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

Linguaggi di Programmazione

Author
Alessio Bandiera

12 gennaio 2024

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

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

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

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

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

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

v) per assurdo, sia $S \subseteq \aleph$ tale che $0_\aleph \in S$ e $\forall x_S \in S \quad \mathrm{succ}_\aleph(x_S) \in S$ 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 $\mathrm{succ}_\aleph$ per il secondo assioma di Peano, e dunque $\zeta_\aleph \neq 0_\aleph \implies \exists \zeta'_\aleph \in \aleph \mid \mathrm{succ}_\aleph(\zeta'_\aleph) = \zeta_\aleph$, e sicuramente $\zeta'_\aleph \notin S$, poiché altrimenti $\zeta_\aleph \in S$ anch'esso in quanto S è chiuso rispetto a $\mathrm{succ}_\aleph$; 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

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

Osservazione 1.1.1.1: Quinto assioma di Peano

Si noti che il quinto degli assiomi di Peano (della Definizione 1.1.1.1) equivale al principio di induzione (descritto nel 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 è detto **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 un insieme sostegno, e $\gamma_1, \ldots \gamma_n$ sono delle operazioni definite su A, allora con

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

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

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

- semigruppi
- monoidi
- gruppi
- gruppi abeliani

Esempio 1.1.2.2 (Algebre)**.** 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 avente cardinalità pari ad 1. L'insieme unità verrà indicato attraverso il simbolo $\mathbb{1}$, e dunque $|\mathbb{1}| = 1$.

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 $\gamma_1, \ldots, \gamma_n$ funzioni definite su A di arbitraria arietà; allora, $(A, \gamma_1, \ldots, \gamma_n)$ è definita **algebra induttiva** se si verificano le seguenti:

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

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

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

Le funzioni $\gamma_1, \ldots, \gamma_n$ 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 $a_1, \ldots, a_k \in \varnothing$, in esso ogni qualificazione è vera a vuoto; allora ogni algebra che ammette l'insieme vuoto 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, e dunque grazie al costruttore nullario è sempre presente almeno un elemento all'interno di essa.

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

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

Proposizione 1.1.2.1: Algebra induttiva di $\mathbb{N}$

Sia zero la funzione definita come segue

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

allora l'algebra $(\mathbb{N}, \text{zero}, \text{succ})$ è induttiva.

Dimostrazione. Si noti che:

i) zero e succ sono iniettive, poiché

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

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

iii) sia $S \subseteq \mathbb{N}$ tale che $\forall x \in S \quad \begin{cases} \text{zero}(x) \in S \\ \text{succ}(x) \in S \end{cases}$; si noti che, poiché $\forall x \in S \quad \text{zero}(x) \in S$, allora $0 \in S$, e poiché S è anche chiuso rispetto a succ, per la Definizione 1.1.1.1 si ha che $S = \mathbb{N}$ necessariamente.

Allora, segue la tesi. $\square$

Definizione 1.1.2.6: Omomorfismo

Un **omomorfismo** è una funzione tra due algebre aventi stessa segnatura, tale da preservare le loro strutture.

Formalmente, siano $(A, \mu_1, \ldots, \mu_n)$ e $(B, \delta_1, \ldots, \delta_n)$ 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; allora, una funzione $f : A \to B$ è detta **omomorfismo** tra le due algebre, se e solo se

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

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

Esempio 1.1.2.5 (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 1.1.2.2: 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, queste sono dette **isomorfe**, e tale proprietà è indicata dal simbolismo

$$A \cong B$$

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

Esempio 1.1.2.6 (Isomorfismi)**.** Si consideri l'omomorfismo dell'Esempio 1.1.2.5; 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 $(A, \gamma_1, \ldots, \gamma_n)$, per ogni algebra — *non necessariamente induttiva* — $(B, \delta_1, \ldots, \delta_n)$, 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 $(A, \gamma_1, \ldots, \gamma_n)$ e $(B, \delta_1, \ldots, \delta_n)$ 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 $f : A \to B$; viceversa, poiché B è un'algebra induttiva, ed A ha la stessa segnatura di B, esiste unico un omomorfismo $g : B \to A$. Si noti che, per la Proposizione 1.1.2.2, 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` identificherà l'insieme di tutti gli oggetti aventi tipo `T`.

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

$$\mathtt{[}a_1, \ldots, a_n\mathtt{]}$$

Definizione 1.2.1.2: Algebra delle liste finite

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

$$(\mathtt{List<T>}, \mathrm{empty}, \mathrm{cons})$$

dove i costruttori sono i seguenti:

$$\begin{gathered} \mathrm{empty} : \mathbb{1} \to \mathtt{List<T>} : x \mapsto \mathtt{[]} \\ \mathrm{cons} : \mathtt{List<T>} \times \mathtt{T} \to \mathtt{List<T>} : (\mathtt{[}a_1, \ldots, a_n\mathtt{]}, x) \mapsto \mathtt{[}a_1, \ldots, a_n, x\mathtt{]} \end{gathered}$$

Proposizione 1.2.1.1: Liste finite induttive

L'algebra delle liste finite è induttiva.

Dimostrazione. Si noti che:

- empty è iniettiva per l'Osservazione 1.1.2.1
- $\forall l, l' \in \mathtt{List<T>}, x, x' \in \mathtt{T} \quad \mathrm{cons}(l, x) = \mathrm{cons}(l', x') \implies \begin{cases} l = l' \\ x = x' \end{cases}$ altrimenti l ed l' avrebbero avuto lunghezze diverse, oppure avrebbero contenuto diversi elementi;
- $\mathrm{im}(\mathrm{empty}) \cap \mathrm{im}(\mathrm{cons}) = \varnothing$, poiché solo empty può restituire `[]`, in quanto cons restituisce sempre una lista contenente almeno l'elemento fornito in input;
- sia $S \subseteq \mathtt{List<T>}$ tale da essere chiuso rispetto ad empty e cons — dunque contenente la lista vuota; per assurdo, sia $\mathtt{List<T>} - S \neq \varnothing \iff \exists l \in \mathtt{List<T>} - S$, ma `List<T>` è chiuso rispetto a cons, ed in particolare $\exists x \in \mathtt{T}, l' \in \mathtt{List<T>} \mid \mathrm{cons}(l', x) = l$, ma poichè $l \notin S$, allora necessariamente $l' \notin S$ poiché S è chiuso rispetto a cons; dunque, ripetendo tale ragionamento induttivamente, si ottiene che S è vuoto, ma questo è impossibile poiché $\mathtt{[]} \in S$ per chiusura su empty $\lightning$.

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:

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

Esempio 1.2.1.1 (Concatenazioni di liste finite)**.** Per concatenare le liste

$$\begin{gathered} \texttt{[}1, 2\texttt{]} \\ \texttt{[}3, 4, 5\texttt{]} \end{gathered}$$

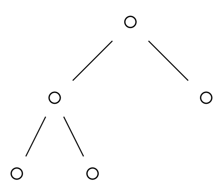
è sufficiente applicare la funzione concat, e dunque si ottiene

$$\begin{gathered} \mathrm{concat}(\texttt{[}1,2\texttt{]}, \texttt{[}3,4,5\texttt{]}) = \\ = \mathrm{cons}(\mathrm{concat}(\texttt{[}1,2\texttt{]}, \texttt{[}3,4\texttt{]}), 5) = \mathrm{cons}(\mathrm{cons}(\mathrm{concat}(\texttt{[}1,2\texttt{]}, \texttt{[}3\texttt{]}), 4), 5) = \\ = \mathrm{cons}(\mathrm{cons}(\mathrm{cons}(\mathrm{concat}(\texttt{[}1,2\texttt{]}, \texttt{[]}), 3), 4), 5) = \mathrm{cons}(\mathrm{cons}(\mathrm{cons}(\texttt{[}1,2\texttt{]}, 3), 4), 5) = \\ = \mathrm{cons}(\mathrm{cons}(\texttt{[}1,2,3\texttt{]}, 4), 5) = \mathrm{cons}(\texttt{[}1,2,3,4\texttt{]}, 5) = \texttt{[}1,2,3,4,5\texttt{]} \end{gathered}$$

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:

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

$$\text{branch} : \texttt{B-tree} \times \texttt{B-tree} \to \texttt{B-tree} : (b, b') \mapsto \begin{array}{ccc} & \circ & \\ / & & \backslash \\ b & & b' \end{array}$$

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

$$\begin{array}{ccc} & \circ & \\ / & & \backslash \\ t & & t' \end{array}$$

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

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

$\square$

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*, in quanto può essere sostituita con regole definite dalla grammatica;
- 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**, oppure **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:

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

essa definisce le regole per utilizzare i numeri in $\mathbb{N}$, ammettendo inoltre le operazioni sintattiche di somma e prodotto. Dunque, essa è composta da:

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

Dunque, segue che

$$\text{Var} = \{x\}$$
$$\text{Val} = \{0, 1, \ldots\}$$

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*)**.** Considerando ad esempio le stringhe `"4"` e `"23"`, si può ottenere la stringa

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

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

Osservazione 2.2.1.1: 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:

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

Osservazione 2.2.1.2: 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 $\mathrm{im}(+) \cap \mathrm{im}(*) \neq \varnothing$.

