aglaia norza

Linguaggi di Programmazione

appunti delle lezioni
libro del corso: non usato, integrati con le dispense del professor Cenciarelli

12/11/2025

thisisaglaia@gmail.com
github.com/AglaiaNorza

Contents

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 **9**

2.1 Linguaggi 9

2.2 Exp 9

2.2.1 Semantica operazionale 10

2.3 Valutazioni Eager e Lazy 12

2.4 Scoping 14

2.4.1 Riassunto delle regole in Exp 15

2.5 Fun 16

3 Lambda calcolo **20**

3.1 Numeri di Church 20

3.2 Primi cenni di SML 22

4 Paradigma imperativo **24**

4.1 Imp: un semplice linguaggio imperativo 24

4.2 All: un linguaggio con procedure 27

1. Algebre 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) $n \in \mathbb{N} \;\Rightarrow\; \text{succ}(n) \in \mathbb{N}$
3) $\nexists n \in \mathbb{N} \mid 0 = \text{succ}(n)$
4) $\forall n, m \; \text{succ}(n) = \text{succ}(m) \;\Rightarrow\; n = m$ (iniettività)
5) $\forall S \subseteq \mathbb{N} \; \left(0 \in S \wedge (n \in S \;\Rightarrow\; \text{succ}(n) \in S) \;\Rightarrow\; S = \mathbb{N}\right)$ (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:

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

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

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 $P(n) \;\Rightarrow\; P(n+1)$ equivale a dire $n \in S \;\Rightarrow\; n+1 \in S$.
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:

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

(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 $\mathbb{1} \to A$ (l'insieme di funzioni $\mathbb{1} \to A$ 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")
- $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_i\}$, è l'insieme di funzioni chiamate "operazioni fondamentali" o "costruttori" dell'algebra

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

> 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 $f : A^n \times K \to A$ un'operazione su A con parametri esterni $K = (K_1 \times \cdots \times K_m)$.
Un insieme $S \subseteq A$ si dice **chiuso** rispetto ad f quando:

$$a_1, \ldots, a_n \in S \ \Rightarrow\ f(a_1, \ldots, a_n, k_1, \ldots, k_n) \in S$$

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

Def. 6: **Algebra induttiva**

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

(1) tutte le $\gamma_i \in \Gamma$ sono iniettive

(2) $\forall i, j \mid i \neq j,\ \mathrm{Im}(\gamma_i) \cap \mathrm{Im}(\gamma_j) = \emptyset$, 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:

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

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

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 ($\nexists n \in \mathbb{N} \mid 0 = \mathrm{succ}(n)$), 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 $\mathbb{0}$ e $n \in \mathbb{N} \Rightarrow \mathrm{succ}(n) \in \mathbb{N}$ corrisponde alla chiusura rispetto a succ)

> alberi binari come algebre induttive
>
> L'insieme degli alberi binari finiti (`B-trees`, leaf, branch), dove:
> - `B-trees` $= \{t \mid t$ è una foglia, oppure $t = \langle t_1, t_2 \rangle$ con $t_1, t_2 \in$ `B-trees`$\}$
> - leaf: $1 \to$ `B-trees` (foglia)
> - branch: `B-trees` $\times$ `B-trees` $\to$ `B-trees` $: (t_{sx}, t_{dx}) \to t$ (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 $A - list$ l'insieme delle liste finite di elementi di A.
La tupla $(A - list, \text{ empty, cons})$ è un'algebra induttiva, dove:

- empty: $\mathbb{1} \to A - list$ è la funzione costante che restituisce la **lista vuota** "$\langle\rangle$".
- cons: $A \times A - list \to A - list : \text{cons}(3, \langle 5, 7\rangle) = \langle 3, 5, 7\rangle$ 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 $A - list$, 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 not: $B \to B : b \to \neg b$.
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 $\emptyset$).
$(\emptyset, \text{not})$ 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 ob($\mathcal{C}$), i cui elementi sono chiamati **oggetti**
- una classe mor($\mathcal{C}$), i cui elementi sono chiamati **morfismi** (o mappe o frecce); ogni morfismo $f : a \to b$ 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 $\mathrm{mor}(b, c) \times \mathrm{mor}(a, b) \to \mathrm{mor}(a, c)$ chiamata **composizione di morfismi**. La composizione di $f : b \to c$ con $g : a \to b$ si indica con $f \circ g : a \to c$

  la composizione deve soddisfare i seguenti assiomi:

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

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

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 $f : B \to C$ 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:

$$\forall A,\ \forall h, k : A \to B, \quad h \circ f = k \circ f \ \Rightarrow \ h = k$$

(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 $f : A \to B$ 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** $A \to B$.

