SAPIENZA
UNIVERSITÀ DI ROMA

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

Linguaggi di Programmazione

Appunti integrati con il libro "TODO", TODO 1, Autore 2, ...

Author
Alessio Bandiera

20 novembre 2023

Indice

Informazioni e Contatti

Prerequisiti consigliati:

- Algebra
- TODO

Segnalazione errori ed eventuali migliorie:

Per segnalare eventuali errori e/o migliorie possibili, si prega di utilizzare il **sistema di Issues fornito da GitHub** all'interno della pagina della repository stessa contenente questi ed altri appunti (link fornito al di sotto), utilizzando uno dei template già forniti compilando direttamente i campi richiesti.

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se l'errore sia ancora presente nella versione più recente.

Licenza di distribuzione:

Contatti dell'autore e ulteriori link:

- Github: **https://github.com/ph04**
- Email: **alessio.bandiera02@gmail.com**
- LinkedIn: **Alessio Bandiera**

1 Induzione

1.1 Algebre induttive

1.1.1 Assiomi di Peano

Definizione 1.1.1.1: Assiomi di Peano

Gli **assiomi di Peano** sono 5 assiomi che definiscono l'insieme $\mathbb{N}$, e sono i seguenti:

i) $0 \in \mathbb{N}$

ii) $\exists \text{succ} : \mathbb{N} \to \mathbb{N}$, o equivalentemente, $\forall x \in \mathbb{N} \quad \text{succ}(x) \in \mathbb{N}$

iii) $\forall x, y \in \mathbb{N} \quad x \neq y \implies \text{succ}(x) \neq \text{succ}(y)$

iv) $\nexists x \in \mathbb{N} \mid \text{succ}(x) = 0$

v) $\forall S \subseteq \mathbb{N} \quad (0 \in S \land (\forall x \in S \quad \text{succ}(x) \in S)) \implies S = \mathbb{N}$

Esempio 1.1.1.1 ($\mathbb{N}$ di von Neumann)**.** Una rappresentazione dell'insieme dei numeri naturali $\mathbb{N}$ alternativa alla canonica

$$\mathbb{N} := \{0, 1, 2, \ldots\}$$

è stata fornita da John von Neumann. Indicando tale rappresentazione con $\aleph$, si ha che, per Neumann

$$\begin{gathered} 0_\aleph := \varnothing = \{\} \\ 1_\aleph := \{0_\aleph\} = \{\{\}\} \\ 2_\aleph := \{0_\aleph, 1_\aleph\} = \{\{\}, \{\{\}\}\} \\ \vdots \end{gathered}$$

e la funzione $\text{succ}_\aleph$ è definita come segue

$$\text{succ}_\aleph : \aleph \to \aleph : x_\aleph \mapsto x_\aleph \cup \{x_\aleph\} = \{\mu_\aleph \in \aleph \mid |\mu_\aleph| \leq |x_\aleph|\}$$

ed in particolare $\forall x_{\aleph} \in \aleph \quad |x_{\aleph}| + 1 = |\text{succ}_{\aleph}(x_{\aleph})|$.

È possibile verificare che tale rappresentazione di $\mathbb{N}$ soddisfa gli assiomi di Peano, in quanto

i) $0_{\aleph} := \varnothing \in \aleph$;

ii) $\exists \text{succ}_{\aleph} : \aleph \to \aleph$, definita precedentemente;

iii) $\forall x_{\aleph}, y_{\aleph} \in \aleph \quad x_{\aleph} \neq y_{\aleph} \implies |x_{\aleph}| \neq |y_{\aleph}| \implies |\text{succ}_{\aleph}(x_{\aleph})| \neq |\text{succ}_{\aleph}(y_{\aleph})| \implies \text{succ}_{\aleph}(x_{\aleph}) \neq \text{succ}_{\aleph}(y_{\aleph})$;

iv) per assurdo, sia $x_{\aleph} \in \aleph$ tale che $\text{succ}_{\aleph}(x_{\aleph}) = 0_{\aleph} := \varnothing$; per definizione $\text{succ}_{\aleph}(x_{\aleph}) := \{\mu_{\aleph} \in \aleph \mid |\mu_{\aleph}| \leq |x_{\aleph}|\}$, ma non esiste $\mu_{\aleph} \in \aleph$ con cardinalità minore o uguale 0, e dunque $\nexists x_{\aleph} \in \aleph \mid \text{succ}_{\aleph}(x_{\aleph}) = 0_{\aleph}$;

v) per assurdo, sia $S \subseteq \aleph$ tale che $0_{\aleph} \in S$ e $\forall x_S \in S \quad \text{succ}_{\aleph}(x_S) \in S$ ma $S \neq \aleph \iff \aleph - S \neq \varnothing \implies \exists \zeta_{\aleph} \in \aleph - S$, ed in particolare $\zeta_{\aleph} \neq 0_{\aleph}$; $\aleph$ è chiuso su $\text{succ}_{\aleph}$ per il secondo assioma di Peano, e dunque $\zeta_{\aleph} \neq 0_{\aleph} \implies \exists \zeta'_{\aleph} \in \aleph \mid \text{succ}_{\aleph}(\zeta'_{\aleph}) = \zeta_{\aleph}$, e sicuramente $\zeta'_{\aleph} \notin S$, poiché altrimenti $\zeta_{\aleph} \in S$ anch'esso in quanto S è chiuso rispetto a $\text{succ}_{\aleph}$; allora, ripetendo il ragionamento analogo per l'intera catena di predecessori, S risulterebbe essere vuoto, ma ciò è impossibile poiché $0_{\aleph} \in S$ in ipotesi $\lightning$.

Principio 1.1.1.1: Principio di Induzione

Sia P una proprietà che vale per $n = 0$, e dunque $P(0)$ è vera; inoltre, per ogni $n \in \mathbb{N}$ si ha che $P(n) \implies P(n+1)$; allora, $P(n)$ è vera per ogni $n \in \mathbb{N}$.

In simboli, utilizzando la notazione della logica formale, si ha che

$$\frac{P(0) \quad \dfrac{\forall n \in \mathbb{N} \quad P(n)}{P(n+1)}}{\forall n \in \mathbb{N} \quad P(n)}$$

Osservazione 1.1.1.1: Quinto assioma di Peano

Si noti che il quinto degli assiomi di Peano (Definizione 1.1.1.1) equivale al principio di induzione (Principio 1.1.1.1). Infatti, il quinto assioma afferma che qualsiasi sottoinsieme S di $\mathbb{N}$ avente lo 0, e caratterizzato dalla chiusura sulla funzione di successore succ, coincide con $\mathbb{N}$ stesso.

1.1.2 Algebre induttive

Definizione 1.1.2.1: Segnatura di una funzione

Data una funzione f, si definisce

$$f : A \to B$$

come **segnatura della funzione** f, dove A è detto **dominio**, denotato con $\mathrm{dom}\,(f)$ e B **codominio** di f.

Definizione 1.1.2.2: Algebra

Una **struttura algebrica**, o più semplicemente **algebra**, consiste di un insieme *non vuoto*, talvolta chiamato **insieme sostegno** (*carrier set* o *domain*), fornito di una o più operazioni su tale insieme, quest'ultime caratterizzate da un numero finito di assiomi da soddisfare.

Se A è l'insieme sostegno, e $\gamma_1, \dots \gamma_n$ sono delle operazioni definite su A, allora con

$$(A, \gamma_1, \dots, \gamma_n)$$

si indica l'algebra costituita da tali componenti, e questo simbolismo prende il nome di **segnatura dell'algebra**.

Esempio 1.1.2.1 (Strutture algebriche con singola operazione)**.** Esempi di strutture algebriche con un'operazione binaria sono i seguenti:

- semigruppi
- monoidi
- gruppi
- gruppi abeliani

Esempio 1.1.2.2 (Strutture algebriche con due operazioni)**.** Esempi di strutture algebriche con due operazioni binarie sono i seguenti:

- semianelli
- anelli
- campi

Definizione 1.1.2.3: Insieme unità

Con **insieme unità** si intende un qualsiasi insieme tale che $|\mathbb{1}| = 1$, e verrà indicato attraverso il simbolo $\mathbb{1}$.

Definizione 1.1.2.4: Funzione nullaria

Dato un insieme A, con **funzione nullaria** si intende una qualsiasi funzione con segnatura

$$f : \mathbb{1} \to A$$

Osservazione 1.1.2.1: Iniettività della funzione nullaria

Si noti che ogni funzione nullaria è iniettiva, poiché il dominio è costituito da un solo elemento.

Definizione 1.1.2.5: Algebra induttiva

Sia A un insieme, e siano $\gamma_1, \dots, \gamma_n$ funzioni definite su A di arbitraria arietà; allora, $(A, \gamma_1, \dots, \gamma_n)$ è definita **algebra induttiva** se si verificano le seguenti:

i) $\gamma_1, \dots, \gamma_n$ sono iniettive

ii) $\forall i, j \in [1, n] \mid i \neq j \quad \text{im}(\gamma_i) \cap \text{im}(\gamma_j) = \varnothing$, ovvero, le immagini dei costruttori sono a due a due disgiunte

iii) $\forall S \subseteq A \quad (\forall i \in [1, n], a_1, \dots a_k \in S, k \in \mathbb{N} \quad \gamma_i(a_1, \dots, a_k) \in S) \implies S = A$, o equivalentemente, in A non devono essere contenute algebre induttive.

Le funzioni $\gamma_1, \dots, \gamma_n$ prendono il nome di **costruttori dell'algebra**.

Osservazione 1.1.2.2: Terzo assioma delle algebre induttive

Si noti che nel terzo assioma della Definizione 1.1.2.5 anche $S = \varnothing$ è un valido sottoinsieme di A, ma poiché non esistono $a_1, \dots, a_k \in \varnothing$, in esso ogni qualificazione è vera a vuoto. Di conseguenza, nel momento in cui si ammette $S = \varnothing$ nel terzo assioma, l'algebra risulta essere non induttiva necessariamente (a meno dell'algebra vuota).

Di conseguenza, questo terzo assioma forza la necessità della presenza di un costruttore nullario all'interno di ogni algebra induttiva, in modo da non poter ammettere $S = \varnothing$, poiché l'algebra deve essere chiusa su ognuno dei suoi costruttori.

Esempio 1.1.2.3 (Numeri naturali)**.** $(\mathbb{N}, +)$ non è un algebra induttiva, poiché esistono $x_1, x_2, x_3, x_4 \in \mathbb{N}$ con $x_1 \neq x_3$ e $x_2 \neq x_4$ tali che $x_1 + x_2 = x_3 + x_4$; ad esempio, $2 + 3 = 5 = 1 + 4$, e $2 \neq 1$, $3 \neq 4$.

Esempio 1.1.2.4 (Algebra di Boole)**.** Dato l'insieme $B = \{\text{true}, \text{false}\}$, e la funzione $\neg$ definita come segue:

$$\neg : B \to B : x \mapsto \begin{cases} \text{false} & x = \text{true} \\ \text{true} & x = \text{false} \end{cases}$$

è possibile dimostrare che l'algebra $(B, \neg)$ non è induttiva; infatti, nonostante $\neg$ sia iniettiva, e la seconda proprietà della Definizione 1.1.2.5 sia vera a vuoto, $(B, \neg)$ non

presenta costruttore nullario, e dunque non può costituire un'algebra induttiva (si noti l'Osservazione 1.1.2.2).

Inoltre, si noti che anche l'algebra $(B, \neg, \text{true}_f)$, dove true_f è la funzione nullaria definita come segue:

$$\text{true}_f : \mathbb{1} \to B : x \mapsto \text{true}$$

non è induttiva, poiché

$$\text{true} \in \text{im}(\neg) \implies \text{im}(\text{true}_f) \cap \text{im}(\neg) \neq \varnothing$$

Esempio 1.1.2.5 (Algebre induttive)**.** Sia zero la funzione definita come segue

$$\text{zero} : \mathbb{1} \to \mathbb{N} : x \mapsto 0$$

e si prenda in esame l'algebra $(\mathbb{N}, \text{succ}, \text{zero})$; allora si ha che

i) succ e zero sono iniettive, poiché

- succ è iniettiva per il terzo assioma di Peano (Definizione 1.1.1.1)
- zero è iniettiva per l'Osservazione 1.1.2.1

ii) $\text{im}(\text{succ}) \cap \text{im}(\text{zero}) = (\mathbb{N} - \{0\}) \cap \{0\} = \varnothing$

iii) TODO

Definizione 1.1.2.6: Omomorfismo

Un **omomorfismo** è una funzione, tra due algebre con stessa segnatura, tale da preservarne le strutture.

Formalmente, siano $(A, \mu_1, \dots, \mu_n)$ e $(B, \delta_1, \dots, \delta_n)$ due algebre tali che ogni funzione μ_i abbia la stessa arietà e lo stesso numero di parametri esterni (denotati con k) di δ_i, per ogni $i \in [1, n]$; allora, una funzione $f : A \to B$ è detta **omomorfismo** tra le due algebre, se e solo se

$$\begin{gathered} f(\mu_1(a_\alpha, \dots, a_\beta, k_\alpha, \dots, k_\gamma)) = \delta_1(f(a_\alpha), \dots, f(a_\beta), k_\alpha, \dots, k_\gamma) \\ \vdots \\ f(\mu_n(a_\eta, \dots, a_\omega, k_\eta, \dots k_\nu)) = \delta_n(f(a_\eta), \dots, f(a_\omega), k_\eta, \dots, k_\nu) \end{gathered}$$

Se l'omomorfismo è da un'algebra in sé stessa, prende il nome di **endomorfismo**.

Esempio 1.1.2.6 (Omomorfismi)**.** Si considerino i gruppi $(\mathbb{R}, +)$ e $(\mathbb{R}_{>0}, \cdot)$, e sia f definita come segue:

$$f : \mathbb{R} \to \mathbb{R}_{>0} : x \mapsto e^x$$

allora, si ha che

$$\forall x, y \in \mathbb{R} \quad f(x) \cdot f(y) = e^x \cdot e^y = e^{x+y} = f(x + y)$$

dunque f è un omomorfismo di gruppi.

Proposizione 1.1.2.1: Composizione di omomorfismi

Siano $(A, \gamma_1, \ldots, \gamma_n)$, $(B, \delta_1, \ldots, \delta_n)$ e $(C, \mu_1, \ldots, \mu_n)$ tre algebre aventi la stessa segnatura, e siano $f : A \to B$ e $g : B \to C$ due omomorfismi; allora $g \circ f$ è un omomorfismo.

Dimostrazione. Per dimostrare la tesi, è necessario dimostrare che

$$\forall i \in [1, k] \quad (g \circ f)(\gamma_i(a_{\alpha_i}, \ldots, a_{\beta_i})) = \mu_i((g \circ f)(a_{\alpha_i}), \ldots (g \circ f)(a_{\beta_i}))$$

allora, si ha che

$$\forall i \in [1, k] \quad g(f(\gamma_i(a_{\alpha_i}, \ldots, a_{\beta_i}))) = g(\delta_i(f(a_{\alpha_i}), \ldots, f(a_{\beta_i}))) = \mu_i(g(f(a_{\alpha_i})), \ldots, g(f(a_{\beta_i})))$$

e dunque segue la tesi. □

Definizione 1.1.2.7: Isomorfismo

Un **isomorfismo** è un omomorfismo biettivo.