Osservazione 2.2.1.3: Disambiguazione di *Exp*

Si noti che l'ambiguità trattata nell'Osservazione 2.2.1.2 non permetterebbe di poter definire la funzione eval, descritta nell'Osservazione 2.2.1.1. 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*

La clausola *let* verrà utilizzata attraverso la sintassi

$$let\ \texttt{variable} = \texttt{expression}_1\ in\ \texttt{expression}_2$$

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

Una variabile alla quale non è stata assegnata nessuna espressione prende il nome di **variabile libera** (*free variable*); 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:

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

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

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 $x, y \in \mathrm{Var}$, 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:

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

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

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

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

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

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

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 $E \in \mathrm{Env}$, 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

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

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

$$E_1E_2 = \{(z, 3), (y, 9)\}\{(z, 4)\} = \{(z, 4), (y, 9)\}$$

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**:
$$\exists v \in \text{Val} \mid E(x) = v \implies [vars]\ E \vdash x \rightsquigarrow v$$
- **somme**:
$$\exists v \in \text{Val} \mid v = v' + v'' \implies [plus]\ \frac{E \vdash M \rightsquigarrow v' \quad E \vdash N \rightsquigarrow v''}{E \vdash M + N \rightsquigarrow v}$$
- **prodotti**:
$$\exists v \in \text{Val} \mid v = v' \cdot v'' \implies [times]\ \frac{E \vdash M \rightsquigarrow v' \quad E \vdash N \rightsquigarrow v''}{E \vdash M * N \rightsquigarrow v}$$
- **dichiarazioni ed assegnazioni**:
$$[let]\ \frac{E \vdash M \rightsquigarrow v' \quad E\{(x, v')\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$

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:

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

e viene indicato con il simbolismo

$$M \sim N$$

Definizione 2.2.4.3: Albero di valutazione

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

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

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

è il seguente

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

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

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** (detto anche *lexical scope*) valuta una data espressione della grammatica utilizzando l'ambiente definito in tempo di interpretazione. 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 una data espressione della grammatica utilizzando l'ambiente definito in tempo di valutazione. 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 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

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

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:

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

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:

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

- **valutazioni**:

$$x \in \mathrm{Var} \quad x \in \mathrm{dom}(E) \land M := E(x) \implies \frac{E \vdash M \leadsto v}{E \vdash x \leadsto v}$$

- **assegnazioni**:

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

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:

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

- **valutazioni**:

$$x \in \mathrm{dom}(E) \land (M, E') := E(x) \implies \frac{E' \vdash M \leadsto v}{E \vdash x \leadsto v}$$

- **assegnazioni**

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

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

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

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

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

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

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

dove

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

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

La clausola fn 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

La clausola di applicazione verrà utilizzata attraverso la sintassi

```
expression expression
```

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:

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

Esempio 2.3.1.1 (Funzioni come argomenti)**.** Sia la seguente

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

un'espressione di Fun; 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 la seguente

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

un'espressione di Fun; 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:

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

Il processo inverso prende il nome di *uncurryficazione*.

Esempio 2.3.1.3 (Curryficazioni)**.** Sia la seguente

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

un'espressione di Fun; 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:

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

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

$$\mathrm{Val} := \{\mathtt{0}, \mathtt{1}, \dots\} \cup (\mathrm{Var} \times Fun)$$

- **funzioni**:

$$[fn]\ E \vdash fn\ x \Rightarrow M \leadsto (x, M)$$

- **applicazioni**:

$$[appl]\ \frac{E \vdash fn\ x \Rightarrow L \leadsto (x, L) \quad E \vdash N \leadsto v' \quad E\{(x, v')\} \vdash L \leadsto v}{E \vdash (fn\ x \Rightarrow L)N \leadsto v}$$

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:

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

- **funzioni**:

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

- **applicazioni**:

$$[appl]\ \frac{E \vdash fn\ x \Rightarrow L \rightsquigarrow (x, L, E') \quad E \vdash N \rightsquigarrow v' \quad E'\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash (fn\ x \Rightarrow L)N \rightsquigarrow v}$$

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

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

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

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

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

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:

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

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

dove

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

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:

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

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

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

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

- **funzioni**:

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

- **applicazioni**:

$$[appl]\ \dfrac{E \vdash fn\ x \Rightarrow L \rightsquigarrow (x, L) \quad E'\{(x,N)\} \vdash L \rightsquigarrow v}{E \vdash (fn\ x \Rightarrow L)N \rightsquigarrow v}$$

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:

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

- **funzioni**:

$$\forall E \in \text{Env}, x \in \text{Var}, M \in Fun \quad E \vdash fn\ x \Rightarrow M \rightsquigarrow (x, M, E)$$

- **applicazioni**:

$$[appl]\ \frac{E \vdash fn\ x \Rightarrow L \rightsquigarrow (x, L, E') \quad E'\{(x, (N, E))\} \vdash L \rightsquigarrow v}{E \vdash (fn\ x \Rightarrow L)N \rightsquigarrow v}$$

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

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

Dimostrazione. Si noti che:

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

mentre, per quanto riguarda $Fun_{\mathrm{ED}} \not\equiv Fun_{\mathrm{LD}}$ e $Fun_{\mathrm{ES}} \not\equiv Fun_{\mathrm{LS}}$, 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:

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

[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:

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

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

$$\begin{gathered}(fn\ zxy \Rightarrow zx(xy))(fn\ xy \Rightarrow y) \longrightarrow fn\ xy \Rightarrow (fn\ xy \Rightarrow y)x(xy) \longrightarrow \\ \longrightarrow fn\ xy \Rightarrow (fn\ y \Rightarrow y)(xy) \longrightarrow fn\ xy \Rightarrow xy =: 1_\lambda\end{gathered}$$

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

$$\begin{gathered}(fn\ zxy \Rightarrow zx(xy))(fn\ xy \Rightarrow xy) \longrightarrow fn\ xy \Rightarrow (fn\ xy \Rightarrow xy)x(xy) \longrightarrow \\ \longrightarrow fn\ xy \Rightarrow (fn\ y \Rightarrow xy)(xy) \longrightarrow fn\ xy \Rightarrow xxy =: 2_\lambda\end{gathered}$$

Lemma 2.4.1.1: Algebra dei numeri di Church

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

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

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

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

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 $(\mathbb{N}, \mathrm{zero}, \mathrm{succ})$ e $(\mathbb{N}_\lambda, \mathrm{zero}_\lambda, \mathrm{succ}_\lambda)$:

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

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:

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

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

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

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

Si noti che 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

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

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.