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

  Ne segue necessariamente che $g \circ a = \text{Id}_A$. (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) $\Rightarrow$ g, f sono isomorfismi $\Rightarrow$ $A \cong B$

2. 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 $eval : L \to \mathbb{N}$, 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
- $M + N : Exp \times Exp \to Exp$ è la somma tra due espressioni
- $let : Var \times Exp \times Exp \to Exp$ 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 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 $free : Exp \to \mathcal{P}(Var)$, 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:

$$(E_1E_2)(x) = \begin{cases} E_2(x) & \text{se } x \in dom(E_2) \\ E_1(x) & \text{altrimenti} \end{cases}$$

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

Def. 14: Semantica operazionale 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 $u + w$)

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

(essenzialmente, per valutare una let, si:

- valuta M $\quad (E \vdash M \rightsquigarrow v_1)$
- 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 una funzione $Exp \times Env \to Val$, 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:

$$let\ \ x = 3\ \ in\ \ let\ \ x =\ \ let\ \ y = 2\ \ in\ \ x + y\ \ in\ \ x + 7 + x$$

in assenza di parentesi, va valutata "partendo dall'interno".

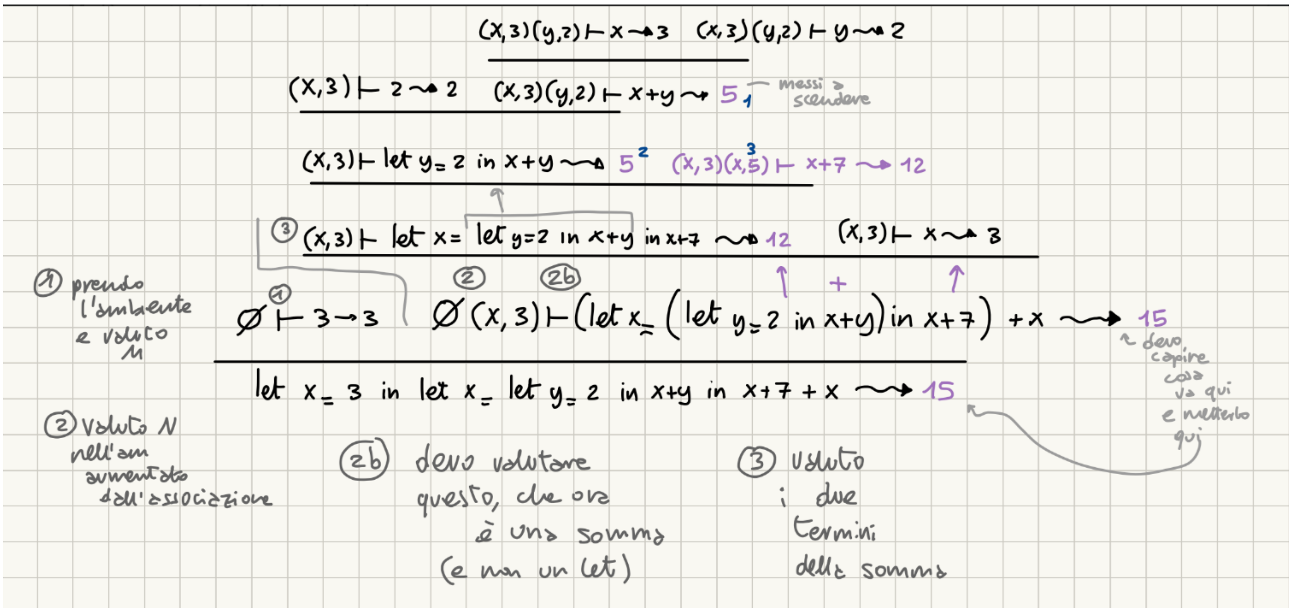
Corrisponde quindi a

$$let\ \ x = 3\ \ in\ \ [\,let\ \ x = \ (let\ \ y = 2\ \ in\ \ x + y)\ \ in\ \ x + 7\,] + x$$

E si ha quindi che:

- la x in $x + y$ e quella finale $(+x)$ sono quelle valutate dal let iniziale
- il valore della x in $x + 7$ è invece dato dal risultato di $let\ x = (let\ y = 2\ \ in\ \ x + y)$

Facciamo un esempio di valutazione di un'espressione:



(copierò appena ho tempo...)