Se tra due strutture algebriche A e B esiste un isomorfismo, tali strutture sono dette **isomorfe**, ed è indicato col seguente simbolismo:

$$A \cong B$$

Se l'isomorfismo è da un'algebra in sé stessa, prende il nome di **automorfismo**.

Esempio 1.1.2.7 (Isomorfismi)**.** Si consideri l'omomorfismo dell'Esempio 1.1.2.6; si noti che

$$\forall x, y \in \mathbb{R} \mid x \neq \mathbb{R} \quad e^x \neq e^y \implies f(x) \neq f(y)$$

e dunque f è iniettiva; inoltre

$$\forall y \in \mathbb{R}_{>0} \quad \exists x \in \mathbb{R} \mid f(x) = e^x = y \iff y = \ln(x)$$

e dunque f è suriettiva. Allora, f è biettiva, e poiché è un omomorfismo, risulta essere un isomorfismo.

Osservazione 1.1.2.3: L'automorfismo dell'identità

Sia $(A, \gamma_1, \ldots, \gamma_n)$ un'algebra, e si consideri la funzione identità su A

$$\mathrm{id} : A \to A : x \mapsto x$$

Si noti che

$$\forall i \in [1, n] \quad \mathrm{id}(\gamma_i(a_{\alpha_i}, \ldots, a_{\beta_i})) = \gamma_i(a_{\alpha_i}, \ldots, a_{\beta_i}) = \gamma_i(\mathrm{id}(a_{\alpha_i}), \ldots, \mathrm{id}(a_{\beta_i}))$$

dunque id è un endomorfismo su A, e poiché la funzione identità è sia iniettiva che suriettiva, id è sempre un automorfismo per qualsiasi algebra A.

1.1.3 Lemma di Lambek

Proposizione 1.1.3.1: Biettività ed invertibilità

Sia f una funzione, allora f è biettiva se e solo se esiste la sua inversa.

Dimostrazione. Omessa. □

Proposizione 1.1.3.2: Algebre con stessa segnatura

Data un'algebra induttiva $(A, \gamma_1, \dots, \gamma_n)$, per ogni algebra — *non necessariamente induttiva* — $(B, \delta_1, \dots, \delta_n)$, avente la stessa segnatura di A, esiste unico un omomorfismo $f : A \to B$.

Dimostrazione. Omessa. □

Corollario 1.1.3.1: Identità nelle algebre induttive

Sia A un'algebra induttiva, e sia $f : A \to A$ un suo endomorfismo; allora $f = \mathrm{id}$.

Dimostrazione. Si noti che, per l'Osservazione 1.1.2.3, esiste l'automorfismo id su A, ma per la Proposizione 1.1.3.2, poiché A è un'algebra induttiva, esiste un unico omomorfismo $f : A \to A$, che deve allora necessariamente coincidere con id. □

Lemma 1.1.3.1: Lemma di Lambek (versione ridotta)

Siano $(A, \gamma_1, \dots, \gamma_n)$ e $(B, \delta_1, \dots, \delta_n)$ due algebre induttive aventi stessa segnatura; allora $A \cong B$.

Dimostrazione. Per la Proposizione 1.1.3.2, poiché A è un'algebra induttiva, e B ha la stessa segnatura di A, esiste unico un omomorfismo $f : A \to B$; si noti però che è vero anche il viceversa, infatti poiché B è un'algebra induttiva, e A ha la stessa segnatura di B, esiste unico un omomorfismo $g : B \to A$. Si noti che, per la Proposizione 1.1.2.1, la funzione $g \circ f : A \to A : x \mapsto g(f(x))$ risulta essere un omomorfismo; allora, poiché A è un'algebra induttiva, per il Corollario 1.1.3.1, si ha che

$$g \circ f = \mathrm{id} \iff f^{-1} = g \iff f \text{ biettiva}$$

(si veda la Proposizione 1.1.3.1) e poiché f è un omomorfismo, segue che $A \cong B$. □

1.2 Strutture dati induttive

1.2.1 Liste

Definizione 1.2.1.1: Liste

Una **lista** è una collezione ordinata di elementi, e l'insieme delle liste di lunghezza finita viene denotato con `List<T>`, dove `T` è il tipo degli elementi che le liste contengono, ed il simobolo `T` verrà identificato con l'insieme di tutti gli oggetti aventi tipo `T`.

Dati $a_1, \dots, a_n \in \texttt{T}$, una lista $l \in \texttt{List<T>}$ contenente tali elementi può essere rappresentata come segue:

$$[a_1, \dots, a_n]$$

Definizione 1.2.1.2: Algebra delle liste finite

L'**algebra delle liste finite** è definita come segue:

$$(\texttt{List<T>}, \text{empty}, \text{cons})$$

dove i costruttori sono i seguenti:

$$\text{empty} : \mathbb{1} \to \texttt{List<T>} : x \mapsto []$$
$$\text{cons} : \texttt{List<T>} \times \texttt{T} \to \texttt{List<T>} : ([a_1, \dots, a_n], x) \mapsto [a_1, \dots, a_n, x]$$

Proposizione 1.2.1.1: Liste finite induttive

L'algebra delle liste finite è induttiva.

Dimostrazione. Si noti che:

- empty ha dominio in $\mathbb{1}$, e poiché questo contiene un solo elemento, empty è necessariamente iniettiva;
- $\forall l, l' \in \texttt{List<T>}, x, x' \in \texttt{T} \quad \text{cons}(l, x) = \text{cons}(l', x') \implies \begin{cases} l = l' \\ x = x' \end{cases}$ altrimenti l ed l' avrebbero avuto lunghezze diverse, oppure avrebbero contenuto diversi elementi;
- $\text{im}\,(\text{empty}) \cap \text{im}\,(\text{cons}) = \varnothing$, poiché solo empty può restituire `[]`, in quanto cons restituisce sempre una lista contenente almeno l'elemento fornito in input;
- sia $S \subseteq \texttt{List<T>}$ tale da essere chiuso rispetto a cons, e contenente la lista vuota; per assurdo, sia $\texttt{List<T>} - S \neq \varnothing \implies \exists l \in \texttt{List<T>} - S$, ma `List<T>` è chiuso rispetto a cons, ed in particolare $\exists x \in \texttt{T}, l' \in \texttt{List<T>} \mid \text{cons}(l', x) = l$, ma poichè $l \notin S$, allora necessariamente $l' \notin S$, poiché S è chiuso rispetto a cons, e quindi $l' \in S \implies l \in S$, ma l è stato scelto in $\texttt{List<T>} - S$; ripetendo tale ragionamento induttivamente, si ottiene che S è vuoto, ma questo è impossibile poiché $[] \in \texttt{List<T>}$ in ipotesi $\lightning$.

Dunque, l'algebra delle liste finite risulta essere induttiva. □

Osservazione 1.2.1.1: Algebra delle liste infinite

Se all'algebra delle liste finite venissero aggiunte anche le liste infinite, l'algebra risultante non sarebbe induttiva, in quanto conterrebbe l'algebra delle liste finite, la quale è induttiva per la Proposizione 1.2.1.1, e verrebbe dunque contraddetto il terzo assioma della Definizione 1.1.2.5.

Osservazione 1.2.1.2: Concatenazione di liste finite

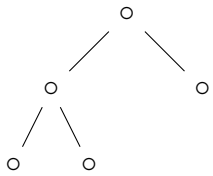
È possibile estendere l'algebra delle liste finite per supportare l'operazione di concatenazione tra liste, come segue:

$$\text{concat} : \texttt{List<T>} \times \texttt{List<T>} \to \texttt{List<T>} : (l, l') \mapsto \begin{cases} l' & l = \texttt{[]} \\ \text{cons}(x, \text{concat}(t, l')) & \exists x \in \texttt{T}, t \in \texttt{List<T>} \mid l = \text{cons}(t, x) \end{cases}$$

1.2.2 Alberi binari

Definizione 1.2.2.1: Albero binario

Un **albero binario** è una struttura dati che è possibile rappresentare graficamente come segue:



Il primo nodo, poiché non è figlio di nessuno, è detto **radice**, e poiché l'albero è *binario*, ogni nodo ha 0 — nel qual caso è definito **foglia** — oppure 2 figli. L'insieme degli alberi binari viene denotato con `B-tree`.

Definizione 1.2.2.2: Algebra degli alberi binari finiti

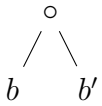
L'**algebra degli alberi binari finiti** è definita come segue:

$$(\texttt{B-tree}, \text{leaf}, \text{branch})$$

dove i costruttori sono i seguenti:

$$\text{leaf} : \mathbb{1} \to \texttt{B-tree} : x \mapsto \circ$$

$$\text{branch} : \texttt{B-tree} \times \texttt{B-tree} \to \texttt{B-tree} : (b, b') \mapsto$$



Proposizione 1.2.2.1: Alberi binari finiti induttivi

L'algebra degli alberi binari finiti è induttiva.

Dimostrazione. Omessa. □

Osservazione 1.2.2.1: Algebra degli alberi binari infiniti

Analogamente all'Osservazione 1.2.1.1, l'algebra degli alberi binari finiti ed infiniti non è induttiva.

Osservazione 1.2.2.2: Nodi di un albero binario finito

È possibile estendere l'algebra degli alberi binari finiti per supportare l'operazione per contare i nodi di un albero, come segue:

$$\text{nodes} : \texttt{B-tree} \to \mathbb{N} : b \mapsto \begin{cases} 1 & b = \circ \\ 1 + \text{nodes}(t) + \text{nodes}(t') & \exists t, t' \in \texttt{B-tree} \mid b = \text{branch}(t, t') \end{cases}$$

Osservazione 1.2.2.3: Foglie di un albero binario finito

È possibile estendere l'algebra degli alberi binari finiti per supportare l'operazione per contare le foglie di un albero, come segue:

$$\text{leaves} : \texttt{B-tree} \to \mathbb{N} : b \mapsto \begin{cases} 1 & b = \circ \\ \text{leaves}(t) + \text{leaves}(t') & \exists t, t' \in \texttt{B-tree} \mid b = \text{branch}(t, t') \end{cases}$$

Teorema 1.2.2.1: Relazione tra foglie e nodi

Ogni albero binario finito, avente n foglie, ha $2n - 1$ nodi.

Dimostrazione. La seguente dimostrazione procede per *induzione strutturale*, dunque effettuando l'induzione sulla morfologia della struttura dati, e non sul numero n di foglie.

Caso base. Il caso base è costituito dunque da $\circ$, l'albero ottenuto attraverso il costruttore nullario leaf, ed infatti si ha che

$$\text{leaves}(\circ) = 1 \implies 2 \cdot 1 - 1 = 1$$

e $\circ$ ha esattamente 1 nodo.

Ipotesi induttiva. Un albero binario finito, avente n foglie, ha $2n - 1$ nodi.

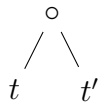
Passo induttivo. Sia $b \in$ `B-tree` tale che esistano $t, t' \in$ `B-tree` tali che $\text{branch}(t, t') = b$, e siano

$$\begin{cases} \text{leaves}(t) = n \\ \text{leaves}(t') = n' \end{cases}$$

Si noti che, per ipotesi induttiva, si ha che

$$\begin{cases} \text{nodes}(t) = 2n - 1 \\ \text{nodes}(t') = 2n' - 1 \end{cases}$$

ed inoltre, poiché $b = \text{branch}(t, t')$, b ha la forma seguente



dunque, per definizione di leaves si ha che

$$\text{leaves}(b) = \text{leaves}(t) + \text{leaves}(t') = n + n'$$

e, dalla morfologia di b, segue che

$$\text{nodes}(b) = \text{nodes}(t) + \text{notes}(t') + 1 = 2n - 1 + 2n' - 1 + 1 = 2(n + n') - 1$$

ed è quindi verificata la tesi, poiché

$$\text{leaves}(b) = n + n' \implies \text{nodes}(b) = 2(n + n') - 1$$

□

2 Paradigma funzionale

2.1 Grammatiche

2.1.1 Definizioni

Definizione 2.1.1.1: Grammatica

Una **grammatica** è un insieme di regole che definiscono come manipolare un insieme di stringhe, agendo su elementi sintattici detti **termini**.

Definizione 2.1.1.2: Variabili

Data una grammatica di G, con Var si indica l'**insieme delle variabili** di G.

Definizione 2.1.1.3: Valori

Data una grammatica, con Val si indica l'**insieme dei valori** che ogni termine della grammatica può assumere.

Definizione 2.1.1.4: Forma di Backus-Naur (BNF)

La **forma di Backus-Naur** (*Backus-Naur Form*) è una notazione utilizzata per descrivere la sintassi di grammatiche, ed è definita come segue:

$$\texttt{<symbol>}, \ldots, \texttt{<symbol>} ::= \texttt{\_expression\_} \mid \ldots \mid \texttt{\_expression\_}$$

dove

- `<symbol>` è una *metavariabile non terminale*, ovvero, può essere sostituita con regole definite dalla grammatica; si noti che le regole possono essere utilizzate ricorsivamente;
- il simbolo ::= indica che ciò che è posto alla sua sinistra deve essere sostituito con ciò che è alla sua destra;
- `_expression_` è un espressione che verrà usata per rimpiazzare le metavariabili non terminali, attraverso le regole definite dalla grammatica; le *metavarabili* che compongono le espressioni possono essere **costanti**, **variabili**, **termini** o **espressioni** contenenti combinazioni delle precedenti, presentando eventualmente anche operazioni sintattiche specifiche.

2.2 *Exp*: un primo linguaggio

2.2.1 Definizioni

Definizione 2.2.1.1: Grammatica *Exp*

Sia *Exp* la seguente grammatica:

$$M, N ::= \texttt{0} \mid \texttt{1} \mid \ldots \mid x \mid M \texttt{ + } N \mid M \texttt{ * } N$$

essa definisce le regole per utilizzare i numeri in $\mathbb{N}$, ammettendo inoltre le operazioni sintattiche di `+` e `*`.

Osservazione 2.2.1.1: Metavariabili di *Exp*

Sia *Exp* la grammatica della Definizione 2.2.1.1; allora, le sue metavariabili sono le seguenti:

- *costanti*: $\texttt{0}, \texttt{1}, \ldots$
- *variabili*: x
- *termini*: M ed N
- *espressioni*: M `+` N e M `*` N (si noti che tecnicamente anche le precedenti sono espressioni)

Dunque, segue che

$$\begin{aligned} \text{Var} &= \{x\} \\ \text{Val} &= \{\texttt{0}, \texttt{1}, \ldots\} \end{aligned}$$

Definizione 2.2.1.2: Linguaggio di una grammatica

Sia G una grammatica; allora, il suo **linguaggio** è l'insieme delle stringhe che è possibile costruire attraverso le regole di G.

