Università Degli Studi Di Catania

Corso Di Laurea in Informatica

Riassunti di Fondamenti Di Informatica

Autore
Salvo Polizzi

Docenti
Prof. Franco Barbanera
Prof. Maria Serafina Madonia

Indice

Parte I

Linguaggi Formali

Capitolo 1

Elementi di Teoria dei linguaggi formali

I **linguaggi formali** sono una *versione semplice* dei linguaggi naturali. Si tratta infatti di un **insieme di parole** che può avere diverse applicazioni:

- Formalizzare le **dimostrazioni matematiche**
- Applicazione nella creazione di **compilatori**
- Applicazioni **linguistiche**
- Applicazioni nella **biologia**

Nel corso di questi riassunti capiremo come si sviluppano e come anche per gli informatici siano degli strumenti molto utili. Le informazioni presenti in questa parte sono tratte dal testo *linguaggi e complessità* [1]

1.1 Alfabeto, Stringa, Linguaggio

Iniziamo la nostra spiegazione sui linguaggi formali partendo della definizione di **alfabeto**, per poi spiegare sequenzialmente i concetti di **stringa** e **linguaggio**.

1.1.1 Alfabeto

Definizione 1.1.1. *Un **alfabeto** è un **insieme finito di simboli***

I simboli in questione possono essere *cifre*, *caratteri*, ecc. Inoltre l'alfabeto si indica con la lettera greca Σ. Un esempio di alfabeto può essere l'insieme di simboli dalla 'a' alla 'z':

$$\Sigma = \{a, b, c, ..., z\} \tag{1.1}$$

oppure possiamo rappresentare l'alfabeto binario:

$$\Sigma = \{0, 1\} \tag{1.2}$$

1.1.2 Stringhe o parole

Una *successione di simboli* appartenenti a un alfabeto Σ viene invece detta **stringa** o **parola** su Σ. Ovviamente se un alfabeto ha cardinalità $|\Sigma| = k$, allora *tutte le possibili stringhe su un alfabeto* Σ sono **infinite**, ovvero il loro numero è infinito, mentre avranno una **lunghezza finita** n. Quindi possiamo dire che sono la sommatoria:

$$\sum_{n=0}^{\infty} k^n \tag{1.3}$$

Inoltre può esistere la **stringa vuota**, indicata con la lettera greca ε, che non contiene **nessun simbolo**. Da tali concetti possiamo arrivare alla definizione formale [1] di stringa e, in particolare, dell'**insieme delle stringhe** su un alfabeto Σ:

Definizione 1.1.2. *Dato un alfabeto Σ, l'**insieme delle stringhe** su Σ si indica con Σ^* ed è così gestito:*

- $\varepsilon \in \Sigma^*$
- $\forall a \in \Sigma, x \in \Sigma^*$, *allora* $xa \in \Sigma^*$

Supponiamo per esempio di avere l'alfabeto $\Sigma = \{0, 1, 2\}$. **Dimostriamo** che 01 è una **stringa** su Σ:

- Rifacendoci alla definizione 1.1.2, la *stringa vuota* $\varepsilon \in \Sigma^*$ è una stringa poichè appartiene all'insieme delle stringhe per definizione;
- Abbiamo, sempre per la definizione 1.1.2, che per ogni elemento appartenente all'alfabeto e per ogni elemento appartenente all'insieme delle stringhe, la loro *concatenazione* è una stringa. Di conseguenza:
  - Prendiamo $0 \in \Sigma$ e $\varepsilon \in \Sigma^*$, avremo che $\epsilon 0 \in \Sigma^*$
  - Poichè ε rappresenta la stringa vuota, non conterrà alcun elemento, e, di conseguenza, concatenando ε a qualsiasi stringa otterrò la stringa stessa. Possiamo quindi dire che $0 \in \Sigma^*$
  - Iteriamo il processo precedente prendendo $0 \in \Sigma^*$ e $1 \in \Sigma$. Allora per la definizione di stringa la concatenazione tra l'elemento $a = 1 \in \Sigma$ e la stringa $x = 0 \in \Sigma^*$ ci darà una nuova stringa $xa = 01 \in \Sigma^*$. Abbiamo dimostrato dunque che 01 è una stringa sull'alfabeto $\Sigma = \{0, 1, 2\}$

Ma allora ci possiamo chiedere $\Sigma \subset \Sigma^*$? Formalmente voglio dimostrare che $\forall a \in \Sigma \Rightarrow a \in \Sigma^*$:

- Riprendendo sempre la definizione formale, abbiamo $a \in \Sigma$ e $\varepsilon \in \Sigma^*$

[1] Quasi sempre in questo corso vi ritroverete le definizioni formali definite per **induzione**. L'induzione è fondamentale per noi informatici perchè ci permette da un **caso base**, di costruire la nostra definizione nei successivi **passi induttivi**. Per chi quindi pensa di avere lacune sull'induzione consiglio di ripassarla.

- Concatenando a ed ε, otteniamo la stringa $\varepsilon a = a \in \Sigma^*$. La risposta alla domanda è quindi positiva, ovvero $\Sigma \subset \Sigma^*$.

Ovviamente **non vale il viceversa**, ovvero $\Sigma^* \not\subset \Sigma$, poichè i simboli dell'alfabeto possono anche essere considerate stringhe, ma non vale il contrario. Possiamo inoltre dire che se abbiamo l'alfabeto $\Sigma = \{0, 1, 2\}$, allora il *sottoinsieme* $\{0, 1\} \subseteq \Sigma^*$, ma non possiamo dire che appartiene a Σ^*, poichè si tratta appunto di un sottoinsieme, non di un elemento.

Concatenazione fra stringhe

Abbiamo visto in precedenza l'applicazione dell'operazione di **concatenazione di stringhe**. Vediamo adesso la sua definizione formale:

Definizione 1.1.3. *Siano $x, y \in \Sigma^*$. La **concatenazione** fra x e y, si indica con $x \cdot y$ ed è così definita:*

$$x \cdot y = xy \tag{1.4}$$

Inoltre si possono notare delle **proprietà** che si applicano a questa operazione:

- L'operazione è **interna**, ovvero:

$$\forall x, y \in \Sigma^* \Rightarrow x \cdot y \in \Sigma^* \tag{1.5}$$

- Esiste l'**elemento neutro**, ed è ε, che rappresenta la stringa vuota
- Vale la **proprietà associativa**, ovvero:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{1.6}$$

Da tali proprietà si può dedurre che Σ^* è un **monoide libero** su Σ. Ovvero un insieme che gode delle proprietà sopra elencate sull'operazione di concatenazione fra stringhe. Ovviamente, però, **non vale** la **proprietà commutativa**

Potenza di una stringa

Oltre alla concatenazione, un'altra operazione possibile con le stringhe è quella di **potenza**. Data infatti la stringa $x \in \Sigma^*$, la potenza x^n sarà la *concatenazione della stringa con se stessa **n-volte***. Inoltre, in analogia con la potenza di un numero naturale $x^0 = 1$, la potenza $x^0 = \epsilon$. Diamo quindi la definizione formale:

Definizione 1.1.4. *Sia $x \in \Sigma^*$, la **potenza n-esima** di x, con $n \geq 0$, è così definita:*

- $x^0 = \varepsilon$
- $x^{n+1} = x \cdot x^n \quad \forall n \geq 0$

Esempio Facciamo un esempio di come applicare la definizione di potenza di una stringa. Consideriamo la stringa $x = ciao \in \Sigma^*$, vogliamo trovare x^2:

- Dalla definizione l'informazione principale che abbiamo è che $x^0 = \varepsilon$. Procediamo quindi per induzione.
- Per definizione sappiamo che $x^1 = x \cdot x^0 = ciao\varepsilon = ciao$
- Sapendo x^1, possiamo ricavarci x^2, infatti: $x^2 = x \cdot x^1 = ciao \cdot ciao = ciaociao$

Abbiamo quindi utilizzato la definizione per trovarci la stringa finale.

Lunghezza di una stringa

Non abbiamo accennato ancora al concetto di **lunghezza di una stringa**. Se ad esempio prendessimo un stringa abb, allora intuitivamente possiamo dire che la sua lunghezza è $|abb| = 3$. Diamo però una definizione formale:

Definizione 1.1.5. *La **lunghezza di una stringa** $x \in \Sigma^*$ si indica con $|x|$ ed è così definita:*

- $|\varepsilon| = 0$
- $|xa| = |x| + 1$, *con* $x \in \Sigma^*, a \in \Sigma$

Dobbiamo fare molta attenzione a **non confondere** la lunghezza di una stringa con la **cardinalità**, nonostante si utilizzi lo stesso simbolo per indicarle. La cardinalità infatti riguarda solo il numero di elementi di un insieme, cosa che non c'entra assolutamente con la lunghezza di una stringa, che è un elemento dell'insieme Σ^*.

1.1.3 Linguaggio formale

Diamo inizialmente la definizione formale di **linguaggio formale**:

Definizione 1.1.6. *Sia Σ alfabeto, si definisce **linguaggio formale** ogni sottoinsieme di Σ^**

Possiamo quindi dire che l'**alfabeto** è un **linguaggio**, infatti $\Sigma \subset \Sigma^*$ è un sottoinsieme di Σ^*, oppure che $\varnothing$ è un **linguaggio**. Infatti $\varnothing$ è un sottoinsieme di Σ^* con cardinalità 0. Ovviamente anche il sottoinsieme $\{\epsilon\}$ è un **linguaggio**, che non è da confondere con $\varnothing$; infatti il primo è un sottoinsieme con cardinalità 1, mentre il secondo con cardinalità 0 (in questo particolare caso il linguaggio si indica con Λ e si dice **linguaggio vuoto**). Possiamo inoltre ricavare il fatto che i linguaggi formali **possono** essere **infiniti**, proprio perchè essendo Σ^* di cardinalità infinita, può avere infiniti sottoinsiemi.

1.2 Operazioni fra Linguaggi

Dopo aver definito un *linguaggio formale*, cerchiamo di analizzare e capire le **operazioni** definite su esso. Innanzitutto, dalla definizione di linguaggio stesso (definizione 1.1.6), possiamo dire che i linguaggi, essendo insiemi, ammettono tutte quelle **operazioni tra insiemi**, che sicuramente conoscerete, tra le quali:

- **Unione**: dati 2 linguaggi L_1, L_2, l'unione tra 2 linguaggi è definita come $L_1 \cup L_2 = \{x \in \Sigma^* |\ x \in L_1 \vee x \in L_2\}$, ovvero l'elemento appartiene ad almeno uno fra L_1 e L_2. Possiamo dire inoltre che, dato un linguaggio L, $L \cup \Lambda = L$;

- **Intersezione**: dati 2 linguaggi L_1, L_2, l'intersezione tra 2 linguaggi è definita come $L_1 \cap L_2 = \{x \in \Sigma^* |\ x \in L_1 \wedge x \in L_2\}$, ovvero l'elemento appartiene ad sia ad L_1 che L_2. Possiamo dire inoltre che, dato un linguaggio L, $L \cap \Lambda = \Lambda$;

- **Complemento** o **Differenza**: dato il linguaggio L, si dirà complemento il linguaggio $\bar{L}$, definito come $\bar{L} = \{x \in \Sigma^* |\ x \notin L\}$

Poichè già conosciamo abbastanza bene questo tipo di operazioni (in caso contrario converrebbe un ripasso), affronteremo in questa sezione delle **operazioni specifiche** sugli *insiemi di parole*, ovvero sui linguaggi.

1.2.1 Concatenazione

Cominciamo dalla definizione formale di **concatenazione fra linguaggi**[2], in modo tale da applicarla poi nel pratico:

Definizione 1.2.1. *Dati 2 linguaggi L_1, L_2 la **concatenazione** di L_1 ed L_2, denotata con $L_1 \cdot L_2$, è così definita:*

$$L_1 \cdot L_2 = \{x_1 \cdot x_2 |\ \forall x_1 \in L_1, \forall x_2 \in L_2\} \tag{1.7}$$

Ciò che ci vuole dire in questo caso la definizione formale è che la *concatenazione* prende in input 2 linguaggi (operazione binaria) e ci restituisce un **nuovo linguaggio** in cui ogni suo elemento è dato dalla concatenazione di un elemento $x_1 \in L_1$ con un elemento $x_2 \in L_2$. Possiamo inoltre dire che dato un linguaggio L, se lo concateniamo con il linguaggio vuoto Λ otteniamo $L \cdot \Lambda = \Lambda \cdot L = \Lambda$. Infatti se per definizione di concatenazione devo prendere una stringa di L_1 e una di L_2, se in L_2 non c'è nessuna stringa non posso concatenare. Di conseguenza otterrò il *linguaggio vuoto*

[2]Notare che quando parliamo di concatenazione o potenza di linguaggi sono operazioni completamente diverse da quelli che abbiamo visto con le stringhe. Si utilizza la stessa notazione per evitare di usare troppi simboli diversi.

