightarrow Bool \rightarrow Nat}} \text{ [spec]}
 \end{array}$$

auto-applicazione e impredicatività

Come abbiamo visto, in *F1* il termine di auto-applicazione $fn\ x \Rightarrow x\ x$ è impossibile da tipare. Ciò deriva da un vincolo strutturale: se assumiamo che x abbia tipo T , la prima occorrenza di x (in posizione di funzione) deve necessariamente avere un tipo $T \rightarrow R$ per poter accettare la seconda x come argomento. Questo porta all'equazione ricorsiva $T \equiv T \rightarrow R$, che non ammette soluzioni finite in *F1* (richiederebbe tipi ricorsivi infiniti).

In *F2*, invece, l'auto-applicazione è derivabile grazie alla proprietà di impredicatività (capacità di istanziare un quantificatore $\forall X$ con un qualsiasi tipo, incluso il tipo stesso che si sta definendo o utilizzando).

Per tipare l'auto-applicazione, assegniamo a x il tipo polimorfo dell'identità:

$$U = \forall X. X \rightarrow X$$

Ma nel Sistema *F2*, non possiamo applicare direttamente un termine polimorfo a un dato: xx è un'applicazione e la regola delle applicazioni richiede un tipo funzione $A \rightarrow B$, mentre noi abbiamo un tipo quantificato $(\forall X. X \rightarrow X)$ - per poterlo usare come tipo funzione, dobbiamo prima specializzarlo.

Il termine che andremo a derivare è quindi:

$$fn\ x : U \Rightarrow (x\ U)\ x$$

Dove il frammento $(x\ U)$ rappresenta l'istanziazione sintattica della prima x con il tipo U .

Formalmente:

$$\frac{\begin{array}{c} x : \forall X. X \rightarrow X \vdash x : \forall X. X \rightarrow X \\ \hline x : \forall X. X \rightarrow X \vdash x : (\forall X. X \rightarrow X) : (\forall X. X \rightarrow X) \rightarrow (\forall X. X \rightarrow X) \end{array} \text{ [spec]} \quad x : \forall X. X \rightarrow X \vdash x : \forall X. X \rightarrow X}{\begin{array}{c} x : \forall X. X \rightarrow X \vdash (x : (\forall X. X \rightarrow X))\ x : \forall X. X \rightarrow X \\ \hline \emptyset \vdash fn\ x : (\forall X. X \rightarrow X) \Rightarrow (x : (\forall X. X \rightarrow X))\ x : (\forall X. X \rightarrow X) \rightarrow (\forall X. X \rightarrow X) \end{array} \text{ [apply]}}$$

6.3. Fun_τ : Il sistema dei tipi di ML

Il linguaggio Fun_τ rappresenta il formalismo teorico alla base del sistema dei tipi di ML (Standard ML). A differenza del System F (F2), dove i tipi sono completamente esplicati, in ML i tipi vengono spesso omessi e calcolati tramite un algoritmo di inferenza. Dal punto di vista della potenza espressiva, Fun_τ si colloca tra il lambda calcolo tipato semplice (F1) e il sistema F2, introducendo una forma limitata di polimorfismo nota come **sistema di Hindley-Milner**.

6.3.1. Monotipi e Politipi

La caratteristica fondamentale che distingue Fun_τ da F2 è la stratificazione dei tipi in due categorie distinte. Questa restrizione è necessaria per garantire l'inferenza automatica dei tipi.

Def. 61: Grammatica dei Tipi

In Fun_τ , i tipi sono divisi in due livelli gerarchici:

- **Tipi Primitivi** (τ , monotipi):

Non contengono quantificatori universali (\forall). Rappresentano i tipi concreti o variabili di tipi semplici.

$$\tau ::= K \mid X \mid \tau_1 \rightarrow \tau_2$$

dove K sono costanti (es. *int*, *bool*) e X variabili di tipo.