Esempio 2.2.1.1 (Linguaggio di *Exp*)**.** Sia *Exp* la grammatica della Definizione 2.2.1.1; in essa, considerando ad esempio le stringhe `"4"` e `"23"`, si può ottenere la stringa

$$\texttt{+}(\texttt{"4"}, \texttt{"23"}) = \texttt{"4 + 23"}$$

dove la *polish notation* — alla sinistra dell'uguale — e la forma sintattica canonica — alla sua destra — verranno utilizzate intercambiabilmente, poiché puro *syntactic sugar*.

Osservazione 2.2.1.2: Valutazione di *Exp*

Si prenda in considerazione la grammatica *Exp* della Definizione 2.2.1.1; su di essa, è possibile definire ricorsivamente una funzione eval, in grado di valutare le stringhe che tale grammatica può produrre, come segue:

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

Osservazione 2.2.1.3: Ambiguità di *Exp*

Si prenda in considerazione la grammatica *Exp* della Definizione 2.2.1.1; si noti che tale grammatica è ambigua, poichè ad esempio

$$+(\texttt{"5"}, *(\texttt{"6"}, \texttt{"7"})) = \texttt{"5 + 6 * 7"} = *(+(\texttt{"5"}, \texttt{"6"}), \texttt{"7"})$$

e da ciò segue anche che $\text{im}\,(+) \cap \text{im}\,(*) \neq \varnothing$.

Osservazione 2.2.1.4: Disambiguazione di *Exp*

Si noti che l'ambiguità trattata nell'Osservazione 2.2.1.3 non permetterebbe di poter definire la funzione eval, descritta nell'Osservazione 2.2.1.2. Dunque, per risolvere tale ambiguità, a meno di parentesi (che *non* sono definite all'interno della grammatica) o dell'esplicitazione della composizione di funzioni utilizzata, verrà sottintesa la normale precedenza degli operatori aritmetica durante la valutazione delle stringhe.

2.2.2 Assegnazioni

Definizione 2.2.2.1: Clausola *let*

Sia G una grammatica; allora, è possibile definire su G una funzione *let*, come segue:

$$let : \text{Var} \times G \times G \to G$$

e verrà utilizzata attraverso la sintassi

$$let\ \texttt{*variable*} = \texttt{\_expression\_}_1\ in\ \texttt{\_expression\_}_2$$

dove alla variabile `*variable*` verrà assegnata l'espressione `_expression_`$_1$ *durante la valutazione* di `_expression_`$_2$; la variabile `*variable*`, all'interno di `_expression_`$_2$, prende il nome di **variabile locale**.

Una variabile alla quale non è stata assegnata nessuna espressione prende il nome di **variable libera** (*free variable* in inglese); una variabile non libera è detta **variabile legata** (*bound variable*). L'azione di legare o liberare una variabile è detta **variable binding**.

Definizione 2.2.2.2: Estensione di *Exp*

Sia *Exp* la seguente estensione della grammatica presente all'interno della Definizione 2.2.1.1:

$$M, N ::= k \mid x \mid M \texttt{ + } N \mid M \texttt{ * } N \mid let\ x = M\ in\ N$$

In essa, sono presenti:

- *costanti*: indicate con k, che sta ad indicare che in *Exp* è ammessa qualsiasi costante; di fatto, è possibile pensare a k come una funzione definita come segue:

$$k : \mathbb{N} \to Exp : x \mapsto \texttt{"x"}$$

- *variabili*: x
- *termini*: M ed N
- *espressioni*: M `+` N, M `*` N e $let\ x = M\ in\ N$

Osservazione 2.2.2.1: Convenzione per le espressioni di grammatiche

Si noti che, d'ora in avanti, le espressioni delle grammatiche presentate non verranno indicate con il `"virgolettato"`, poiché verrà sottointesa la trasformazione tra i simboli, sintattici, delle grammatiche, e le vere operazioni e/o costanti che rappresentano, semanticamente.

Esempio 2.2.2.1 (Clausole *let*)**.** Sia *Exp* la grammatica della Definizione 2.2.2.2; un esempio di espressione su *Exp*, che utilizza la clausola *let* della Definizione 2.2.2.1, è la seguente:

$$let\ x = 3\ in\ (x + 1)$$

e nel momento in cui viene valutata tale espressione, si ha che

$$x = 3 \implies x + 1 = 3 + 1 = 4$$

e dunque il valore dell'espressione è 4.

Esempio 2.2.2.2 (Variabili libere)**.** Sia *Exp* la grammatica della Definizione 2.2.2.2, ed ammettendo la variabile y in essa, si consideri la seguente espressione:

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

in essa, la variabile x è posta pari a 3, ma ad y non è stato assegnato alcuna espressione, e dunque risulta essere una variabile libera.

Osservazione 2.2.2.2: Ambiguità di *let*

Sia *Exp* la grammatica della Definizione 2.2.2.2, e si consideri la sua seguente espressione

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

per qualche espressione $M \in Exp$, e due variabili $x, y \in \text{Var}$, ammettendo dunque y tra le variabili di *Exp*; si noti che tale espressione è ambigua, poiché potrebbe equivalere a

$$(let\ x = M\ in\ x) + y$$

oppure a

$$let\ x = M\ in\ (x + y)$$

Per convenzione, all'interno di questi appunti, in assenza di parentesi che descrivano la precedenza degli operatori, si assume la precedenza della seconda espressione mostrata.

Osservazione 2.2.2.3: Variabili libere di *Exp*

Sia *Exp* la grammatica della Definizione 2.2.2.2; su di essa, è possibile definire, ricorsivamente, una funzione in grado di restituire le variabili free di una data espressione, come segue:

$$\text{free} : Exp \to \mathcal{P}(\text{Var}) : e \mapsto \begin{cases} \varnothing & \exists \eta \in \mathbb{N} \mid e = k(\eta) \\ \{x\} & \exists x \in \text{Var} \mid e = x \\ \text{free}(M) \cup \text{free}(N) & \exists M, N \in Exp \mid e = M + N \vee e = M * N \\ \text{free}(M) \cup (\text{free}(N) - \{x\}) & \exists x \in \text{Var}, M, N \in Exp \mid e = (let\ x = M\ in\ N) \end{cases}$$

2.2.3 Ambienti

Definizione 2.2.3.1: Ambiente di una grammatica

Data una grammatica tale che Val sia un insieme finito, un **ambiente** della grammatica è una funzione della forma

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

che associa dunque una variabile ad un possibile valore che può assumere (la notazione fin indica che E è una funzione *parziale*, dunque non necessariamente definita su tutto il dominio). L'insieme di tutti gli ambienti della grammatica è denotato con

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

In simboli, gli ambienti verranno scritti come insiemi di coppie (x, k) con $x \in \text{Var}, k \in \text{Val}$, che descriveranno la mappa definita dall'ambiente stesso. Si noti che, per un certo ambiente $E \in \text{Env}$, $E(x)$ è indefinito per ogni $x \in \text{Var} - \text{dom}(E)$.

Esempio 2.2.3.1 (Ambienti di *Exp*)**.** Sia *Exp* la grammatica della Definizione 2.2.2.2;

allora, un possibile ambiente di *Exp*, denotato con $E \in \text{Env}$, è il seguente:

$$E := \{(z, 3), (y, 9)\}$$

ed esso esprime la possibilità che in *Exp* z possa essere valutato pari a 3, mentre y pari a 9 (tecnicamente, le variabili z ed y andrebbero ammesse all'interno della grammatica, ma d'ora in avanti tale precisazione verrà sottintesa).

Definizione 2.2.3.2: Concatenazione di ambienti

Siano E_1 ed E_2 due ambienti di una grammatica; allora, si definisce **concatenazione** di E_1 ed E_2 la seguente funzione

$$E_1E_2 : \text{Env} \times \text{Env} \to \text{Env} : x \mapsto \begin{cases} E_2 & x \in \text{dom}(E_2) \vee x \in \text{dom}(E_1) \cap \text{dom}(E_2) \\ E_1(x) & x \in \text{dom}(E_1) \end{cases}$$

dunque, nella concatenazione E_2 sovrascrive le tuple che sono presenti anche in E_1.

Esempio 2.2.3.2 (Concatenazioni di ambienti)**.** Sia *Exp* la grammatica descritta all'interno della Definizione 2.2.2.2, e siano

$$E_1 := \{(z, 3), (y, 9)\}$$
$$E_2 := \{(z, 4)\}$$

due suoi ambienti; allora si ha che

$$E_1E_2 := \{(z, 4), (y, 9)\}$$

2.2.4 Semantica operazionale di *Exp*

Definizione 2.2.4.1: Semantica operazionale di una grammatica

Data una grammatica G, si definisce **semantica operazionale** della grammatica una relazione, indicata col simbolo $\rightsquigarrow$, definita come segue:

$$\rightsquigarrow \subseteq \text{Env} \times G \times \text{Val}$$

Un elemento $(E, M, v) \in \rightsquigarrow$ è detto **giudizio operazionale**, e viene scritto attraverso il seguente simbolismo:

$$E \vdash M \rightsquigarrow v$$

e si legge "valutando M, nell'ambiente E, si ottiene v".

Proposizione 2.2.4.1: Semantica operazionale di *Exp*

Sia *Exp* la grammatica definita all'interno della Definizione 2.2.2.2, e sia E un suo ambiente; allora, si definiscono le seguenti regole operazionali:

- **costanti**:

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

- **variabili**:

$$\forall x \in \text{Var} \quad \exists v \in \text{Val} \mid E(x) = v \implies [vars]\ E \vdash x \rightsquigarrow v$$

- **somme**:

$$\forall v', v'' \in \text{Val}, M, N \in Exp \quad \exists v \in \text{Val} \mid v = v' + v'' \implies [plus]\ \frac{E \vdash M \rightsquigarrow v' \quad E \vdash N \rightsquigarrow v''}{E \vdash M + N \rightsquigarrow v}$$

- **prodotti**:

$$\forall v', v'' \in \text{Val}, M, N \in Exp \quad \exists v \in \text{Val} \mid v = v' \cdot v'' \implies [times]\ \frac{E \vdash M \rightsquigarrow v' \quad E \vdash N \rightsquigarrow v''}{E \vdash M * N \rightsquigarrow v}$$

- **dichiarazioni ed assegnazioni**:

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

Definizione 2.2.4.2: Equivalenza operazionale

Sia G una grammatica, e siano M ed N due sue espressioni; queste sono dette **operazionalmente equivalenti**, se è vera la seguente proposizione:

$$\forall E \in \text{Env}, v \in \text{Val} \quad E \vdash M \rightsquigarrow v \iff E \vdash N \rightsquigarrow v$$

e viene indicato con $M \sim N$.

Definizione 2.2.4.3: Albero di valutazione

Con **albero di valutazione** di un'espressione e, si definisce l'albero, composto da inferenze logiche, ottenuto dalla valutazione di e.

Osservazione 2.2.4.1: Ambiente iniziale

Per qualsiasi grammatica — a meno di specifiche — si assume che, all'interno di una valutazione, l'ambiente iniziale sia $\varnothing \in \text{Env}$.

Esempio 2.2.4.1 (Alberi di valutazione su *Exp*)**.** Sia *Exp* la grammatica definita all'in-

terno della Definizione 2.2.2.2; allora, l'albero di valutazione dell'espressione

$$let\ x = 3\ in\ x + 4$$

è il seguente

$$\dfrac{\varnothing \vdash 3 \rightsquigarrow 3 \quad \dfrac{\{(x,3)\} \vdash x \rightsquigarrow 3 \quad \{(x,3)\} \vdash 4 \rightsquigarrow 4}{\{(x,3)\} \vdash x + 4 \rightsquigarrow 7}}{\varnothing \vdash let\ x = 3\ in\ x + 4 \rightsquigarrow 7}$$

e l'espressione è valutabile poiché $x \in \mathrm{dom}(\{(x,1)\}) = \{x\}$.

2.2.5 Valutazioni e scoping

Definizione 2.2.5.1: Valutazione eager

Data una grammatica, la **valutazione eager** valuta una data espressione della grammatica non appena questa viene legata ad una variabile. In simboli, la valutazione eager verrà indicata con il pedice E.

Definizione 2.2.5.2: Valutazione lazy

Data una grammatica, la **valutazione lazy** valuta una data espressione della grammatica solo quando il suo valore viene richiesto da un'altra espressione. In simboli, la valutazione lazy verrà indicata con il pedice L.

Definizione 2.2.5.3: Scoping statico

Data una grammatica, la valutazione a **scoping statico** (*lexical scope*) valuta le variabili TODO. In simboli, lo scoping statico verrà indicato con il pedice S.

Definizione 2.2.5.4: Scoping dinamico

Data una grammatica, la valutazione a **scoping dinamico** valuta le variabili utilizzando l'ambiente definito in tempo di valutazione. TODO NON MI PIACE CAMBIA. In simboli, lo scoping dinamico verrà indicato con il pedice D.

Definizione 2.2.5.5: Equivalenza di semantiche operazionali

Data una grammatica, due sue semantiche operazionali sono dette **equivalenti** se, presa una qualunque espressione di G, quando questa viene valutata attraverso le due semantiche, produce lo stesso risultato.

In simboli, data una grammatica G, e due sue semantiche operazionali A e $\mathbb{B}$, se queste sono equivalenti, la loro equivalenza viene denotata con il seguente simbolismo:

$$G_{\mathrm{A}} \equiv G_{\mathrm{B}}$$

Lemma 2.2.5.1: Exp_{ES} **e** Exp_{ED}

Sia *Exp* la grammatica definita all'interno della Definizione 2.2.2.2, avente le clausole definite nell'Proposizione 2.2.4.1; allora, si ha che

$$Exp_{\text{ES}} \equiv Exp_{\text{ED}}$$

Dimostrazione. Omessa. □

Esempio 2.2.5.1 (Exp_{E})**.** Sia *Exp* la grammatica definita nella Definizione 2.2.2.2, e si consideri la seguente espressione:

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

essa, valutata attraverso valutazione eager, produce il seguente albero di derivazione:

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

Proposizione 2.2.5.1: Exp_{LD}

Sia *Exp* la grammatica definita nella Definizione 2.2.2.2; per poter valutare le sue espressioni in maniera lazy dinamica, è necessario ridefinire alcune regole di inferenza definite all'interno dell'Proposizione 2.2.4.1:

- l'insieme degli ambienti di *Exp* viene ridefinito come segue:

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

  dunque gli ambienti possono associare delle variabili anche a delle espressioni, in modo da poter ritardare la valutazione di quest'ultime fin quando non diventa strettamente necessario conoscerne il valore;

- $\forall E \in \text{Env}, x \in \text{Var} \quad x \in \text{dom}(E) \wedge M := E(x) \implies \dfrac{E \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v}$, per rendere M calcolabile, attraverso l'ambiente corrente, nel momento in cui viene assegnata ad una variabile;

- $\forall x \in \text{Var}, M, N \in Exp \quad [let]\ \dfrac{E\{(x,M)\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$, in modo da ritardare la valutazione di M.

Si noti che tale valutazione utilizza lo scoping dinamico, poiché viene utilizzato l'ambiente corrente per valutare le variabili.