Esempio Prendiamo per esempio il linguaggio $L_1 = \{a, ab\}$ e il linguaggio $L_2 = \{bb, ba\}$, avremo allora che la *concatenazione tra i 2 linguaggi* sarà:

$$L_1 \cdot L_2 = \{abb, aba, abbb, abba\} \tag{1.8}$$

Se inoltre volessimo concatenare inoltre due linguaggi L_1, L_2, sapendo che $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, allora possiamo dire che $L_1 \cdot L_2 \subseteq (\Sigma_1 \cup \Sigma_2)^*$, ovvero $L_1 \cdot L_2$ è un *linguaggio* sull'alfabeto $\Sigma_3 = \Sigma_1 \cup \Sigma_2$. Facciamo un esempio: supponiamo di avere $L_1 = \{a, ab\}$ definito sull'alfabeto $\Sigma_1 = \{a, b\}$ e supponiamo di avere $L_2 = \{00, 10\}$ definito sull'alfabeto $\Sigma_2 = \{0, 1\}$. Avremo che $L_1 \cdot L_2 = \{a00, ab00, a10, ab10\}$ e che $\Sigma_1 \cup \Sigma_2 = \{a, b, 0, 1\}$. Di conseguenza possiamo dire che $L_1 \cdot L_2$ è un linguaggio sull'alfabeto $\Sigma_1 \cup \Sigma_2$.

Sottoinsiemi della concatenazione Adesso poniamoci una domanda: dati 2 linguaggi L_1, L_2, posso affermare in generale che $L_1 \subseteq L_1 \cdot L_2$? In generale posso dire che quest'affermazione è **falsa**, però vi sono 2 casi particolari:

- Nel caso in cui $L_2 = \{\varepsilon\}$, allora avremo che $L_1 = L_1 \cdot L_2$, ovvero i 2 insiemi **coincidono**. Dimostriamolo:
  - La nostra **ipotesi** quindi è che $L_2 = \{\varepsilon\}$. La **tesi** a cui dobbiamo arrivare è che $\forall x \in L_1 \Rightarrow x \in L_1 \cdot L_2$
  - Poichè abbiamo che $\varepsilon \in L_2 \Rightarrow x \cdot \varepsilon \in L_1 \cdot L_2$.
  - Sappiamo però che $x \cdot \varepsilon = x$, quindi abbiamo che $x \in L_1 \cdot L_2 \quad \forall x \in L_1$, come volevasi dimostrare.
- Nel caso in cui L_2 contiene ε, che differisce dall'insieme precedente poichè in questo caso contiene anche altri elementi, allora in questo caso possiamo affermare che $L_1 \subseteq L_1 \cdot L_2$. Questo perchè per quanto dimostrato prima, tutti gli elementi di L_1 concatenati con ε saranno anche elementi di L_2. Di conseguenza L_1 è un *sottoinsieme* di L_2.

1.2.2 Potenza

Passiamo adesso a un altra operazione specifica dei linguaggi, ovvero la **potenza di un linguaggio**. A differenza di quanto visto con la *concatenazione di 2 linguaggi*, che era un *operazione binaria*, la potenza è un **operazione unaria** ovvero prende in input un linguaggio e ne restituisce un altro. L'operazione di concatenazione, invece, che è binaria, prende in input 2 linguaggi e ne restituisce uno nuovo. Iniziamo, quindi, definendo formalmente l'operazione di potenza e vediamo poi anche come applicarla:

Definizione 1.2.2. *Sia $L \subseteq \Sigma^*$. Si definisce **potenza n-esima** di L e si denota con L^n, l'operazione così definita:*

- $L^0 = \{\varepsilon\}$
- $L^{n+1} = L \cdot L^n$

Secondo la convenzione, quindi, possiamo dire che $\Lambda^0 = \{\varepsilon\}$.

Esempio Facciamo un esempio per capire subito come applichiamo questa operazione. Dato $L = \{ab, bbb\}$, troviamo L^2. Potremmo dire intuitivamente che L^2 è costituito da tutte le possibili stringhe ottenute concatenando 2 elementi di L. Come però dobbiamo imparare durante questo corso, cerchiamo di applicare la definizione:

- Partiamo quindi dall'informazione certa, che ci viene data dal *caso base*, ovvero $L^0 = \{\varepsilon\}$;
- Sappiamo, per il passo induttivo, che $L^1 = L \cdot L^0 = L \cdot \{\varepsilon\} = L = \{ab, bbb\}$
- Procedendo sempre induttivamente posso dire che $L^2 = L \cdot L^1 = L \cdot L = \{ab, bbb\} \cdot \{ab, bbb\} = \{abab, abbbb, bbbab, bbbbbb\}$

1.2.3 Chiusura (riflessiva) di Klenee

Vediamo adesso l'operazione detta **chiusura riflessiva di Klenee**. Diamo, quindi, la definizione formale e poi le sue applicazioni:

Definizione 1.2.3. *Sia $L \subseteq \Sigma^*$. La **chiusura riflessiva di Klenee** di L si denota con L^* ed è così definita:*

$$L^* = \bigcup_{n=0}^{\infty} L^n \tag{1.9}$$

Quindi, secondo la definizione, dato un linguaggio L, la sua chiusura di Klenee sarà l'unione di tutte le potenze n-esime dei linguaggi, cioè:

$$L^* = \{L^0 \cup L^1 \cup L^2 \cup L^3 ....\} \tag{1.10}$$

Inoltre possiamo dire che, in generale, L^* è **infinito**, infatti posso concatenare un qualsiasi numero di elementi di L. Se prendiamo ad esempio $L = \{a\}$, $L^* = \{\varepsilon, a, aa, aaa, aaaa...\}$. Gli unici casi in cui L^* è **finito** sono:

- Se $L = \{\varepsilon\}$, allora:

$$L^* = L^0 \cup L^1 \cup L^2 ... = \{\epsilon\} \cup \{\epsilon\} \cup \{\epsilon\} ... = \{\epsilon\} \tag{1.11}$$

- Se $L = \Lambda$, allora:

$$L^* = L^0 \cup L^1 \cup L^2 ... = \{\varepsilon\} \cup \{\Lambda\} \cup \{\Lambda\} ... = \{\varepsilon\} \tag{1.12}$$

Se ci facciamo caso, inoltre, se consideriamo $L = \Sigma$, infatti un alfabeto è un linguaggio, allora Σ^* rappresenta la **chiusura di Klenee sull'alfabeto** Σ, ovvero $L^* = \Sigma^*$. Ecco, quindi, da dove prendiamo la notazione per l'insieme di stringhe.

1.2.4 Chiusura non riflessiva

Un'operazione simile ma non uguale alla chiusura di Klenee, è la **chiusura non riflessiva**. Vediamo, quindi, la definizione formale:

Definizione 1.2.4. *Dato $L \in \Sigma^*$, la **chiusura non riflessiva** si denota con L^+ ed è così definita:*

$$L^+ = \bigcup_{n=1}^{\infty} L^n \tag{1.13}$$

Quindi, se nella chiusura di Klenee $\varepsilon \in L^*$, qui possiamo avere 2 casi:

- Se $\varepsilon \in L$, allora $\varepsilon \in L^+$
- Se $\varepsilon \notin L$, allora $\varepsilon \notin L^+$

Capitolo 2

Riconoscimento di Linguaggi

Nel capitolo precedente abbiamo visto come si compone un *linguaggio* e quali sono le *operazioni* che possiamo effettuare tra linguaggi appunto. Vediamo adesso come **rappresentare** opportunamente **un linguaggio**. Ci sono essenzialmente 2 metodi:

- **Metodi Riconoscitivi**: sono quei metodi che prendono in input una stringa e mi dicono si la stringa appartiene al linguaggio
- **Metodi Generativi**: sono quei metodi attraverso il quale grazie a un insieme finito di regole, se generano le stringhe del linguaggio.

Vedremo nel corso di questo capitolo di approfondire i *metodi riconoscitivi*, e nel prossimo i *metodi generativi*.

2.1 Accettazione e Riconoscimento di Linguaggi

Cominciamo quindi dando una breve introduzione sui *metodi riconoscitivi*, che abbiamo detto che hanno come obiettivo quello di riconoscere un linguaggio. Il concetto fondamentale è che tali metodi si basano su **riconoscitori**, o anche detti **automi**, che sono *modelli teorici* che hanno una struttura particolare. Un riconoscitore è dotato di:

- Uno o più **nastri**, che sono dispositivi di memoria, formati da **celle** che possono contenere simboli appartenenti a un generico alfabeto Σ. Una stringa, quindi, non è altro che la sequenza di simboli contenuti in celle consecutive tale che non ci sia una cella *vuota*, che viene indicata con B o $\not{b}$. Inoltre, il nastro è **infinito da entrambe le direzioni** in modo tale da poter scrivere quante stringhe di lunghezza finita vogliamo.
- I caratteri vengono **letti**/**scritti** tramite **testine** che possono muoversi lungo i nastri e posizionarsi sulle diverse celle, come si può vedere dalla seguente figura:

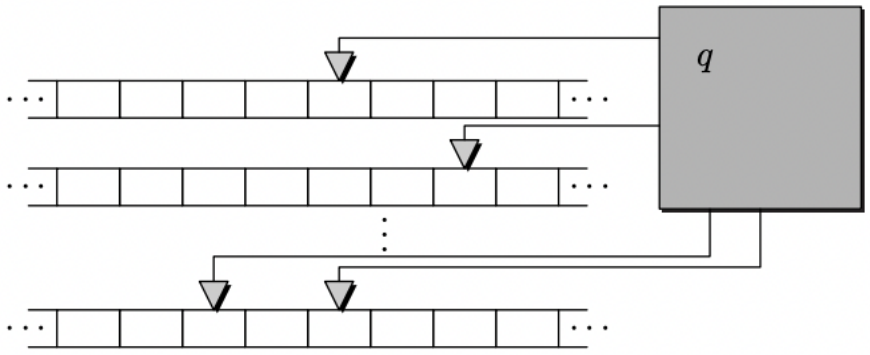


Figure 2.1: Testine che si posizionano in celle diverse

- La struttura dell'automa prevede inoltre un dispositivo, chiamato **controllo**, che ad ogni istante assume uno **stato** che rappresenta l'informazione associata al funzionamento dell'automa in quell'istante. Per capire bene il concetto di stato si può fare l'esempio dell'algoritmo di somma: quando sommiamo in colonna, se dalle unità alle decine non abbiamo nessun riporto lo stato non cambia, altrimenti, con un qualsiasi riporto, lo stato quando dovremo sommare le decine cambierà proprio perchè dovremo considerare il riporto. Lo stato iniziale viene denominato, per un riconoscitore con q_0, e vi è un **insieme di stati finiti** Q che contiene tutti gli stati dell'automa, e un **insieme di stati finali** $F \subseteq Q$ che è l'insieme di stati che ci permette di dire se una stringa appartiene al linguaggio o meno.

Abbiamo accennato sulla struttura di un automa, ma dobbiamo capire su cosa si basa il **funzionamento dell'automa**: questo si basa principalmente sui concetti di **configurazione** e **funzione di transizione**:

- La **configurazione** rappresenta il comportamento che l'automa assume in un determinato momento, ovvero una fotografia della "situazione momentanea" dell'automa. Tale configurazione si costituisce di una **tripla**:

  1. Contenuto dei nastri
  2. Posizione della testina
  3. Stato del controllo

  Quindi capiamo che lo *stato* cambia in base alla configurazione ed è una componente stessa della configurazione. Inoltre, per convenzione, si è deciso che il *primo momento della configurazione dell'automa* è chiamato **configurazione iniziale**. Tra le configurazioni, vi è anche la cosiddetta *configurazione di accettazione*, che è quella configurazione che ci permette di dire se la stringa in input è accettata o meno.

- La **funzione di transizione** è un *programma che ci dice come funziona l'automa* e, in particolare, è una **relazione tra configurazioni** che mi dice come passare da una configurazione alla *configurazione successiva* e, per questo, si dice che tale funzione detta il **movimento**, o **passo computazionale** dell'automa che essenzialmente può fare le seguenti azioni:

  1. Cambiare la testina
  2. Cambiare lo stato
  3. Scrivere un carattere

  Possiamo dunque concludere che la funzione di transizione è una funzione che prende in **input** un simbolo e uno stato, e restituisce in **output** un simbolo, uno stato e un movimento:

$$f : \ (simbolo, stato) \longrightarrow (simbolo, stato, movimento) \tag{2.1}$$

2.2 Automi a Stati Finiti Deterministici

Un esempio di *rinconoscitori* o *automi* sono gli **automi a stati finiti**, detti anche Automi a Stati Finiti Deterministici (ASFD). Si tratta di automi in cui la *testina* può soltanto *leggere*, quindi si dice che la testina è di **sola lettura**, e, inoltre, si può **muovere soltanto verso destra**, ovvero non può tornare su un input già letto, e inoltre hanno un numero finito di stati (stranamente). Vediamo quindi la *definizione formale*, per poi cercare di sviscerarla per bene:

Definizione 2.2.1. *Un Automi a Stati Finiti Deterministici (ASFD) è una quintupla* $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ *dove:*

- Σ *è l'****alfabeto*** *in input (finito)*
- Q *è l'****insieme di stati finiti***
- δ *è la* ***funzione di transizione*** *tale che:*

$$\delta : \ Q \times \Sigma \to Q \tag{2.2}$$

- q_0 *è lo* ***stato iniziale***
- $F \subseteq Q$ *è l'****insieme di stati finali***

Vediamo di analizzare alcuni aspetti importanti di questa definizione:

- Poichè si tratta di un *riconoscitore*, abbiamo ovviamente un *alfabeto in input* e un *insieme di stati finiti* Q, che rappresentano tutti gli stati che può assumere il controllo dell'automa, come avevamo già detto nella sezione 2.1.

