

Contents

1	Algebre e Algebre induttive	2
1.1	Introduzione e definizione	2
1.2	Lemma di Lambek	4
1.2.1	Precisazione sulla notazione	4
1.3	Algebra induttiva di alberi binari	5
1.4	Esercizi	5
2	Linguaggi di espressioni	8
2.1	Introduzione	8
2.2	Linguaggio <i>Exp</i>	9
2.2.1	Operatore <i>let</i> , variabili <i>libere e legate</i>	9
2.3	Semantica operativa di <i>Exp</i>	10
2.3.1	Semantica operativa " <i>lazy</i> "	11
2.3.2	Semantica <i>lazy</i> con <i>scoping statico</i>	12
3	Linguaggio <i>Fun</i>	13
3.1	Semantica operativa	14
3.2	Il lambda calcolo	15
3.2.1	I numeri di Church	15

1 Algebre e Algebre induttive

1.1 Introduzione e definizione

Definizione 1.1 (Assiomi di Peano).

- I. $\emptyset \in \mathbb{N}$
- II. se $n \in \mathbb{N} \implies \sigma(n) \in \mathbb{N}$
- III. $\nexists n \text{ t.c. } \sigma(n) = \emptyset$
- IV. $\forall m, n \quad \sigma(m) = \sigma(n) \implies n = m$
- V. $\forall S \subseteq \mathbb{N} \text{ (se } \emptyset \in S \text{ e } n \in S \implies \sigma(n) \in S) \implies S = \mathbb{N}$

Il quinto assioma è equivalente all'induzione, ovvero: sia P una proprietà su \mathbb{N} , se $P(0)$ è vera e si assume sia vera fino ad n , quindi $P(n)$ se si dimostra che $P(n+1)$ è vero allora P è vera per ogni n . Attraverso questi assiomi abbiamo costruito la "struttura" dei numeri naturali, che non sono altro che un caso particolare di algebra. La definizione di peano quindi è un modo per definire un'algebra induttiva. Le algebre sono degli insiemi dotati di operazioni definite sugli elementi dell'insieme stesso, nel caso delle algebre eterogenee vengono coinvolti anche parametri esterni. Le operazioni definite in questi insiemi hanno il codominio nell'insieme stesso, consideriamo ad esempio la seguente algebra:

Esempio 1.2. Sia A l'insieme dotato di un'operazione γ definita come segue:

$$\gamma : A \times K \rightarrow A$$

dove K è una collezione di elementi diversa da A . Anche l'operazione $\eta : K \rightarrow A$ che non prende elementi da A può essere un'operazione valida.

Definizione 1.3. Un insieme S si dice chiuso rispetto ad un'operazione γ se:

- 1. $a \in S \implies \gamma(a) \in S$
- 2. $a_1, a_2, \dots, a_k \in S \implies \gamma(a_1, a_2, \dots, a_k) \in S$
- 3. Preso $m \in M$ e $a_1, a_2 \in S \implies \gamma(m, a_1, a_2) \in S$

(nel (3) m può essere un qualunque elemento di M)

Definizione 1.4. Un'algebra (A, γ) , dove γ rappresenta una famiglia di operazioni $\{\gamma_i\}$, si dice induttiva quando:

- 1. Tutte le γ_i sono iniettive.
- 2. Le γ_i hanno immagini disgiunte.
- 3. $\forall S \subseteq A$ se S è chiuso rispetto a tutte le γ_i allora $S = A$.

L'insieme A è chiamato *insieme sottostante* all'algebra e rappresenta il codominio di ogni operazione $\gamma_i \in \gamma$.

Quindi \mathbb{N} è solo un caso particolare di algebra induttiva definita su gli interi positivi, dotata dell'operazione σ e dell'operazione \emptyset . Quest'ultima merita un piccolo approfondimento, infatti il terzo assioma di Peano impone che \emptyset non sia immagine di alcun $\sigma(n)$, abbiamo quindi bisogno di definire un'operazione speciale che mappa nello zero:

$$\emptyset : \mathbb{1} \rightarrow \mathbb{N}$$

Dove $\mathbb{1}$ denota l'insieme banale (o \mathbb{N}^0 ovvero un insieme composto da un solo elemento). La definizione di algebra induttiva ci serve per definire una collezione di oggetti in cui si esclude tutto ciò che non è possibile costruire a partire dalle operazioni definite.