Proposizione 2.2.5.2: Exp_{LS}

Sia *Exp* la grammatica definita all'interno della Definizione 2.2.2.2; per poter valutare le sue espressioni in maniera lazy statica, è necessario ridefinire alcune regole di inferenza definite all'interno dell'Proposizione 2.2.4.1:

- l'insieme degli ambienti di *Exp* viene ridefinito come segue:

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

  dunque gli ambienti possono associare delle variabili anche a delle tuple contenenti espressioni ed ambienti; in particolare, per una tupla $(x, (M, E))$, si ha che x vale M valutata nell'ambiente E; questa correzione nella definizione di Env permette di tenere traccia degli ambienti in cui sono state fatte le assegnazioni;

- $\forall E \in \text{Env}, x \in \text{Var} \quad x \in \text{dom}(E) \land (M, E') := E(x) \implies \dfrac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v}$, affinché sia possibile valutare le variabili esattamente nell'ambiente in cui gli è stata assegnata l'espressione M, e non nell'ambiente corrente;

- $\forall x \in \text{Var}, M, N \in Exp \quad [let]\ \dfrac{E\{(x, (M, E))\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$, in modo da salvare anche l'ambiente in cui alla variabile x è stata assegnata l'espressione M.

Si noti che tale valutazione utilizza lo scoping statico, poiché vengono salvati anche gli ambienti in cui sono state fatte le assegnazioni.

Lemma 2.2.5.2: Exp_{LS} **e** Exp_{LD}

Sia *Exp* la grammatica definita nella Definizione 2.2.2.2; allora, si ha che

$$Exp_{\text{LS}} \not\equiv Exp_{\text{LD}}$$

Dimostrazione. Si consideri l'espressione definita nell'Esempio 2.2.5.1; essa, valutata attraverso valutazione lazy dinamica, produce il seguente albero di derivazione:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{(x,3),(y,x)\}\{(x,7)\} \vdash 7 \rightsquigarrow 7}{\{(x,3),(y,x)\}\{(x,7)\} \vdash x \rightsquigarrow 7} \quad \dfrac{\{(x,3),(y,x)\}\{(x,7)\} \vdash 7 \rightsquigarrow 7}{\{(x,3),(y,x)\}\{(x,7)\} \vdash x \rightsquigarrow 7}}{\{(x,3),(y,x)\}\{(x,7)\} \vdash y + x \rightsquigarrow 14}}{\{(x,3),(y,x)\} \vdash let\ x = 7\ in\ y + x \rightsquigarrow 14}}{\{(x,3)\} \vdash let\ y = x\ in\ (let\ x = 7\ in\ y + x) \rightsquigarrow 14}}{\varnothing \vdash let\ x = 3\ in\ (let\ y = x\ in\ (let\ x = 7\ in\ y + x)) \rightsquigarrow 14}$$

Differentemente, valutando tale espressione attraverso valutazione lazy statica, produce

il seguente albero di derivazione:

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

dove

$$E := \{(x,(3,\varnothing))\}$$
$$E' := E\{(y,(x,E))\}$$
$$E'' := E'\{(x,(7,E'))\}$$

Allora, poiché le due valutazioni producono risultati differenti, per la Definizione 2.2.5.5, segue la tesi. □

2.3 *Fun*: un linguaggio funzionale

2.3.1 Definizioni

Definizione 2.3.1.1: Clausola fn

Sia G una grammatica; allora, è possibile definire su G una funzione fn, come segue:

$$fn : \text{Var} \times G \to G$$

e verrà utilizzata attraverso la sintassi

$$fn\ \texttt{*variable*} \Rightarrow \texttt{\_expression\_}$$

che restituisce una funzione avente come parametro `*variable*`, il cui valore sarà utilizzato per valutare `_expression_`.

Definizione 2.3.1.2: Applicazione

Sia G una grammatica; allora, dati due suoi termini M, N, è possibile definire su G l'applicazione di M ad N, attraverso la seguente sintassi:

$$MN$$

Tale sintassi è presa in prestito dal lambda calcolo.

Si noti che un'espressione MNL applica prima M ad N, e poi MN ad L, dunque la precedenza è da sinistra verso destra, ovvero $(MN)L$.

Definizione 2.3.1.3: Grammatica *Fun*

Sia *Fun* la seguente estensione della grammatica *Exp*, definita all'interno della Definizione 2.2.2.2:

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

Esempio 2.3.1.1 (Funzioni come argomenti)**.** Sia *Fun* la grammatica definita all'interno della Definizione 2.3.1.3, e sia

$$(fn\ x \Rightarrow x + 1)7$$

una sua espressione; essa, poiché applica la funzione $fn\ x \Rightarrow x+1$ all'espressione 7, viene valutata a

$$x = 7 \implies x + 1 = 7 + 1 = 8$$

Esempio 2.3.1.2 (Espressioni su *Fun*)**.** Sia *Fun* la grammatica definita all'interno della Definizione 2.3.1.3, e sia

$$(fn\ x \Rightarrow x3)(fn\ x \Rightarrow x + 1)$$

una sua espressione; essa, una volta valutata, applica la funzione $fn\ x \Rightarrow x+1$ all'espressione 3, e dunque il suo valore è pari a

$$x = 3 \implies x + 1 = 3 + 1 = 4$$

Definizione 2.3.1.4: Curryficazione

Si consideri la clausola fn della Definizione 2.3.1.1; è possibile definirne una notazione contratta, che prende il nome di *curryficazione*, ed è definita come segue:

$$fn\ x_1 x_2 \ldots x_n \Rightarrow M \iff fn\ x_1 \Rightarrow (fn\ x_2 \Rightarrow \ldots (fn\ x_n \Rightarrow M) \ldots)$$

Il processo inverso prende il nome di *uncurryficazione*.

Esempio 2.3.1.3 (Curryficazioni)**.** Sia *Fun* la grammatica della Definizione 2.3.1.3, e sia

$$(fn\ xy \Rightarrow yx)7(fn\ x \Rightarrow x + 1)$$

una sua espressione; una volta effettuata l'uncurryficazione, si ottiene la seguente espressione:

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

che, una volta valutata, diventa

$$(fn\ y \Rightarrow y7)(fn\ x \Rightarrow x + 1)$$

e dunque, analogamente all'Esempio 2.3.1.2, il risultato è

$$x = 7 \implies x + 1 = 7 + 1 = 8$$

Osservazione 2.3.1.1: Significato della curryficazione

Si noti che la curryficazione *ha significato*, in quanto è possibile considerare la seguente funzione, la quale restituisce la somma di due costanti

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

come se fosse una *funzione a due argomenti*, poiché per utilizzarla sarà necessario fornire 2 interi, come nel seguente esempio:

$$(fn\ x \Rightarrow fn\ y \Rightarrow x + y)5\ 7 \longrightarrow (fn\ y \Rightarrow 5 + y)7 \longrightarrow 5 + 7 \longrightarrow 12$$

Di conseguenza, la versione curryficata della funzione presentata, ovvero

$$fn\ x \Rightarrow fn\ y \Rightarrow x + y \iff fn\ xy \Rightarrow x + y$$

può essere interpretata come una funzione che lavora con 2 argomenti.

2.3.2 Semantica operazionale di *Fun*

Proposizione 2.3.2.1: Fun_{ED}

Sia *Fun* la grammatica definita nella Definizione 2.3.1.3; per poter valutare le sue espressioni in maniera eager dinamica, è necessario ridefinire alcune regole di inferenza:

- l'insieme degli ambienti di *Fun* viene definito come segue:

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

- l'insieme dei valori di *Fun* viene ridefinito come segue:

$$\text{Val} := \{\texttt{0}, \texttt{1}, \ldots\} \cup (\text{Var} \times Fun)$$

  affinché i giudizi operazionali di *Fun* possano contenere tuple;

- $\forall E \in \text{Env}, x \in \text{Var}, M \in Fun \quad E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M)$ per la valutazione delle funzioni;

- $\forall E \in \text{Env}, M, N \in Fun \quad \dfrac{E \vdash M \rightsquigarrow (x, L) \quad E \vdash N \rightsquigarrow v' \quad E\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$
  per certi $x \in \text{Var}, L \in Fun$ tali che M sia una funzione della forma $fn\ x \Rightarrow L$; dunque, per le applicazioni, si ha che il primo giudizio operazionale forza M ad essere una funzione, il secondo valuta N, ed il terzo valuta L con x pari al valore di N.

Si noti che tale valutazione risulta essere eager, poiché N viene valutata immediatamente.

Proposizione 2.3.2.2: Fun_{ES}

Sia *Fun* la grammatica definita nella Definizione 2.3.1.3; per poter valutare le sue espressioni in maniera eager statica, è necessario ridefinire alcune regole di inferenza:

- l'insieme degli ambienti di *Fun* viene definito come segue:

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

- l'insieme dei valori di *Fun* viene ridefinito come segue:

$$\text{Val} := \{\texttt{0}, \texttt{1}, \ldots\} \cup (\text{Var} \times \mathit{Fun} \times \text{Env})$$

  affinché i giudizi operazionali di *Fun* possano contenere triple;

- $\forall E \in \text{Env}, x \in \text{Var}, M \in \mathit{Fun} \quad E \vdash \mathit{fn}\ x \Rightarrow M \rightsquigarrow (x, M, E)$ per la valutazione delle funzioni;

- $\forall E \in \text{Env}, M, N \in \mathit{Fun} \quad \dfrac{E \vdash M \rightsquigarrow (x, L, E') \quad E \vdash N \rightsquigarrow v' \quad E'\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$
  per certi $E' \in \text{Env}, x \in \text{Var}, L \in \mathit{Fun}$ tali che M sia una funzione della forma $\mathit{fn}\ x \Rightarrow L$; dunque, per le applicazioni, si ha che il primo giudizio operazionale forza M ad essere una funzione — salvando inoltre l'ambiente in cui la funzione M è stata valutata — il secondo valuta N, ed il terzo valuta L con x pari al valore di N, ma all'interno dell'ambiente che era stato salvato valutando la funzione M.

Si noti che tale valutazione risulta essere eager, poiché N viene valutata immediatamente, ed è a scoping statico poiché viene riportato l'ambiente in cui le valutazioni vengono effettuate.

Lemma 2.3.2.1: Fun_{ES} **e** Fun_{ED}

Sia *Fun* la grammatica definita nella Definizione 2.3.1.3; allora, si ha che

$$Fun_{\text{ES}} \not\equiv Fun_{\text{ED}}$$

Dimostrazione. Si consideri la seguente espressione

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

definita sulla grammatica *Fun*; essa, valutata attraverso valutazione eager dinamica, produce il seguente albero di derivazione:

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

$$(*) \quad \dfrac{E' \vdash 3 \rightsquigarrow 3 \qquad \dfrac{E'' \vdash y \rightsquigarrow (z, x) \quad E'' \vdash x \rightsquigarrow 3 \quad E''\{(z, 3)\} \vdash x \rightsquigarrow 3}{E'\{(x, 3)\} \vdash yx \rightsquigarrow 3}}{E\{(y, (z, x))\} \vdash \mathit{let}\ x = 3\ \mathit{in}\ yx \rightsquigarrow 3}$$

dove

$$E := \{(x, 7)\}$$
$$E' := E\{(y, (z, x))\}$$
$$E'' := E'\{(x, 3)\}$$

Differentemente, valutando tale espressione attraverso valutazione eager statica, produce il seguente albero di derivazione:

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

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

dove

$$E := \{(x, 7)\}$$
$$E' := E\{(y, (z, x, E))\}$$
$$E'' := E'\{(x, 3)\}$$

Allora, poiché le due valutazioni producono risultati differenti, per la Definizione 2.2.5.5, segue la tesi. □

Proposizione 2.3.2.3: Fun_{LD}

Sia *Fun* la grammatica definita nella Definizione 2.3.1.3; per poter valutare le sue espressioni in maniera lazy dinamica, è necessario ridefinire alcune regole di inferenza:

- l'insieme degli ambienti di *Fun* viene ridefinito come segue:

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

- l'insieme dei valori di *Fun* viene ridefinito come segue:

  $$\text{Val} := \{0, 1, \ldots\} \cup (\text{Var} \times Fun)$$

  affinché i giudizi operazionali di *Fun* possano contenere tuple;

- $\forall E \in \text{Env}, x \in \text{Var}, M \in Fun \quad E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M)$ per la valutazione delle funzioni;

- $\forall E \in \text{Env}, M, N \in Fun \quad \dfrac{E \vdash M \rightsquigarrow (x, L) \quad E'\{(x, N)\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$ per certi $x \in \text{Var}, L \in Fun$ tali che M sia una funzione della forma $fn\ x \Rightarrow L$; dunque, per le applicazioni, si ha che il primo giudizio operazionale forza M ad essere una funzione, mentre il secondo valuta L con x pari ad N — e non al valore che questa assume.

Si noti che tale valutazione risulta essere lazy, poiché N non viene valutata immediatamente, ma ne viene ritardato il calcolo finché non strettamente necessario.

Proposizione 2.3.2.4: Fun_{LS}

Sia Fun la grammatica definita nella Definizione 2.3.1.3; per poter valutare le sue espressioni in maniera lazy statica, è necessario ridefinire alcune regole di inferenza:

- l'insieme degli ambienti di Fun viene ridefinito come segue:

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

  dunque, la sua definizione è ricorsiva

- l'insieme dei valori di Fun viene ridefinito come segue:

$$\text{Val} := \{0, 1, \ldots\} \cup (\text{Var} \times Fun \times \text{Env})$$

- $\forall E \in \text{Env}, x \in \text{Var}, M \in Fun \quad E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M, E)$ per la valutazione delle funzioni;
- $\forall E \in \text{Env}, M, N \in Fun \quad \dfrac{E \vdash M \rightsquigarrow (x, L, E') \quad E'\{(x, (N, E))\} \vdash L \rightsquigarrow v}{E \vdash MN \rightsquigarrow v}$ per certi $E' \in \text{Env}, x \in \text{Var}, L \in Fun$ tali che M sia una funzione della forma $fn\ x \Rightarrow L$; dunque, per le applicazioni, si ha che il primo giudizio operazionale forza M ad essere una funzione — salvando inoltre l'ambiente in cui la funzione M è stata valutata — mentre il secondo valuta L senza valutare l'espressione N, ma ponendo solamente x pari ad N – e non al suo valore — all'interno dell'ambiente in cui è stata effettuata l'applicazione MN.

Si noti che tale valutazione risulta essere lazy, poiché N non viene valutata immediatamente, ed è a scoping statico poiché viene riportato l'ambiente in cui le valutazioni vengono effettuate.

Lemma 2.3.2.2: Fun_{LS} **e** Fun_{LD}

Sia Fun la grammatica definita nella Definizione 2.3.1.3; allora, si ha che

$$Fun_{\text{LS}} \not\equiv Fun_{\text{LD}}$$

Dimostrazione. Omessa. □

Definizione 2.3.2.1: Espressione ω

Data una grammatica, l'**espressione** ω è un'espressione composta dalla più piccola funzione che entra in ricorsione infinita senza chiamare sé stessa.