2.4.4 Ricorsione

Definizione 2.4.4.1: 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 2.4.4.1 (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 2.4.4.2 (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

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

Definizione 2.4.4.2: 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 2.4.4.1: 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

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

$\square$

Osservazione 2.4.4.1: Ricorsione attraverso Y

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

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

Questo permette di implementare la ricorsione all'interno del paradigma funzionale, come segue: si consideri la funzione $F(g)$ definita all'interno del Esempio 2.4.4.2; è possibile definire dunque una funzione simile ad essa, ovvero

$$h := fn\ rn \Rightarrow \begin{cases} 1 & n = 0 \\ n \cdot (r\ (n-1)) & n > 0 \end{cases}$$

il cui punto fisso è ancora la funzione che computa il fattoriale di n. Inoltre, si noti che:

$$\begin{gathered} (\mathrm{Y}h)\ k = \\ = h\ (\mathrm{Y}h)\ k = \\ = \left(fn\ rn \Rightarrow \begin{cases} 1 & n = 0 \\ n \cdot (r\ (n-1)) & n > 0 \end{cases} \right) (\mathrm{Y}h)\ k = \\ = \left(fn\ n \Rightarrow \begin{cases} 1 & n = 0 \\ n \cdot ((\mathrm{Y}h)\ (n-1)) & n > 0 \end{cases} \right) k = \\ = \begin{cases} 1 & k = 0 \\ k \cdot ((\mathrm{Y}h)\ (k-1)) & k > 0 \end{cases} \end{gathered}$$

dunque, segue che

$$\begin{gathered} (\mathrm{Y}h)\ 3 = \\ = 3 \cdot ((\mathrm{Y}h)\ 2) = \\ = 3 \cdot 2 \cdot ((\mathrm{Y}h)\ 1) = \\ = 3 \cdot 2 \cdot 1 \cdot (1) = \mathrm{fact}(3) \end{gathered}$$

2.5 Fun_ρ: un linguaggio funzionale ricorsivo

2.5.1 Operatori di ricorsione

Lemma 2.5.1.1: Ricorsione tramite numeri naturali

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

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

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. Poiché le algebre $(\mathbb{N}, \text{succ}, \text{zero})$ e (A, h, zero_A) hanno la stessa segnatura, e l'algebra dei numeri naturali è induttiva — come mostrato nell'Proposizione 1.1.2.1 — 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

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

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

Definizione 2.5.1.1: 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 2.5.1.1, dove $\texttt{expression}_1$ rappresenta a — ovvero lo zero dell'algebra (A, h, zero_A) — ed $\texttt{expression}_2$ costituisce il costruttore $h : A \to A$.

Osservazione 2.5.1.1: Operatore ρ

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

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

espresso attravreso i termini delle funzioni definite all'interno del Lemma 2.5.1.1.

Esempio 2.5.1.1 (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

$$\begin{gathered} (\rho \ \text{true}_\lambda \ \text{not}_\lambda) \equiv \text{isEven} \\ (\rho \ \text{false}_\lambda \ \text{not}_\lambda) \equiv \text{isFalse} \end{gathered}$$

Osservazione 2.5.1.2: Operatore ρ e $\mathbb{N}_\lambda$

Si consideri un'algebra (A, h, zero_A) tale che a sia il suo zero; per l'Osservazione 2.5.1.1, dunque, si verifica che

$$\begin{gathered} \rho \ a \ h \ (\underbrace{\text{succ} \ (\ \dots \ (\text{succ}}_{n \ \text{volte}} \ 0))) = \\ = h \ (\rho \ a \ h \ (\underbrace{\text{succ} \ (\ \dots \ (\text{succ}}_{n-1 \ \text{volte}} \ 0)))) = \dots = \\ \dots = \underbrace{h \ (\ \dots \ (h}_{n-1 \ \text{volte}} \ (\rho \ a \ h \ (\text{succ} \ 0)))) = \\ = \underbrace{h \ (\ \dots \ (h}_{n \ \text{volte}} \ (\rho \ a \ h \ 0))) = \underbrace{h \ (\ \dots \ (h}_{n \ \text{volte}} \ 0)) = n_\lambda \ 0 \end{gathered}$$

dunque vi è una stretta corrispondenza tra l'operatore ρ ed i numeri di Church.

Esempio 2.5.1.2 ($\mathbb{N}_\lambda$ con ρ)**.** Per l'Osservazione 2.5.1.2, ad esempio, si ha che

$$\begin{gathered} \rho \ a \ h \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))) = h \ (\rho \ a \ h \ (\text{succ} \ (\text{succ} \ 0))) = \\ = h \ (h \ (\rho \ a \ h \ (\text{succ} \ 0))) = h \ (h \ (h \ \rho \ a \ h \ 0)) = h \ (h \ (h \ 0)) \equiv 3_\lambda \ 0 \end{gathered}$$

Lemma 2.5.1.2: 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 $\text{zero}_A : \mathbb{1} \to A : x \mapsto a$.

Definizione 2.5.1.2: 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 2.5.1.1, dove $\texttt{expression}_1$ rappresenta a — ovvero lo zero dell'algebra (A, h, zero_A) — ed $\texttt{expression}_2$ costituisce il costruttore $h : A \times \mathbb{N} \to A$.

Osservazione 2.5.1.3: Operatore rec

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

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

espresso attraverso i termini delle funzioni definite all'interno del Lemma 2.5.1.2.

Proposizione 2.5.1.1: Equivalenza tra ρ e rec

Si consideri la grammatica Fun della Definizione 2.3.1.3; allora, date due sue espressioni M ed N, si ha che

$$\rho\ M\ \ N \equiv rec\ M\ \ (fn\ xy \Rightarrow Nx)$$

Dimostrazione. La dimostrazione procede per induzione su n.

Caso base. Per $n = 0$, si ha che

$$\rho\ M\ \ N\ 0 = M = rec\ M\ \ (fn\ xy \Rightarrow Nx)$$

per l'Osservazione 2.5.1.1 e l'Osservazione 2.5.1.3.

Ipotesi induttiva. Dato $n > 0$, si ha che

$$\rho\ M\ \ N\ \ n = rec\ M\ \ (fn\ xy \Rightarrow Nx)\ n$$

Passo induttivo. Per $n + 1 = \text{succ}(n)$, si ha che

$$rec\ M\ \ (fn\ xy \Rightarrow Nx)\ (\text{succ}\ \ n) = (fn\ xy \Rightarrow Nx)\ (rec\ M\ \ fn\ xy \Rightarrow Nx\ n)\ n =$$
$$= (fn\ xy \Rightarrow Nx)\ (\rho\ M\ \ N\ \ n)\ n \longrightarrow N\ (\rho\ M\ \ N\ \ n)$$

per ipotesi induttiva.

□

Esempio 2.5.1.3 (Equivalenze tra ρ e rec)**.** È possibile esprimere la funzione isEven definita nell'Esempio 2.5.1.1, 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 2.5.1.3: Grammatica Fun_ρ

Si consideri la grammatica Fun 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:

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

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 2.5.1.4 (Operatore rec in Fun_ρ)**.** Sia $\mathrm{plus} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} : (x, y) \mapsto x + y$ la funzione che somma x ad y, definita ricorsivamente — attraverso le espressioni della grammatica Fun_ρ definita nella Definizione 2.5.1.3 — come segue:

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

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

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

poiché

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

(si noti l'Osservazione 2.5.1.3), ed inoltre

$$\begin{aligned} \mathrm{plus}\ (\mathrm{succ}\ \ n)\ y = rec\ y\ (fn\ wz \Rightarrow \mathrm{succ}\ \ w)\ (\mathrm{succ}\ \ n) = \\ = (fn\ wz \Rightarrow \mathrm{succ}\ \ w)\ (rec\ y\ (fn\ wz \Rightarrow \mathrm{succ}\ \ w)\ n)\ n = \\ = (fn\ wz \Rightarrow \mathrm{succ}\ \ n)\ (\mathrm{plus}\ n\ y)\ n \longrightarrow \mathrm{succ}\ (\mathrm{plus}\ n\ y) \end{aligned}$$