Esempio 1.5 (Algebra di liste). Definiamo L l'insieme delle liste ordinarie di interi e una famiglia di operazioni γ_L formata da due operazioni, *cons*, *empty*, diciamo che (L, γ_L) è un'algebra induttiva. Definiamo *cons* come l'operazione che dato un naturale e una lista, aggiunge quel numero in coda alla lista:

$$\text{cons}(n, (n_1, \dots, n_q)) = (n, n_1, \dots, n_q)$$

empty invece è l'operazione con dominio in $\mathbb{1}$ che mappa nella lista vuota $()$. Quindi *cons* e *empty* rispettano gli assiomi di algebra induttiva, in particolare: le immagini sono disgiunte, le operazioni sono iniettive e non esistono sotto-algebre chiuse per *cons* e *empty*. Grazie alla struttura induttiva appena costruita possiamo definire l'operazione *append*, che prende due liste e le unisce. Ecco un esempio di definizione ricorsiva:

$$\text{append}(), l = l$$

$$\text{append}(\text{cons}(n, l), l') = \text{cons}(n, \text{append}(l, l')) \quad \text{con } n \in \mathbb{N} \text{ e } l, l' \in L$$

Esempio 1.6. (Booleani) sia $\mathbb{B} = \{\text{True}, \text{False}\}$ e siano t, f due operazioni:

$$t : \mathbb{1} \rightarrow \mathbb{B}$$

$$f : \mathbb{1} \rightarrow \mathbb{B}$$

Quindi $t(\mathbb{1}) = \text{True}$ e $f(\mathbb{1}) = \text{False}$, da questa definizione segue che $(\mathbb{B}, \{f, t\})$ è un'algebra induttiva.

Teorema 1.7. *Un'algebra induttiva è finita se e solo se i costruttori hanno solo parametri esterni.*

Un esempio banale è \mathbb{B} (1.8).

1.2 Lemma di Lambek

1.2.1 Precisazione sulla notazione

La segnatura algebrica delle operazioni di un'algebra è rappresentata da un'insieme I , di nomi di funzione e per ogni $i \in I$ corrisponde un $\alpha_i \geq 0$, che indica il numero di parametri che l'operazione prende dall'insieme sottostante all'algebra, e un vettore $\mathbf{K}_i = (K_{i1}, \dots, K_{iq})$, contenente i domini da cui vengono presi i parametri esterni per l'operazione, quindi la dimensione del vettore rappresenta il numero dei parametri esterni. Due signature di due algebre sono equivalenti se è possibile ottenere l'una dall'altra semplicemente scambiando gli insiemi sottostanti all'algebra nei singoli costruttori.

Esempio 1.8. Consideriamo (A, f_A) e (B, f_B) , con $f_A : A \times K \rightarrow A$ e $f_B : B \times K \rightarrow B$, le due algebre hanno la stessa segnatura perché è possibile ottenere f_A semplicemente sostituendo in f_B B con A . Quindi l'equivalenza di signature è semplicemente un'equivalenza nella "forma" di ogni costruttore dell'algebra.

Definizione 1.9. $h : (A, \gamma_A) \rightarrow (B, \gamma_B)$ è un omomorfismo di algebre se per ogni $i \in I$

$$h(\gamma_{A_i}(a_1, \dots, a_{\alpha_i}, k_1, \dots, k_{\beta_i})) = \gamma_{B_i}(h(a_1), \dots, h(a_{\alpha_i}), k_1, \dots, k_{\beta_i}).$$

Un omomorfismo bigettivo è chiamato *isomorfismo*.

Teorema 1.10. Siano (A, γ_A) un'algebra induttiva e (B, γ_B) un'algebra (non necessariamente induttiva), con operazioni con la stessa segnatura, allora esiste un unico omomorfismo h :

$$h : (A, \gamma_A) \rightarrow (B, \gamma_B)$$

Esempio 1.11. Consideriamo l'algebra induttiva dei naturali e $(\mathbb{B}, \text{true}, \text{not})$, dove $\text{not}(\text{true}) = \text{false}$ e $\text{not}(\text{false}) = \text{true}$. Per il teorema (1.10) esiste un unico omomorfismo di algebre $h : \mathbb{N} \rightarrow \mathbb{B}$ definito come :

$$h(\emptyset) = \text{True}$$

$$h(\sigma(n)) = \text{not}(h(n))$$

Lemma 1.12 (Lambek). Due algebre induttive con stessa segnatura sono isomorfe.

Dimostrazione. Siano A e B due algebre induttive, per (1.11) esiste un unico omomorfismo $h : A \rightarrow B$ e viceversa $h' : B \rightarrow A$.

$$A \xrightarrow{h} B \xrightarrow{h'} A$$

Consideriamo ora $h' \circ h : A \rightarrow A$, ovviamente la composizione di due omomorfismi è anch'esso un omomorfismo, per esempio la funzione identità $Id : A \rightarrow A$ è un omomorfismo da A in A . Ricordiamo che dal precedente teorema sappiamo che tale omomorfismo è unico, segue che $h' \circ h = Id$ e quindi $h' = h^{-1}$. \square

Affinchè sia presente un isomorfismo è necessaria una bigezione tra gli insiemi sottostanti all'algebra, ma quest'ultima non è sufficiente a garantire l'uguaglianza nella struttura, infatti è necessario che anche le segnature siano le stesse.

Esempio 1.13. Prendiamo il caso di due insiemi di interi positivi \mathbb{N} e $\mathbb{N}_* := \{0, 1, \dots, *\}$, esiste certamente una mappa biettiva tra i due insiemi, ma non è possibile stabilire un isomorfismo tra le due algebre, in quanto la segnatura delle rispettive famiglie di operazioni sarà diversa.