Esempio 2.3.2.1 (Espressione ω di Fun)**.** Sia Fun la grammatica definita all'interno della Definizione 2.3.1.3; allora, una sua espressione ω è la seguente:

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

Essa risulta essere un'espressione ω per *Fun*, poiché la sua valutazione entra in ricorsione infinita indipendentemente dalla semantica scelta.

Lemma 2.3.2.3: Semantiche di *Fun*

Sia *Fun* la grammatica definita nella Definizione 2.3.1.3; allora, si ha che

$$Fun_{\text{LD}} \not\equiv Fun_{\text{ED}} \not\equiv Fun_{\text{ES}} \not\equiv Fun_{\text{LS}}$$

Dimostrazione. Si noti che:

- $Fun_{\text{ED}} \not\equiv Fun_{\text{ES}}$ per il Lemma 2.3.2.1
- $Fun_{\text{LS}} \not\equiv Fun_{\text{LD}}$ per il Lemma 2.3.2.2

mentre, per quanto riguarda $Fun_{\text{ED}} \not\equiv Fun_{\text{LD}}$ e $Fun_{\text{ES}} \not\equiv Fun_{\text{LS}}$, si prenda l'espressione ω di *Fun*, presentata all'interno dell'Esempio 2.3.2.1, e si consideri la seguente espressione

$$let\ x = \omega\ in\ 69^{1}$$

questa, quando valutata in maniera eager — indipendentemente dallo scoping — richiederebbe di valutare immediatamente l'espressione ω, la quale è invalutabile per definizione; mentre, quando valutata in maniera lazy — indipendentemente dallo scoping — rimanderebbe il calcolo dell'espressione ω, restituendo 69 come risultato. Dunque, poiché si ottengono risultati diversi a seconda della semantica utilizzata per valutare tale espressione, segue la tesi. □

Osservazione 2.3.2.1: Variabili libere di *Fun*

Sia *Fun* la grammatica della Definizione 2.3.1.3; su di essa, è possibile definire, ricorsivamente, una funzione in grado di restituire le variabili free di una data espressione, come segue:

$$\text{free} : Fun \to \mathcal{P}(\text{Var}) : e \mapsto \begin{cases} \varnothing & \exists \eta \in \mathbb{N} \mid e = k(\eta) \\ \{x\} & \exists x \in \text{Var} \mid e = x \\ \text{free}(M) \cup \text{free}(N) & \exists M, N \in Fun \mid e = M + N \vee e = M * N \\ \text{free}(M) \cup (\text{free}(N) - \{x\}) & \exists x \in \text{Var}, M, N \in Fun \mid e = (let\ x = M\ in\ N) \\ \text{free}(M) - \{x\} & \exists x \in \text{Var}, M \in Fun \mid e = (fn\ x \Rightarrow M) \\ \text{free}(M) \cup \text{free}(N) & \exists M, N \in Fun \mid e = (MN) \end{cases}$$

[1] Nice.

2.4 Lambda calcolo

2.4.1 Numeri di Church

Definizione 2.4.1.1: Numeri di Church

La **rappresentazione di Church dei numeri naturali**, denotata con $\mathbb{N}_\lambda$, è la seguente:

- $0_\lambda := fn\ x \Rightarrow fn\ y \Rightarrow y \iff fn\ xy \Rightarrow y$
- $\text{succ}_\lambda := fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(xy)) \iff fn\ zxy \Rightarrow zx(xy)$

Esempio 2.4.1.1 (1_λ di Church)**.** Per calcolare l'$1_\lambda \in \mathbb{N}_\lambda$ di Church, è sufficiente valutare $\text{succ}_\lambda(0_\lambda)$, e dunque

$$(fn\ zxy \Rightarrow zx(xy))(fn\ xy \Rightarrow y) \longrightarrow fn\ xy \Rightarrow (fn\ xy \Rightarrow y)x(xy) \longrightarrow$$
$$\longrightarrow fn\ xy \Rightarrow (fn\ y \Rightarrow y)(xy) \longrightarrow fn\ xy \Rightarrow xy =: 1_\lambda$$

Esempio 2.4.1.2 (2_λ di Church)**.** Per calcolare il $2_\lambda \in \mathbb{N}_\lambda$ di Church, è sufficiente valutare $\text{succ}_\lambda(1_\lambda)$, e dunque

$$(fn\ zxy \Rightarrow zx(xy))(fn\ xy \Rightarrow xy) \longrightarrow fn\ xy \Rightarrow (fn\ xy \Rightarrow xy)x(xy) \longrightarrow$$
$$\longrightarrow fn\ xy \Rightarrow (fn\ y \Rightarrow xy)(xy) \longrightarrow fn\ xy \Rightarrow xxy =: 2_\lambda$$

Lemma 2.4.1.1: Algebra dei numeri di Church

Si consideri l'algebra dei numeri di Church, definita come $(\mathbb{N}_\lambda, \text{zero}_\lambda, \text{succ}_\lambda)$, dove

$$\text{zero}_\lambda : \mathbb{1} \to \mathbb{N}_\lambda : x \mapsto 0_\lambda$$

Essa è un'algebra induttiva.

Dimostrazione. Omessa. □

Osservazione 2.4.1.1: Significato di $\mathbb{N}_\lambda$

Si considerino l'Esempio 2.4.1.1 e l'Esempio 2.4.1.2, e si noti che

$$\begin{aligned} 0_\lambda &:= fn\ xy \Rightarrow y \\ 1_\lambda &:= fn\ xy \Rightarrow xy \\ 2_\lambda &:= fn\ xy \Rightarrow x(xy) \\ &\vdots \end{aligned}$$

dunque, la corrispondenza tra $\mathbb{N}$ e $\mathbb{N}_\lambda$ è data dal *numero di applicazioni effettuate* ad una qualche variabile. Infatti è possibile costruire il seguente isomorfismo tra le algebre $(\mathbb{N}, \text{zero}, \text{succ})$ e $(\mathbb{N}_\lambda, \text{zero}_\lambda, \text{succ}_\lambda)$:

$$\varphi : \mathbb{N} \to \mathbb{N}_\lambda : n \mapsto fn\ xy \Rightarrow \underbrace{x \cdots (x}_{n \text{ volte}}\, y)$$

dove x è dunque una funzione, che può essere applicata ad una certa variabile y.

Proposizione 2.4.1.1: Funzione eval_λ

È possibile definire una funzione che, dato un numero di Church, restituisce il corrispondente numero naturale, come segue:

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

poiché applica la funzione $\text{succ}_\mathbb{N}$ esattamente $z \in \mathbb{N}_\lambda$ volte a 0.

Esempio 2.4.1.3 (Corrispondenze tra $\mathbb{N}_\lambda$ e $\mathbb{N}$)**.** Per valutare $\text{eval}_\lambda(1_\lambda)$ è necessario svolgere i seguenti calcoli:

$$\begin{aligned} (fn\ z \Rightarrow z(fn\ x \Rightarrow x + 1)0)(fn\ xy \Rightarrow xy) &\longrightarrow (fn\ xy \Rightarrow xy)(fn\ x \Rightarrow x + 1)0 \longrightarrow \\ &\longrightarrow (fn\ y \Rightarrow (fn\ x \Rightarrow x + 1)y)0 \longrightarrow (fn\ x \Rightarrow x + 1)0 \longrightarrow 1 \end{aligned}$$

Proposizione 2.4.1.2: Operazione sum_λ

Si consideri l'algebra dei numeri di Church; su di essa, è possibile definire l'operazione di somma, come segue:

$$\text{sum}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zx(wxy)) \iff fn\ zwxy \Rightarrow zx(wxy)$$

poiché alla variabile y viene prima applicata x esattamente $w \in \mathbb{N}_\lambda$ volte ad y, e successivamente ad esso viene applicata x altre $z \in \mathbb{N}_\lambda$ volte. Inoltre, è possibile fornire una definizione altenrativa della funzione, come segue:

$$\text{sum}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow z\ \text{succ}_\lambda\ w \iff fn\ zw \Rightarrow z\ \text{succ}_\lambda\ w$$

poiché viene applicata la funzione di successore succ_λ a $w \in \mathbb{N}_\lambda$, esattamente $z \in \mathbb{N}_\lambda$ volte.

Esempio 2.4.1.4 (Somme in $\mathbb{N}_\lambda$)**.** Per calcolare $\text{sum}_\lambda(2_\lambda, 1_\lambda)$ è necessario svolgere i seguenti calcoli:

$$\begin{gathered}(fn\ zwxy \Rightarrow zx(wxy))(fn\ xy \Rightarrow x(xy))(fn\ xy \Rightarrow xy) \longrightarrow \\ \longrightarrow (fn\ wxy \Rightarrow (fn\ xy \Rightarrow x(xy))x(wxy))(fn\ xy \Rightarrow xy) \longrightarrow \\ \longrightarrow (fn\ wxy \Rightarrow (fn\ y \Rightarrow x(xy))(wxy))(fn\ xy \Rightarrow xy) \longrightarrow \\ \longrightarrow (fn\ wxy \Rightarrow x(x(wxy)))(fn\ xy \Rightarrow xy) \longrightarrow fn\ xy \Rightarrow x(x((fn\ xy \Rightarrow xy)xy)) \longrightarrow \\ \longrightarrow fn\ xy \Rightarrow x(x((fn\ y \Rightarrow xy)y)) \longrightarrow fn\ xy \Rightarrow x(x(xy)) =: 3_\lambda\end{gathered}$$

Proposizione 2.4.1.3: Operazione prod_λ

Si consideri l'algebra dei numeri di Church; su di essa, è possibile definire l'operazione di prodotto, come segue:

$$\text{prod}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow z(wx)y) \iff fn\ zwxy \Rightarrow z(wx)y$$

poiché alla variabile y viene applicata la funzione $z(wx)$, che equivale alla funzione wx composta su sé stessa z volte — e dunque $z, w \in \mathbb{N}_\lambda$. Inoltre, è possibile fornire una definizione altenrativa della funzione, come segue:

$$\text{prod}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow z(\text{sum}_\lambda\ w)0_\lambda \iff fn\ zw \Rightarrow z(\text{sum}_\lambda\ w)0_\lambda$$

poiché allo 0_λ viene applicata la funzione $z(\text{sum}_\lambda\ w)$, che equivale alla funzione $(\text{sum}_\lambda\ w)$ — la quale restituisce una funzione che si aspetta il secondo argomento a cui applicare la somma, che sarà 0_λ — composta su sé stessa $z \in \mathbb{N}_\lambda$ volte.

Proposizione 2.4.1.4: Operazione power_λ

Si consideri l'algebra dei numeri di Church; su di essa, è possibile definire l'operazione di elevamento a potenza, come segue:

$$\text{power}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow wz$$

TODO.

2.4.2 Logica booleana di Church

Definizione 2.4.2.1: Logica booleana di Church

La **rappresentazione di Church della logica booleana**, denotata con $\mathbb{B}_\lambda$, è la seguente

- $\text{true}_\lambda := fn\ x \Rightarrow fn\ y \Rightarrow x \iff fn\ xy \Rightarrow x$
- $\text{false}_\lambda := fn\ x \Rightarrow fn\ y \Rightarrow y \iff fn\ xy \Rightarrow y$

Proposizione 2.4.2.1: Funzione evalBool_λ

Si consideri la grammatica definita all'interno della Definizione 2.3.1.3; è possibile estenderla affinché includa anche i valori booleani *true* e *false*. Dunque, è possibile definire una funzione che, dato un booleano di Church, restituisce il corrispondente valore booleano, come segue:

$$\text{evalBool}_\lambda := fn\ z \Rightarrow z\ true\ false$$

Esempio 2.4.2.1 (Valutazione di true_λ)**.** Per valutare $\text{evalBool}_\lambda(\text{true}_\lambda)$, è sufficiente svolgere i seguenti calcoli:

$$(fn\ z \Rightarrow z\ true\ false)(fn\ xy \Rightarrow x) \longrightarrow (fn\ xy \Rightarrow x)true\ false \longrightarrow$$
$$\longrightarrow (fn\ y \Rightarrow true)false \longrightarrow true$$

Proposizione 2.4.2.2: Operazione ite_λ

Si consideri l'algebra dei booleani di Church; su di essa, è possibile definire l'operazione logica *if-then-else*, come segue:

$$\text{ite}_\lambda := fn\ z \Rightarrow fn\ u \Rightarrow fn\ v \Rightarrow zuv \iff fn\ zuv \Rightarrow zuv$$

poiché dato un $z \in \mathbb{B}_\lambda$, la funzione restituirà u se z è true_λ, altrimenti v.

Proposizione 2.4.2.3: Operazione if_λ

Si consideri l'algebra dei booleani di Church; su di essa, è possibile definire l'operazione logica *if*, come segue:

$$\text{if}_\lambda := fn\ z \Rightarrow fn\ u \Rightarrow z\ u\ \text{true}_\lambda \iff fn\ zu \Rightarrow z\ u\ \text{true}_\lambda$$

poiché l'operazione è equivalente ad ite_λ definita all'interno dell'Proposizione 2.4.2.2, ma il branch dell'*else* restituisce sempre true_λ.

Proposizione 2.4.2.4: Operazione not_λ

Si consideri l'algebra dei booleani di Church; su di essa, è possibile definire l'operazione logica di negazione come segue:

$$\text{not}_\lambda := fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow zyx) \iff fn\ zxy \Rightarrow zyx$$

poiché tale funzione passa ad un certo $z \in \mathbb{B}_\lambda$ le variabili x ed y scambiate di posto. Si noti che l'operazione è simile all'operazione ite_λ definita nella Proposizione 2.4.2.2, poiché è possibile interpretarla come l'inverso dell'operazione *if-then-else*.

Proposizione 2.4.2.5: Operazione or_λ

Si consideri l'algebra dei booleani di Church; su di essa, è possibile definire l'operazione logica di *or*, come segue:

$$\text{or}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow \text{if}_\lambda(\text{not}_\lambda\ z)w \iff fn\ zw \Rightarrow \text{if}_\lambda(\text{not}_\lambda\ z)w$$

poiché TODO

Proposizione 2.4.2.6: Operazione and_λ

Si consideri l'algebra dei booleani di Church; su di essa, è possibile definire l'operazione logica di *and*, come segue:

$$\text{and}_\lambda := fn\ z \Rightarrow fn\ w \Rightarrow \text{not}_\lambda(\text{if}_\lambda\ z(\text{not}_\lambda\ w))$$

poiché TODO

2.4.3 Lambda calcolo

Definizione 2.4.3.1: Lambda calcolo

Il **lambda calcolo** è un sistema formale atto ad analizzare le funzioni e le loro applicazioni. La grammatica del lambda calcolo è la seguente

$$M, N ::= x \mid \lambda x.M \mid MN$$

dove $\lambda x.M$ indica una funzione della forma $fn\ x \Rightarrow M$, e prende il nome di **lambda astrazione**. Le espressioni del lambda calcolo sono dette **lambda espressioni**.