- La *funzione di transizione* δ è leggermente diversa dall'esempio generico che avevamo fatto nella sezione 2.1. In questo caso, infatti, poichè la testina è di sola lettura e il movimento avviene solo verso destra, la funzione di transizione **può cambiare solamente lo stato**. Quindi, la funzione, prenderà in input la coppia ordinata (*stato, simbolo*), ovvero $(q, a) \in Q \times \Sigma$, e restituirà semplicemente lo stato della configurazione successiva $q \in Q$. Per questo tipo di automi, quindi, so che quando inizierò la computazione avrò sempre un un *numero finito di celle diverse da* B, proprio perchè la testina è di sola lettura e quindi non può cambiare i simboli all'interno del nastro.

- Infine abbiamo lo *stato iniziale*, che è lo stato con cui inizia la computazione, e l'*insieme di stati finali* che indicano quell'insieme di stati che mi dicono se la stringa è accettata o meno, ma non implicano la fine della computazione.

Soffermandoci un attimo sulla funzione di transizione è importante fare una considerazione: infatti per tale funzione, negli automi a stati finiti, sono definite tutte le coppie (*stato,simbolo*) proprio perchè si tratta di **automi a stati finiti**. Può sembrare banale, ma in realtà non tutti gli automi hanno questa caratteristica. In particolare un automa a stati finiti, quindi, data una qualsiasi stringa in input può **decidere se la stringa appartiene al linguaggio o meno**. In questo caso si dice che **linguaggio è riconosciuto** dagli ASFD. Vedremo, invece, per altri riconoscitori, come le *macchine di Turing*, che si dirà che il **linguaggio è accettato**, ovvero data una stringa il riconoscitore mi dice cosa succede se appartiene al linguaggio, ma non sappiamo cosa succede se non gli appartiene. In questo senso capiamo che per questo la funzione di transizione degli ASFD viene detta **funzione totale**, ovvero, come già accennato, sono definite tutte le coppie stato-simbolo della funzione transizione tale che a una coppia (stato,simbolo) corrisponde uno e un solo stato. Di conseguenza l'automa sa sempre darmi la risposta se una stringa appartiene o meno al linguaggio. Per rappresentare inoltre il comportamento dell'automa per ogni coppia di input della funzione di transizione si utilizza una *tabella*, che viene chiamata **tabella di transizione**.

Esempio di Tabella di Transizione Dato l'automa $\mathcal{A} =< \Sigma = \{a, b\}, Q = \{q_0, q_1, q_2\}, \delta, q_0, F = \{q_1\} >$ che ha funzione di transizione definita nel seguente modo:

- $\delta(q_0, a) = q_0$
- $\delta(q_0, b) = q_1$
- $\delta(q_1, a) = q_2$
- $\delta(q_1, b) = q_2$
- $\delta(q_2, a) = q_2$
- $\delta(q_2, b) = q_2$

Notiamo, quindi, che abbiamo definito la funzione per ogni coppia stato-simbolo, di conseguenza, come abbiamo già detto, è una *funzione totale*. Possiamo rappresentare tale funzione di transizione tramite la seguente *tabella di transizione*:

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

Table 2.1: Tabella della funzione di transizione

2.2.1 Diagrammi di Stato di un automa

Un altro modo per *rappresentare* molto efficientemente *un automa*, oltre alla tabella di transizione, è tramite il **diagramma degli stati** che è un *grafo* che ha:

- I **nodi** che rappresentano gli **stati**, e, quindi, sono tanti quanto il numero di stati.
- Gli **archi** che rappresentano le **transizioni**, per cui sono etichettati con il simbolo la cui lettura determina lo stato della transizione.
- Gli **stati finali** sono rappresentati tramite un doppio cerchio o circolo, mentre lo **stato iniziale** è rappresentato tramite una freccia entrante.

Vediamo un esempio prendendo come esempio l'automa visto in precedenza, utilizzando la funzione di transizione vista nella tabella 2.1:

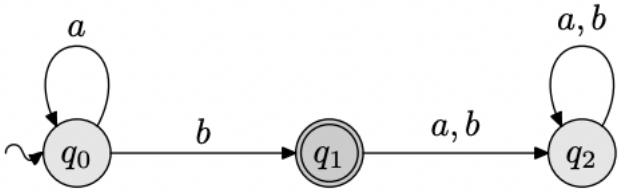
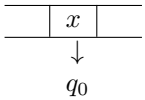


Figure 2.2: Diagramma degli Stati di un automa

2.2.2 Computazione di un Automa a Stati Finiti

Supponiamo di avere la seguente stringa x nel nastro:



In base alla funzione di transizione abbiamo visto che si *scorre la stringa*, fino a che l'automa si rende conto di aver trovato un carattere B, che indica la fine della

stringa. Una volta trovato tale carattere la computazione **termina**. Per sapere se la stringa è accettata o meno si guarda lo stato:

- Se $q_n \in F$, ovvero lo stato appartiene all'insieme di stati finali, allora la stringa è **accettata**
- Se $q_n \notin F$, ovvero lo stato non appartiene all'insieme di stati finali, allora la stringa è **non accettata**

Approfondiremo tale argomento, dando una definizione formale di stringa accettata dall'automa nella sezione 2.2.4

2.2.3 Configurazioni degli Automi a Stati Finiti

Abbiamo già accennato alla configurazione dell'automa in precedenza. Diamo adesso, quindi, una definizione formale, per poi vedere i *tipi di configurazioni* nel quale si può trovare l'automa:

Definizione 2.2.2. *Dato un ASFD* $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ *una* ***configurazione*** *di* $\mathcal{A}$ *è una* ***coppia*** (q, x) *con* $q \in Q$ *e* $x \in \Sigma^*$

Quindi, quella vista nella sezione precedente era la *configurazione iniziale* dell'automa, di cui daremo fra poco la definizione formale. Poichè quindi $x \in \Sigma^*$, una configurazione può anche essere la coppia (q, ε), che corrispondono nel nastro ai caratteri di B. Vediamo quindi di dare una definizione formale che ci descriva formalmente appunto, alcuni tipi particolari di configurazioni [1]:

Definizione 2.2.3. *Una configurazione* (q, x) *con* $q \in Q,\ x \in \Sigma^*$ *è detta:*

- ***Iniziale*** *se* $q = q_0$
- ***Finale*** *se* $x = \varepsilon$
- ***Accettante*** *se* $q \in F$ *e* $x = \varepsilon$

Quindi capiamo che partendo nella computazione da una configurazione iniziale (q_0, x), la stringa è accettata se e solo se l'automa a fine computazione si trova in una configurazione *accettante* tale per cui si ha la coppia (q, ε) con $q \in F$, ovvero q appartiene all'insieme di stati finali. Inoltre, da quanto visto nella definizione, il fatto che q appartenga ad F non implica che la computazione termini, infatti la condizione per cui la computazione termini è che si trovi un carattere di B, ovvero ci troviamo in una configurazione finale (q, ε).

[1] Da adesso in poi useremo una convenzione sui linguaggi formali, molto usata anche nei libri di testo, per cui le lettere (a, b, c) indicano *simboli dell'alfabeto* e le lettere x, y, z indicano invece le *stringhe*

Configurazione Successiva e Relazioni fra Configurazioni

Affinchè la computazione termini, è fondamentale comprendere quale sia la relazione fra configurazioni e il concetto di **configurazione successiva**. Diamo, dunque, la seguente definizione:

Definizione 2.2.4. *Dato un ASFD* $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ *e due configurazioni* (q, x) *e* (q', y) *avremo che la configurazione* (q', y) *è* ***successiva*** *a* (q, x)*, e scriveremo* $(q, x) \vdash_{\mathcal{A}} (q', y)$ *se valgono le seguenti condizioni:*

- $\exists a \in \Sigma$ *tale che* $x = ay$
- $\delta(q, a) = q'$

Esempio 2.2.1. *Prendiamo ad esempio, considerando sempre la funzione di transizione della tabella 2.1, la stringa* $x = aab$*. In questo caso possiamo notare che una possibile configurazione successiva è:*

$$(q_0, aab) \vdash_{\mathcal{A}} (q_0, ab) \tag{2.3}$$

Secondo la tabella di transizione, infatti, abbiamo che $\delta(q_0, a) = q_0$*. Inoltre considerando* $x = aab$ *e* $y = ab$*, abbiamo che esiste un carattere appartenente a* Σ*, che è* a*, tale che* $x = ay \rightarrow aab = a \cdot ab$*. Quindi, poichè sono soddisfatte entrambe le condizioni,* (q_0, ab) *è una possibile configurazione successiva di* (q_0, aab)*.*

Chiusura Riflessiva e Transitiva della relazione $\vdash$

Nella computazione di ASFD passiamo da una configurazione (q_0, x_1) a una configurazione (q_n, x_n) con una successione di configurazioni successive del tipo $(q_0, x_1) \vdash_{\mathcal{A}} (q_1, x_1) \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} (q_n, x_n)$. Un modo equivalente di scrivere tale espressione è:

$$(q_0, x_1) \vdash^*_{\mathcal{A}} (q_n, x_n) \tag{2.4}$$

dove il simbolo $\vdash^*_{\mathcal{A}}$ indica la **chiusura riflessiva e transitiva della relazione** $\vdash$ ed indica il *passaggio di una configurazione all'altra in un numero di passi maggiori o uguali a 0*. Per cui, utilizzando il simbolo $\vdash^*_{\mathcal{A}}$ posso scrivere anche $(q, x) \vdash^*_{\mathcal{A}} (q, x)$, mentre non posso scrivere $(q, x) \vdash_{\mathcal{A}} (q, x)$. Allo stesso modo, esiste il simbolo $\vdash^+_{\mathcal{A}}$ che indica che il numero di passi per passare da una configurazione all'altra deve essere maggiore o uguale a 1.

2.2.4 Stringa Accettata e Linguaggio Riconosciuto

Abbiamo accennato al concetto di **linguaggio riconosciuto** da un ASFD all'inizio della sezione 2.2. Ora che abbiamo introdotto anche il simbolo $\vdash^*_{\mathcal{A}}$, possiamo dare una definizione formale di cosa si intenda per **stringa accettata** e **linguaggio riconosciuto** in un ASFD.

Definizione 2.2.5. *Dato un ASFD* $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ *e* $x \in \Sigma^*$*, diciamo che* x *è* ***accettata*** *da* $\mathcal{A}$ *se e solo se* $(q_0, x) \vdash^*_{\mathcal{A}} (q, \varepsilon)$ *con* $q \in F$*. Il* ***linguaggio riconosciuto*** *da* $\mathcal{A}$ *è l'insieme:*

$$L(\mathcal{A}) = \{x \in \Sigma^* |\ (q_0, x) \vdash^*_{\mathcal{A}} (q, \varepsilon) \text{ con } q \in F\} \tag{2.5}$$

Di conseguenza un **linguaggio riconosciuto** da un ASFD è l'insieme di **stringhe accettate**. Vediamo un esempio.

Esempio 2.2.2. *Utilizzando l'ASFD che ha tabella di transizione 2.1 dimostriamo che:*

$$x = aab \in L(\mathcal{A}) \tag{2.6}$$

ovvero dobbiamo dimostrare che esiste un passaggio dalla configurazione (q_0, aab) *a una configurazione* (q_1, ε) *con un numero di passi maggiori o uguali a 0:*

$$(q_0, aab) \vdash_{\mathcal{A}} (q_0, ab) \vdash_{\mathcal{A}} (q_0, b) \vdash_{\mathcal{A}} (q_1, \varepsilon) \tag{2.7}$$

quindi la stringa $x = aab \in L(\mathcal{A})$.

La domanda che ci possiamo porre sul *linguaggio riconosciuto* da un automa è: $\varepsilon \in L(\mathcal{A})$? A riguardo possiamo dire che:

$$\varepsilon \in L(\mathcal{A}) \text{ se e solo se } q_0 \in F$$

Dimostrazione. Poichè si tratta di una *doppia implicazione* dimostriamo la condizione necessaria ($\Rightarrow$) e la condizione sufficiente ($\Leftarrow$):

- Condizione Necessaria ($\Rightarrow$): se $\varepsilon \in L(\mathcal{A})$ allora esisterà una *derivazione* tale che:
$$(q_0, \varepsilon) \vdash^* (q, \varepsilon) \tag{2.8}$$
Poichè per essere accettata, la stringa deve essere $x = \varepsilon$ e $q \in F$, allora possiamo dire che $q_0 \in F$.
- Condizione Sufficiente ($\Leftarrow$): se $q_0 \in F$ allora possiamo dire che esiste una *derivazione* tale che:
$$(q_0, \varepsilon) \vdash^* (q_0, \varepsilon) \tag{2.9}$$
poichè $q_0 \in F$.