- **Schemi di Tipo** (σ , politipi):

Possono contenere quantificatori, ma *solo al livello più esterno* (Forma Normale Prenessa).

$$\sigma ::= \tau \mid \forall X. \sigma$$

Gli schemi rappresentano tipi polimorfi generici.

Restrizione del Polimorfismo

Questa grammatica implica che il costruttore di funzione \rightarrow può collegare solo tipi primitivi τ , mai schemi σ .

- Il tipo $(\forall X.X) \rightarrow (\forall X.X)$ è *legale in F2 ma non in ML*.
- In Fun_τ non esistono funzioni che prendono in input oggetti polimorfi; l'astrazione è possibile solo sui tipi primitivi

Def. 62: Grammatica dei Termini

I termini di Fun_τ sono del tipo:

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

6.3.2. Istanza Generica

Il meccanismo che permette di utilizzare uno schema polimorfo in contesti concreti è la relazione di *specializzazione* o *istanza generica*.

Def. 63: Istanza generica (\succ)

Uno schema $\sigma' = \forall Y_1 \dots Y_m. \tau'$ è un'**istanza generica** di $\sigma = \forall X_1 \dots X_n. \tau$, scritto $\sigma \succ \sigma'$ (oppure $\sigma' \sqsubseteq \sigma$), se:

$$\tau' = [\tau_1, \dots, \tau_n / X_1, \dots, X_n] \tau$$

dove le τ_i sono tipi primitivi e nessuna variabile Y_j appare libera in σ .

(possiamo ottenere σ' sostituendo le variabili X_i (solo quelle quantificate da un $\forall X_i$) con dei monotipi)

I quantificatori $\forall Y_1 \dots Y_m$ in σ' sono necessari per chiudere lo schema: se la sostituzione introduce nuove variabili di tipo (contenute nei τ_i), queste risulterebbero libere in τ' ; la regola impone quindi di quantificarle tutte (e sole quelle) per mantenere lo schema chiuso (vedi esempio sotto).

Esempi di specializzazione:

- dato $\sigma = \forall X. X \rightarrow X$, il tipo $int \rightarrow int$ è sua istanza generica ($\sigma \succ int \rightarrow int$).
- dato $\sigma = \forall X, Y. X \rightarrow Y$, lo schema $\forall Z. (int \rightarrow Z) \rightarrow bool$ è sua istanza generica.
 - ↪ abbiamo sostituito X con il tipo ($int \rightarrow Z$)
 - ↪ la variabile Z è nuova ed è libera dopo la sostituzione
 - ↪ dobbiamo aggiungere $\forall Z$ in testa per chiudere lo schema
- *non valido:* Se $\sigma = \forall X. Y \rightarrow X$, allora $\forall Y. (int \rightarrow Y) \rightarrow X$ non è istanza generica perché Y è libera in σ e non può essere catturata dalla quantificazione (violazione della condizione laterale).

cattura di variabile

$$\sigma = \forall X. (Y \rightarrow X) \quad \succ^{??} \quad \sigma' = \forall Y. ((int \rightarrow Y) \rightarrow \dots)$$

Nello schema originale $\forall X. (Y \rightarrow X)$, la variabile Y è libera - è fissata dal contesto esterno.

La formula originale dice: “dato quel preciso tipo Y (es. $bool$), restituisco una funzione che per ogni X ...”

Nello schema σ' , abbiamo aggiunto un quantificatore $\forall Y$ - diciamo quindi “per qualsiasi tipo Y ...”

Se permettessimo questo passaggio, la Y che prima indicava un tipo specifico verrebbe “catturata” dal nuovo $\forall Y$ e diventerebbe generica.

6.3.3. Regole di Inferenza di Fun_τ

Il sistema di tipi utilizza giudizi della forma $\Gamma \vdash M : \sigma$. Oltre alle regole standard per variabili e costanti, Fun_τ introduce regole specifiche per gestire la generalizzazione e specializzazione.