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 $let\ =$ [espressione lunghissima] $in\ 7$, 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 $let\ x = \ N\ in\ M$ 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:

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

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

Mettiamo i due approcci a confronto sull'espressione

$$let\ x = 2\ in\ let\ y = x\ in\ let\ x = 7\ in\ y \rightsquigarrow 3$$

- **approccio eager**:

$$\dfrac{\dfrac{(x,2) \vdash x \rightsquigarrow 2 \qquad (x,2) \vdash 1 \rightsquigarrow 1}{(x,2) \vdash x + 1 \rightsquigarrow 3} \qquad \dfrac{(x,2)(y,3) \vdash 7 \rightsquigarrow 7 \qquad (x,2)(y,3)(x,7) \vdash y \rightsquigarrow 3}{(x,2)(y,3) \vdash \text{let } x = 7 \text{ in } y \rightsquigarrow 3}}{\dfrac{(x,2) \vdash \text{let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}}$$

- **approccio lazy**:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(x,2)(y,x+1)(x,7) \vdash 7 \rightsquigarrow 7}{(x,2)(y,x+1)(x,7) \vdash x \rightsquigarrow 7} \qquad (x,2)(y,x+1)(x,7) \vdash 1 \rightsquigarrow 1}{(x,2)(y,x+1)(x,7) \vdash x + 1 \rightsquigarrow 8}}{(x,2)(y,x+1)(x,7) \vdash y \rightsquigarrow 8}}{(x,2)(y,x+1) \text{ let } x = 7 \text{ in } y \rightsquigarrow 8}}{(x,2) \vdash \text{let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 8}}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x + 1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 8}$$

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'interno 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.

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

Per farlo, estendiamo nuovamente Env in questo modo:

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

Def. 19: Regole della semantica lazy statica 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:
$$\frac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \text{ se } E(x) = (M, E')$$
- la nuova regola per il let è:
$$\frac{E(x, M, E) \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$

Valutiamo la stessa espressione anche con questo approccio:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{(x,2,\emptyset) \vdash x \rightsquigarrow 2 \qquad (x,2,\emptyset) \vdash 1 \rightsquigarrow 1}{(x,2,\emptyset) \vdash x+1 \rightsquigarrow 3}}{E(x,7,E) \vdash y \rightsquigarrow 3}}{(x,2,\emptyset)(y,x+1,(x,2,\emptyset)) \vdash \text{ let } x = 7 \text{ in } y \rightsquigarrow 3}}{(x,2,\emptyset) \vdash \text{let } y = x+1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}}{\emptyset \vdash \text{let } x = 2 \text{ in let } y = x+1 \text{ in let } x = 7 \text{ in } y \rightsquigarrow 3}$$

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:

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

- 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

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

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

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

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

lazy statico

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

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

(lazy dinamico)

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

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

esercizi

2.5. Fun

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

Def. 20: Fun

La grammatica di Fun è:

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

dove:

- le regole presenti in Exp (1-4) rimangono invariate, con gli appropriati cambi di dominio (es. $let : Var \times Fun \times Fun \to Fun$)
- $fn : Var \times Fun \to Fun$ è una **funzione** (anonima) con parametro x
  - una funzione $fn\ x \ \Rightarrow\ M$ si può rappresentare in maniera alternativa attraverso la sua **chiusura**, $(x, M) \in Var \times Fun$
- $\cdot : Fun \times Fun \to Fun$ 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 $Var \cup (Var \times Fun)$ (variabili $\cup$ chiusure)

Quindi, per esempio:

- $(fn\ x \ \Rightarrow\ x + 1)\ 5 = 6$ (la funzione $x + 1$ è applicata all'argomento 6)
- $(fn\ x \ \Rightarrow\ x\ 5)(fn\ y \ \Rightarrow\ y + 1) = 6$

  la funzione prende in input una funzione (in questo caso "successore"), e la applica a 5
- $(fn\ x \ \Rightarrow\ (fn\ y \ \Rightarrow\ y\ x))\ 3\ (fn\ z \ \Rightarrow\ z + 1)\ = 4$

  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 $fn\ x \ \Rightarrow\ x\ 10$ "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

$$MNL \equiv (MN)L$$

Def. 21: Semantica eager dinamica di Fun

[fn dinamico eager]

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

[$apply$ dinamico eager]

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

Esempio: esempio

Def. 22: Semantica eager statica di Fun

$$\frac{[fn \text{ statico eager}]}{E \vdash fn\, x \;\Rightarrow\; M \rightsquigarrow (x, M, E)}$$

$$[apply \text{ statico eager}]\quad \frac{E \vdash M \rightsquigarrow (x, M', E') \qquad E \vdash N \rightsquigarrow v \qquad E'(x, v) \vdash M' \rightsquigarrow v'}{E \vdash MN \rightsquigarrow v'}$$

Lemma 1: Eager dinamico e statico in Fun

(Al contrario di Exp), si ha che:

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

Def. 23: Semantica lazy dinamica di Fun

$$[fn \text{ lazy statico}]\quad E \vdash fn\, x \;\Rightarrow\; M \rightsquigarrow (x, M)$$

$$[apply \text{ lazy statico}]\quad \frac{E \vdash M \rightsquigarrow (x, M') \qquad E(x, N) \vdash M' \rightsquigarrow v}{E \vdash M\; N \rightsquigarrow v}$$

Def. 24: Semantica lazy statica di Fun

$$[fn \text{ lazy statico}]\quad E \vdash fn\, x \;\Rightarrow\; M \rightsquigarrow (x, (M, E))$$

$$[apply \text{ lazy statico}]\quad \frac{E \vdash M \rightsquigarrow (x, M', E') \qquad E'(x, (N, E)) \vdash M' \rightsquigarrow v}{E \vdash M\; N \rightsquigarrow v}$$

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

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

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.

Def. 25: Curryficazione

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

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

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

Si trasforma quindi una funzione che prende due argomenti in una funzione ad un argomento che ritorna un'altra funzione t.c. $curry((f))(x)(y) = f(x, y)$.
(Il processo inverso prende il nome di **decurryficazione**).

Curryficazione in Fun

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

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

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

Lemma 2: Eager dinamico e statico in Fun

(Al contrario di Exp), si ha che:

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

ESEMPIO

$$\dfrac{Y \qquad X \qquad \dfrac{A \qquad B \qquad C}{Z}}{F}$$

Def. 26: Semantica operazionale di Fun lazy dinamico

$$[fn \text{ lazy statico}]$$
$$E \vdash fn\ x\ \Rightarrow\ M \rightsquigarrow (x, M)$$

$$[apply \text{ lazy statico}]$$
$$\dfrac{E \vdash M \rightsquigarrow (x, M') \qquad E(x, N) \vdash M' \rightsquigarrow v}{E \vdash M\ N \rightsquigarrow v}$$

Def. 27: Semantica operazionale di Fun lazy statico

$$[fn \text{ lazy statico}]$$
$$E \vdash fn\ x\ \Rightarrow\ M \rightsquigarrow (x, (M, E))$$

$$[apply \text{ lazy statico}]$$

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

Def. 28: **Curryficazione**

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

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

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

3. Lambda calcolo

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. 29: Numeri di Church in Fun

Più in generale, indicando con $M^n N$ 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:

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

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

Def. 30: $succ$ 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:

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

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

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:

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

che corrisponde al due di Church !

Def. 31: 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.