□

2.2.5 Funzione di Transizione Estesa

Una notazione che si utilizza molto spesso è quella di **funzione di transizione estesa**. Se $\delta : Q \times \Sigma \to Q$ è definita per coppie *stato-simbolo*, la funzione di transizione estesa è definita per coppie *stato-stringa* e restituisce lo stato a cui arriviamo dopo applicato δ a ogni simbolo della stringa. In particolare è definita come $\bar{\delta} : Q \times \Sigma^* \to Q$. Riprendendo la tabella di transizione 2.1 vediamo un esempio di come si applica la funzione di transizione estesa, calcolando $\bar{\delta}(q_0, aab)$:

$$\delta(q_0, a) = q_0 \longrightarrow \delta(q_0, a) = q_0 \longrightarrow \delta(q_0, b) = q_1 \tag{2.10}$$

quindi $\bar{\delta}(q_0, aab) = q_1$. Vediamo, quindi, di formalizzare la definizione di funzione di transizione estesa:

Definizione 2.2.6. *La **funzione di transizione estesa** di un ASFD* $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ *è la funzione:*

$$\bar{\delta} : \ Q \times \Sigma^* \to Q \tag{2.11}$$

definita nel seguente modo:

- $\bar{\delta}(q, \varepsilon) = q$
- $\bar{\delta}(q, xa) = \delta(\bar{\delta}(q, x), a)$ *con* $a \in \Sigma$ *e* $x \in \Sigma^*$

Relazione tra Funzione Estesa e configurazioni successive

Vi è un *teorema* molto importante a proposito della funzione di transizione estesa che la mette in relazione col concetto di configurazione successiva, e, in particolare dice che:

Teorema 2.2.1.

$$\bar{\delta}(q, x) = p \Leftrightarrow (q, x) \vdash^* (p, \varepsilon) \tag{2.12}$$

Tale teorema non lo dimostreremo, ma, per esercitazione vi posso dire che si dimostra per **induzione sulla lunghezza della stringa**, vedere sezione 5.2.1 sull'induzione. Tale teorema ci permette inoltre di dare una definizione equivalente di **linguaggio riconosciuto** da un ASFD utilizzando la funzione di transizione estesa. In particolare diciamo che:

$$L(\mathcal{A}) = \{x \in \Sigma^* | \ \bar{\delta}(q_0, x) = q \text{ con } q \in F\} \tag{2.13}$$

2.3 Automi a Stati Finiti Non Deterministici

Per quanto riguarda gli automi, fino ad adesso abbiamo parlato di automi **deterministici**, che hanno la caratteristica che

*ogni configurazione ammette **al più una configurazione successiva***

Infatti sappiamo che se abbiamo la configurazione del tipo (q, ε) la computazione termina, mentre per tutti gli altri casi sappiamo che la funzione di transizione è definita in modo tale che per ogni coppia stato simbolo corrisponde uno e un solo stato, ovvero o abbiamo una configurazione successiva o altrimenti non l'abbiamo e la computazione termina, per questo usiamo il termine *al più*. Parlando, invece, di automi **non deterministici** essi sono così chiamati poichè

*possono ammettere **più configurazioni successive***

Essenzialmente, se parliamo di automi a stati finiti non deterministici, quindi, la differenza principale con quelli deterministici è la **differente funzione di transizione**, che viene chiamata δ_n ed è definita come:

$$\delta_n : \ Q \times \Sigma \to Pow(Q) \tag{2.14}$$

dove $Pow(Q)$ rappresenta l'*insieme delle parti di* Q, ovvero l'insieme i cui elementi sono sottoinsiemi dell'insieme Q, ovvero un *insieme di stati*, come ad esempio $Pow(Q) = \{q_0, q_1\}, \{q_1, q_2\}$ ecc.

2.3.1 Definizione Formale

La definizione formale di **automa a stati finiti non deterministico**, quindi, è del tutto simile a quella vista per gli automi a stati finiti deterministici, se non per la funzione di transizione.

Definizione 2.3.1. *Un Automi a Stati Finiti Non Deterministici (ASFND) $\mathcal{A}_N$ è una quintupla $\mathcal{A}_n =< \Sigma, Q, \delta_N, q_0, F >$ dove:*

- Σ *è l'**alfabeto** (finito)*
- Q *è l'insieme degli stati (finito)*
- δ_N *è la **funzione di transizione***

$$\delta_n : \ Q \times \Sigma \to Pow(Q) \tag{2.15}$$

dove $Pow(Q)$ è l'insieme delle parti di Q [2]

- $q_0 \in Q$ *è lo **stato iniziale***
- $F \subseteq Q$ *è l'**insieme di stati finali***

La domanda che ci si può porre, in questo caso è: **perchè introduciamo anche gli automi a stati finiti non deterministici**? Si può dimostrare che a livello di capacità di riconoscimento di un linguaggio sono perfettamente equivalenti, ovvero con automi deterministici o non deterministici riconosciamo gli stessi linguaggi (sezione 2.4). Ciò che però conta è che *ci permettono con più facilità di riconoscere certe classi di linguaggi*, ovvero quelli dove può essere utile avere **diverse scelte nella computazione**. Facciamo un esempio:

Esempio 2.3.1. *Dato l'alfabeto $\Sigma = \{a, b\}$, vogliamo costruire un automa che riconosca le stringhe che abbiano b come penultimo carattere, considerando che $Q = \{q_0, q_1, q_2\}, F = \{q_2\}$.*

Risoluzione. Costruiamo, dunque, la nostra tabella di transizione, considerando la richiesta dell'esercizio:

δ_n	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\varnothing$	$\varnothing$

In questo caso, tale funzione di transizione, che è quella di un ASFND, ci permette molto facilmente di riconoscere stringhe che abbiano b come penultimo carattere. Se avessimo dovuto creare un automa deterministico, sarebbe stato parecchio più complesso. □

[2] Per insieme delle parti di Q si intende un insieme i cui elementi sono anch'essi sottoinsiemi dell'insieme Q.

Da tale esempio possiamo fare delle considerazioni: Nel caso degli ASFND la funzione **non è** sempre **totale**. Ovvero non sempre è definita per tutte le coppie stato-simbolo, a differenza degli ASFD. Ciò, però, non implica che la computazione non termini, infatti stiamo parlando sempre di automi a stati finiti, ovvero automi che hanno la testina che si muove soltanto verso destra con stringhe di lunghezza finita. Riprendendo l'esempio, quindi, abbiamo che se mi trovo nel caso in cui $\delta(q_2, a) = \varnothing$, allora in tal caso la computazione termina.

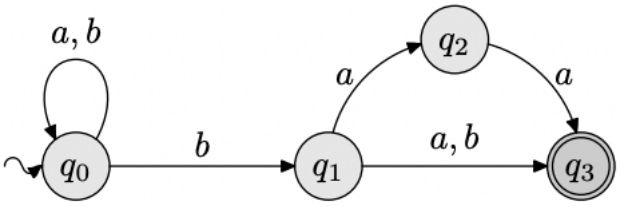
Computazione di un ASFND

Per quanto riguarda la **computazione** di un ASFND, è una computazione **non deterministica**. Poichè, infatti, la funzione di transizione è definita in modo tale che a una coppia stato-simbolo corrisponda un insieme di stati, allora si formeranno diversi *alberi di computazione* per tutte le possibili scelte per ogni stato. Se, quindi, almeno uno dei possibili alberi di computazione è tale che:

$$(q_0, x) \vdash^* (q, \varepsilon) \text{ con } q \in F \tag{2.16}$$

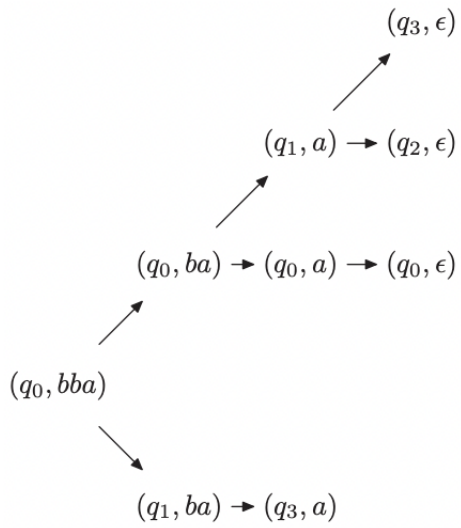
allora la **stringa** è **accettata**.

Esempio 2.3.2. *Prendiamo in considerazione l'automa che ha il seguente diagramma di stati:*



Data la stringa $x = bba$, vogliamo vedere tale stringa è accettata dall'automa.

Risoluzione. Analizziamo dunque tutti i possibili alberi di computazione:



Come possiamo notare, poichè tra tutte le possibili computazioni vi è una configurazione (q_3, ε), con $q_3 \in F$, allora possiamo dire che $x = bba$ è accettata dall'automa.

□

2.3.2 Configurazioni Successive e Linguaggio Riconosciuto da un ASFND

Abbiamo notato che negli ASFND, le configurazioni successive possono essere più di una. Se quindi avessimo, ad esempio, 2 configurazioni successive del tipo:

$$\begin{array}{l}(q_0, bb) \vdash_{\mathcal{A}_N} (q_0, b) \\ (q_0, bb) \vdash_{\mathcal{A}_N} (q_1, b)\end{array} \tag{2.17}$$

allora la notazione più *compatta* per le **configurazioni successive** è la seguente:

$$(\{q_0\}, bb) \vdash_{\mathcal{A}_N} (\{q_0, q_1\}, b) \tag{2.18}$$

Ovvero partendo dall'insieme di stati $\{q_0\}$, le possibili configurazioni successive sono raffigurate dall'insieme di stati $\{q_0, q_1\}$. Ovviamente possiamo avere anche una situazione del tipo:

$$(\{q_0, q_1\}, b) \vdash_{\mathcal{A}_N} (\{q_0, q_1, q_2\}, \varepsilon) \tag{2.19}$$

Quindi utilizzando la chiusura riflessiva possiamo dire che in un numero di passi maggiore o uguale a 0:

$$(\{q_0\}, bb) \vdash^*_{\mathcal{A}_N} (\{q_0, q_1, q_2\}, \varepsilon) \tag{2.20}$$

Dopo aver fatto tali considerazioni, possiamo dunque dare la definizione di **linguaggio riconosciuto da un ASFND**:

Definizione 2.3.2. *Sia $\mathcal{A}_N =< \Sigma, Q, \delta_N, q_0, F >$ ASFND. Il **linguaggio riconosciuto** da un ASFND è così definito:*

$$L(\mathcal{A}_N) = \{x \in \Sigma^* |\ (\{q_0\}, x) \vdash^*_{\mathcal{A}_N} (T, \varepsilon)\} \tag{2.21}$$

con $T \in P(Q)$ [3] *e $T \cap F \neq \varnothing$*

Con tale definizione stiamo dunque dicendo che in zero, uno o più passi dobbiamo arrivare alla configurazione (T, ε) dove T appartiene all'*insieme delle parti di Q*, ovvero è un insieme di stati, proprio per la definizione di ASFND, e in modo tale che T abbia almeno uno stato finale, il ché equivale a dire che l'intersezione $T \cap F \neq \varnothing$.

2.3.3 Funzione di Transizione Estesa di un ASFND

Vediamo adesso la definizione di **funzione di transizione estesa** di un ASFND, che sembrerà leggermente diversa rispetto a quella vista per gli ASFD, ma che in realtà rispecchia esattamente quanto visto nella definizione precedente di funzione di transizione estesa per gli ASFD (sezione 2.2.5):

Definizione 2.3.3. *Sia $\mathcal{A}_N =< \Sigma, Q, \delta_N, q_0, F >$ un ASFND. La **funzione di transizione estesa***

$$\bar{\delta}_N :\ Q \times \Sigma^* \to P(Q) \tag{2.22}$$

è così definita:

- $\bar{\delta}_N(q, \varepsilon) = \{q\} \quad \forall q \in Q$
- $\bar{\delta}_N(q, xa) = \bigcup_{p \in \bar{\delta}_N(q,x)} \delta_N(p, a) \quad \forall q \in Q, a \in \Sigma, x \in \Sigma^*$

In questo caso, poichè la funzione di transizione estesa ci restituisce un *insieme di stati*, e poichè è definita per coppie (stato,stringa) allora, per il passo induttivo, la definiamo come l'unione delle funzioni di transizioni per tutti gli stati appartenenti alla funzione di transizione estesa applicata sulla stringa x. Utilizzando la funzione di transizione estesa, possiamo quindi dare la definizione di linguaggio riconosciuto conoscendo la relazione che vi è tra configurazioni successive e funzione di transizione estesa:

Definizione 2.3.4. *Sia $\mathcal{A}_N =< \Sigma, Q, \delta_N, q_0, F >$ un ASFND. Il **linguaggio riconosciuto da un ASFND** è*

$$L(\mathcal{A}_N) = \{x \in \Sigma^* |\ \bar{\delta}_N(q_0, x) = T \text{ con } T \in P(Q),\ T \cap F \neq \varnothing\} \tag{2.23}$$

2.4 Relazione tra ASFD e ASFND

Abbiamo già accennato al fatto che ASFD e ASFND sono *equivalenti* dal punto di vista della *capacità di riconoscimento di linguaggi*. Possiamo affermare, infatti, che c'è un teorema che formalizza tale concetto:

[3] La notazione $P(Q)$ e $Pow(Q)$ sono equivalenti e stanno ad indicare l'**insieme delle parti** di Q

Teorema 2.4.1. *Dato un linguaggio L allora:*

$$L \text{ è riconosciuto da un ASFND} \Leftrightarrow L \text{ è riconosciuto da un ASFD} \tag{2.24}$$

Non vediamo la dimostrazione per intero, ma vediamo nel dettaglio come vengono costruiti gli automi affinchè il teorema possa essere dimostrato. Inoltre tale costruzione rappresenta un *algoritmo* che ci dice come costruire un ASFD da un ASFND che riconoscono lo stesso linguaggio e viceversa. Si tratta quindi di una *dimostrazione costruttiva*.