Def. 64: Regole standard

- **Costanti:**

$$\overline{\Gamma \vdash k : K}$$

- **Variabili:**

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

■ **Funzione:**

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

- non sappiamo ancora che tipo abbia x , ma assumiamo che sia un monotipo τ
- sotto questa ipotesi, analizziamo M e determiniamo che tipo τ' ritorni (il tipo del fn sarà $\tau \rightarrow \tau'$)
- NB: la regola usa τ (tipo monomorfo): gli argomenti di una funzione non possono essere polimorfi

■ **Applicazione:**

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

Def. 65: Regole del Polimorfismo

■ **Generalizzazione (Gen):**

Permette di introdurre il polimorfismo astraendo su una variabile di tipo X , a patto che non sia vincolata nel contesto.

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

■ **Specializzazione (Spec):**

Permette di utilizzare un termine polimorfo come se avesse un tipo più specifico (istanza generica).

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

In ML questa regola è implicita: l'interprete specializza automaticamente il tipo quando necessario per far combaciare i tipi nell'applicazione.

6.3.4. Il Let-Polimorfismo

Mentre le funzioni accettano solo argomenti monomorfi, il costrutto `let` permette di legare una variabile a uno schema polimorfo.

Def. 66: Regola Let-in

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

Nota: La variabile x viene inserita nel contesto per la valutazione del corpo N con tipo σ (schema), e non τ (primitivo).

6.3.4.1. Let vs Fn

Sebbene operazionalmente `let x = M in N` possa sembrare simile a `(fn x => N) M`, il sistema dei tipi li tratta in modo profondamente diverso.

Consideriamo il termine:

$$\text{let } x = (\text{fn } y \Rightarrow y) \text{ in } (x\ 5, \ x\ \text{true})$$

- (1) L'identità $\text{fn } y \Rightarrow y$ ha tipo $\forall Y.Y \rightarrow Y$.
- (2) Grazie alla regola `let`, x nel corpo ha questo schema polimorfo.
- (3) Nella prima occorrenza ($x\ 5$), x si specializza in $\text{int} \rightarrow \text{int}$.
- (4) Nella seconda occorrenza ($x\ \text{true}$), x si specializza in $\text{bool} \rightarrow \text{bool}$.

Al contrario, il termine equivalente con lambda:

$$(\text{fn } x \Rightarrow (x\ 5, \ x\ \text{true}))(\text{fn } y \Rightarrow y)$$

Non è tipabile in Fun_τ . La regola di astrazione (fn) obbligherebbe il parametro formale x ad avere un unico tipo primitivo τ (dovrebbe essere contemporaneamente $\text{int} \rightarrow \dots$ e $\text{bool} \rightarrow \dots$).

Perché il Let supporta il polimorfismo e la Fn no?

La distinzione nasce da un compromesso fondamentale tra potenza espressiva e decidibilità dell'inferenza.

- costrutto `let` - conosciamo il valore !!

In `let x = M in N`, il compilatore ha accesso immediato al termine M .

- il sistema analizza M prima di guardare N .
- calcola il tipo di M , vede che è generico e lo generalizza in uno schema σ (es. $\forall \alpha. \alpha \rightarrow \alpha$)
- quindi, ogni volta che x compare nel corpo N , si applica la regola della specializzazione

- costrutto `fn` - non conosciamo il valore

In `fn x => N`, stiamo definendo una funzione che verrà chiamata da qualcun altro in futuro.

- il compilatore non sa cosa verrà passato al posto di x .
- per poter inferire automaticamente i tipi, il sistema assume che x abbia un solo tipo concreto τ (monotipo).
- all'interno del corpo N , tutte le occorrenze di x devono essere coerenti con questo unico tipo τ .