1.3 Algebra induttiva di alberi binari

Teorema 1.14. *Ogni albero binario con n foglie ha $2n - 1$ nodi.*

Per dimostrare questo teorema possiamo utilizzare l'induzione completa, ma ai fini del nostro studio, risulta più istruttivo utilizzare l'*induzione strutturale* su un'algebra induttiva di alberi binari. Sia B_{tree} l'insieme di tutti gli alberi binari finiti. Dotiamo B_{tree} di due costruttori:

- $root : 1 \rightarrow B_{tree}$, un costruttore di base che mappa nell'albero binario formato da un solo nodo.
- $branch : B_{tree} \times B_{tree} \rightarrow B_{tree}$, un costruttore che unisce due alberi binari, aggiungendo una radice e attaccando i due alberi alla radice, uno come sottoalbero destro e uno come sotto albero sinistro.

Le due operazioni rispettano gli assiomi di algebra induttiva, quindi $(B_{tree}, root, branch)$ è un'algebra induttiva. Possiamo ora applicare l'induzione sull'algebra di alberi binari, modificando l'induzione sui naturali:

$$\frac{P(0) \quad P(n) \implies P(\sigma(n))}{\forall n \quad P(n)} \rightarrow \frac{P(root) \quad P(t_1), P(t_2) \implies P(branch(t_1, t_2))}{\forall t \quad P(t)}$$

Dimostrazione (1.14). Il caso base è banale infatti $|root| = 2(1) - 1 = 1$. Appliciamo il passo induttivo e dimostriamo che, dati t_1 e t_2 due alberi binari con $|t_1| = 2n_1 - 1$, $|t_2| = 2n_2 - 1$, allora $|branch(t_1, t_2)| = 2n_1 - 1 + 2n_2 - 1 + 1 = 2(n_1 + n_2) - 1$. \square

Durante la costruzione dell'algebra abbiamo specificato la presenza solo di alberi finiti, in quanto un elemento in sé infinito (in questo caso un albero), violerebbe gli assiomi di algebra induttiva. Quindi le collezioni di elementi con operazioni che contengono elementi di questo tipo vengono chiamate algebre *co-induttive*.

1.4 Esercizi

Definizione 1.15. Un costruttore è ogni γ_i appartenente alla famiglia di operazioni di un'algebra (A, γ) . Un costruttore di base non ha parametri presi dall'insieme sottostante all'algebra, ovvero $\alpha_i = 0$.

Esercizio 1. Dimostrare che ogni algebra induttiva non vuota ha almeno un costruttore base.

Soluzione. Sia (A, γ) , con $\gamma = (\gamma_1, \dots, \gamma_k)$, un'algebra induttiva. Consideriamo $\emptyset \subsetneq A$, questo è chiaramente chiuso per ogni γ_i che non sia di base, quindi se supponiamo che non esistano in γ costruttori di base allora (\emptyset, γ) è un'algebra induttiva, andando in contrapposizione con il terzo assioma. Segue l'esistenza di almeno un costruttore base. \square

Esercizio 2. Mostrare che esiste un isomorfismo algebrico tra i naturali e $P = \{0, 2, 4, \dots\}$ dove, $0_P = 0$ e $\sigma_P(n) = n + 2$.

Soluzione. Per prima cosa verifichiamo che $(P, \emptyset_P, \sigma_P)$ sia un'algebra induttiva:

1. σ_P e $\emptyset_P : \mathbb{1} \rightarrow P$, sono chiaramente iniettive, infatti se $\sigma_P(n) = \sigma_P(m)$ e quindi $n + 2 = m + 2 \implies n = m$.
2. $\text{Im } \sigma_P \cap \text{Im } \emptyset_P = \emptyset$
3. Chiaramente se $S \subseteq P$ è chiuso rispetto a \emptyset_P e σ_P , allora S deve essere necessariamente P .

Consideriamo la funzione tra le due algebre induttive che hanno signature equivalenti:

$$\begin{aligned} f : \mathbb{N} &\rightarrow P \\ n &\xrightarrow{f} 2n \end{aligned}$$

Chiaramente f è bigettiva ed è anche un omomorfismo, infatti

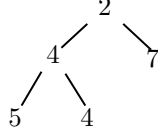
$$\begin{aligned} f(\sigma(n)) &= \sigma_P(f(n)) \\ \emptyset &= \emptyset_P \end{aligned}$$

quindi f è un isomorfismo. \square

Esercizio 3. Dimostrare che ogni algebra induttiva non vuota con un costruttore non base è necessariamente infinita.