Dimostrazione. Poichè si tratta di una doppia implicazione, dobbiamo dimostrare sia il caso ($\Rightarrow$), che il caso ($\Leftarrow$):

- ($\Rightarrow$): in tal caso ciò che si deve fare è costruire un ASFD tale che il linguaggio L riconosciuto dal rispettivo ASFND sia riconosciuto anche dall'ASFD. L'idea che vi è alla base è quella di *considerare come **stati singoli** dell'ASFD **insiemi di stati** dell'ASFND*. La costruzione, formalmente, avviene in tal modo: sia $\mathcal{A}_N =< \Sigma, Q, \delta_N, q_0, F >$ ASFND tale che $L = L(\mathcal{A}_N)$, definisco l'ASFD $\mathcal{A}' =< \Sigma, Q', \delta', q_0', F' >$ in tal modo:
  - $Q' = P(Q)$
  - $q_0' = \{q_0\}$
  - $\delta' : \ Q' \times \Sigma \to Q'$ è definita come:
    - $\delta'(T, a) = \bigcup_{q \in T} \delta_N(q, a) \quad \forall T \in Q', a \in \Sigma$
  - $F' = \{T \in Q' |\ T \cap F \neq \varnothing\}$

  Quindi, considerando come insieme di stati Q, tutti i possibili sottoinsiemi di Q, facciamo sì che a un singolo stato dell'ASFD corrisponda un insieme di stati dell'ASFND. Inoltre la funzione di transizione δ' ci restituirà un insieme di stati dato dall'unione degli stati ottenuti applicando la funzione di transizione ad ogni stato appartenente a T, che quindi sarà il nostro insieme di stati, che nell'ASFD corrisponde al singolo stato. Possiamo notare, quindi, la differente **cardinalità** di Q e Q' Se, infatti, abbiamo che $|Q| = n$, allora la cardinalità di $|Q'| = |P(Q)| = 2^{|Q|} = 2^n$, ovvero la cardinalità dell'insieme delle parti di Q è di 2^n. Da tale considerazione possiamo notare quindi che per ogni ASFND con cardinalità n, esiste il corrispettivo ASFD con cardinalità 2^n.

- ($\Leftarrow$): in tal caso dobbiamo costruire l'ASFND dal rispettivo ASFD che riconosce il linguaggio L. In tal caso risulta molto più semplice la costruzione in quanto *l'ASFD è un particolare tipo di ASFND nel quale la funzione di transizione restituisce un **insieme di stati con cardinalità 1***. Formalmente, dunque, abbiamo che: dato $\mathcal{A} =< \Sigma, Q, \delta, q_0, F >$ ASFD, costruisco $\mathcal{A}_N =< \Sigma, Q, \delta_N, q_0, F >$ dove la funzione di transizione $\delta_N : \ Q \times \Sigma \to P(Q)$ è tale che:

$$\delta_N(q, a) = \{\delta(q, a)\} \tag{2.25}$$

□

2.5 Pumping Lemma per Linguaggi Regolari

Una volta vista la correlazione stretta che vi è tra ASFD e ASFND, analizziamo adesso un teorema molto importante per gli automi a stati finiti in generale che riguarda i **linguaggi regolari**. Diamo prima, però, la definizione di linguaggio regolare:

Definizione 2.5.1 (Linguaggio Regolare)**.** *Un linguaggio L è detto **regolare** se esiste un automa a stati finiti che riconosce L*

Il **pumping lemma**, in tal senso, rappresenta la *condizione necessaria*, ma non sufficiente, affinchè un linguaggio sia regolare. Ovvero possiamo affermare che:

$$L \text{ è regolare} \Rightarrow L \text{ soddisfa il pumping lemma} \tag{2.26}$$

oppure equivalentemente possiamo dire:

$$L \text{ non soddisfa il pumping lemma} \Rightarrow L \text{ non è regolare} \tag{2.27}$$

2.5.1 Enunciato e Dimostrazione del Teorema

L'idea del **pumping lemma** è quello che presa una stringa appartenente al linguaggio tale che soddisfi determinate caratteristiche, allora potrò suddividere tale stringa quante volte voglio ottenendo stringhe appartenenti al linguaggio. Il termine "*pumping*" sta a significare proprio il fatto che posso ripetere tale processo di suddivisione tante volte. Vediamo quindi l'enunciato:

Teorema 2.5.1. *Per ogni linguaggio L esiste una costante $n \geq 0$ tale che, se $z \in L$ e $|z| \geq n$, allora $\exists u, v, w \in \Sigma^*$ tali che $z = uvw$, con $|uv| \leq n$, $|v| \geq 1$ e ottenere che $uv^i w \in L \quad \forall i \geq 0$*

Osservazione 1. *Notiamo che come ipotesi del teorema stiamo dicendo che $|v| \geq 1 \Rightarrow v \neq \varepsilon$. Infatti se $v = \varepsilon$ allora $v^i = \varepsilon \ \forall i \geq 0$. Ma ciò comporta dire che $z = uw \in L$ e che $uv^i w = uw$ e ciò è una banalità, infatti non ci da alcuna informazione in quanto già sapevamo che $uw \in L$, proprio perchè $z = uw$ in questo caso.*

Prima di dimostrare formalmente il teorema vediamo di dare un'idea di come viene affrontata la dimostrazione. Ciò che facciamo inizialmente è porre $n = |Q|$, in tal modo sapendo che $|z| \geq n$, allora sappiamo anche che $|z| \geq Q$. Per il principio dei cassetti o *pigeonhole principle* possiamo dire che poichè la lunghezza della stringa è maggiore del numero di stati dell'automa, allora sicuramente nella computazione ritornerò a uno stato da cui ero già "passato". In particolare troverò un ciclo per v^i.

Dimostrazione. Sia $A =< \Sigma, Q, \delta, q_0, F >$ ASFD tale che $L = L(\mathcal{A})$. Supponiamo di avere una costante $n = |Q|$. Sia $z \in L, |z| = k \geq n$. Sia z_h il *prefisso* di lunghezza $0 \leq h \leq n$. Se ad esempio abbiamo $z = ababb$, allora $z_0 = \varepsilon, z_1 = a, z_2 = ab$, ecc. Sia inoltre q_{ih} lo stato *stato in cui arrivo partendo da q_0 e leggendo*

z_h, ovvero $q_{ih} = \bar{\delta}(q_0, z_h)$. Dal momento che $k \geq n$, deve esistere almeno uno stato in cui il controllo si troverà per almeno 2 volte, ovvero abbiamo 2 indici r, s con $r < s \leq n$, ovvero r, s sono differenti, tali che

$$q_{ir} = q_{is} \longrightarrow \bar{\delta}(q_0, z_r) = q_{ir} = q_{is} = \bar{\delta}(q_0, z_s) \tag{2.28}$$

Poniamo adesso $u = z_r$, $uv = z_s$ e $z = uvw$. Dimostriamo innanzitutto che $|uv| \leq n$ e $|v| \geq 1$:

- Per quanto riguarda $|uv|$, sappiamo che:

  $$|uv| = |z_s| = s \leq n \Rightarrow |uv| \leq n \tag{2.29}$$

  quindi il primo punto è dimostrato.

- Per quanto riguarda $|v|$ sappiamo che:

  $$|u| = |z_r| = r < s = |z_s| = |uv| \tag{2.30}$$

  Quindi, poichè $r < s$, ovvero gli indici sono distinti, sappiamo che $s - r > 0 \rightarrow |uv| - |u| > 0 \rightarrow v \geq 1$.

Dobbiamo dimostrare adesso che $\forall i \geq 0$ abbiamo che $uv^i w \in L$. Per dimostrare tale punto, utilizziamo l'**induzione su** i:

- **Passo Base** $(i = 0)$: devo dunque dimostrare che $uv^0 w = uw \in L$, sapendo che:

  $$uvw \in L \Rightarrow \bar{\delta}(q_0, uvw) \in F \tag{2.31}$$

  Utilizziamo una *proprietà della funzione di transizione estesa*, che non dimostriamo, per la quale:

  $$\bar{\delta}(q, xy) = \bar{\delta}(\bar{\delta}(q, x), y) \tag{2.32}$$

  La dimostrazione prosegue, quindi, nel seguente modo:

  $$\begin{aligned} \bar{\delta}(q_0, uw) = \bar{\delta}(\bar{\delta}(q_0, u), w) = \\ = \bar{\delta}(\bar{\delta}(q_0, z_r), w) = \\ = \bar{\delta}(\bar{\delta}(q_0, z_s), w) = \\ = \bar{\delta}(\bar{\delta}(q_0, uv), w) = \\ = \bar{\delta}(q_0, uvw) = \bar{\delta}(q_0, z) \in F \end{aligned} \tag{2.33}$$

- **Passo Induttivo**: Sia $i > 0$. Supponiamo che $uv^{i-1} w \in L$ e dimostriamo che $uv^i w \in L$. Per ipotesi induttiva sappiamo quindi che:

  $$uv^{i-1} w \in L \Rightarrow \bar{\delta}(q_0, uv^{i-1} w) \in F \tag{2.34}$$

Dimostriamo, dunque, che $\bar{\delta}(q_0, uv^i w) \in F$ utilizzando sempre la proprietà della funzione di transizione:

$$\begin{aligned} \bar{\delta}(q_0, uv^i w) = \bar{\delta}(q_0, uvv^{i-1} w) = \\ = \bar{\delta}(\bar{\delta}(q_0, uv), v^{i-1} w) = \\ = \bar{\delta}(\bar{\delta}(q_0, z_s), v^{i-1} w) = \\ = \bar{\delta}(\bar{\delta}(q_0, z_r), v^{i-1} w) = \\ = \bar{\delta}(\bar{\delta}(q_0, u), v^{i-1} w) = \\ = \bar{\delta}(q_0, uv^{i-1} w) \in F \Rightarrow uv^i w \in L \end{aligned} \tag{2.35}$$

□

2.5.2 Utilizzo del Pumping Lemma

Abbiamo affermato all'inizio di questa sezione che il *pumping lemma* è una *condizione necessaria* affinchè un linguaggio sia regolare. In tal senso, è possibile utilizzare il pumping lemma per dimostrare che un linguaggio non è regolare. Vediamo quindi un esempio sul come utilizzare il pumping lemma per dimostrare che un linguaggio non è regolare:

Esempio 2.5.1. *Dimostrare che il linguaggio* $L = \{a^m b^m | \ m \geq 0\}$ ***non è regolare***.

Risoluzione. Le stringhe di tale linguaggio saranno quindi del tipo "$ab, aabb, aaabbb, ...$". Prima di svolgere la richiesta dell'esercizio, è importante fare una considerazione. Infatti se avessimo un linguaggio $L' = \{ab, aabb, aaabbb\}$ esso sarebbe regolare, infatti potremmo costruire un automa che per ogni stringa esegue una computazione diversa, con un numero di stati pari alla somma delle lunghezze delle stringhe. Ciò che invece possiamo dire sul linguaggio $L = \{a^m b^m\}$, è il fatto che il numero di stringhe di tale linguaggio è **infinito**, il che comporterebbe avere un *automa con un numero infinito di stati* , che è impossibile per la definizione stessa di automa. Tale considerazione ci porta ad affermare la seguente cosa:

- Se il linguaggio è **finito**, allora è sicuramente **regolare**, infatti si può sempre costruire l'automa.
- Se il linguaggio è **infinito**, può essere **regolare o non regolare**.

Iniziamo quindi con la vera dimostrazione: affinchè la dimostrazione che L non sia regolare sia corretta, dobbiamo **negare il pumping lemma**, il ché equivale a dire:

Dato un linguaggio L, per ogni costante $n \geq 0$, esiste una stringa $z \in L : \ |z| \geq n$ tale che per ogni u, v, w con $z = uvw$ e $|uv| \leq n$, $|v| \geq 1$ allora otteniamo che esiste $i \geq 0$ tale che $uv^i w \notin L$.

Quello che si fa quindi, negando il teorema, è sostituire i *quantificatori*, in particolare scambiando il quantificatore esistenziale con quello universale. Per quanto riguarda il linguaggio che ci interessa, prendiamo una stringa $z = a^n b^n$, che quindi ha

lunghezza $|z| = 2n \geq n$. Quando suddividerò la stringa, quindi, avrò che $uv \in \{a\}^*$ e $v \in \{a\}^*$, ovvero sia la stringa uv che la stringa v conterranno solo 'a', infatti sappiamo che i primi n simboli sono a e gli ultimi n sono b, e, poichè $|uv| \leq n$, vale quanto scritto sopra. Allora possiamo dire che $\exists i : \ i = 0$ tale che $uv^0w = uw \notin L$, poichè infatti in questo caso avremmo un numero diverso di a e b, il ché va in contraddizione con la costruzione del linguaggio.

□

2.6 Proprietà di chiusura dei linguaggi regolari

Una volta compreso il significato effettivo di *linguaggio regolare* e l'utilizzo del *pumping lemma* per dimostrare che un linguaggio regolare, vediamo adesso una serie di teoremi che mettono in luce le **proprietà di chiusura dei linguaggi regolari** rispetto a moltissime operazioni.

2.6.1 Unione

Per quanto riguarda l'operazione di **unione**, vale il seguente teorema:

Teorema 2.6.1. *Dati 2 linguaggi regolari L_1, L_2, la loro **unione** $L_1 \cup L_2$ è anch'esso un linguaggio regolare*

Non dimostriamo il teorema, ma diciamo che anche in questo caso si tratta di una *dimostrazione costruttiva*, ovvero che mostra un algoritmo che ci indica come costruire gli automi. In questo caso dati 2 automi $\mathcal{A}_1, \mathcal{A}_2$ che riconoscono, rispettivamente, L_1, L_2, si crea un nuovo automa con un *nuovo stato iniziale* tale che nella sua computazione ci sia un cammino che dallo stato iniziale vada a finire sia nella computazione di un automa che nell'altro. Nel caso ci siano stati che hanno stesso nome si rinominano.