Nota storica: se permettessimo il polimorfismo anche negli argomenti di funzione (polimorfismo di rango superiore), l'inferenza dei tipi diventerebbe indecidibile (dimostrato da Wells, 1994). ML sceglie di limitare la `fn` per garantire che l'inferenza funzioni sempre.

Isomorfismo di Curry-Howard

L'isomorfismo di Curry-Howard stabilisce una corrispondenza tra la logica formale e la teoria dei tipi. Esso rivela che le regole di derivazione logica e le regole di tipizzazione del lambda calcolo non sono semplicemente simili, ma sono sostanzialmente la stessa cosa vista sotto due lenti diverse.

7.0.1. Corrispondenza tra Regole

Per intuire l'isomorfismo, osserviamo il parallelismo tra le regole fondamentali della logica intuizionista e quelle del lambda calcolo tipato.

7.0.1.1. Modus Ponens e Applicazione

In logica, la regola del Modus Ponens afferma che se A implica B ed A è vero, allora B è vero. Nei tipi, la regola di *Applicazione* afferma che se abbiamo una funzione da A a B e un argomento di tipo A , otteniamo un risultato di tipo B .

| Logica (Modus Ponens) | Tipi (Applicazione) |
|---|--|
| $\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$ | $\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$ |

7.0.1.2. Deduzione e Astrazione

Analogamente, il teorema di deduzione corrisponde alla creazione di una funzione (*Astrazione*). Se assumendo l'ipotesi A possiamo dimostrare B , allora anche l'implicazione $A \rightarrow B$ è dimostrabile. Parallelamente, se in un contesto con una variabile $x : A$ possiamo costruire un termine $M : B$, allora esiste la funzione $A \rightarrow B$.

| Logica (teo. di Deduzione) | Tipi (Astrazione) |
|--|---|
| $\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$ | $\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{fn } x \Rightarrow M : A \rightarrow B}$ |

7.0.2. Tipi e programmi, proposizioni e dimostrazioni

L'isomorfismo si articola su due livelli: la corrispondenza tra le formule (proposizioni/tipi) e quella tra i meccanismi di prova (deduzioni/programmi).

Def. 67: Proposizioni come Tipi

Esiste una mappatura diretta tra i connettivi logici e i costruttori di tipo:

| Logica Proposizionale | Sistema dei Tipi |
|------------------------------------|--|
| Implicazione ($A \rightarrow B$) | Tipo Funzione ($A \rightarrow B$) |
| Congiunzione ($A \wedge B$) | Tipo Prodotto ($A \times B$) |
| Disgiunzione ($A \vee B$) | Unione Disgiunta ($A + B$) |
| Vero (<i>True</i>) | Tipo Unitario (1 o <i>Unit</i>) |
| Falso (<i>False</i>) | Tipo Vuoto (\emptyset o <i>Void</i>) |

Def. 68: Dimostrazioni come Programmi

Esiste una corrispondenza tra la struttura di una dimostrazione logica e la struttura di un programma:

| Deduzione Naturale | Lambda Calcolo Tipato |
|--------------------------------|-------------------------------|
| Ipotesi | Variabili libere |
| Eliminazione dell'implicazione | Applicazione di funzione |
| Introduzione dell'implicazione | Astrazione (def. di funzione) |
| Dimostrazione completa | Termine chiuso (programma) |

7.0.2.1. Dimostrazioni come Programmi

Una conseguenza fondamentale di questo isomorfismo è la possibilità di verificare la validità logica attraverso la programmazione.

Validità Logica e Abitabilità

Dato un tipo A e la sua proposizione corrispondente φ_A :

Esiste un termine chiuso M tale che $\vdash M : A \iff \varphi_A$ è una tautologia

- **Tipo Abitato:** Se riusciamo a scrivere una funzione (un programma che termina) con un certo tipo, abbiamo effettivamente fornito una *dimostrazione costruttiva* della proposizione logica corrispondente.
- **Tipo Vuoto:** Se un tipo non ha abitanti (non è possibile scrivere un termine per quel tipo), la proposizione corrispondente è logicamente falsa (o non dimostrabile).