Soluzione. Sia $b \in A$ l'elemento mappato dal costruttore di base β (di cui abbiamo verificato l'esistenza nell'esercizio sopra), se A è dotato anche di un costruttore non di base γ , allora $\gamma(b) = a_1$, $\gamma(a_1) = a_2$ e così via senza mai giungere ad una fine. Infatti se volessimo provare a chiudere la sequenza, ad esempio proprio con $\gamma(a_n) = b$, otterremmo delle immagini di γ e β non disgiunte contraddicendo gli assiomi di algebra induttiva, segue la non finitezza di A . \square

Esercizio 4. Consideriamo alberi binari con nodi etichettati da numeri naturali. Eccone uno:



Definire l'algebra induttiva *BN-trees* di questi alberi. Definire un'algebra di uguale segnatura sull'insieme S delle sequenze finite di numeri naturali in modo che la funzione $f : BN-trees \rightarrow S$ che associa a ciascun albero la sequenza di etichette ottenuta con una visita depth-first sia un omomorfismo. Applicando f all'albero dell'esempio, ci aspettiamo di ottenere la sequenza $\langle 2, 4, 5, 4, 7 \rangle$.

Soluzione. Definiamo un'estensione delle operazioni create in precedenza per l'algebra di B_{tree} :

$$branch^* : BN-tree \times BN-tree \times \mathbb{N} \rightarrow BN-tree,$$

$$root^* : \mathbb{N} \rightarrow BN-tree.$$

$branch^*$ prende in input due alberi e un intero che etichetterà la radice; $root^*$ invece prende un intero n e crea la radice etichettata con n . In modo del tutto analogo definiamo sull'insieme S l'operazione di concatenazione di due serie numeriche s_1, s_2 di lunghezza dispari che restituisca una serie dispari:

$$\mathcal{C} : S \times S \times \mathbb{N} \rightarrow S$$

$$\mathcal{C}(s, s', n) \mapsto \langle n, s_1, \dots, s_k, s'_1, \dots, s'_q \rangle$$

$$\Lambda : \mathbb{N} \rightarrow S$$

$$\Lambda(n) \mapsto \langle n \rangle$$

Come conseguenza del *lemma di Lambek* (1.12) esiste ed è unico l'omomorfismo $f : B-tree \rightarrow S$. Consideriamo la sequenza $s = \langle s_1 \dots s_k \rangle$ generata visitando con una *DFS* l'albero $t \in B-tree$, quindi $f(t) = s$. In particolare f è un omomorfismo di algebre

$$f(branch(t_1, t_2, a)) = \mathcal{C}(f(t_1), f(t_2), a) \quad (1)$$

$$f(root^*(n)) = \Lambda(n) \quad (2)$$

La (1) è giustificata dal fatto che la *depth-first* visita l'albero da sinistra a destra, e quindi la sequenza risultante avrà inizialmente a (la radice del nuovo albero prodotto da $branch$) $f(t_1)$ ed infine $f(t_2)$. \square

2 Linguaggi di espressioni

2.1 Introduzione

Definizione 2.1. Sia L un linguaggio, un insieme di stringhe generate dalla grammatica

$$M, N ::= 1|2|\dots|M + N|M * N.$$

Esempio 2.2. Ad esempio $3 + 5, 3 * 5$ e $4 * 5 + 2$ sono delle espressioni di L .

Introduciamo la funzione $eval : L \rightarrow \mathbb{N}$, per valutare la sintassi delle espressioni:

$$eval(n) = n$$

$$eval(M + N) = eval(M) + eval(N)$$

$$eval(M * N) = eval(M) * eval(N)$$

Possiamo definire la funzione $eval$ per casi, ma non è stato definito un modo univoco per valutare delle espressioni come " $5 + 4 * 3$ ", viene svolta prima la $*$ o il $+$?

Per come è stato costruito il linguaggio L dotato delle operazioni $+$ e $*$ non può essere un'algebra induttiva. Per disambiguare queste espressioni dobbiamo riformulare il linguaggio in modo da renderlo un algebra induttiva. Per iniziare dobbiamo dare una rappresentazione di senso univoco alle espressioni del tipo " $5 + 4 * 3$ ", scriviamo più formalmente:

$$\mathbf{1} : \mathbb{1} \rightarrow L$$

$$\mathbf{2} : \mathbb{1} \rightarrow L$$

$$\vdots$$

$$times : L \times L \rightarrow L$$

$$plus : L \times L \rightarrow L$$

Allora $(L, \mathbf{1}, \mathbf{2}, \dots, times, plus)$ è un'algebra induttiva. Quindi l' $eval$ di " $5 + 4 * 3$ " può essere:

$$eval("5 + 4 * 3") = \begin{cases} eval(times(plus(\mathbf{5}(\epsilon), \mathbf{4}(\epsilon)), \mathbf{3}(\epsilon))) \\ eval(plus(\mathbf{5}(\epsilon), times(\mathbf{4}(\epsilon), \mathbf{3}(\epsilon)))) \end{cases}$$

Pur essendo quella definita sopra la notazione corretta, utilizzarla per le nostre valutazioni risulterebbe inutilmente complessa, per questo motivo faremo uso della classica notazione con le parentesi per definire le precedenze. Ad esempio

$$plus(\mathbf{5}(\epsilon), times(\mathbf{4}(\epsilon), \mathbf{3}(\epsilon))) = 5 + (4 * 3).$$

2.2 Linguaggio *Exp*

Definiamo il linguaggio ***Exp*** costituito da espressioni di somma e *let*:

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

dove k sono le costanti ed appartengono all'insieme dei valori $Val = \{1, 2, \dots\}$; x sono le variabili in $Var = \{x, y, z, \dots\}$ ed M, N in termini di linguaggio in *Exp*.