2.6.2 Complemento

La classe dei linguaggi regolari è chiusa rispetto al **complemento**. Il teorema recita:

Teorema 2.6.2. *Dato un linguaggio regolare $L \subseteq \Sigma^*$, allora anche $\bar{L} = L^c = \Sigma^* \backslash L$ è regolare*

In tal caso la dimostrazione ci dice come costruire l'automa che riconosca il complemento. In questo caso basta **scambiare gli stati finali con quelli iniziali**. Il problema si pone poichè, essendo non deterministico, non è detto che l'automa legga tutta la stringa. Allora, in questo caso, si costruisce il corrispondente automa deterministico che ha una *funzione totale* che, qualunque sia una stringa in input, leggerà tutta la stringa.

2.6.3 Intersezione, concatenazione e chiusura riflessiva

Oltre a quelle già viste, vi sono altre operazioni rispetto alle quali i linguaggi regolari sono chiusi, e sono:

- **Intersezione**
- **Concatenazione**
- **Chiusura riflessiva di Klenee**

2.7 Espressioni Regolari

Un altro strumento che utilizziamo per *rappresentare linguaggi* sono le **espressioni regolari**, che sono *scritture che usano simboli di un alfabeto e simboli delle operazioni, che corrispondono a un linguaggio*, in particolare sono uno strumento "finito" per rappresentare linguaggi anche infiniti. Vediamo quindi di dare una definizione formale:

Definizione 2.7.1. *Dato un alfabeto Σ e dato l'insieme di simboli $\{+, *, (,), \cdot, \varnothing\}$, si definisce **espressione regolare su** Σ una stringa $r \in (\{+, *, (,), \cdot, \varnothing\} \cup \Sigma)^+$ [4] tale che valga una delle seguenti condizioni:*

1. $r = \varnothing$
2. $r = a$ *con* $a \in \Sigma$
3. $r = (s + t)$ *oppure* $r = (s \cdot t)$ *oppure* $r = (s^*)$ *con* s, t *espressioni regolari su* Σ.

Esempio 2.7.1. *Vediamo degli esempi di espressioni regolari e non, dato un alfabeto* $\Sigma = \{a, b\}$*:*

- $(a + \cdot b \cdot b)$*: vediamo di scomporre tale espressione per vedere se sia regolare o meno. Notiamo che abbiamo la somma di* a *e* $\cdot b \cdot b$*: sappiamo che* a *è espressione regolare, infatti* $a \in \Sigma$*, mentre abbiamo che possiamo suddividere* $\cdot b \cdot b$ *in un prodotto* $\varepsilon \cdot (b \cdot b)$*. Abbiamo però detto che, poichè si trattava della chiusura positiva e non riflessiva dell'insieme* $\{+, *, (,), \cdot, \varnothing\} \cup \Sigma$*, allora un espressione regolare non può avere la stringa vuota, quindi tale espressione non è regolare e quindi non lo è neanche quella finale.*
- $((a^* \cdot b)^* + (a \cdot b))$*: in questo caso abbiamo la somma di due espressioni che sono* $(a^* \cdot b)^*$ *e* $(a \cdot b)$*: sappiamo che* $(r)^*$ *è espressione regolare, e quindi* $(a^* \cdot b)^*$ *è espressione regolare, infatti si tratta di un prodotto di due termini che soddisfano la condizione per essere espressione regolare. Ovviamente anche* $(a \cdot b)$ *è espressione regolare, quindi* $((a^* \cdot b)^* + (a \cdot b))$ *è espressione regolare.*

[4] Con tale scrittura stiamo facendo un riferimento alla chiusura positiva di un linguaggio (sezione 1.2.4) per il quale diciamo che r, essendo una stringa è una concatenazione dei simboli $\{+, *, (,), \cdot, \varnothing\} \cup \Sigma$, ma dal quale vogliamo escludere ε, ovvero la stringa vuota.

2.7.1 Linguaggio rappresentato da un Espressione Regolare

Una volta definite le espressioni regolari, possiamo dare la definizione formale di **linguaggio rappresentato da un espressione regolare**:

Definizione 2.7.2. *Sia r espressione regolare su Σ. Il linguaggio rappresentato da r si indica con $\mathcal{L}(r)$ ed è così definito:*

1. *Se $r = \varnothing$, allora $\mathcal{L}(r) = \Lambda$*
2. *Se $r = a \in \Sigma$, allora $\mathcal{L}(r) = \{a\}$*
3. *Se $r = (s + t)$, allora $\mathcal{L}(r) = \mathcal{L}(s) \cup \mathcal{L}(t)$*
4. *Se $r = (s \cdot t)$, allora $\mathcal{L}(r) = \mathcal{L}(s) \cdot \mathcal{L}(t)$*
5. *Se $r = s^*$, allora $\mathcal{L}(r) = (\mathcal{L}(s))^*$*

Esempio 2.7.2. *Vediamo quindi applicando la definizione alcuni esempi di linguaggi rappresentati dalle espressioni regolari:*

- $r = (a^* \cdot b)$. *Applichiamo la definizione:*

$$\begin{aligned} \mathcal{L}(a^* \cdot b) = \\ \mathcal{L}(a^*) \cdot \mathcal{L}(b) = \\ (\mathcal{L}(a))^* \cdot \mathcal{L}(b) = \\ (\{a\})^* \cdot \{b\} = \{a^n b |\ n \geq 0\} \end{aligned} \tag{2.36}$$

- $r = (\varnothing)^*$*: vediamo questo particolare caso:*

$$\mathcal{L}((\varnothing)^*) = (\mathcal{L}(\varnothing))^* = \Lambda^* = \{\varepsilon\} \tag{2.37}$$

Espressioni Equivalenti

Possiamo dire che data un espressione regolare esiste uno e un solo linguaggio rappresentato dall'espressione stessa, però vi possono essere più espressioni che rappresentano lo stesso linguaggio. In tal caso si parla di **espressioni equivalenti**:

Definizione 2.7.3. *Due espressioni regolari si dicono **equivalenti** se rappresentano lo stesso linguaggio*

Esempio 2.7.3. *Vediamo per esempio le seguenti espressioni regolari: $r = ((a + b) \cdot b)$ e $s = ((a \cdot b) + (b \cdot b))$.*
Analizziamo dunque i linguaggi rappresentati dalle 2 espressioni:

$$\begin{aligned} \mathcal{L}(((a + b) \cdot b)) = \{a, b\} \cdot \{b\} = \{ab, bb\} \\ \mathcal{L}(((a \cdot b) + (b \cdot b))) = \{ab\} \cup \{bb\} = \{ab, bb\} \end{aligned} \tag{2.38}$$

Notiamo che le espressioni rappresentano lo stesso linguaggio e sono dunque equivalenti.

Convenzioni sulle Espressioni Regolari

Per facilitare la scrittura delle espressioni regolari, si utilizzano delle convenzioni sia sulle *operazioni*, che sulle *parentesi*:

- Per quanto riguarda le operazioni diciamo che la **concatenazione '$\cdot$' ha precedenza sulla somma**. E l'operazione di chiusura riflessiva $*$ ha precedenza sulla concatenazione. Per esempio, quindi, abbiamo che:
  - $\mathcal{L}(a + b \cdot a) = \mathcal{L}(a) \cup \mathcal{L}(ba) = \{a, ba\}$
  - $\mathcal{L}((a + b) \cdot a) = \{a, b\} \cdot \{a\} = \{aa, ba\}$
- Per quanto riguarda invece le parentesi diciamo che:

$$\begin{array}{c} (a+b) \rightsquigarrow a+b \\ (a \cdot b) \rightsquigarrow a \cdot b \\ a \cdot b \rightsquigarrow ab \end{array} \tag{2.39}$$

2.7.2 Relazione fra Automi ed Espressioni Regolari

Un importantissimo risultato nello studio delle espressioni regolari è stato quello di trovare una *fortissima* ***correlazione tra automi ed espressioni regolari***. In particolare è stato dimostrato un teorema che dice:

Teorema 2.7.1. *Sia $L \subseteq \Sigma^*$ un linguaggio formale. Allora*

$$L \text{ è regolare} \iff \exists \text{ un'espressione regolare che lo rappresenta}$$

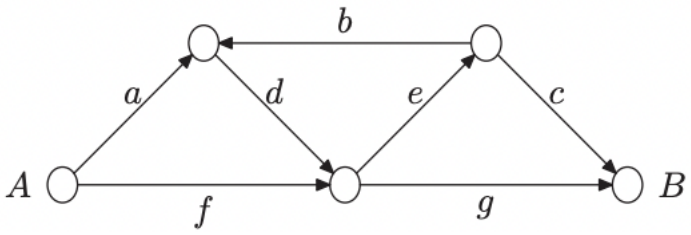
cioè $L = \mathcal{L}(r)$

Il risultato di tale teorema è importantissimo, infatti sappiamo che se riusciamo a costruire per un linguaggio la rispettiva espressione regolare che lo rappresenta, allora esisterà sicuramente l'automa che lo riconosce. Vediamo alcuni esempi:

- Il linguaggio $L = \{a^n b^n |\ n \geq 0\}$ *non è regolare* (esempio 2.5.1), infatti non esiste alcuna espressione regolare che lo rappresenta.
- Il linguaggio $L = \{a^n b^m |\ m \geq 1\}$ è regolare, infatti esiste l'espressione regolare che lo rappresenta, ovvero $r = aa^*bb^*$

Esempio della Rete Stradale

Consideriamo la seguente *rete stradale*:



Il nostro intento è quello di **trovare tutti i possibili cammini dal nodo A al nodo B** scrivendo l'*espressione regolare* che rappresenta le possibili *etichette dal nodo A al nodo B*. Utilizzando il teorema sulla relazione fra automi ed espressioni regolari, possiamo considerare tale rete stradale come un *automa che ha stato iniziale A e stato finale B*. A questo punto possiamo affermare che tutte le stringhe riconosciute dall'automa sono le giuste etichette, e inoltre possiamo affermare che esisterà sicuramente l'espressione regolare che rappresenta il linguaggio. Notiamo che al centro abbiamo il ciclo $(ebd)^*$. L'espressione regolare che possiamo dunque scrivere è la seguente:

$$r = (ad + f) \cdot (ebd)^* \cdot (g + ec) \tag{2.40}$$

Infatti possiamo affermare che partendo da A, i due possibili cammini per arrivare al nodo dove comincia il ciclo sono ad unito ad f. Una volta che termina il ciclo i possibili cammini per arrivare a B sono o ec o g.

2.8 Cardinalità dei Linguaggi

Uno dei problemi più importanti dell'informatica è quello di riconoscere se una stringa appartiene ad un linguaggio. Ad esempio quando un compilatore deve verificare la correttezza sintattica di un determinato programma scritto con un determinato linguaggio, deve verificare se le stringhe scritte in quel determinato programma appartengano a quel linguaggio. Vedremo in questo capitolo, valutando un problema di **cardinalità degli insiemi**, riusciremo a osservare che non esiste un algoritmo di riconoscimento per tutti i linguaggi.

2.8.1 Ordinamento Lessicografico

Introduciamo adesso la definizione di **ordinamento lessicografico**, che ci darà un aiuto per affrontare il nostro problema sulla cardinalità. In parole semplici, possiamo dire che l'ordinamento lessicografico delle stringhe di Σ^* è un *ordinamento sulla lunghezza delle stringhe al quale applichiamo l'ordine alfabetico*. Il motivo per cui non ordiniamo le stringhe per ordine alfabetico è che si potrebbe creare il seguente problema: consideriamo $\Sigma = \{a, b\}$; se applicassimo l'ordinamento alfabetico avremmo il seguente ordine:

$$\begin{array}{c} a \\ aa \\ aaa \\ aaaa \\ \vdots \\ b \\ ba \end{array} \tag{2.41}$$

Come possiamo notare si pone il problema di avere dei simboli che non sono ordinati in maniera corretta (esempio: b, ba), e, per ovviare a tale problema utilizziamo un ordinamento lessicografico.

Definizione 2.8.1. *Dato un alfabeto $\Sigma = \{a_1, a_2, ..., a_n\}$, si definisce **ordinamento lessicografico** delle stringhe di Σ^*, l'ordinamento $<$ ottenuto stabilendo un ordinamento fra i caratteri di Σ (ad esempio $a_1 < a_2 < ... < a_n$) e definendo l'ordinamento di due stringhe $x, y \in \Sigma^*$ in modo tale che $x < y$ se e solo se una delle due condizioni seguenti è verificata:*

1. $|x| < |y|$
2. $|x| = |y|$, *ed esiste* $z \in \Sigma^*$ *tale che* $x = za_iu$ *e* $y = za_jv$ *con* $u, v \in \Sigma_*$ *e* $i < j$

Quindi, tornando sul nostro alfabeto $\Sigma = \{a, b\}$ avremo un ordinamento lessicografico di questo tipo:

$$\begin{array}{c} \varepsilon \\ a \\ b \\ aa \\ ab \\ ba \\ bb \\ \vdots \end{array} \tag{2.42}$$

2.8.2 Cardinalità di Linguaggi e Cardinalità di Programmi

Torniamo adesso alla nostra domanda iniziale, ovvero *esistono più linguaggi rispetto a programmi che riconoscono un certo linguaggio?*
Considerando che esiste un ordinamento lessicografico su Σ^*, possiamo dire che gli elementi di Σ^* sono *numerabili*, ovvero sono infiniti quanto i numeri naturali, dove $|\mathbb{N}| = \aleph_0$. L'insieme di tutti i linguaggi, possiamo quindi dire che rappresenta tutti i possibili sottoinsiemi di Σ^*, sono $|P(\mathbb{N})| = 2^{\aleph_0}$, ovvero una quantità non numerabile. Per quanto riguarda i linguaggi di programmazione possiamo dire che l'insieme di programmi corrisponde al più all'insieme di stringhe di Σ^*, ovvero $\aleph_0$. Infatti vengono generate le stringhe di Σ_p^* le quali vengono accettate dal compilatore solo se sono sintatticamente corrette per quel linguaggio. Ma, allora se abbiamo visto che la cardinalità dell'insieme dei linguaggi è una quantità non numerabile mentre la cardinalità dell'insieme dei programmi lo è, allora possiamo affermare che non possono essere messi in corrispondenza, ovvero esistono linguaggi che non possono essere riconosciuti da programmi.