Esempio: Consideriamo il tipo $\forall X, Y. X \rightarrow (Y \rightarrow X)$.

- **Logica:** Corrisponde all'assioma $A \rightarrow (B \rightarrow A)$.
- **Programma:** Corrisponde al termine $\lambda x. \lambda y. y \Rightarrow x$. Questo programma è valido e ben tipato, il che conferma che la proposizione logica è una tautologia.

7.0.2.2. Esempi di tautologie e tipi abitati

(aggiunti da me per curiosità personale)

Altri casi in cui la verità logica si traduce in programmi:

- *Transitività dell'implicazione*

- **Logica:** $(A \rightarrow B) \wedge (B \rightarrow C) \rightarrow (A \rightarrow C)$.
- equivalente logicamente a $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$.
- **Programma:** composizione di funzioni.
Termine: $fn\ f \Rightarrow fn\ g \Rightarrow fn\ x \Rightarrow g(f(x))$
Tipo: $(A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$.
Dato che possiamo scrivere una funzione che compone due funzioni, la proposizione logica è dimostrata.
- *Scambio di premesse (flip)*
- **Logica:** $(A \rightarrow (B \rightarrow C)) \rightarrow (B \rightarrow (A \rightarrow C))$.
Se assumendo A e poi B posso dimostrare C , allora posso arrivare a C anche assumendo prima B e poi A (l'ordine delle ipotesi è irrilevante)
- **Programma:** Una funzione che inverte l'ordine degli argomenti.
Termine: $fn\ f \Rightarrow fn\ y \Rightarrow fn\ x \Rightarrow f\ x\ y$
Tipo: $(A \rightarrow B \rightarrow C) \rightarrow (B \rightarrow A \rightarrow C)$.

7.0.2.3. Proposizioni non tipabili

Nella logica classica esistono tautologie che non sono dimostrabili nella logica intuizionista e, di conseguenza, corrispondono a tipi che non possono essere abitati da un programma puro (l'informatica richiede una costruzione esplicita del risultato).

In questo contesto, la negazione $\neg A$ è definita come $A \rightarrow \perp$ (logicamente equivalenti).
Un programma di tipo $\neg A$ è una funzione che, se alimentata con un input di tipo A , non restituisce mai un valore (crasha o diverge), producendo l'assurdo.

Esempi di formule logicamente valide in logica classica ma computazionalmente impossibili:

- *Tertium non datur*
 - **Logica:** $A \vee \neg A$.
 - **Tipo:** $A + (A \rightarrow \perp)$
per abitare questo tipo, dovremmo scrivere una funzione universale che, per un qualsiasi tipo generico A , sia in grado di decidere se restituire un'istanza di A (ramo sinistro) o una dimostrazione che A è impossibile (ramo destro) - non esiste un algoritmo onnisciente in grado di determinare l'abitabilità di un tipo arbitrario.
- *Eliminazione della doppia negazione*
 - **Logica:** $\neg\neg A \rightarrow A$.
 - **Tipo:** $((A \rightarrow \perp) \rightarrow \perp) \rightarrow A$.
questo richiederebbe una funzione che prende in input la "prova dell'impossibilità che A non esista" e restituisca un'istanza concreta di A - ma sapere che un dato non può non esistere non equivale ad averlo costruito in memoria.
- *Legge di Peirce*
 - **Logica:** $((A \rightarrow B) \rightarrow A) \rightarrow A$.
 - **Tipo:** stesso della logica.
questo tipo richiede di produrre un A avendo solo una funzione che promette di restituire un A se noi gli forniamo un modo per andare da A a B . È un vicolo cieco computazionale: non possiamo ottenere l' A iniziale senza invocare la funzione, ma per invocarla ci serve un A che ancora non abbiamo.