2.2.1 Operatore *let*, variabili libere e legate

L'operatore *let* derivante dalla sintassi SML, indicato con $\text{let } x = M \text{ in } N$ e con segnatura:

$$\text{let} : Var \times Exp \times Exp \rightarrow Exp$$

In questo modo definiamo una variabile locale x inizializzata al valore dell'espressione M , ed un corpo N che può contenere un riferimento ad x . Ad esempio

$$\text{let } x = 4 \text{ in } x + x$$

verrà valutata assegnando alla variabile x il valore 4, sommando $x + x$ e restituisce 8. Una variabile è *libera* in un termine T quando non compare in nessun N , sottotermini di T , della forma $\text{let } x = M \text{ in } N$. Ogni occorrenza di x in N viene detta *legata* alla dichiarazione di x nel termine $\text{let } x = M \text{ in } N$. Definiamo induttivamente la funzione $free : Exp \rightarrow \mathcal{P}(Var)$, che restituisce l'insieme delle variabili che occorrono libere in un'espressione :

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

A questo proposito definiamo lo *scoping* di una variabile come l'insieme delle regole che determinano la visibilità di una variabile all'interno del programma. Per *scope* di una variabile invece, si intende la porzione di programma in cui la variabile può essere riferita.

Esempio 2.3. Consideriamo le due espressioni in ***Exp***:

$$\text{let } x = M \text{ in let } y = N \text{ in } L \quad (I)$$

$$\text{let } y = (\text{let } x = M \text{ in } N) \text{ in } L \quad (II)$$

Se procediamo alla valutazione di (I) e (II), le due espressioni sembrerebbero essere equivalenti, quindi se le intercambiassimo all'interno di uno stesso programma, questo dovrebbe produrre lo stesso risultato. Ciò non succede nell'espressione qui sotto se valutata:

$$\text{let } x = 1 \text{ in } [\text{let } x = M \text{ in let } y = N \text{ in } L] \quad (I')$$

$$\text{let } x = 1 \text{ in } [\text{let } y = (\text{let } x = M \text{ in } N) \text{ in } L] \quad (II')$$

In questo caso (I') vale 6, mentre (II') vale 4.

2.3 Semantica operativa di Exp

Il precedente esempio ci porta a riflettere sull'esistenza di un modo per dimostrare l'equivalenza di due programmi. Quest'ultimo è un compito molto più difficile rispetto alla dimostrazione della "non equivalenza", la quale solitamente necessita solo di un controesempio. Dobbiamo definire una **semantica operativa** per il linguaggio **Exp** affinché sia possibile assegnare un significato alle sue operazioni. Assumeremo, con un'ipotesi semplificata, che il **significato** di un'espressione sia semplicemente il suo valore. Quello che intendiamo creare è un sistema formale con regole di inferenza simile a quello logico, che ci consenta di derivare delle conseguenze a partire da delle premesse iniziali. Per definire in modo formale una semantica operativa è necessario fornire alcune nozioni preliminari.

Definizione 2.4. Una funzione f si dice **parziale** se può essere definita solo per un sottoinsieme del suo dominio, ed è indicata con la seguente notazione

$$f : D \rightharpoonup C.$$

f viene detta **parziale finita**, se il sottoinsieme del dominio per cui è definita è finito (scritta con \xrightarrow{fin}).

Definizione 2.5. Un *ambiente* è un'associazione tra variabili in Var e valori in Val , formalmente definiamo una funzione *parziale finita*

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

Definizione 2.6. Siano E_1, E_2 sono due ambienti, allora la loro composizione $(E_1 E_2)$ è quell'ambiente che se applicato ad una variabile risulta:

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

Quindi per prima cosa si controlla se x è definito nell'ambiente E_2 , se non definito controlla E_1 .

Esempio 2.7. $(E_1 E_2)(y) = (\{(x, 2)(y, 3)\}\{(y, 4)\})(y) = 4$

Osservazione 2.8. Ricordiamo inoltre la forma di una regola logica:

$$\frac{premesse_1 \dots premesse_k}{conseguenza} \quad (\text{condizione laterale} \dots)$$

Risulta necessario sviluppare un insieme di regole che mi permettano, a partire da delle condizioni iniziali, di derivare dei sequenti come $E \vdash M \rightsquigarrow v$ dove, come specificato precedentemente, E è una funzione parziale Env , M è un termine di Exp e v è un valore in Val .