Capitolo 3

Generazione di Linguaggi

Nel capitolo precedente abbiamo trattato dei *metodi riconoscitivi di un linguaggio*. Adesso tratteremo i **metodi generativi**, ovvero quei metodi che mi permettono di **generare tutte le stringhe di un linguaggio**

3.1 Grammatiche

Lo strumento grazie al quale riusciamo a generare tutte le stringhe di un linguaggio sono le **grammatiche**. Esse sono formate da un *insieme di regole* che mi permettono di generare appunto le stringhe di un determinato linguaggio. Le regole sono *coppie ordinate di stringhe* di questo tipo:

$$(S, 0A1) \quad (0A, 00A1) \quad (A, \varepsilon) \tag{3.1}$$

dove i simboli $\{0, 1\}$ sono chiamati *simboli terminali*, e sono quei simboli che ci danno l'alfabeto sul quale genero le stringhe di un linguaggio; mentre i simboli $\{S, A\}$ sono chiamati *simboli non terminali* e servono per ottenere le stringhe del linguaggio. Per ottenere le stringhe di un linguaggio applico le *regole di sostituzione*. Considerando quindi le regole scritte in precedenza, un esempio di sostituzione può essere il seguente:

$$S \quad 0A1 \quad 00A11 \quad 00\varepsilon 11 = 0011 \tag{3.2}$$

Vediamo quindi di dare una definizione formale:

Definizione 3.1.1. *Una **grammatica formale** G è una quadrupla $G =< V_T, V_N, P, S >$ dove:*

- *V_T è l'insieme finito e non vuoto di **simboli terminali***
- *V_N è l'insieme finito e non vuoto di **simboli non terminali***
- *$S \in V_N$ è il simbolo iniziale della grammatica*

- P *è l'insieme finito di* ***regole di produzione*** *ed è una relazione binaria, ovvero una coppia ordinata di stringhe, definita come:*

$$((V_T \cup V_N)^* \cdot V_N \cdot (V_T \cup V_N)^*) \times (V_T \cup V_N)^* \tag{3.3}$$

dove la coppia $(\alpha, \beta) \subseteq P$ *si può rappresentare come* $\alpha \to \beta$*, e si legge "*α *produce* β *"*

Soffermandoci un attimo su come è definita la relazione binaria di una regola, notiamo di avere come primo elemento della relazione una concatenazione di tre insiemi di stringhe; questo perchè dobbiamo avere almeno un simbolo non terminale in ogni regola.

3.1.1 Linguaggi generati da una Grammatica

Una volta compreso che le grammatiche sono appunto strumenti atti alla generazione di stringhe; vogliamo adesso dare una definizione formale di cosa è un **linguaggio generato da una grammatica**. Per dare tale definizione dobbiamo anche dare la definizione di *forma proposizionale* e di *frase generata da una grammatica*:

Definizione 3.1.2. *Data una grammatica* $G =< V_N, V_T, P, S >$*, le* ***forme proposizionali*** *di* G *si definiscono come segue:*

- S *è una forma proposizionale*
- *Se* $\alpha\beta\gamma$ *è una forma proposizionale e* $\beta \to \delta$ *è una regola di produzione, allora* $\alpha\delta\gamma$ *è una forma proposizionale*

Una forma proposizionale generata in V_T^* *è detta* ***frase generata da*** G*.*

Una volta formalizzato il concetto di forma proposizionale, che formalizza il concetto di sostituzione nelle regole di produzione, e formalizzato il concetto di frase come stringa di simboli terminali, possiamo dare la definizione di linguaggio generato da G:

Definizione 3.1.3. *Il* ***linguaggio generato da*** G*, detto* $L(G)$*, è l'insieme delle frasi generate da* G*.*

Considerando la grammatica G, che aveva le seguenti regole di produzione: $P = \{S \to 0A1,\ 0A \to 00A1,\ A \to \varepsilon\}$, allora il linguaggio generato da tale grammatica è il seguente:

$$L(G) = \{0^n 1^n |\ n \geq 1\} \tag{3.4}$$

Infatti possiamo notare che tutte le frasi generate da G hanno lo stesso numero di 0 e 1. Inoltre, possiamo dire che le regole del tipo $A \to \varepsilon$ vengono dette ε-**produzioni**. Oltre a tale esempio, potrebbe esistere una *grammatica che genera un linguaggio vuoto*, come la grammatica che ha le seguenti regole di produzione: $P = \{S \to Ab,\ A \to Sa\}$. Infatti con tali regole non troveremo mai frasi generate da G, di conseguenza

$$L(G) = \Lambda \tag{3.5}$$

3.1.2 Derivazioni

Nelle grammatiche possiamo definire anche il concetto di **derivazione**. Cioè, applicando le regole di produzione, partendo da una stringa ne otteniamo una nuova che abbiamo derivato dalla precedente. Inoltre, possiamo avere 3 tipi di derivazione:

- **Derivazione Diretta**
- **Derivazione non banale**
- **Derivazione**

Andiamo quindi a definire questi 3 tipi di derivazione formalmente.

Derivazione Diretta

Definizione 3.1.4. *Data una grammatica G, se $\alpha, \beta, \gamma \in (V_T \cup V_N)^*$ e $\beta \to \delta \in P$, allora dirò che $\alpha\beta\gamma$ **deriva direttamente** $\alpha\delta\gamma$ e scriverò:*

$$\alpha\beta\gamma \Rightarrow_G \alpha\delta\gamma \tag{3.6}$$

Per esempio se considerassimo la regola $S \to 0A1$, possiamo dire che $S \Rightarrow_G 0A1$: infatti se consideriamo la stringa $\alpha\beta\gamma = \varepsilon S\varepsilon$ dove $\alpha, \gamma = \varepsilon$ e $\beta = S$, allora possiamo dire che $\alpha\delta\gamma = \varepsilon 0A1\varepsilon$, ovvero S deriva direttamente $0A1$. Allo stesso modo possiamo dire che, considerando la regola $0A \to 00A1$ possiamo dire che:

$$01010A1 \Rightarrow_G 010100A11 \tag{3.7}$$

dove $\alpha = 0101, \gamma = 1$ e $\beta = 0A$.

Derivazione non banale

Definizione 3.1.5. *Data una grammatica G, siano $\alpha, \beta \in (V_T \cup V_N)^*$. Diciamo che α **deriva in modo non banale** β e scriviamo $\alpha \Rightarrow_G^+ \beta$ se e solo se $\exists \alpha_0, \alpha_1, ..., \alpha_n$ con $n \geq 1$ tali che:*

$$\alpha = \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G ... \Rightarrow_G \alpha_n = \beta \tag{3.8}$$

Se ad esempio considerassimo la grammatica che ha le regole $P = \{S \to 0A1,\ 0A \to 0A11\}$ allora potremmo dire che $S \Rightarrow_G^+ 00A11$, ma non possiamo dire che $S \Rightarrow_G^+ 0A1$, infatti il numero di passi della derivazione deve essere maggiore o uguale a 1, ovvero non può essere una semplice derivazione diretta.

Derivazioni

Definizione 3.1.6. *Data una grammatica G, siano $\alpha, \beta \in (V_T \cup V_N)^*$. Diciamo che α **deriva** β e scriviamo $\alpha \Rightarrow_G^* \beta$, se $\exists \alpha_0, \alpha_1, ..., \alpha_n$ con $n \geq 0$ tali che:*

$$\alpha = \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G ... \Rightarrow_G \alpha_n = \beta \tag{3.9}$$

In tal caso questo tipo di derivazione ammette anche la derivazione $\alpha \Rightarrow_G^* \alpha$, proprio perchè gode della *proprietà riflessiva*, a differenza della derivazione non banale.

Linguaggio generato da una grammatica con derivazioni

A questo punto, una volta che abbiamo introdotto il concetto di *derivazione*, possiamo dare la definizione di **linguaggio generato da una grammatica** come:

$$L(G) = \{x \in V_T^* |\ S \Rightarrow_G^* x\} \tag{3.10}$$

In realtà, sarebbe semanticamente più corretto indicare una *derivazione non banale*: infatti in 0 passi la stringa che otterremmo sarebbe $S \notin V_T^*$. Per convenzione, però, si decide di utilizzare la derivazione con il simbolo '$*$'.

3.2 Classificazione delle Grammatiche

Le grammatiche sono uno strumento molto potente che attraverso un insieme di relazioni finito, riesce a generare linguaggi anche infiniti. Proprio per tale motivo si decide di **classificare le grammatiche** in base a quali *classi di linguaggi* la grammatica riesce a generare. In particolare abbiamo:

- **Grammatiche di tipo 0**, dette *senza restrizioni* o *unrestricted*: questo tipo di grammatica ha regole di produzione del tipo $\alpha \to \beta$, in cui $\alpha \in (V_N \cup V_T)^* V_N (V_N \cup V_T)^*$ e $\beta \in (V_N \cup V_T)^*$. La classe di linguaggi generata da tali grammatiche corrisponde ai *linguaggi riconosciuti da macchine di Turing*.

- **Grammatiche di tipo 1** dette *contestuali* o *context-sensitive*: questo tipo di grammatiche ha regole di produzione del tipo $\alpha \to \beta$ nel quale $|\alpha| \leq |\beta|$, ovvero la lunghezza della stringa α è minore o uguale rispetto alla lunghezza di β. La classe di linguaggi generata da tali grammatiche corrisponde ai *linguaggi accettati da automi limitati linearmente*.

- **Grammatiche di tipo 2** dette *acontestuali* o *context-free*: sono le grammatiche che hanno regole di produzione del tipo $A \to \beta$ con $A \in V_N$ e $\beta \in (V_N \cup V_T)^+$ (escludiamo in β la parola vuota ε).

- **Grammatiche di tipo 3** dette *regolari*: sono le grammatiche che hanno regole di produzione del tipo $A \to \delta$ con $A \in V_N$ e $\delta \in (V_T \cdot V_N) \cup V_T$, dove δ è un simbolo terminale seguito da uno non terminale, oppure un singolo simbolo terminale. La classe del linguaggi generati da questo tipo di grammatica è la classe dei *linguaggi regolari*. In particolare, per tale tipo di linguaggio sappiamo sicuramente che esiste un automa a stati finiti che lo riconosce, proprio per la definizione di linguaggio regolare. Da notare come i linguaggi regolari siano caratterizzati da molti punti di vista (grammatiche di tipo 3, automi a stati finiti, espressioni regolari).

3.2.1 Gerarchia Di Chomsky

Dalle definizioni che abbiamo dato sulla classificazione delle grammatiche, è facilmente intuibile che dato un $0 \leq n \leq 2$, ogni grammatica di tipo $n+1$ è una

particolare grammatica di tipo n, ovvero vale la seguente gerarchia, detta **gerarchia di Chomsky**:

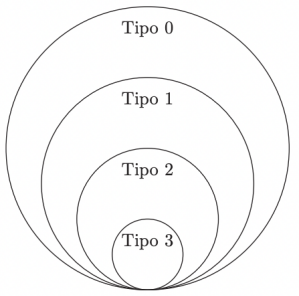


Figure 3.1: Gerarchia di Chomsky

Ci si può chiedere dunque l'utilità di avere tale classificazione di grammatiche, e la risposta, molto banalmente, è che il nostro obiettivo è quello di trovare la grammatica più semplice possibile che riesca a generare un linguaggio. Da qui nasce la seguente definizione:

Definizione 3.2.1. *Un linguaggio è detto* ***strettamente di tipo*** n *se esiste una grammatica di tipo* n *che lo genera e non esiste alcuna grammatica di tipo* m*, con* $m > n$*, che lo genera.*

Una nota a margine da fare riguarda la differenza tra gramatiche *contestuali* e grammatiche *context-free*, e in particolare capire perchè viene dato tale nome:

- Per quanto riguarda le grammatiche *contestuali*, è possibile dimostrare che le regole di produzione del tipo $\alpha \to \beta$ con $|\alpha| \leq |\beta|$ sono equivalenti alle regole del tipo $\alpha A \beta \to \alpha \delta \beta$. Notiamo che affichè A possa essere sostituito, deve trovarsi tra α e β, ovvero deve trovarsi in uno specifico **contesto**. Da qui il nome dato a tale tipo di grammatica.
- Per quanto riguarda le grammatiche *contestuali* o *context-free*, le regole sono del tipo $A \to \delta$. In tal caso non ci interessa del contesto, ma posso sostituire A con δ quando voglio. Per questo sono chiamate *context-free*.