Esempio 2.4.3.1 (Lambda espressioni)**.** I seguenti sono esempi di espressioni del lambda calcolo:

- $(\lambda x.x + 1)2$ corrisponde a $(fn\ x \Rightarrow x + 1)2$ ed equivale a 3
- $\lambda xy.x(x(xy))$ corrisponde a $fn\ xy \Rightarrow x(x(xy))$, ovvero $3_\lambda \in \mathbb{N}_\lambda$
- $(\lambda x.xy)(\lambda x.x)$ equivale ad y

Definizione 2.4.3.2: Sostituzione

Date due espressioni M ed N, ed una variabile x, l'operazione di **sostituzione** rimpiazza ogni occorrenza della variabile x — all'interno dell'espressione M — con il termine N. In simboli

$$M[N/x]$$

Esempio 2.4.3.2 (Sostituzioni)**.** I seguenti sono esempi di sostitutioni all'interno di espressioni:

- $(xy)[\lambda z.z/x]$ corrisponde a $(\lambda z.z)y$, ovvero y
- $(fn\ x \Rightarrow y)[x/y]$ corrisponde a $fn\ x \Rightarrow x$

Osservazione 2.4.3.1: Cattura di variabili

L'operazione di sostituzione, definita nella Definizione 2.4.3.2, potrebbe cambiare il binding delle variabili definite; tale fenomeno prende il nome di **cattura di variabili**.

Esempio 2.4.3.3 (Catture di variabili)**.** Si consideri la seguente espressione:

$$(\lambda y.M)[N/x]$$

se N contenesse la variabile y in modo libero, si avrebbe che

$$\lambda y.(M[N/x])$$

non sarebbe equivalente all'espressione di partenza, poiché y diverrebbe legata. Dunque, la loro equivalenza è garantita solamente se $y \notin \text{free}(N)$.

Definizione 2.4.3.3: Alfa conversione

Data una lambda astrazione $\lambda x.M$, si definisce **alfa conversione** la regola secondo la quale ogni occorrenza di x nella lambda astrazione viene rimpiazzata con un altra variabile. In simboli

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

Esempio 2.4.3.4 (Alfa conversioni)**.** Si consideri la seguente lambda astrazione

$$\lambda x.x(\lambda z.zw)$$

allora, ad esempio, è vero che

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

avendo rimpiazzato x con z.

Definizione 2.4.3.4: Alfa equivalenza

Due lambda astrazioni $\lambda x.M$ e $\lambda y.N$ sono dette **alfa equivalenti**, indicato con il simbolo $\equiv_\alpha$ se è vera la seguente:

$$\lambda x.M \equiv_\alpha \lambda.N \iff \begin{cases} \lambda x.M \xrightarrow{\alpha} \lambda y.(N[y/x]) \\ \lambda y.N \xrightarrow{\alpha} \lambda x.(M[x/y]) \end{cases}$$

Esempio 2.4.3.5 (Alfa equivalenze)**.** Si considerino le due seguenti lambda astrazioni

$$\lambda x.xy$$
$$\lambda z.zy$$

e si noti che

$$\begin{cases} \lambda x.xy \xrightarrow{\alpha} \lambda z.zy \\ \lambda z.zy \xrightarrow{\alpha} \lambda x.xy \end{cases} \iff \lambda x.xy \equiv_\alpha \lambda z.zy$$

dunque le due lambda astrazioni sono alfa equivalenti. Si considerino invece le due seguenti lambda astrazioni

$$\lambda x.x(\lambda z.zw)$$
$$\lambda z.z(\lambda z.zw)$$

e si noti, differentemente, che

$$\begin{cases} \lambda x.x(\lambda z.zw) \xrightarrow{\alpha} \lambda z.z(\lambda z.zw) \\ \lambda z.z(\lambda z.zw) \not\xrightarrow{}_\alpha \lambda x.x(\lambda z.zw) \end{cases} \implies \lambda x.x(\lambda z.zw) \not\equiv_\alpha \lambda z.z(\lambda z.zw)$$

Definizione 2.4.3.5: Beta conversione

Data una lambda espressione $(\lambda x.M)$, si definisce **beta conversione** la regola secondo la quale ogni occorrenza di x all'interno di M viene rimpiazzata dal termine N. In simboli

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

Esempio 2.4.3.6 (Beta conversioni)**.** Si consideri la seguente lambda espressione

$$(\lambda x.xy)(\lambda z.z)$$

applicando la beta conversione, si ottiene

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

Osservazione 2.4.3.2: Beta conversioni

Di fatto, la beta conversione corrisponde ad un passo computazionale.

Osservazione 2.4.3.3: Semantica delle beta conversioni

Si noti che la beta conversione ha significato solamente in un contesto lazy, poiché considerando ad esempio la lambda espressione

$$(\lambda x.7)\omega$$

(dove ω è rappresenta l'espressione ω della Definizione 2.3.2.1) essa è beta equivalente alla lambda espressione $(\lambda x.7)[\omega/x] \xrightarrow{\beta} 7$ soltanto se l'espressione ω non viene valutata.

Definizione 2.4.3.6: Eta conversione

Si definisce **eta conversione** la regola secondo la quale una lambda espressione $(\lambda x.Mx)$ può essere rimpiazzata con M, solo se x non è libera. In simboli

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

Esempio 2.4.3.7 (Eta conversioni)**.** Si consideri la seguente lambda espressione

$$\lambda x.(\lambda y.y)x$$

si noti che $\text{free}(\lambda y.y) = \varnothing \implies x \notin \text{free}(\lambda y.y)$ e dunque è possibile applicare l'eta conversione, ottenendo quindi

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

Osservazione 2.4.3.4: Cattura nelle eta conversioni

Si noti che, all'interno della Definizione 2.4.3.6, la condizione per cui x non sia libera garantisce che la conversione produca espressioni equivalenti; infatti, se x fosse libera in M, poiché non lo sarebbe comunque in $(\lambda x.Mx)$, l'eta conversione non avrebbe mantenuto l'equivalenza.

3 Paradigma imperativo

3.1 Programmi

3.1.1 Memoria

Definizione 3.1.1.1: Programma

Nel paradigma imperativo, un **programma** è un componente semantico che non restituisce valori — dunque non ha *side effect* — ma cambiano il valore della memoria.

Definizione 3.1.1.2: Memoria

Sia G una grammatica composta da programmi; per simulare la **memoria**, all'interno del paradigma imperativo verranno utilizzati ambienti definiti come segue:

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

dove Loc è un insieme di locazioni di memoria; inoltre, si definisce il seguente insieme

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

dunque, le funzioni $E \in \text{Env}$ associano le variabili ad una locazione in memoria, mentre le funzioni $S \in \text{Store}$ associano le locazioni ai valori, simulando quindi la struttura dei *puntatori*, i quali caratterizzano i linguaggi imperativi.

3.1.2 Clausole imperative

Definizione 3.1.2.1: Clausola *skip*

Sia G una grammatica; allora, è possibile definire su G la clausola *skip*, che equivale a non effettuare alcuna operazione.

Definizione 3.1.2.2: Clausola *seq*

Sia G una grammatica composta da programmi, e sia G_p l'insieme dei suoi programmi; allora, è possibile definire su G una funzione *seq*, come segue:

$$seq : G_p \times G_p \to G_p$$

e verrà utilizzata attraverso la sintassi

$$\texttt{<program>}_1 ; \texttt{<program>}_2$$

che sta ad indicare che verrà prima eseguito $\texttt{<program>}_1$, e successivamente $\texttt{<program>}_2$.

Definizione 3.1.2.3: Clausola *ite*

Sia G una grammatica composta da programmi, e siano G_p ed G_M rispettivamente l'insieme dei suoi programmi e delle sue espressioni; allora, è possibile definire su G una funzione *ite* (*if-then-else*), come segue:

$$ite : G_M \times G_p \times G_p \to G_p$$

e verrà utilizzata attraverso la sintassi

$$if\ \texttt{\_expression\_}\ then\ \texttt{<program>}_1\ else\ \texttt{<program>}_2$$

dove se M è un espressione che può essere valutata a *true*, verrà eseguito $\texttt{<program>}_1$, altrimenti verrà eseguito $\texttt{<program>}_2$.

Definizione 3.1.2.4: Clausola *while*

Sia G una grammatica composta da programmi, e siano G_p ed G_M rispettivamente l'insieme dei suoi programmi e delle sue espressioni; allora, è possibile definire su G una funzione *while*, come segue:

$$while : G_M \times G_p \to G_p$$

e verrà utilizzata attraverso la sintassi

while `_expression_` *do* `<program>`

dove — se `_expression_` è un espressione che può essere valutata a *true* o $false$ — viene eseguito `<program>` fintanto che `_expression_` viene valutata a *true*.

Definizione 3.1.2.5: Clausola *var*

TODO

Definizione 3.1.2.6: Clausola *assign*

Sia G una grammatica composta da programmi, e sia G_M l'insieme delle sue espressioni; allora, è possibile deifnire su G una funzione *assign*, come segue:

$$assign : \text{Var} \times G_M \to G$$

e verrà utilizzata attraverso la sintassi

`*variable*` := `_expression_`

all'interno della quale, a `*variable*`, verrà assegnato il valore di `_expression_`.

3.2 *Imp*: un linguaggio imperativo

3.2.1 Definizioni

Definizione 3.2.1.1: Grammatica *Imp*

Si consideri la grammatica Exp definita all'interno della Definizione 2.2.1.1 (si noti che non si tratta di Exp estesa); è possibile estendere essa come segue:

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

dove $true$ e $false$ sono valori booleani di verità. Allora, sia Imp la grammatica composta da tale estensione di Exp, e dalla seguente:

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

L'insieme degli ambienti di Imp è strutturato come nella Definizione 3.1.1.2. Inoltre, per effettuare le valutazioni, vengono definite le seguenti semantiche operazionali:

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

ed i loro giudizi operazionali verranno indicati come segue:

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

Osservazione 3.2.1.1: Semantiche di *Imp*

Sia Imp la grammatica della Definizione 3.2.1.1, e si considerino le sue semantiche operazionali; esse sono definite tali che la semantica $\overset{M}{\rightsquigarrow}$ delle espressioni sia in grado di restituire valori, ma non di cambiare la memoria, mentre la semantica $\overset{p}{\rightsquigarrow}$ dei programmi non restituisca valori, ma alteri lo stato della memoria.

3.2.2 Semantica operazionale di *Imp*

Proposizione 3.2.2.1: Semantica operazionale di *Imp*

Sia Imp la grammatica definita all'interno della Definizione 3.2.1.1, e siano $E \in \text{Env}$ e $S \in \text{Store}$; allora, si definiscono le seguenti regole operazionali (si noti che, per brevità, verrà utilizzato il simbolo $\rightsquigarrow$ all'interno dei giudizi di entrambe le semantiche di Imp):

- **costanti**:

$$[const]\ E \vdash k,\ S \rightsquigarrow S$$

- **variabili**:

$$\forall x \in \text{Var} \quad \exists v \in \text{Val} \mid S(E(x)) = v \implies [vars]\ E \vdash x,\ S \rightsquigarrow v$$

- **somme**:

$$\forall M, N \in Exp, v \in \text{Val} \quad \exists v_1, v_2 \in \text{Val} \mid v = v_1 + v_2 \implies [plus]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M + N,\ S \rightsquigarrow v}$$

- **prodotti**:

$$\forall M, N \in Exp, v \in \text{Val} \quad \exists v_1, v_2 \in \text{Val} \mid v = v_1 \cdot v_2 \implies [times]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M * N,\ S \rightsquigarrow v}$$

- **minorazioni**:

$$\forall M, N \in Exp, v_1, v_2 \in \text{Store} \quad v_1 < v_2 \implies [lt_1]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M\ <\ N,\ S \rightsquigarrow true}$$

$$\forall M, N \in Exp, v_1, v_2 \in \text{Store} \quad v_1 \geq v_2 \implies [lt_2]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M\ <\ N,\ S \rightsquigarrow false}$$

- **skip**:

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

- **composizioni sequenziali**:

$$\forall p, q \in Imp, S', S'' \in \text{Store} \quad [seq]\ \frac{E \vdash p,\ S \rightsquigarrow S' \quad E \vdash q,\ S' \rightsquigarrow S''}{E \vdash p; q,\ S \rightsquigarrow S''}$$

- **if-then-else**:

$$\forall p, q \in Imp, S' \in \text{Store} \quad [ite_1]\ \frac{E \vdash M,\ S \rightsquigarrow true \quad E \vdash p,\ S \rightsquigarrow S'}{E \vdash if\ M\ then\ p\ else\ q,\ S \rightsquigarrow S'}$$

$$\forall p, q \in Imp, S'' \in \text{Store} \quad [ite_2]\ \frac{E \vdash M,\ S \rightsquigarrow false \quad E \vdash p,\ S \rightsquigarrow S''}{E \vdash if\ M\ then\ p\ else\ q,\ S \rightsquigarrow S''}$$

- **while**:

$$\forall M \in Exp, S', S'' \in \text{Store} \quad [while_1]\ \frac{E \vdash M,\ S \rightsquigarrow true \quad E \vdash p,\ S \rightsquigarrow S' \quad E \vdash while\ M\ do\ p,\ S' \rightsquigarrow S''}{E \vdash while\ M\ do\ p,\ S \rightsquigarrow S''}$$

$$\forall M \in Exp \quad [while_2]\ \frac{E \vdash M,\ S \rightsquigarrow false}{E \vdash while\ M\ do\ p,\ S \rightsquigarrow S}$$

- **dichiarazioni ed assegnazioni**:

$$\forall x \in \text{Var}, M \in Exp, p \in Imp, S' \in \text{Store}, v \in \text{Val} \quad \exists l \in \text{Loc} \mid l \notin \text{dom}(S) \implies$$

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

- **assegnazioni**:

$$\forall x \in \text{Var}, M \in Exp \quad \exists l \in \text{Loc} \mid E(x) = l \implies [assign]\ \frac{E \vdash M,\ S \rightsquigarrow v}{E \vdash x := M,\ S \rightsquigarrow S\{(l,v)\}}$$

3.3 Memoria contigua

3.3.1 Definizioni

Definizione 3.3.1.1: Memoria contigua

Sia G una grammatica composta da programmi, e siano G_p e G_M rispettivamente l'insieme dei suoi programmi e delle sue espressioni; al fine di implementare gli *array* all'interno del paradigma imperativo, è necessario definire la **memoria contigua**, dunque si definisce il seguente insieme di locazioni

$$\mathrm{Loc}^+ := \bigcup_{n \in \mathbb{N}} \mathrm{Loc}^n$$

poiché è in grado di supportare infinite sequenze di elementi, e si assume che queste siano contigue in memoria.

Definizione 3.3.1.2: Clausola *arr*

Sia G una grammatica composta da programmi, e siano G_p e G_M rispettivamente l'insieme dei suoi programmi e delle sue espressioni; allora, è possibile definire su G una funzione arr, come segue:

$$arr : \mathrm{Var} \times G_M^+ \times G_p \to G_p$$

e verrà utilizzata attraverso la sintassi

$$arr\ \texttt{*variable*} = [\texttt{\_expression\_}_1, \ldots, \texttt{\_expression\_}_n]\ in\ \texttt{<program>}$$

la quale pone `*variable*` pari all'array $[\texttt{\_expression\_}_1, \ldots, \texttt{\_expression\_}_n]$, all'interno di `<program>`.