Definizione 2.9. La semantica operativa di Exp è definita come una relazione

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

ovvero una relazione tra Env ed Exp che restituisce un valore in Val . Diamo ora una costruzione induttiva delle regole della relazione " \rightsquigarrow ":

- *Regola della costante*: $E \vdash k \rightsquigarrow k$
- *Regola della variabile*: $E \vdash x \rightsquigarrow v$ (se $E(x) = v$)
- *Regola del plus*:

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

- *Regola del let*:

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

dove con $E(x, v)$ componiamo l'ambiente E con l'associazione tra x e v .

Osservazione 2.10. Le premesse in una regola logica non hanno un ordine da seguire, da questo possiamo dedurre che la funzione *plus* sia commutativa. Ovviamente questo non è vero in ogni linguaggio, ad esempio

`x + (x++) != (x++) + x (Java)`

Per come è stata costruito il *let*, si potrebbe pensare che provochi una modifica all'ambiente, e che quindi sia importante tenere conto dell'ordine con cui viene eseguito, in realtà non è così. Infatti la dichiarazione fatta nel *let* è limitata all'espressione in M o N che sono fatte al suo interno, di conseguenza creeremo solo variabili locali.

Esempio 2.11. La valutazione delle espressioni attraverso la semantica operativa appena definita viene detta valutazione *eager* ovvero, valuta le espressioni appena le incontra. Diamo la prova utilizzando le regole di derivazione sulla seguente espressione (*la valutazione parte dal basso e va verso l'alto*):

$$\frac{\frac{\frac{(x, 1)(y, 1) \vdash 2 \rightsquigarrow 2 \quad (x, 1)(y, 1)(x, 2) \vdash y \rightsquigarrow 1}{(x, 1) \vdash x \rightsquigarrow 1} \quad (x, 1)(y, 1) \vdash \text{let } x = 2 \text{ in } y \rightsquigarrow 1}{\emptyset \vdash 1 \rightsquigarrow 1} \quad (x, 1) \vdash \text{let } y = x \text{ in let } x = 2 \text{ in } y \rightsquigarrow 1}{\emptyset \vdash \text{let } x = 1 \text{ in let } y = x \text{ in let } x = 2 \text{ in } y \rightsquigarrow 1}$$

Con le regole definite l'espressione è valutata 1, ma nel caso in cui la valutazione fosse stata *lazy* avremmo ottenuto come risultato 2, questo perché l'espressione viene valutata solo quando dev'essere utilizzata.

2.3.1 Semantica operativa "lazy"

Per ottenere una valutazione *lazy* è necessario modificare la *Regola del let* e la *Regola della variabile*:

- *Regola del let modificata*:

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

dove $E(x, M)$ indica la composizione dell'ambiente E con l'associazione della variabile x all'espressione non valutata M .

- Regola della variabile modificata

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

In questo modo dobbiamo valutare una certa espressione M solo se è richiesto il suo valore.

2.3.2 Semantica *lazy* con *scoping statico*

La modifica delle regole oltre a cambiare il tipo di valutazione, sta cambiando anche lo *scoping* del linguaggio. Infatti con la valutazione *eager* lo *scoping* era *statico*, ma con l'introduzione delle regole *lazy* lo *scoping* è diventato *dinamico*. Per ottenere una semantica *lazy statica* dobbiamo modificare ulteriormente le regole:

$$\begin{aligned} [let]_{LS} & \frac{E(x, M, E) \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v} \\ [var]_{LS} & \frac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \quad (\text{se } E(x) = (M, E')) \end{aligned}$$

Quello che stiamo facendo è tenere conto dell'ambiente E in cui inizialmente abbiamo l'associazione (x, M) , quindi è necessario estendere la funzione Env :

$$Env : Var \xrightarrow{fin} Exp \times Env \quad (I)$$

La (I) è chiamata *equazione ricorsiva di domini*, la soluzione dell'equazione è ogni insieme X per cui

$$X : Var \xrightarrow{fin} Exp \times X$$

in particolare possiamo avere una soluzione con *ambienti alti*, ovvero una serie di ambienti infinitamente annidati; altrimenti avremo degli *ambienti bassi*. Risulta chiaro che un ambiente *alto* ad esempio

$$E = \{(x, \{(x, \{(x, \{\dots\})\dots\})\})\}$$

non terminerà mai la valutazione di x nel caso in cui ne avesse bisogno. Nel caso dello *scoping statico* tengo conto dell'ambiente in cui ho eseguito una determinata associazione; se invece lo *scoping* è *dinamico* utilizzo l'ambiente attuale per eseguire la valutazione.

Osservazione 2.12. La valutazione *lazy* non è sempre più efficiente di quella *eager*, infatti ogni volta che avremo bisogno di un valore, se utilizziamo la *laziness* dobbiamo ricalcolarlo per ogni occorrenza. Ad esempio

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

se M è un termine complesso, la valutazione *lazy* è estremamente inefficiente.