Esempio 2.5.1.5 (Operatore rec in Fun_ρ)**.** Sia $\mathrm{twice} : \mathbb{N} \to \mathbb{N} : n \mapsto 2 \cdot n$ la funzione che raddoppia l'input fornito, definita ricorsivamente — attraverso le espressioni della grammatica Fun_ρ definita nella Definizione 2.5.1.3 — come segue:

$$\mathrm{twice} : \begin{cases} \mathrm{twice}\ 0 = 0 \\ \mathrm{twice}\ (\mathrm{succ}\ \ n) = \mathrm{succ}\ (\mathrm{succ}\ (\mathrm{twice}\ n)) \end{cases}$$

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

$$\mathrm{twice} \equiv rec\ 0\ (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))$$

poiché

$$\mathrm{twice}\ 0 = rec\ 0\ (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))\ 0 = 0$$

(si noti l'Osservazione 2.5.1.3), ed inoltre

$$\begin{aligned} \mathrm{twice}\ (\mathrm{succ}\ \ n) = rec\ 0\ (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))\ (\mathrm{succ}\ \ n) = \\ = (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))\ (rec\ 0\ (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))\ n)\ n = \\ = (fn\ xy \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ \ x))\ (\mathrm{twice}\ n)\ n \longrightarrow \\ \longrightarrow (fn\ y \Rightarrow \mathrm{succ}\ (\mathrm{succ}\ (\mathrm{twice}\ n)))\ n \longrightarrow \mathrm{succ}\ (\mathrm{succ}\ (\mathrm{twice}\ n)) \end{aligned}$$

3

Paradigma imperativo

3.1 Programmi

3.1.1 Memoria

Definizione 3.1.1.1: Memoria

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

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

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

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

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

Definizione 3.1.1.2: Programma

Nel paradigma imperativo, un **programma** è un componente semantico che non restituisce valori — dunque non ha *side effect* — ma cambia lo stato memoria.

3.1.2 Clausole imperative

Definizione 3.1.2.1: Clausola *skip*

La clausola *skip* verrà utilizata attraverso la sintassi

$$skip$$

ed equivale a non effettuare alcuna operazione.

Definizione 3.1.2.2: Clausola *seq*

La clausola *seq* 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*

La clausola *ite* 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*

La clasuola *while* verrà utilizzata attraverso la sintassi

$$while\ \texttt{expression}\ do\ \texttt{program}$$

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*

La clausola *assign* 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:

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

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

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

L'insieme degli ambienti di *Imp* è strutturato come nella Definizione 3.1.1.1. Inoltre, per effettuare le valutazioni, vengono definite le seguenti semantiche operazionali:

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

ed i loro giudizi operazionali verranno indicati come segue:

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

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 $\overset{M}{\leadsto}$ delle espressioni sia in grado di restituire valori, ma non di cambiare la memoria, mentre la semantica $\overset{p}{\leadsto}$ 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**:
$$\exists v \in \text{Val} \mid S(E(x)) = v \implies [vars]\ E \vdash x,\ S \rightsquigarrow v$$

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

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

- **minorazioni**:
$$v_1 < v_2 \implies [lt_1]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M\ <\ N,\ S \rightsquigarrow true}$$
$$v_1 \geq v_2 \implies [lt_2]\ \frac{E \vdash M,\ S \rightsquigarrow v_1 \quad E \vdash N,\ S \rightsquigarrow v_2}{E \vdash M\ <\ N,\ S \rightsquigarrow false}$$

- **skip**:
$$[skip]\ E \vdash skip,\ S \rightsquigarrow S$$

- **composizioni sequenziali**:
$$[seq]\ \frac{E \vdash p,\ S \rightsquigarrow S' \quad E \vdash q,\ S' \rightsquigarrow S''}{E \vdash p; q,\ S \rightsquigarrow S''}$$

- **if-then-else**:
$$[ite_1]\ \frac{E \vdash M,\ S \rightsquigarrow true \quad E \vdash p,\ S \rightsquigarrow S'}{E \vdash if\ M\ then\ p\ else\ q,\ S \rightsquigarrow S'}$$
$$[ite_2]\ \frac{E \vdash M,\ S \rightsquigarrow false \quad E \vdash p,\ S \rightsquigarrow S''}{E \vdash if\ M\ then\ p\ else\ q,\ S \rightsquigarrow S''}$$

- **while**:

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

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

- **dichiarazioni**:

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

- **assegnazioni**:

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

3.3 Memoria contigua

3.3.1 Definizioni

Definizione 3.3.1.1: Memoria contigua

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*

La clausola *arr* 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*

la clausola *proc* verrà utilizzata attraverso la sintassi

$$\mathit{proc}\ \texttt{variable}_1(\texttt{variable}_2)\ \mathit{is}\ \texttt{program}_1\ \mathit{in}\ \texttt{program}_2$$

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*

La clausola *call* verrà utilizzata attraverso la sintassi

$$\mathit{call}\ \texttt{variable}(\texttt{expression})$$

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:

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

Dunque, essa è composta da:

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

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

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

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

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

- **locazioni in array**:
$$\exists (l_0, \dots, l_n) \in \text{Loc}^+ \mid E(x) = \langle l_0, \dots, l_n \rangle \implies$$
$$\implies \forall m \in [0, n] \quad [loc_2]\ \frac{E \vdash M,\ S \overset{M}{\rightsquigarrow} m}{E \vdash x[M],\ S \overset{V}{\rightsquigarrow} l_m}$$

- **reference**:
$$\exists v \in \text{Val} \mid S(l) = v \implies [ref]\ \frac{E \vdash V,\ S \overset{V}{\rightsquigarrow} l}{E \vdash V,\ S \overset{M}{\rightsquigarrow} v}$$

- **assegnazioni**:
$$[assign]\ \frac{E \vdash M,\ S \overset{M}{\rightsquigarrow} v \quad E \vdash V,\ S \overset{V}{\rightsquigarrow} l}{E \vdash V := M,\ S \overset{p}{\rightsquigarrow} S\{(l, v)\}}$$

- **array**:
$$\exists (l_0, \dots, l_n) \in \text{Loc}^+ \mid l_0, \dots, l_n \notin \text{dom}(S) \implies$$
$$\implies [arr]\ \frac{E \vdash M_0,\ S \overset{M}{\rightsquigarrow} v_0 \quad \dots \quad E \vdash M_n,\ S \overset{M}{\rightsquigarrow} v_n \quad E\{(x, (l_0, \dots, l_n))\} \vdash p,\ S\{(l_0, v_0), \dots, (l_n, v_n)\} \overset{p}{\rightsquigarrow} S'}{E \vdash arr\ x = [M_0, \dots, M_n]\ in\ p,\ S \overset{p}{\rightsquigarrow} S'}$$

- **procedure**:
$$[proc]\ \frac{E\{(y, (x, p, E))\} \vdash q,\ S \overset{p}{\rightsquigarrow} S'}{E \vdash proc\ y(x)\ is\ p\ in\ q,\ S \overset{p}{\rightsquigarrow} S'}$$

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**:

$$\exists l \in \mathrm{Loc} - \mathrm{dom}(S) \wedge E(y) = (x, p, E') \implies$$

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

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**:

$$\exists l \in \mathrm{Loc} - \mathrm{dom}(S) \wedge E(y) = (x, p, E') \implies$$

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

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**:

$$E(x) = (V, E') \implies [loc_3]\ \frac{E' \vdash V,\ S \overset{V}{\leadsto} l}{E \vdash x,\ S \overset{V}{\leadsto} l}$$

- **call-by-name**:

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

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 Formule imperative

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 imperativa

Si definisce **formula imperativa** un'espressione appartenente alla seguente grammatica:

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

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

Proposizione 4.1.1.1: Inferenza delle formule imperative

Si considerino le formule imperative 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**:
$$\frac{\{A\}\ p\ \{B_0\} \quad \{A\}\ p\ \{B_1\} \quad \ldots \quad \{A\}\ p\ \{B_n\}}{\{A\}\ p\ \{B_0 \land B_1 \land \ldots \land B_n\}}$$
poiché, se partendo da uno stato che soddisfa A, una volta terminata l'esecuzione, p raggiunge uno stato che soddisfa B_0 e $B_1, \ldots, B_n$, allora soddisferà sicuramente anche $B_0 \land B_1 \land \ldots \land B_n$;