Definizione 3.3.1.3: Clausola *loc*

TODO

Definizione 3.3.1.4: Clausola *proc*

Sia G una grammatica composta da programmi, e sia G_p l'insieme dei suoi programmi; allora, è possibile definire su G una funzione *proc*, come segue:

$$proc : \text{Var} \times \text{Var} \times G_p \times G_p \to G_p$$

e verrà utilizzata attraverso la sintassi

$$proc\ \texttt{*variable*}_1(\texttt{*variable*}_2)\ is\ \texttt{<program>}_1\ in\ \texttt{<program>}_2$$

la quale definisce la *procedura* $\texttt{*variable*}_1(\texttt{*variable*}_2)$, il cui corpo è costituito da $\texttt{<program>}_1$, ed è possibile utilizzarla all'interno di $\texttt{<program>}_2$.

Definizione 3.3.1.5: Clausola *call*

Sia G una grammatica composta da programmi, e siano G_p e G_M rispettivamente l'insieme dei suoi programmi e delle sue espressioni; allora, è possibile definire su G una funzione *call*, come segue:

$$call : \text{Var} \times G_M \to G_p$$

e verrà utilizzata attraverso la sintassi

$$call\ \texttt{*variable*}(\texttt{\_expression\_})$$

la quale effettua una chiamata alla procedura `*variable*`, fornendole come parametro `_expression_`.

3.4 *All*: un linguaggio imperativo completo

3.4.1 Definizioni

Definizione 3.4.1.1: Grammatica *All*

Sia *All* la grammatica, estensione di *Imp* definita nella Definizione 3.2.1.1, composta dalle seguenti grammatiche:

$$
\begin{aligned}
&k ::= \texttt{0} \mid \texttt{1} \mid \ \ldots \ \mid true \mid false \\
&V ::= x \mid x[M] \\
&M, N ::= k \mid V \mid M \texttt{ + } N \mid M \texttt{ * } N \mid M \texttt{ < } N \\
&p, q ::= skip \mid p; q \mid if\ M\ then\ p\ else\ q \\
&\qquad\quad while\ M\ do\ p \mid var\ x = M\ in\ p \mid arr\ x = [M_0, \ldots, M_n]\ in\ p \\
&\qquad\quad V := M \mid proc\ y(x)\ is\ p\ in\ x \mid call\ y(M)
\end{aligned}
$$

Dunque, essa è composta da:

i) una grammatica per le *costanti*;

ii) una grammatica per le *espressioni assegnabili*, che prende il nome di *LExp* (*left expressions*); per valutare le sue espressioni, si introduce la seguente semantica:

$$\overset{V}{\rightsquigarrow} \subseteq \text{Env} \times LExp \times \text{Store} \times \text{Loc}$$

iii) una grammatica per le *espressioni valutabili*, la quale consiste in un'estensione di *Exp* definita nella Definizione 2.2.1.1;

iv) una grammatica per i *programmi*, la quale consiste in un'estensione di *Imp* definita nella Definizione 3.2.1.1.

Si noti che, poiché tale grammatica supporta gli array, è necessario fornire ad *All* locazioni di memoria contigue. Inoltre, l'insieme degli ambienti di *All* è definito come segue:

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

3.4.2 Semantica operazionale di *All*

Proposizione 3.4.2.1: Semantica operazionale di *All*

Sia *All* la grammatica definita all'interno della Definizione 3.4.1.1, e siano $E \in \text{Env}$ e $S \in \text{Store}$; allora, in aggiunta alle regole

$$[const],\ [plus],\ [times],\ [lt_1],\ [lt_2],\ [skip],\ [seq],\ [ite_1],\ [ite_2],\ [while_1],\ [while_2]$$

descritte nell'Proposizione 3.2.2.1, si definiscono le seguenti:

- **locazioni**:

$$\forall x \in \text{Var} \quad \exists l \in \text{Loc}^+ \mid E(x) = l \implies [loc_1]\ E \vdash x,\ S \overset{V}{\rightsquigarrow} l$$

- **locazioni in array**:

$$\forall n \in \mathbb{N}, x \in \text{Var} \quad \exists (l_0, \dots, l_n) \in \text{Loc}^+ \mid E(x) = \langle l_0, \dots, l_n \rangle \implies$$

$$\implies \forall m \in [0, n] \quad [loc_2]\ \frac{E \vdash M,\ S \overset{M}{\rightsquigarrow} m}{E \vdash x[M],\ S \overset{V}{\rightsquigarrow} l_m}$$

- **reference**:

$$\forall V \in LExp, l \in \text{Loc} \quad \exists v \in \text{Val} \mid S(l) = v \implies [ref]\ \frac{E \vdash V,\ S \overset{V}{\rightsquigarrow} l}{E \vdash V,\ S \overset{M}{\rightsquigarrow} v}$$

- **assegnazioni**:

$$\forall V \in LExp, M \in Exp, l \in \text{Loc}, v \in \text{Val} \quad [assign]\ \frac{E \vdash M,\ S \overset{M}{\rightsquigarrow} v \quad E \vdash V,\ S \overset{V}{\rightsquigarrow} l}{E \vdash V := M,\ S \overset{p}{\rightsquigarrow} S\{(l, v)\}}$$

- **array**:

$$\forall n \in \mathbb{N}, S' \in \text{Store}, x \in \text{Var}, M_0, \dots, M_n \in Exp, p \in Imp, v_0, \dots, v_n \in \text{Val}$$

$$\exists (l_0, \dots, l_n) \in \text{Loc}^+ \mid l_0, \dots, l_n \notin \text{dom}(S) \implies$$

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

- **procedure**:

$$\forall y, x \in \text{Var}, p, q \in Imp, S' \in \text{Store} \quad [proc]\ \frac{E\{(y, (x, p, E))\} \vdash q,\ S \overset{p}{\rightsquigarrow} S'}{E \vdash proc\ y(x)\ is\ p\ in\ q,\ S \overset{p}{\rightsquigarrow} S'}$$

Per quanto concerne le regole della clausola *call*, si consultino la Proposizione 3.4.2.2, la Proposizione 3.4.2.3 e la Proposizione 3.4.2.4.

Osservazione 3.4.2.1: Variabili in *All*

Si considerino le regole della grammatica *All*, definite all'interno dell'Proposizione 3.4.2.1, e si noti che in essa non è presente la regola di inferenza $[vars]$; infatti, all'interno di *All*, l'unico modo per accedere al valore di una variabile è tramite *reference*, dunque utilizzando $[loc_1]$ assieme a $[ref]$. Questo è necessario poiché *All* supporta gli array, e dunque TODO.

Definizione 3.4.2.1: Semantiche di *call*

È possibile effettuare le chiamate alle procedure attraverso le seguenti semantiche operazionali:

- **call-by-value**, la quale corrisponde ad una semantica *eager statica*, e gli argomenti alle procedure sono espressioni non assegnabili, e vengono valutate;
- **call-by-reference**, la quale corrisponde ad una semantica *eager statica*, e gli argomenti alle procedure sono espressioni assegnabili, e vengono valutate;
- **call-by-name**, la quale corrisponde ad una semantica *lazy statica*, e gli argomenti alle procedure sono espressioni assegnabili, ma non vengono valutate.

Proposizione 3.4.2.2: All_{V}

Sia *All* la grammatica definita nella Definizione 3.4.1.1; per poter valutare le sue espressioni attraverso la semantica *call-by-value*, è necessario definire la seguente regola di inferenza:

- **call-by-value**:

$$\forall y \in \mathrm{Var}, M \in Exp, v \in \mathrm{Val}, p \in Imp \quad \exists l \in \mathrm{Loc} - \mathrm{dom}(S) \wedge E(y) = (x, p, E') \implies$$

$$\implies [call]_{value} \frac{E \vdash M,\ S \overset{M}{\rightsquigarrow} v \quad E'\{(x,l)\} \vdash p,\ S\{(l,v)\} \overset{p}{\rightsquigarrow} S'}{E \vdash call\ y(M),\ S \overset{p}{\rightsquigarrow} S'}$$

per certi $E, E' \in \mathrm{Env}, S, S' \in \mathrm{Store}$.

Proposizione 3.4.2.3: All_{R}

Sia *All* la grammatica definita nella Definizione 3.4.1.1; per poter valutare le sue espressioni attraverso la semantica *call-by-reference*, è necessario definire la seguente regola di inferenza:

- **call-by-reference**:

$$\forall y \in \mathrm{Var}, M \in Exp, v \in \mathrm{Val}, p \in Imp \quad \exists l \in \mathrm{Loc} - \mathrm{dom}(S) \wedge E(y) = (x, p, E') \implies$$

$$\implies [call]_{ref} \frac{E \vdash V,\ S \overset{V}{\rightsquigarrow} l \quad E'\{(x,l)\} \vdash p,\ S \overset{p}{\rightsquigarrow} S'}{E \vdash call\ y(V),\ S \overset{p}{\rightsquigarrow} S'}$$

per certi $E, E' \in \mathrm{Env}, S, S' \in \mathrm{Store}$.

Proposizione 3.4.2.4: All_{N}

Sia *All* la grammatica definita nella Definizione 3.4.1.1; per poter valutare le sue espressioni attraverso la semantica *call-by-name*, è necessario ridefinire l'insieme degli ambienti, come segue

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

ed inoltre vengono aggiunte le due regole di inferenza che seguono:

- **locazioni**:

$$\forall x \in \mathrm{Var}, l \in \mathrm{Loc}, V \in LExp \quad E(x) = (V, E') \implies [loc_3] \frac{E' \vdash V,\ S \overset{V}{\rightsquigarrow} l}{E \vdash x,\ S \overset{V}{\rightsquigarrow} l}$$

- **call-by-name**:

$$\forall x, y \in \mathrm{Var}, V \in LExp, p \in Imp$$

$$E(y) = (x, p, E') \implies [call]_{name} \frac{E'\{(x, (V, E))\} \vdash p,\ S \overset{p}{\rightsquigarrow} S'}{E \vdash call\ y(V),\ S \overset{p}{\rightsquigarrow} S'}$$

per certi $E, E' \in \mathrm{Env}, S, S' \in \mathrm{Store}$.

Lemma 3.4.2.1: Semantiche di *All*

Sia *All* la grammatica definita nella Definizione 3.4.1.1; allora, si ha che

$$All_{\mathrm{V}} \not\equiv All_{\mathrm{R}} \not\equiv All_{\mathrm{N}}$$

Dimostrazione. TODO. □

4

Correttezza dei programmi

4.1 Correttezza nel paradigma imperativo

4.1.1 Logica di Hoare

Definizione 4.1.1.1: Tripla di Hoare

Si definisce **tripla di Hoare** la seguente clausola:

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

dove A — che prende il nome di **precondizione** — e B — che prende il nome di **postcondizione** — sono espressioni booleane, e p è un programma. La sua *interpretazione di correttezza parziale* è la seguente: "stando in uno stato che soddisfa A, ed eseguendo p a partire da quest'ultimo, se p termina, allora esso terminerà in uno stato che soddisfa B".

Definizione 4.1.1.2: Formula

Si definisce **formula** una proposizione appartenente alla seguente grammatica:

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

dove A, B e C sono espressioni booleane, e p è un programma.

Proposizione 4.1.1.1: Inferenza delle triple di Hoare

Si considerino le formule descritte all'interno della Definizione 4.1.1.2; allora, si definiscono le seguenti regole di inferenza:

- **true**:
$$\{A\}\ p\ \{true\}$$
poiché, per qualsiasi programma p, indipendentemente dalla precondizione A, ogni espressione booleana di postcondizione soddisferà $true$;
- **false**:
$$\{false\}\ p\ \{B\}$$
poiché la tripla di Hoare è vera a vuoto, in quando la precondizione non può essere soddisfatta poiché è $false$;
- **strengthening**:
$$\frac{A \supset B \quad \{B\}\ p\ \{C\}}{\{A\}\ p\ \{C\}}$$
poiché se da A implica B, e si raggiunge uno stato che soddisfa C quando p termina partendo da uno stato che soddisfa B, allora sicuramente partendo da uno stato che soddisfa A, quando p termina, C sarà soddisfatta dallo stato terminale;
- **weakening**:
$$\frac{B \supset A \quad \{C\}\ p\ \{B\}}{\{C\}\ p\ \{A\}}$$
poiché se da B implica A, e si raggiunge uno stato che soddisfa B quando p termina partendo da uno stato che soddifa C, allora sicuramente partendo da uno stato che soddisfa C, quando p termina, A sarà soddisfatta dallo stato terminale; si noti che le regole di *strengthening* e di *weakening* sono l'una il duale dell'altra;
- **and**:
$$\frac{\{A\}\ p\ \{B_0\} \quad \{A\}\ p\ \{B_1\} \quad \dots \quad \{A\}\ p\ \{B_n\}}{\{A\}\ p\ \{B_0 \wedge B_1 \wedge \dots \wedge B_n\}}$$
poiché, se partendo da uno stato che soddisfa A, una volta terminata l'esecuzione, p raggiunge uno stato che soddisfa B_0 e $B_1, \dots, B_n$, allora soddisferà sicuramente anche $B_0 \wedge B_1 \wedge \dots \wedge B_n$;
- **or**:
$$\frac{\{B_0\}\ p\ \{A\} \quad \{B_1\}\ p\ \{A\} \quad \dots \quad \{B_n\}\ p\ \{A\}}{\{B_0 \vee B_1 \vee \dots \vee B_n\}\ p\ \{A\}}$$
poiché, se partendo da un qualsiasi stato tra B_0 e $B_1, \dots, B_n$, una volta terminata l'esecuzione, p raggiunge uno stato che soddisfa A, allora lo stato iniziale soddisferà anche $B_0 \vee B_1 \vee \dots \vee B_n$; si noti che le regole *and* ed *or* sono l'una il duale dell'altra;

Definizione 4.1.1.3: Logica di Hoare

Si definisce **logica di Hoare** la seguente grammatica:

$$M, N ::= k \mid x \mid M + N \mid M * N$$
$$A, B ::= true \mid false \mid A \supset B \mid M < N \mid M = N$$
$$p, q ::= skip \mid p; q \mid x := M \mid if\ B\ then\ p\ else\ q \mid while\ B\ do\ p$$

dove la prima grammatica comprende le espressioni, la seconda le espressioni booleane, e la terza i programmi.