Esempio 2.13. Valutiamo la seguente espressione con la semantica *lazy statica*:

$$\frac{\frac{\frac{\frac{\frac{\frac{\emptyset \vdash 3 \rightsquigarrow 3}{E \vdash x \rightsquigarrow 3} \quad (E(x) = (3, \emptyset))}{E'' = E'(x, 5, E') \vdash y \rightsquigarrow 3} \quad (E''(y) = (x, E))}{E' = E(y, x, E) \vdash \text{let } x = 5 \text{ in } y \rightsquigarrow 3}}{E = (x, 3, \emptyset) \vdash \text{let } y = x \text{ in let } x = 5 \text{ in } y \rightsquigarrow 3}}{\emptyset \vdash \text{let } x = 3 \text{ in let } y = x \text{ in let } x = 5 \text{ in } y \rightsquigarrow 3}$$

La valutazione sarebbe stata la stessa se avessimo utilizzato le regole della semantica *eager*.

Definizione 2.14. Un ambiente *eager* E si dice *equivalente* ad un ambiente *lazy statico* E' , se per ogni variabile x , con $E(x) = v$, esistono M ed E'' tali che $E'(x) = (M, E'')$ e $E'' \vdash M \rightsquigarrow v$.

Teorema 2.15. Le semantiche *lazy statica* e *eager* sono equivalenti se e solo se E ed E' non sono ambienti alti.

$$E \vdash M \rightsquigarrow_E v \iff E' \vdash M \rightsquigarrow_{LS} v$$

3 Linguaggio *Fun*

Il linguaggio *Fun* estende il linguaggio *Exp* con la nozione di *funzione*, quindi possiamo aggiungere al precedente linguaggio le seguenti clausole:

$$fn \ x \implies M|MN,$$

quindi

$$MN ::= k|x|M + N | \text{let } x = M \text{ in } N | fn \ x \implies M|MN$$

Successivamente mostreremo che le clausole definite nel linguaggio *Exp*, ad eccezione delle *variabili* non sono necessarie in *Fun*. L'operazione *fn* ha seguente natura:

$$fn : Var \times Fun \rightarrow Fun$$

Osservazione 3.1. L'operazione *fn* prende una sola variabile in input, per costruire "funzioni a più variabili" possiamo comporre più operatori *fn*

$$fn \ x_1 \implies (fn \ x_2 \implies \dots (fn \ x_k \implies M))$$

Per comodità la notazione sopra sarà equivalente a $fn \ x_1 \dots x_k \implies M$.

Esempio 3.2. $fn \ xy \Rightarrow x + y$, nella scrittura completa $fn \ x \Rightarrow fn \ y \Rightarrow x + y$. Utilizzando la clausola $fn \ x \Rightarrow MN$ stiamo applicando M ad N , ad esempio

$$(fn \ xy \rightarrow yx)(fn \ x \Rightarrow x + 1)(fn \ x \Rightarrow x)3$$

In questo caso a la prima funzione a partire da sinistra inverte l'ordine di applicazione delle due funzioni che la seguono, applicando la funzione identica e poi la funzione successore, successivamente passo 3 alla funzione successore ottenendo 4.

3.1 Semantica operativa

Essendo *Fun* un'estensione del linguaggio *Exp* diamo ora un'estensione anche delle regole semantiche. Per iniziare nell'espressione $fn\ x \Rightarrow M$ non c'è nulla da calcolare, per questo potremmo dire che

$$fn\ x \Rightarrow M \rightsquigarrow fn\ x \Rightarrow M \quad (*)$$

Useremo la notazione della *chiusura* per esprimere la parte sinistra della relazione (*), indicata con

$$[fn] \quad fn\ x \Rightarrow M \rightsquigarrow (x, M) \quad (I)$$

dove M è il *corpo* della *chiusura*, mentre x è il *parametro*. A seguito di questa modifica è necessario un ulteriore cambiamento nel dominio *Val* nella relazione \rightsquigarrow :

$$Val = Const \cup (Var \times Fun)$$

dove $Const = \{5, 6, \dots\}$. Studiamo ora il significato dell'applicazione di M ad N espressa con MN . Affinché tale espressione possa essere valutata è necessario che M sia una funzione, inoltre in questa prima definizione adotteremo una semantica *eager* con *scoping dinamico*.

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

quindi, per valutare MN dobbiamo valutare per prima M all'interno di E e restituire la sua *chiusura*, successivamente "passare" alla funzione appena valutata N (creando l'associazione (x, v)), infine valutare M' , ovvero il corpo della *chiusura*, nell'ambiente E aumentato dell'associazione di v ad x .

Osservazione 3.3. Bisogna ricordare che v e v' non rappresentano più solo dei valori.