- **or**:
$$\frac{\{B_0\}\ p\ \{A\} \quad \{B_1\}\ p\ \{A\} \quad \ldots \quad \{B_n\}\ p\ \{A\}}{\{B_0 \lor B_1 \lor \ldots \lor B_n\}\ p\ \{A\}}$$
poiché, se partendo da un qualsiasi stato tra B_0 e $B_1, \ldots, B_n$, una volta terminata l'esecuzione, p raggiunge uno stato che soddisfa A, allora lo stato iniziale soddisferà anche $B_0 \lor B_1 \lor \ldots \lor B_n$; si noti che le regole *and* ed *or* sono l'una il duale dell'altra;

4.1.2 Logica di Hoare

Definizione 4.1.2.1: Logica di Hoare

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

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

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

Proposizione 4.1.2.1: 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 4.1.2.2: 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 4.1.2.1 (Correttezza di programmi)**.** Si consideri il seguente programma — espresso in termini della logica di Hoare, definita nella Definizione 4.1.2.1 — 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:

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

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

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

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 Formule funzionali

Definizione 4.2.1.1: Formula funzionale

Si definisce **formula funzionale** un'espressione appartenente alla seguente grammatica:

$$\varphi \ ::= \ \forall x.\varphi \ \mid \ M = N$$

dunque l'unico predicato presente è l'uguaglianza, denotato con il simbolo $=$.

Proposizione 4.2.1.1: Inferenza delle formule funzionali

Si considerino le formule funzionali descritte all'interno della Definizione 4.2.1.1; allora, si definiscono le seguenti regole di inferenza, espresse attraverso la sintassi della grammatica Fun_ρ definita nella Definizione 2.5.1.3:

- **regola** α:
$$fn\ x \Rightarrow M = fn\ y \Rightarrow M[y/x]$$
si noti che questa regola coincide con l'alfa equivalenza descritta nella Definizione 2.4.3.4 del lambda calcolo;
- **regola** β:
$$(fn\ x \Rightarrow M)\ N = M[N/x]$$
si noti che questa regola coincide con la beta conversione, descritta nella Definizione 2.4.3.5, del lambda calcolo;
- **regola del caso base**:
$$rec\ M\ N\ 0 = M$$
(in accordo con l'Osservazione 2.5.1.3);
- **regola del passo ricorsivo**:
$$rec\ M\ N\ (\text{succ}\ \ L) = N\ (rec\ M\ N\ L)\ L$$
(in accordo con l'Osservazione 2.5.1.3);
- **regola dell'induzione**:
$$\frac{P(0) \quad P(n) \implies P(\text{succ}\ \ n)}{\forall n \quad P(n)}$$
- **regola dell'uguaglianza**:
$$\frac{M = N \quad M = L}{N = L}$$

- **regola del contesto**:

$$\frac{M = N \quad M' = N'}{MN = M'N'}$$

- **regola ξ**:

$$\frac{M = N}{fn\ x \Rightarrow M = fn\ x \Rightarrow N}$$

Proposizione 4.2.1.2: Proprietà del predicato di uguaglianza

TODO

Dimostrazione. TODO □

5

Elementi di Teoria dei Tipi

5.1 Lambda calcolo tipato semplice

5.1.1 Definizioni

Definizione 5.1.1.1: Lambda calcolo tipato semplice

La grammatica del **lambda calcolo tipato semplice** (noto in letteratura come *System F1*) è costituita dalle seguenti grammatiche:

$$
\begin{aligned}
A, B &::= K \mid A \to B \\
M, N &::= k \mid x \mid \lambda x : A.M \mid MN
\end{aligned}
$$

Dunque, essa è composta da da:

- una grammatica per i *tipi*, che verrà indicata con $Types$; si noti che il simbolismo $A \to B$ indica il tipo di una funzione che prende in ingresso un termine di tipo A, e ne restituisce uno di tipo B;
- una grammatica per i *termini*, che verrà indicata con $Terms$; si noti che il simbolismo $\lambda x : A.M$ indica una funzione che prende in ingresso un termine di tipo A, ed ha l'espressione M come suo corpo.

Il *System F1* è **monomorfo**, poiché un termine ha *uno ed un solo tipo*.

Definizione 5.1.1.2: Contesto dei tipi di una grammatica

Data una grammatica tipata G, le cui grammatiche delle espressioni e dei tipi sono indicate rispettivamente con $Terms$ e $Types$, un **contesto dei tipi** della grammatica è una funzione della forma

$$\Gamma : \mathrm{Var} \overset{fin}{\to} Types$$

che associa dunque una variabile ad un possibile tipo. L'insieme di tutti i contesti della grammatica è denotato con

$$\mathrm{Ctx} := \{f \mid \mathrm{Var} \overset{fin}{\to} Types\}$$

In simboli, i contesti verranno scritti come liste di coppie $M : A$ con $M \in Terms, A \in Types$, che descriveranno la mappa definita dal contesto stesso. Si noti che, per un certo contesto $\Gamma \in \mathrm{Ctx}$, $\Gamma(x)$ è indefinito per ogni $A \in Types - \mathrm{dom}(\Gamma)$.

Esempio 5.1.1.1 (TODO)**.** TODO

Definizione 5.1.1.3: Concatenazione di contesti

Siano Γ_1 e Γ_2 due contesti di una grammatica tipata; allora, si definisce **concatenazione** di Γ_1 e Γ_2 la seguente funzione

$$\Gamma_1, \Gamma_2 : \mathrm{Ctx} \times \mathrm{Ctx} \to \mathrm{Ctx} : x \mapsto \begin{cases} \Gamma_2(x) & x \in \mathrm{dom}(\Gamma_2) \lor x \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) \\ \Gamma_1(x) & x \in \mathrm{dom}(\Gamma_1) \end{cases}$$

dunque, nella concatenazione Γ_2 sovrascrive le tuple che sono presenti anche in Γ_1.

Esempio 5.1.1.2 (Concatenazione di contesti)**.** Si considerino i due contesti seguenti, definiti all'interno del lambda calcolo tipato

$$\Gamma_1 := M : A, N : B$$
$$\Gamma_2 := M : C$$

allora, si ha che

$$\Gamma_1, \Gamma_2 = M : C, N : B$$

Definizione 5.1.1.4: Giudizi di tipi

Data una grammatica tipata G, si definisce **semantica di tipi** della grammatica una relazione, indicata col simbolo :, definita come segue:

$$: \subseteq \mathrm{Ctx} \times \mathit{Terms} \times \mathit{Types}$$

dove Terms e Types indicano rispettivamente le grammatiche delle espressioni e dei tipi di G. Un elemento $(\Gamma, M, A) \in\ :$ è detto **giudizio di tipo**, e viene scritto attraverso il seguente simbolismo:

$$\Gamma \vdash M : A$$

per qualche espressione M e tipo A, e si legge "M è un termine legale di tipo A nel contesto dei tipi Γ".