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

> 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 $dechurch(c2)$ in questo modo:

$$\begin{aligned}
dechurch(c2) &= (fn\ x\ \Rightarrow\ fn\ y\ \Rightarrow\ x\ (x\ y))\ (fn\ x\ \Rightarrow\ x + 1)\ 0 \\
&\text{sostituisco } x: \\
&= (fn\ y\ \Rightarrow\ (fn\ x\ \Rightarrow\ x + 1)((fn\ x\ \Rightarrow\ x + 1)\ y))\ 0 \\
&\text{sostituisco } y: \\
&= (fn\ x\ \Rightarrow\ x + 1)((fn\ x\ \Rightarrow\ x + 1)\ 0) \\
&\text{applico:} \\
&= (fn\ x\ \Rightarrow\ x + 1)(0 + 1) \\
&= (fn\ x\ \Rightarrow\ x + 1)\ 1 \\
&= 1 + 1 \\
&= 2
\end{aligned}$$

Thm. 4

Si ha:

$$dechurch(church(M)) \rightsquigarrow k \iff M \rightsquigarrow k$$

Def. 32: 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).

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

(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. 33: Prodotto di Church

Allo stesso modo, possiamo calcolare il **prodotto di Church**.
Sappiamo che $u \times v = \underbrace{u + u + u + \ldots + u}_{v \text{ volte}}$.
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: $fn\ x\ \Rightarrow\ (plus\ x\ v)$ (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 v volte a partire da zero.

$$times \equiv fn\, v \;\Rightarrow\; fn\, u \;\Rightarrow\; v\,(fn\, x \;\Rightarrow\; (plus\, x\, u))\; c_0$$

$$\equiv fn\, v\, u \;\Rightarrow\; v\,(plus_u)\; c_0$$

3.2. Primi 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. 34: **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);
```

4. 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. 35: **Grammatica del linguaggio** Imp

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

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

$$\begin{aligned} p, q ::= {} & skip \mid p; q \mid if\ M\ then\ p\ else\ q \mid while\ M\ do\ p \mid \\ & var\ x = M\ in\ p \mid x := M \end{aligned}$$

dove:

- $M,\ N \in Exp$
- $p,\ q \in Imp$
- $skip$ 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 semantici

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:

$$(S_1S_2)(x) = \begin{cases} S_2(x) & \text{se } x \in dom(S_2) \\ S_1(x) & \text{altrimenti} \end{cases}$$

Def. 36: **Semantiche operazionali**

Imp definisce due relazioni di valutazione:

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

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

$$E \vdash M, S \overset{M}{\rightsquigarrow} v$$

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

$$\overset{p}{\rightsquigarrow} Env \times Imp \times Store \times Store$$

$$E \vdash p, S \overset{p}{\rightsquigarrow} S'$$

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

- Costante:

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

- Variabile:

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

- Somma:

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

- Minore (vero):

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

- Minore (falso):

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

- Skip:

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

- Sequenza:

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

- If (condizione vera):

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

- If (condizione falsa):

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

- While (condizione vera):

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

- While (condizione falsa):

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

- Inizializzazione:

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

- Assegnamento:

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

> Notiamo che questa semantica non fa “garbage collection” !

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. 37: Grammatica del linguaggio *All*

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

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

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

$$\begin{aligned} p, q ::= {} & skip \mid p; q \mid if\ M\ then\ p\ else\ q \mid while\ M\ do\ p \mid \\ & var\ x = M\ in\ p \mid arr\ x = [M_0, \ldots, M_n]\ in\ p \mid V := M \mid \\ & proc\ y(x)\ is\ p\ in\ q \mid call\ y(M) \end{aligned}$$

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 $M_0, \ldots, M_n$ 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 semantici

Definiamo 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. 38: Semantiche operazionali

- introduciamo una nuova relazione di valutazione per le **espressioni assegnabili**:

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

- oltre a quelle (già esistenti) per le **espressioni** M:

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

$$E \vdash M, S \overset{M}{\rightsquigarrow} v$$

- e per i **programmi** p:

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

$$E \vdash p, S \overset{p}{\rightsquigarrow} S'$$

Oltre alle regole già definite in Imp, il linguaggio All definisce:

- Loc 1:

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

- Loc 2:

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

- Ref:

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

- Assign:

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

- Arr:

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

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

- Proc:

$$\frac{E\{(y, \langle x, p, E \rangle)\} \vdash q, S \overset{v}{\rightsquigarrow} S'}{E \vdash \text{proc } y(x) \text{ is } p \text{ in } q, S \overset{v}{\rightsquigarrow} S'}$$

- Call by value:

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

$(*)$ se $E(y) = \langle x, p, E' \rangle$ e l è nuova