Esempio 3.4. Deriviamo con la semantica appena definita la seguente espressione:

$$\frac{\frac{E \vdash y \rightsquigarrow (z, x + z) \quad E \vdash 7 \rightsquigarrow 7 \quad E(z, 7) \vdash x + z \rightsquigarrow 10}{(x, 3) \vdash fn\ z \Rightarrow x + z \rightsquigarrow (z, x + z) \quad E = (x, 3)(y, (z, (x + z))) \vdash let\ x = 4\ in\ y7 \rightsquigarrow 10}}{\frac{\emptyset \vdash 3 \rightsquigarrow 3 \quad (x, 3) \vdash let\ y = (fn\ z \Rightarrow x)\ in\ y7 \rightsquigarrow 10}{\emptyset \vdash let\ x = 3\ in\ let\ y = (fn\ z \Rightarrow x)\ in\ y7 \rightsquigarrow 10}}$$

Definendo la semantica di *Fun* con approccio *eager* abbiamo ottenuto uno *scoping dinamico*, questo perché il *corpo* della *chiusura* non viene valutato immediatamente (appena viene dichiarata la funzione), in aggiunta al fatto che non memorizziamo l'ambiente in cui inizialmente è dichiarata. Per ottenere quindi un ***eager statico*** è sufficiente modificare la *chiusura* memorizzando oltre al parametro e al corpo, anche l'ambiente del momento in cui incontriamo la funzione. Quindi la (I) diventa

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

mentre la regola (II) (regola di applicazione) sarà

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

Possiamo convincerci della validità di questa modifica provando a valutare il termine qui sotto, che avrà valore 4 nel caso di *eager dinamico*, 3 con quello *statico*:

$$\emptyset \vdash \text{let } x = 3 \text{ in let } y = \text{fn } z \Rightarrow x \text{ in let } x = 4 \text{ in } y7.$$

Per ottenere una versione **lazy** di *Fun* possiamo estendere la versione *lazy* di *Exp*, modificando anche le regole di *applicazione* e quella di *fn*.

3.2 Il lambda calcolo

Avremmo potuto scrivere il linguaggio *Fun* in termini più semplici eliminando delle clausole che sono derivabili da altre. Ad esempio la clausola

$$\text{let } x = M \text{ in } N \equiv (\text{fn } x \Rightarrow N)M$$

intuitivamente sia a sinistra che a destra stiamo applicando *M* in *N*.

Dimostrazione. La regola *[let]* di *Fun* può essere derivata dalle regole *[appl]* ed *[fn]*. Infatti *[let]* deriva $\text{let } x = M \text{ in } E \vdash N \rightsquigarrow v$ a partire dalle premesse $E \vdash M \rightsquigarrow v'$ e $E(x, v') \vdash N \rightsquigarrow v$.

Quindi traducendo il termine *let* in $(\text{fn } x \Rightarrow N)M$ riusciamo comunque a derivare *v*, a partire dalle stesse premesse con l'aggiunta della assioma *[fn]*:

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

□

In modo simile possiamo eliminare anche tutti i valori in *Val* e la somma $(M + N)$ ottenendo

$$M, N ::= x | \text{fn } x \Rightarrow M | MN$$

questo linguaggio più semplice sarà chiamato *lambda calcolo* e i suoi termini, *lambda termini*.

3.2.1 I numeri di Church

Il *lambda calcolo* appena definito deve essere equivalente al linguaggio *Fun*, quindi deve esistere un modo per rappresentare i numeri in *Val* che abbiamo eliminato, e di conseguenza anche la loro aritmetica. Una rappresentazione di questi numeri tramite il *lambda calcolo* è stata data da *Alonzo Church*. Essendo il λ -calcolo un linguaggio di sole funzioni, per esempio, *tre* viene definito come :

$$c_3 = \text{fn } xy \Rightarrow x(x(xy))$$

in parole "applico 3 volte x a y ". In generale un numero $n \in \omega$ può essere scritto

$$\begin{aligned}c_n &= fn \ xy \Rightarrow x^n y \\c_0 &= fn \ xy \Rightarrow y\end{aligned}$$

Definiamo ora la funzione successore per i numeri di Church σ , che prende in input un numero di Church e ne ritorna il successore:

$$\sigma(w) \equiv fn \ w \Rightarrow fn \ xy \Rightarrow wx(xy) \equiv x(wxy)$$

Spiegazione: partiamo da un numero di Church w , e otteniamo il suo successore applicando x ad y una volta (ovvero (xy)), successivamente applichiamo ad (xy) , w volte x (w è la funzione che si applica w volte a qualcosa dato in input). Con l'aiuto della funzione successore possiamo definire l'operazione *plus* per i numeri di Church:

$$plus \ uv = w\sigma v$$

ovvero passo σ a w , cioè la funzione che la applica w volte, e applico questa composizione a v . In modo simile e con risultato del tutto analogo possiamo scriverla come

$$fn \ uv \Rightarrow fn \ xy \Rightarrow wx(vxy)$$

quindi applichiamo la funzione x , v volte ad y e passo il risultato a w , che applica per altre w volte x ad y . Continuando la costruzione dell'aritmetica definiamo la moltiplicazione *times* a partire dal *plus*:

$$fn \ uv \Rightarrow v(plus \ u) \ c_0$$

oppure

$$fn \ uv \Rightarrow fn \ xy \Rightarrow u(vx)y.$$

Spiegazione: applico u alla funzione che applica v volte x a qualcosa, ovvero (vx) , il risultato è la funzione che applica u volte (vx) , ovviamente ad y .