Proposizione 5.1.1.1: Regole di inferenza di *System F1*

Dato un contesto di tipi $\Gamma \in \mathrm{Ctx}$, le regole di inferenza del lambda calcolo tipato sono le seguenti:

- **costanti**:
$$[\mathit{const}]\ \Gamma \vdash k : K$$
dove $K := \{\texttt{int}, \texttt{string}, \texttt{bool}\}$
- **variabili**:
$$\forall x \in \mathrm{Var} \quad \exists v \in \mathit{Types} \mid \Gamma(x) = A \implies [\mathit{vars}]\ \Gamma \vdash x : A$$
- **applicazioni**:
$$\forall M, N \in \mathit{Terms}, A, B \in \mathit{Types} \quad [\mathit{appl}]\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$
- **funzioni**:
$$\forall x \in \mathrm{Var}, M \in \mathit{Terms}, A, B \in \mathit{Types} \quad [\mathit{fn}]\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

Esempio 5.1.1.3 (Derivazione di tipi)**.** Si consideri la seguente espressione, descritta attraverso le espressioni del lambda calcolo tipato:

$$(\lambda x : \texttt{int} \to \texttt{bool}.x((\lambda y : \texttt{string}.7)\texttt{"ciao"}))(\lambda z : \texttt{int}.\mathit{true})$$

dunque, per derivare il suo tipo, è necessario sviluppare il seguente albero di derivazione:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \texttt{int} : \texttt{int} \to \texttt{bool} \quad \dfrac{\dfrac{\Gamma, y : \texttt{string} \vdash 7 : \texttt{int}}{\Gamma \vdash \lambda y : \texttt{string}.7 : \texttt{string} \to \texttt{int}} \quad \Gamma \vdash \texttt{"ciao"} : \texttt{string}}{\Gamma \vdash (\lambda y : \texttt{string}.7)\texttt{"ciao"} : \texttt{int}}}{x : \texttt{int} \to \texttt{bool} \vdash x((\lambda y : \texttt{string}.7)\texttt{"ciao"}) : \texttt{bool}}}{\varnothing \vdash \lambda x : \texttt{int} \to \texttt{bool}.x((\lambda y : \texttt{string}.7)\texttt{"ciao"} : (\texttt{int} \to \texttt{bool}) \to \texttt{bool}} \quad \dfrac{z : \texttt{int} \vdash \mathit{true} : \texttt{bool}}{\varnothing \vdash \lambda z : \texttt{int}.\mathit{true} : \texttt{int} \to \texttt{bool}}}{\varnothing \vdash (\lambda x : \texttt{int} \to \texttt{bool}.x((\lambda y : \texttt{string}.7)\texttt{"ciao"}))(\lambda z : \texttt{int}.\mathit{true}) : \texttt{bool}}$$

dove
$$\Gamma := x : \texttt{int} \to \texttt{bool}$$

Lemma 5.1.1.1: Espressioni non tipabili

Non tutte le espressioni del lambda calcolo tipato semplice sono tipabili.

Dimostrazione. Si consideri la seguente espressione:
$$\lambda x.xx$$
espressa in termini del lambda calcolo non tipato; provando a tipare tale espressione, si ottiene il seguente albero di derivazione:
$$\cfrac{\cfrac{x : A \vdash x : A \to B \quad x : A \vdash x : B}{x : A \vdash xx : B}}{\varnothing \vdash \lambda x : A.xx : A \to B}$$
per certi tipi A e B, ma poiché in tale albero compare il giudizio
$$x : A \vdash x : A \to B$$
segue necessariamente che $A \to B$ deve coincidere con A, ma questo non è possibile poiché $A \to B$ avrà sempre una $\to$ in più di A, indipendentemente dal tipo di quest'ultimo; dunque, segue la tesi. $\square$

Osservazione 5.1.1.1: Tipi infiniti

Si consideri la dimostrazione del Lemma 5.1.1.1; essa verte sul numero di $\to$ presenti all'interno del tipo considerato, dunque un tipo *infinito* come ad esempio
$$C \to C \to C \to C \to C \to \cdots$$
potrebbe rendere falso l'argomento utilizzato all'interno della dimostrazione, poiché entrambe i tipi considerati avrebbero un numero di $\to$ infinito. In realtà, questo non costituisce un controesempio poiché i tipi infiniti non sono presenti nell'algebra induttiva dei tipi del lambda calcolo tipato semplice, definita come
$$(Types, K, \to)$$
Infatti, siano per assurdo inclusi i tipi infiniti all'interno di tale algebra induttiva; allora esisterebbe un sottoinsieme dell'algebra considerata, ovvero $Types$ stesso, tale da costituire un'algebra induttiva, e dunque verrebbe violato l'assioma *iii* della Definizione 1.1.2.5.

Proposizione 5.1.1.2: Ricorsione in *System F1*

Ogni termine di *System F1* termina.

Dimostrazione. Omessa. $\square$

5.2 Lambda calcolo polimorfo

5.2.1 Polimorfismo

Definizione 5.2.1.1: Polimorfismo

Con **polimorfismo** si definisce la possibilità, per un'espressione, di assumere molteplici tipi, in base al contesto considerato. Le grammatiche polimorfe sono caratterizzate da **variabili di tipo**, che permettono di descrivere in maniera *generica* le loro espressioni.

Definizione 5.2.1.2: Clausola dei tipi polimorfi

Sia G una grammatica tipata polimorfa, dove $Types$ e $Terms$ costituiscono rispettivamente le grammatiche dei tipi e dei termini di G; allora, è possibile definire su G una funzione *poly*, come segue:

$$poly : Types \times Terms \to Terms$$

e verrà utilizzata attraverso la sintassi

$$\forall \texttt{type}_1.\texttt{type}_2$$

la quale asserisce che, per ogni tipo che la varibile di tipo $\texttt{type}_1$ può assumere, il tipo polimorfo definito è $\texttt{type}_2$. Si noti che è possibile contrarre l'espressione

$$\forall \texttt{type}_1.(\forall \texttt{type}_2.\ldots.(\forall \texttt{type}_{n-1}.\texttt{type}_n))$$

con l'espressione

$$\forall \texttt{type}_1, \texttt{type}_2, \ldots, \texttt{type}_{n-1}.\texttt{type}_n$$

Esempio 5.2.1.1 (Tipi polimorfi)**.** Si consideri ad esempio il tipo

$$\forall X.X \to X$$

essa, attraverso la variabile di tipo X, definisce il tipo polimorfo $X \to X$ (che rappresenta una funzione che prende in ingresso un'espressione di tipo X e restituisce a sua volta un'espressione di tipo X).

Definizione 5.2.1.3: Clausola di astrazione dei tipi

Sia G una grammatica tipata polimorfa, dove $Types$ e $Terms$ costituiscono rispettivamente le grammatiche dei tipi e dei termini di G; allora, è possibile definire su G una funzione $abstr$, come segue:

$$abstr : Types \times Terms \to Terms$$

e verrà utilizzata attraverso la sintassi

$$\Lambda\texttt{type.expression}$$

la quale dichiara la variabile di tipo `type` all'interno dell'espressione `expression`.

Definizione 5.2.1.4: Clausola di istanziazione dei tipi

Sia G una grammatica tipata polimorfa, dove $Types$ e $Terms$ costituiscono rispettivamente le grammatiche dei tipi e dei termini di G; allora, è possibile definire su G una funzione $inst$, come segue:

$$inst : Types \times Terms \to Terms$$

e verrà utilizzata attraverso la sintassi

`expression type`

la quale istanzia il tipo `type` all'interno dell'espressione `expression`.

Osservazione 5.2.1.1: Istanziazione dei tipi

Si noti che, per utilizzare un'espressione polimorfa, è necessario prima istanziarne (o *specializzarne*) il tipo.

5.2.2 Lambda calcolo polimorfo

Definizione 5.2.2.1: Lambda calcolo polimorfo

La grmamatica del **lambda calcolo polimorfo** (noto in letteratura come *System F*) è costituita dalle seguenti grammatiche

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

Dunque, essa è composta da

- una grammatica per i *tipi*, che verrà indicata con $Types$; si noti che il simbolismo X indica che la grammatica include le variabili di tipo, il cui insieme verrà indicato con $\mathrm{TypeVar}$;
- una grammatica per i *termini*, che verrà indicata con $Terms$.