Proposizione 4.1.1.2: Regole della logica di Hoare

I seguenti sono i *significati assiomatici* (o *regole di inferenza speciali*), della logica di Hoare:

- **skip**:
$$\{A\}\ skip\ \{A\}$$
poichè la clausola *skip* non effettua nulla;
- **composizioni sequenziali**:
$$\frac{\{A\}\ p\ \{B\} \quad \{B\}\ q\ \{C\}}{\{A\}\ p; q\ \{C\}}$$
poiché al fine di effettuare p e q in sequenza è necessaria una condizione intermedia (nell'esempio B);
- **if-then-else**:
$$\frac{\{A \wedge C\}\ p\ \{B\} \quad \{A \wedge \neg C\}\ q\ \{B\}}{\{A\}\ if\ C\ then\ p\ else\ q\ \{B\}}$$
poiché entrambe le triple di Hoare devono avere B come postcondizione, ma deve essere eseguito p se si verifica C, altrimenti q, e dunque seguono le precondizioni descritte; si noti che anche la seguente regola di inferenza è una valida alternativa
$$\frac{\{A\}\ p\ \{B\} \quad \{A\}\ q\ \{B\}}{\{A\}\ if\ C\ then\ p\ else\ q\ \{B\}}$$
ma questa risulta essere una regola *più debole* della precedente, poiché le triple $\{A\}\ p\ \{B\}$ e $\{A\}\ q\ \{B\}$ sono *più forti*, in quanto hanno meno precondizioni da verificare, ma questo le rende anche più difficili da dimostrare;
- **while**:
$$\frac{\{A \wedge C\}\ p\ \{A\}}{\{A\}\ while\ C\ do\ p\ \{A \wedge \neg C\}}$$
TODO
- **assegnazioni**:
$$\{B[M/x]\}\ x := M\ \{B\}$$
TODO

Definizione 4.1.1.4: Invarianti

Sia A un'espressione booleana della logica di Hoare; essa è detta **invariante** se e solo se, per qualche espressione booleana B e programma p, si ha che

$$\{A\}\ \mathit{while}\ B\ \mathit{do}\ p\ \{A \wedge \neg B\}$$

ovvero, la condizione A è verificata *prima e dopo* l'esecuzione di p.

Esempio 4.1.1.1 (Correttezza di programmi)**.** Si consideri il seguente programma — espresso in termini della logica di Hoare, definita nella Definizione 4.1.1.3 — in grado di calcolare quoziente e resto dati dal rapporto di due numeri in input (nel programma, x ed y):

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

È possibile dimostrarne la correttezza attraverso il seguente albero di valutazione:

$$(*)\ \frac{x \geq 0 \supset (x = 0 \cdot y + x \wedge x \geq 0) \qquad \dfrac{\{x = 0 \cdot y + x \wedge x \geq 0\}\ b := x\ \{A_1\}}{\{A_1[x/b]\}\ b := x\ \{A_1\}}}{\{x \geq 0\}\ b := x\ \{x = 0 \cdot y + b \wedge b \geq 0\}}$$

$$(**)\ \frac{A \wedge b \geq y \supset (x = (a+1) \cdot y + (b - y) \wedge b - y \geq 0) \qquad \dfrac{\{x = (a+1) \cdot y + (b-y) \wedge b - y \geq 0\}\ b := b - y\ \{A_2\}}{\{A_2[(b-y)/b]\}\ b := b - y\ \{A_2\}}}{\{A \wedge b \geq y\}\ b := b - y\ \{x = (a+1) \cdot y + b \wedge b \geq 0\}}$$

$$\frac{(*)\ \dfrac{\{x = 0 \cdot y + b \wedge b \geq 0\}\ a := 0\ \{A\}}{\{A[0/a]\}\ a := 0\ \{A\}} \qquad \dfrac{(**)\ \dfrac{\{x = (a+1) \cdot y + b \wedge b \geq 0\}\ a := a + 1\ \{A\}}{\{A[(a+1)/a]\}\ a := a + 1\ \{A\}}}{\{A \wedge b \geq y\}\ b := b - y;\ a := a + 1\ \{A\}}}{\{x \geq 0\}\ b := x;\ a := 0;\ \mathit{while}\ b \geq y\ \mathit{do}\ (b := b - y;\ a := a + 1)\ \{x = a \cdot y + b \wedge 0 \leq b < y\}}$$

dove

$$\begin{aligned} A &:= x = a \cdot y + b \wedge b \geq 0 \\ A_1 &:= A[0/a] \longrightarrow x = 0 \cdot y + b \wedge b \geq 0 \\ A_2 &:= A[(a+1)/a] \longrightarrow x = (a+1) \cdot y + b \wedge b \geq 0 \end{aligned}$$

Si noti che la postcondizione scelta per dimostrare la correttezza del programma, ovvero

$$x = a \cdot y + b \wedge 0 \leq b < y$$

è valida poiché ad ogni ciclo, fintanto che $b \geq y$, il programma sottrae y da b, ed aggiunge 1 ad a, dunque a risulta essere il numero di volte che è stato possibile sottrarre y da b — che rappresenta la parte intera del quoziente $\frac{x}{y}$ — mentre b è il resto della divisione — poiché è la parte che non è stato possibile sottrarre da b.

Inoltre, la tripla di Hoare è sempre soddisfatta, poiché nel caso limite in cui $y \geq 0$, il programma non termina e dunque la tripla è soddisfatta a vuoto.

4.2 Correttezza nel paradigma funzionale

4.2.1 Punto fisso

Definizione 4.2.1.1: Punto fisso

Data una funzione $f : X \to X$, un elemento $x \in X$ è detto **punto fisso di** f se e solo se $f(x) = x$.

Esempio 4.2.1.1 (Punti fissi)**.** Sia $f(x) = x^2 - 3x + 4$; allora, poiché $f(2) = 2^2 - 3 \cdot 2 + 4 = 4 - 6 + 4 = 2$, 2 è un punto fisso di f.

Esempio 4.2.1.2 (Funzioni come punti fissi)**.** Si consideri la funzione

$$F(g) := h(x) = \begin{cases} 1 & x = 0 \\ x \cdot g(x-1) & x > 0 \end{cases}$$

che prende in input una funzione; per vedere come opera, calcolando ad esempio $F(\text{succ})$, si ottiene la funzione

$$F(\text{succ}) := h(x) = \begin{cases} 1 & x = 0 \\ x \cdot \text{succ}(x-1) = x \cdot x = x^2 & x > 0 \end{cases}$$

che restituisce 1 se x è 0, altrimenti restituisce x^2.

Allora, il punto fisso di F è la funzione seguente

$$\text{fact}(x) = \begin{cases} 1 & x = 0 \\ x \cdot \text{fact}(x-1) & x > 0 \end{cases}$$

che computa il fattoriale di un numero; infatti, si ha che

$$F(\text{fact}) := h(x) = \begin{cases} 1 & x = 0 \\ x \cdot \text{fact}(x-1) & x > 0 \end{cases} \equiv: \text{fact}$$

4.2.2 Ricorsione

Definizione 4.2.2.1: Combinatore di Kleene

Si definisce **combinatore di Kleene**, o **operatore di punto fisso**, la seguente espressione esprimibile attraverso la grammatica Fun della Definizione 2.3.1.3:

$$\text{Y} := fn\ f \Rightarrow ((fn\ x \Rightarrow f(xx))(fn\ x \Rightarrow f(xx)))$$

Proposizione 4.2.2.1: Punto fisso di una funzione

Data una funzione, il combinatore di Kleene ne restituisce il punto fisso.

Dimostrazione. Se il combinatore di Kleene è in grado di restituire il punto fisso di una funzione, allora data una funzione h, si ha che $\mathrm{Y}h$ è il suo punto fisso, e dunque per definizione

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

Allora, svolgendo i calcoli, si ottiene che

$$\mathrm{Y}h \xrightarrow{\beta} (fn\ x \Rightarrow h(xx))(fn\ x \Rightarrow h(xx)) \longrightarrow h((fn\ x \Rightarrow h(xx))(fn\ x \Rightarrow h(xx))) \xrightarrow{\beta} h(\mathrm{Y}h)$$

□

Osservazione 4.2.2.1: Ricorsione attraverso Y

Si noti che, poiché per una funzione h è vero che $h(\mathrm{Y}h) = \mathrm{Y}h$, si ha che

$$\begin{gathered} h(\mathrm{Y}h) = \mathrm{Y}h \\ h(h(\mathrm{Y}h)) = \mathrm{Y}h \\ \vdots \\ h(\ldots(h(\mathrm{Y}h))) = \mathrm{Y}h \end{gathered}$$

TODO DA FINIRE

Lemma 4.2.2.1: Ricorsione debole terminante

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

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

Dunque f risulta essere l'unico omomorfismo tra le algebre $(\mathbb{N}, \mathrm{succ}, \mathrm{zero})$ e (A, h, zero_A), per qualche funzione nullaria $\mathrm{zero}_A : \mathbb{1} \to A : x \mapsto a$.

Dimostrazione. Poiché le algebre $(\mathbb{N}, \mathrm{succ}, \mathrm{zero})$ e (A, h, zero_A) hanno la stessa segnatura, e l'algebra dei numeri naturali è induttiva — come mostrato nell'Esempio 1.1.2.5 — per la Proposizione 1.1.3.2 esiste unico un omomorfismo $f : \mathbb{N} \to A$, dunque per definizione stessa di omomorfismo, si verifica che

$$f : \mathbb{N} \to A : \begin{cases} f(0) = f(\mathrm{zero}(x)) = \mathrm{zero}_A(f(x)) = a \\ f(\mathrm{succ}(n)) = h(f(n)) \end{cases}$$

poichè $\forall x \in \mathbb{1} \quad \mathrm{zero}(x) = 0$ ed inoltre $\forall x \in \mathbb{1} \quad \mathrm{zero}_A(x) = a$. □

Definizione 4.2.2.2: Operatore ρ

Si definisce **operatore** ρ l'operatore che, attraverso la sintassi

$$\rho \ \texttt{\_expression\_}_1 \ \texttt{\_expression\_}_2$$

restituisce l'omomorfismo descritto all'interno del Lemma 4.2.2.1, dove `_expression_`$_1$ rappresenta il costruttore nullario dell'algebra, e `_expression_`$_2$ costituisce la funzione $h : A \to A$.

Osservazione 4.2.2.2: Operatore ρ

Si noti che, per definizione dell'operatore ρ, si verifica che

$$\rho : \begin{cases} (\rho \text{ zero } h)\ 0 = a \\ (\rho \text{ zero } h)\ (\text{succ } n) = h\ ((\rho \text{ zero } h)\ n) \end{cases}$$

espresso attravreso i termini delle funzioni definite all'interno del Lemma 4.2.2.1.

Esempio 4.2.2.1 (Operatore ρ)**.** Si considerino le algebre dei numeri naturali $(\mathbb{N}, \text{succ}, \text{zero})$, e dei booleani di Church $(\mathbb{B}_\lambda, \text{not}_\lambda, \text{True}_\lambda)$, dove

$$\text{True}_\lambda : \mathbb{1} \to \mathbb{B}_\lambda : x \mapsto \text{true}_\lambda$$

Allora, si ha che $(\rho\ \text{True}_\lambda\ \text{not}_\lambda)$ è una funzione tale che

$$\begin{cases} (\rho\ \text{True}_\lambda\ \text{not}_\lambda)0 = \text{true}_\lambda \\ (\rho\ \text{True}_\lambda\ \text{not}_\lambda)(\text{succ } n) = \text{not}_\lambda((\rho\ \text{True}_\lambda\ \text{not}_\lambda)n) \end{cases}$$

e, considerando la seguente funzione, scritta in forma ricorsiva

$$\text{isEven} : \mathbb{N} \to \mathbb{B}_\lambda : \begin{cases} \text{isEven}(0) = \text{true}_\lambda \\ \text{isEven}(\text{succ}(n)) = \text{not}_\lambda(\text{isEven}(n)) \end{cases}$$

che è in grado di restituire la parità di un numero naturale, è facilmente verificabile che

$$(\rho\ \text{True}_\lambda\ \text{not}_\lambda) \equiv \text{isEven}$$
$$(\rho\ \text{False}_\lambda\ \text{not}_\lambda) \equiv \text{isFalse}$$

Lemma 4.2.2.2: Funzione ricorsiva primitiva

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

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

Dunque f risulta essere l'unico omomorfismo tra le algebre $(\mathbb{N}, \text{succ}, \text{zero})$ e (A, h, zero_A), per qualche funzione nullaria $\text{zero}_A : \mathbb{1} \to A : x \mapsto a$.

Dimostrazione. TODO □

Definizione 4.2.2.3: Operatore *rec*

Si definisce **operatore** *rec* l'operatore che, attraverso la sintassi

$$rec \ \texttt{\_expression\_}_1 \ \texttt{\_expression\_}_2$$

restituisce l'omomorfismo descritto all'interno del Lemma 4.2.2.2, dove $\texttt{\_expression\_}_1$ rappresenta il costruttore nullario dell'algebra, e $\texttt{\_expression\_}_2$ costituisce la funzione $h : A \times \mathbb{N} \to A$.

Osservazione 4.2.2.3: Operatore *rec*

Si noti che, per definizione dell'operatore *rec*, si verifica che

$$rec : \begin{cases} (rec \text{ zero } h) \ 0 = a \\ (rec \text{ zero } h) \ (\text{succ } \ n) = h \ ((rec \text{ zero } h) \ n) \ n \end{cases}$$

espresso attraverso i termini delle funzioni definite all'interno del Lemma 4.2.2.2.

Esempio 4.2.2.2 (Operatore *rec*)**.** È possibile esprimere la funzione isEven definita nell'Esempio 4.2.2.1, attraverso l'operatore *rec*, come segue:

$$\text{isEven} \equiv (\rho \ \text{True}_\lambda \ \text{not}_\lambda) \equiv (rec \ \text{True}_\lambda \ (fn \ xn \Rightarrow \text{not}_\lambda \ x))$$

Definizione 4.2.2.4: Grammatica Fun_ρ

Si consideri la grammatica definita all'interno della Definizione 2.3.1.3; la seguente è una sua estensione, che prenderà il nome di Fun_ρ, che include al suo interno una clausola con l'operatore *rec*:

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

dove si noti che 0 non sta ad indicare tutte le costanti, ma esclusivamente lo 0, in quanto non è necessario includere le altre poiché la grammatica è fornita della funzione succ.

Esempio 4.2.2.3 (Operatore *rec* in Fun_ρ)**.** Sia $\text{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (x, y) \mapsto x + y$ la funzione che somma x ad y, definita ricorsivamente — attraverso le espressioni della grammatica Fun_ρ definita nella Definizione 4.2.2.4 — come segue:

$$\text{plus} : \begin{cases} \text{plus } 0 \ y = y \\ \text{plus } (\text{succ } \ n) \ y = \text{succ } (\text{plus } n \ y) \end{cases}$$

si noti dunque che è possibile esprimerla attraverso l'operatore *rec* come segue:

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

poiché

$$\text{plus } 0\ y \equiv rec\ y\ (fn\ wz \Rightarrow \text{succ}\ \ w)\ 0 = y$$

(si noti l'Osservazione 4.2.2.3), ed inoltre

$$\begin{gathered}\text{plus (succ}\ \ n)\ y \equiv rec\ y\ (fn\ wz \Rightarrow \text{succ}\ \ w)\ \text{(succ}\ \ n) \equiv \\ \equiv (fn\ wz \Rightarrow \text{succ}\ \ w)\ (rec\ y\ (fn\ wz \Rightarrow \text{succ}\ \ w)\ n)\ n \equiv \\ \equiv (fn\ wz \Rightarrow \text{succ}\ \ n)\ \text{(plus}\ n\ y)\ n \longrightarrow \text{succ (plus}\ n\ y)\end{gathered}$$