Proposizione 5.2.2.1: Regole di inferenza di *System F*

Dato un contesto di tipi $\Gamma \in \mathrm{Ctx}$, le regole di inferenza del lambda calcolo polimorfo estendono quelle definite all'interno della Proposizione 5.1.1.1, attraverso le seguenti:

- **generalizzazioni**:

$$\forall M \in Terms, A \in Types \quad \exists X \in \mathrm{TypeVar} \mid X \notin \mathrm{free}(\Gamma) \implies [gen]\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda X.M : \forall X.A}$$

- **specializzazioni**:

$$\forall M \in Terms, A, B \in Types, X \in \mathrm{TypeVar} \quad [spec]\ \frac{\Gamma \vdash M : \forall X.A}{\Gamma \vdash MB : A[B/X]}$$

Esempio 5.2.2.1 (Tipo dell'identità polimorfa)**.** Si consideri la seguente espressione, che rappresenta l'identità polimorfa:

$$\Lambda X.\lambda x : X.x$$

è possibile dedurne il tipo attraverso il seguente albero di derivazione

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

ed istanziando ad esempio il tipo `int` su di essa, si ottiene il seguente albero:

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

Osservazione 5.2.2.1: Cattura delle variabili di tipo

Si consideri il seguente albero di derivazione:

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

e posto

$$M := \Lambda X.\lambda x : X.(\Lambda X.x)\ \texttt{int}$$

si può ad esempio derivare che

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

il che è necessariamente falso, poiché M rappresenta una lambda astrazione non applicata, e dunque non può in alcun modo avere come tipo `int`. Questo assurdo è stato raggiunto poiché è stata violata la condizione della regola della generalizzazione, descritta all'interno della Proposizione 5.2.2.1: nel primo albero di derivazione, nel passaggio

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

la variabile di tipo X era in free(Γ), ed il tipo $\forall X.X$ l'ha catturata, ma la X presente in $x : X$, e quella presente in $\forall X.X$ sono X *diverse*: infatti, è solo un caso che entrambe le variabili di tipo si chiamino X TODO DA CAPIRE CHE STO A SCRIVE PERCHÉ NON MI È CHIARO NIENTE

Osservazione 5.2.2.2: Espressioni polimorficamente tipabili

Si consideri l'espressione

$$\lambda x.xx$$

discussa all'interno della dimostrazione del Lemma 5.1.1.1; diversamente dal *System F1*, all'interno del *System F*, poiché polimorfo, è possibile tipare tale espressione, ad esempio come segue:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \forall X.X \to \texttt{int}}{\Gamma \vdash x\ (\texttt{int} \to \texttt{int}) : (\texttt{int} \to \texttt{int}) \to \texttt{int}} \quad \dfrac{\Gamma \vdash x : \forall X.X \to \texttt{int}}{\Gamma \vdash x\ \texttt{int} : \texttt{int} \to \texttt{int}}}{x : \forall X.X \to \texttt{int} \vdash x\ (\texttt{int} \to \texttt{int})\ (x\ \texttt{int}) : \texttt{int}}}{\varnothing \vdash \lambda x : \forall X.X \to \texttt{int}.x\ (\texttt{int} \to \texttt{int})\ (x\ \texttt{int}) : (\forall X.X \to \texttt{int}) \to \texttt{int}}$$

dove

$$\Gamma := x : \forall X.X \to \texttt{int}$$

Ciononostante, l'espressione ω della grammatica Fun, che è stata presentata all'interno dell'Esempio 2.3.2.1, ovvero

$$(\lambda x.xx)(\lambda x.xx)$$

resta non tipabile nel *System F* (di cui se ne omette la dimostrazione).

Teorema 5.2.2.1: Sistema di tipi del *System F*

Il sistema di tipi del *System F* è indecidibile.

Dimostrazione. Omessa. □

Proposizione 5.2.2.2: Ricorsione in *System F*

Ogni termine di *System F* termina.

Dimostrazione. Omessa. □

5.3 Fun_τ: un linguaggio tipato polimorfo

5.3.1 Definizioni

Definizione 5.3.1.1: Istanza generica

Siano σ e σ' i seguenti tipi:

$$\sigma := \forall X_1, \dots, X_n.\tau$$
$$\sigma' := \forall Y_1, \dots, Y_m.\tau'$$

Allora, σ' si dice essere un'**istanza generica** di σ — e viene denotato con il simbolismo

$$\sigma > \sigma'$$

se e solo se:

- $\exists \tau_1, \dots, \tau_n \mid \tau' = [\tau_1, \dots, \tau_n / X_1, \dots, X_n]$ (dove quest'ultima indica una sostituzione simultanea)
- $\nexists j \in [1, m] \mid Y_j \in \text{free}(\sigma)$

Esempio 5.3.1.1 (Istanze generiche)**.** Dato il tipo

$$\sigma := \forall X_1, X_2.X_1 \to X_2$$

si verifica che il tipo σ', definito come

$$\sigma' := \forall Z.(\texttt{int} \to Z) \to \texttt{bool}$$

è un'istanza generica di σ, poiché

$$\begin{cases} \tau_1 := \texttt{int} \to Z \\ \tau_2 := \texttt{bool} \end{cases} \implies \exists \tau_1, \tau_2 \mid \tau' := (\texttt{int} \to Z) \to \texttt{bool} = (X_1 \to X_2)[\tau_1, \tau_2 / X_1, X_2]$$

ed inoltre $Z \notin \text{free}(\sigma)$.

Esempio 5.3.1.2 (Istanze generiche)**.** Dato il tipo

$$\sigma := \forall X, Y.X \to Y$$

si verifica che il tipo σ', definito come

$$\sigma' := X \to Y$$

è un'istanza generica di σ, poichè σ' è semplicemente diventato non ulteriormente specializzabile.

Esempio 5.3.1.3 (Istanze generiche)**.** Dato il tipo

$$\sigma := X$$

si verifica che il tipo σ', definito come

$$\sigma' := \forall Y.X$$

è un'istanza generica di σ, poichè X è stato sostituito con sé stesso, e $Y \notin \text{free}(\sigma)$.

Esempio 5.3.1.4 (Istanze generiche)**.** Dato il tipo

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

si verifica che il tipo σ', definito come

$$\sigma' := \forall Y.X$$

è un'istanza generica di σ, poichè X è stato sostituito con sé stesso, e $Y \notin \text{free}(\sigma)$.

Non esempio 5.3.1.1 (Istanze generiche)**.** Dato il tipo

$$\sigma := \forall X.Y \to X$$

si verifica che il tipo σ', definito come

$$\sigma' := \forall Y.(\texttt{int} \to Y) \to X$$

non è un'istanza generica di σ, poiché $Y \in \text{free}(\sigma)$.

Non esempio 5.3.1.2 (Istanze generiche)**.** Dato il tipo

$$\sigma := X \to Y$$

si verifica che il tipo σ', definito come

$$\sigma' := \texttt{int} \to Y$$

non è un'istanza generica di σ, poiché σ non è specializzabile.

Non esempio 5.3.1.3 (Istanze generiche)**.** Dato il tipo

$$\sigma := X$$

si verifica che il tipo σ', definito come

$$\sigma' := \forall X.X$$

non è un'istanza generica di σ, poiché $X \in \text{free}(\sigma)$.

Osservazione 5.3.1.1: Non-antisimmetria delle istanze

Si considerino i due tipi seguenti:

$$\sigma := \forall Z.X$$
$$\sigma' := \forall W.X$$

e si noti che

$$\begin{cases} \tau_1 := X \implies \exists \tau_1 \mid \tau' := X = (X)[\tau_1/Z] \\ W \notin \text{free}(\sigma) \end{cases} \iff \sigma > \sigma'$$

ed inoltre

$$\begin{cases} \tau_1 := X \implies \exists \tau_1 \mid \tau' := X = (X)[\tau_1/W] \\ Z \notin \text{free}(\sigma') \end{cases} \iff \sigma' > \sigma$$

Allora esistono tipi tali per cui

$$\begin{cases} \sigma > \sigma' \\ \sigma' > \sigma \end{cases}$$

nonostante $\sigma \neq \sigma'$. Allora la relazione $>$ non è antisimmetrica.

Definizione 5.3.1.2: Grammatica Fun_τ

Sia Fun_τ la grammatica tipata polimorfa seguente:

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

Dunque, essa è composta da:

- due grammatiche per i *tipi*, che verranno indicate genericamente con $Types$; gli elementi della grammatica prendono il nome di *tipi primitivi* (la cui grammatica verrà indicata con $PrimTypes$), mentre quelli della seconda prendono il nome di *schemi di tipo*;
- una grammatica per i *termini*, che verrà indicata con $Terms$.

Si noti che questa grammatica è la stessa che viene utilizzata all'interno del linguaggio di progarmmazione SML.

Osservazione 5.3.1.2: Tipi di Fun_τ

La definizione dei tipi di Fun_τ è scomposta in due grammatiche differenti, al fine di vietare di costruire tipi come il seguente:

$$\forall X.X \to \texttt{int}$$

infatti, non è possibile avere questo tipo in Fun_τ, poiché una volta costruito il tipo $\forall X.X$, il quale è uno schema di tipo, non è legale utilizzare la regola $\tau_1 \to \tau_2$, poiché τ_1 e τ_2 devono essere tipi primitivi.

Di conseguenza, i tipi di Fun_τ non permettono di definire funzioni che in input si aspettano tipi polimorfi, ed infatti l'espressione

$$\lambda x.xx$$

discussa nell'Osservazione 5.2.2.2 non è tipabile in Fun_τ.

Proposizione 5.3.1.1: Regole di inferenza di Fun_τ

Dato un contesto di tipi $\Gamma \in \text{Ctx}$, le regole di inferenza di Fun_τ estendono le regole

$$[const], \ [vars]$$

definite all'interno della Proposizione 5.1.1.1 attraverso le seguenti:

- **generalizzazioni**:

$$\forall M \in Terms, \sigma \in Types \quad \exists X \in \text{TypeVar} \mid X \notin \text{free}(\Gamma) \implies [gen]\ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall X.\sigma}$$

- **specializzazioni**:

$$\forall M \in Terms, \sigma, \sigma' \in Types \quad \sigma > \sigma' \implies \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma'}$$

- **applicazioni**:

$$\forall M, N \in Terms, \tau, \tau' \in Types \quad [appl]\ \frac{\Gamma \vdash M : \tau' \to \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash MN : \tau}$$

- **funzioni**:

$$\forall M \in Terms, \tau, \tau' \in Types, x \in \text{Var} \quad [fn]\ \frac{\Gamma, x : \tau' \vdash M : \tau}{\Gamma \vdash fn\ x \Rightarrow M : \tau' \to \tau}$$

- **assegnazioni**:

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

Esempio 5.3.1.5 (Derivazione di tipi)**.** Si consideri l'espressione

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

è possibile derivarne il tipo attraverso il seguente albero di derivazione:

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

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

5.3.2 Algoritmo $\mathcal{W}$

Definizione 5.3.2.1: Sostituzione

Si definisce **sostituzione** una morfismo parziale — in cui gli input non presenti nel dominio vengono mandati in loro stessi — sull'operatore $\to$, che mappa variabili di tipo a tipi primitivi; in simboli

$$V : \text{TypeVar} \overset{fin}{\to} PrimTypes$$

Definizione 5.3.2.2: Unificatore

Siano τ_1 e τ_2 due tipi primitivi; allora, una sostituzione V è detta **unificatore di tipi primitivi** se

$$V(\tau_1) = V(\tau_2)$$

Esempio 5.3.2.1 (Unificatori)**.** Siano τ_1 e τ_2 i seguenti tipi primitivi:

$$\tau_1 := A \to B \to C$$
$$\tau_2 := (C \to E) \to D$$

e sia V una sostituzione definita insiemisticamente come

$$V := \{(A, C \to E), (D, B \to C)\}$$

allora, poiché

$$\begin{aligned} V(\tau_1) = V(A \to B \to C) = V(A) \to B \to C = \\ = (C \to E) \to B \to C = \\ = (C \to E) \to V(D) = V((C \to E) \to D) = V(\tau_2) \end{aligned}$$

si ha che V è unificatore di τ_1 e τ_2.

Non esempio 5.3.2.1 (Unificatori)**.** Siano τ_1 e τ_2 i seguenti tipi primitivi:

$$\tau_1 := A \to A$$
$$\tau_2 := \texttt{int} \to \texttt{bool}$$

per assurdo, se esistesse un unificatore V di τ_1 e τ_2, si otterrebbe che $\texttt{int} \equiv \texttt{bool}$ $\lightning$.

Non esempio 5.3.2.2 (Unificatori)**.** Siano τ_1 e τ_2 i seguenti tipi primitivi:

$$\tau_1 := A \to A \to A$$
$$\tau_2 := B \to B$$

per assurdo, se esistesse un unificatore V di τ_1 e τ_2, si otterrebbe che $A \equiv A \to A$ $\lightning$.

Teorema 5.3.2.1: Teorema di unificazione di Robinson

Esiste un algoritmo $\mathcal{U}$ *fallibile* che, dati due tipi primitivi τ_1 e τ_2, se non fallisce, restituisce una sostizuione V tale che

- $V(X) \neq X$ implica che X compare in τ_1 o τ_2
- V è unificatore di τ_1 e τ_2

ed inoltre l'algoritmo è tale che, se esiste un unificatore W di τ_1 e τ_2, allora

- $\mathcal{U}$ non può fallire
- esiste una sostituzione Z tale che $W = Z \circ \mathcal{U}(\tau_1, \tau_2)$, dunque il risultato dell'algoritmo è la sostituzione più generica possibile

Dimostrazione. Omessa. □

Esempio 5.3.2.2 (Genericità delle sostituzioni)**.** Siano τ_1 e τ_2 i seguenti tipi primitivi

$$\tau_1 := X \to Y$$
$$\tau_2 := Z \to Y$$

e siano V e W due sostituzioni definite insiemisticamente come

$$V := \{(X, Z)\}$$
$$W := \{(X, Z), (Y, A \to B)\}$$

si noti che entrambe le sostituzioni sono unificiatori di τ_1 e τ_2, poiché

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

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

ma poiché la sostituzione

$$U := \{(Y, A \to B)\}$$

è tale che

$$W = U \circ V$$

allora V è più generica di W.

Definizione 5.3.2.3: Generalizzazione massima

Data una variabile $x \in \text{Var}$, ed un contesto $\Gamma \in \text{Ctx}$, si definisce **generalizzazione massima di** x il seguente schema di tipo:

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

dove $\Gamma(x) = \tau$ e $\text{TypeVar} - \text{free}(\Gamma) = \{X_1, \dots, X_n\}$.

Definizione 5.3.2.4: Schema principale

TODO

Teorema 5.3.2.2: Algoritmo $\mathcal{W}$

Esiste un algoritmo $\mathcal{W}$ *fallibile* che, dato un contesto $\Gamma \in \text{Ctx}$ ed un termine M, se non fallisce, restituisce una tupla (V, τ) — dove V è una sostituzione e τ è un tipo — tale che:

- se $M \equiv x$, e
$$\Gamma(x) = \forall X_1, \dots, X_n.\tau'$$
allora
$$\begin{cases} V \equiv \text{id} \\ \exists Y_1, \dots, Y_n \in \text{free}(\Gamma(x)) \mid \tau = \tau'[Y_1, \dots, Y_n / X_1, \dots, X_n] \end{cases}$$
- se $M \equiv M_1 M_2$, e
$$\begin{cases} \mathcal{W}(\Gamma, M_1) = (V_1, \tau_1) \\ \mathcal{W}(V_1(\Gamma), M_2) = (V_2, \tau_2) \\ \mathcal{U}(V_2(\tau_1), \tau_2 \to Y) = W \end{cases}$$
dove Y è una nuova variabile — allora
$$\begin{cases} V = W \circ V_2 \circ V_1 \\ \tau = W(Y) \end{cases}$$
- se $M \equiv fn\ x \Rightarrow N$, e
$$\mathcal{W}((\Gamma, x : X), N) = (V_1, \tau_1)$$
dove X è una nuova variabile, allora
$$\begin{cases} V = V_1 \\ \tau = V_1(X) \to \tau_1 \end{cases}$$
- se $M \equiv let\ x = N\ in\ L$, e
$$\mathcal{W}((V_1(\Gamma), x : \tau'), L) = (V_2, \tau_2)$$
dove $\overline{V_1(\Gamma)}(x) = \tau'$, allora
$$\begin{cases} V = V_2 \circ V_1 \\ \tau = \tau_2 \end{cases}$$

Inoltre, se $\mathcal{U}$ fallisce allora $\mathcal{W}$ fallisce.

Dimostrazione. Omessa. □