Parte II

Logica

Capitolo 4

Sistemi Formali

4.1 Introduzione alla Logica e ai Sistemi Formali

Affrontiamo la parte della **logica**. La maggiorparte di noi studenti non ha un idea chiara di cosa significhi programma o algoritmo o di cosa sia un linguaggio di programmazione. Vedremo, quindi, come i **programmi** che scriviamo noi informatici, non sono altro che **dimostrazioni matematiche**. Di conseguenza vedremo cosa si intende per *dimostrazione formale* e cosa si intende per *programma* o *algoritmo*. Capiremo che significa **computare**, e, annesso a questo, cos'è un **calcolo**. Abbiamo accennato alla fortissima correlazione tra dimostrazioni e programmi senza però spiegare effettivamente di cosa si occupi la **logica**. Essa appunto è lo **studio del ragionamento corretto**. In questo ambito introduciamo i **sistemi formali**, che non sono altro che la **descrizione formale di un ragionamento corretto in un contesto specifico**. Ovvero il nostro scopo è dimostrare che formalmente è stato fatto un ragionamento incontrovertibile.

4.2 Definizione di Sistema Formale

Abbiamo compreso che lo scopo di un sistema formale è quello di descrivere formalmente un ragionamento corretto. Ma cos'è un **ragionamento**? Un ragionamento è essenzialmente composto da **affermazioni**. Però, come abbiamo detto, noi vogliamo descrivere formalmente il ragionamento, quindi dobbiamo **formalizzare il concetto di affermazione**. Nei sistemi formali queste possono essere chiamate:

- **Formule ben Formate (fbf)**
- **judgement**
- **asserzioni**

Formalmente, quindi, possiamo affermare che un ragionamento è una **lista di asserzioni**. Siamo però ancora lontani da una definizione precisa di sistema formale.

Adesso quindi diamo una definizione precisa di sistema formale, e spieghiamo ogni passaggio per bene:

Definizione 4.2.1 (**Sistema Formale**). *Un **sistema formale** $\mathcal{D}$ è dato da:*

- *Un **insieme numerabile** S (alfabeto);*
- *Un **insieme decidibile** $W \subseteq S^*$ (insieme delle fbf)*
- *Un **insieme degli assiomi** $Ax \subseteq W$;*
- *Un **insieme finito di regole finitarie** $\mathcal{R} = \{R_i\}_{i \in I}$, con $R_i \subseteq W^{n_i}$ con I ed $n_i \geq 2$ finiti.*

Sintassi di un sistema formale con l'esempio del sistema formale $\mathcal{CL}$

Probabilmente da quanto scritto nella definizione non abbiamo capito nulla. Cerchiamo quindi di spiegare ogni passaggio, in particolare *come si descrive la **sintassi di un sistema formale*** [1], facendo anche utilizzo dell'esempio di un sistema formale chiamato $\mathcal{CL}$:

- Partiamo dall'**insieme numerabile** S: abbiamo specificato con una parentesi che può essere chiamato anche *alfabeto*. Richiamando quanto detto nella parte di linguaggi fomali (definizione 1.1.1), possiamo affermare che un insieme numerabile è un qualsiasi insieme che può essere messo in corrispondenza biunivoca con l'insieme $\mathbb{N}$ dei numeri naturali, quindi è un insieme infinito ma con *cardinalità* dell'insieme $\mathbb{N}$. Nell'esempio del sistema formale $\mathcal{CL}$, l'insieme numerabile è così composto:

  $$S = \{k, s, (,), =\} \tag{4.1}$$

  in tal caso i simboli ")", "(", "=" li dobbiamo vedere solo come simboli appunto. Non è importante in questo caso il loro significato.

- Abbiamo parlato poi dell'**insieme decidibile** $W \subseteq S^*$: come abbiamo sempre detto nella parte di linguaggi formali (definizione 1.1.2), possiamo dire che l'insieme S^*, dato S alfabeto, è l'insieme infinito di tutte le stringhe di lunghezza finita con caratteri in S (nel caso del sistema $\mathcal{CL}$ "k()sk" è una stringa e quindi appartiene a S^*). Per la definizione, però, ci interessa sapere anche cosa sia effettivamente l'insieme W, contenuto in S^*, **decidibile**. Soffermandoci su quest'ultima parola, vogliamo dire che tale insieme presenta un *procedimento effettivo* che mi permette di *decidere* se un elemento **appartiene** o **non appartiene** all'insieme W. Nell'esempio del sistema formale $\mathcal{CL}$ abbiamo che l'insieme W è così formato: $W = \{P = Q |\ P, Q \in \tau\}$:

  1. $k \in \tau, s \in \tau$

[1] Vedremo di spiegare il significato delle affermazioni o cosa significhi affermazione **corretta** in seguito. Adesso cerchiamo di capire la sintassi del sistema formale, cercando di non mescolare le 2 cose.

2. Se $P, Q \in \tau$ allora $(PQ) \in \tau$
3. Nient'altro è un termine

Analizziamo quanto scritto. Nella definizione dell'insieme W, vi è scritto che W è l'insieme delle fbf. Infatti se consideriamo l'insieme S^*, non tutte sono fbf, ma solo alcune. Quindi tale insieme ci specifica cosa possiamo considerare come fbf o meno, ecco quindi che ritorna la definizione di insieme *decidibile*. In questo senso la definizione dell'insieme W, ci indica quali stringhe di S^* sono delle fbf. Inoltre diamo una **definizione induttiva**, ovvero attraverso passi induttivi, di come costruire gli elementi di τ, dove τ rappresenta l'**insieme dei termini**, poichè abbiamo specificato nella definizione dell'insieme W che $P, Q \in \tau$, ovvero P e Q sono termini, e ogni fbf sarà costruita con un uguaglianza di mezzo, del tipo $P = Q$. Tale definizione induttiva di come costruire τ è data dai 3 punti che ci dicono che:

1. Il simbolo k e il simbolo s appartengono all'insieme dei termini τ
2. La seconda clausola ci dice che $P, Q \in \tau$, allora la *concatenazione* con le parentesi $(PQ) \in \tau$. Stiamo quindi descrivendo la sintassi per costruire τ, e quindi come costruire tutte le stringhe di τ (es: $\{k, s, (ks), (k(ks)), (kk)...\}$). In tal caso quindi possiamo immaginare come se P, Q fossero delle variabili che utilizziamo nei nostri programmi, che contengono le stringhe costruite dall'alfabeto S, e possono assumere diversi valori o uguali.
3. La terza clausola è quella fondamentale per la definizione. Ci dice infatti che nient'altro è un termine. Se pensiamo che la definizione è un metodo che ci permette di identificare **univocamente** un oggetto, non specificando questa clausola avrei la definizione di un'infinità di insiemi, e non solo dell'insieme che stiamo analizzando. Nelle definizioni, infatti, questa clausola è quasi sempre omessa, ma in realtà vi è sempre, altrimenti la definizione stessa non avrebbe senso.

- Come terzo punto della definizione vi è un insieme $Ax \subseteq W$, detto **insieme degli assiomi**. Prima di capire cosa sia l'insieme degli assiomi, possiamo fare una considerazione iniziale: in ogni ragionamento, soprattutto in quelli logici-matematici, le nostre asserzioni devono essere giustificate. Nel sistema formale posso decidere appunto di avere delle asserzioni o fbf che posso inserire **senza giustificazione**. Tali fbf sono gli **assiomi**. Tornando al nostro esempio del sistema formale $\mathcal{CL}$, per ogni $P, Q, R \in \tau$ abbiamo i seguenti *schemi di assioma* [2]:
  - $((kP)Q) = P$ (Axk)
  - $P = P$ (assioma di riflessività)
  - $(((sP)Q)R)$ (Axs)

[2] Per schemi di assioma intendiamo un modo per descrivere un numero eventualmente infinito di assiomi tramite un'unica espressione

– Nient'altro è assioma

Nel sistema $\mathcal{CL}$, quindi, queste fbf sono assiomi. Ovvero non hanno bisogno di alcuna giustificazione. Inoltre ogni assioma viene chiamato con un nome che è specificato nelle parentesi (Axk, Axs, assioma di riflessività). Se ci facciamo caso, inoltre, abbiamo la seguente caratteristica riguardante gli insiemi che abbiamo appena descritto, ovvero che:

$$Ax \subseteq W \subseteq S^* \tag{4.2}$$

Stiamo quindi dicendo che l'insieme W delle fbf è un sottoinsieme dell'insieme S^* di stringhe di lunghezza finita. Possiamo affermare quindi che **una fbf è una stringa**. Inoltre abbiamo che l'insieme Ax degli assiomi è un sottoinsieme dell'insieme W delle fbf. Possiamo affermare quindi che **un'assioma è una fbf**. Per la transitività quindi **gli assiomi sono stringhe di lunghezza finita**, ma non si può dire il viceversa.

- Al quarto punto della definizione abbiamo l'**insieme finito di regole finitarie**, dette anche **regole di inferenza**, $\mathcal{R} = \{R_i\}_{i \in I}$. Di questo punto dobbiamo capire cosa indichi il simbolo i, e il significato della scrittura $R_i \subseteq W^{n_i}$. Analizziamo tale punto a partire dal discorso fatto in precedenza sugli assiomi. Abbiamo compreso che gli assiomi sono asserzioni che non hanno bisogno di una giustificazione. Nel resto dei casi, quindi, devo poter giustificare le mie asserzioni attraverso determinate regole *costruite in base ad asserzioni fatte in precedenza* (es: "se piove allora mi porto l'ombrello", giustifico il fatto che "porto l'ombrello" dall'asserzione "piove"). Nell'esempio, quindi, possiamo capire la struttura di una regola: ovvero vi sono alcune **premesse della regola** ("piove"), che sono una serie di fbf, dal quale traiamo una **conclusione** ("mi porto l'ombrello"). Graficamente, inoltre, la regola può essere rappresentata in questo modo:

$$\frac{\alpha_1 \quad \alpha_2 ... \quad \alpha_n}{\beta} \tag{4.3}$$

dove $(\alpha_1, \alpha_2, ..., \alpha_n)$ sono le premesse della regola, ovvero fbf che per essere affermate o devono essere assiomi o devono essere giustificate tramite regole. La linea al centro possiamo vederla come la locuzione "allora" (es: se $\alpha_1, \alpha_2 ...$ allora β); e β è la conclusione della regola. Non abbiamo però ancora dato una *definizione precisa* di regola, ma ne abbiamo dato una rappresentazione grafica. Tornando quindi alla definizione di partenza, avevamo detto che tutte le regole formano un **insieme** (insieme delle regole finitarie). In realtà anche la regola stessa è un insieme; in particolare è una **relazione**, quindi un **sottoinsieme del prodotto cartesiano**. Ecco perchè $R_i \subseteq W^{n_i}$ e anche perchè $n_i \geq 2$ (non può esistere un sottoinsieme del prodotto cartesiano con $n_i < 2$). Parlando quindi di sottoinsiemi del prodotto cartesiano, parliamo quindi di **tuple** (es: se $R_i \subseteq W^2$ allora sarà una coppia di fbf; se $R_i \subseteq W^3$ allora sarà una tripla di fbf ecc.). Per quanto riguarda l'insieme I, esso è l'insieme di indici e, con la scrittura $\mathcal{R} = \{R_i\}_{i \in I}$, intendiamo dire che possiamo prendere qualsiasi i appartenente agli insieme di indici, dove un

indice non deve essere per forza numerico, ma può essere qualsiasi simbolo. Tornando al nostro esempio del sistema $\mathcal{CL}$, abbiamo $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$, ovvero l'insieme di regole finitarie di $\mathcal{CL}$ è formato da 4 regole, dove:

- $R_1 = \{P = Q, Q = P |\ P, Q \in \tau\} \subseteq W^2$. La prima regola viene denominata con la scrittura $(REFL)$ ed è una coppia e afferma quindi che se $P = Q$ allora $Q = P$ e può essere graficamente rappresentata come $\frac{P=Q}{Q=P}$ e, in questo caso, la premessa è $P = Q$ e la conclusione è $Q = P$.
- $R_2 = \{P = Q, Q = R, P = R |\ P, Q, R \in \tau\} \subseteq W^3$. Si tratta dunque di una *tripla*, ed è denominata come $(TRANS)$, ed è simile alla proprietà transitiva di qui sentiamo parlare nelle relazioni di equivalenza, e dice che se $P = Q$ e $Q = R$ allora $P = R$. Graficamente è rappresentata come $\frac{P=Q \quad Q=R}{P=R}$ e presenta quindi 2 premesse e una conclusione.
- Ecc.

Poichè si tratta di un esempio non elenchiamo tutte le regole per motivi di semplicità.

4.3 Derivazioni

Una volta che abbiamo definito il *sistema formale*, vediamo il concetto di **derivazione** o *dimostrazione* o *deduzione*. Enunciamo quindi la definizione, per poi cercare di dare una spiegazione:

Definizione 4.3.1. *Dato un insieme di fbf nel sistema formale $\mathcal{D}$, una $\mathcal{D}-derivazione$ (dimostrazione) a partire dall'insieme è una* ***successione finita di fbf*** *$\alpha_1, ..., \alpha_n$ di $\mathcal{D}$, tale che, per ogni $i = 1, ..., n$ si abbia:*

- $\alpha_i \in Ax$ *oppure*
- $(\alpha_{h_1}, ..., \alpha_{h_{n_j}}) \in R_j$ *per qualche $j \in I$, e $h_1, ..., h_{n_{j-1}} < i$*

Essenzialmente dalla definizione vogliamo dire che una **derivazione in un sistema formale**, che chiamiamo genericamente *sistema formale* $\mathcal{D}$, non è nient'altro che una *sequenza o lista di fbf* che inseriamo o come **assiomi** ($\alpha_i \in Ax$), oppure a partire **regole** ($(\alpha_{h_1}, ..., \alpha_{h_{n_j}}) \in R_j$) *giustificate in base ad asserzioni (fbf) fatte in precedenza* nel quale l'ultima formula $\alpha_i = \alpha h_{n_j}$ è la conclusione della regola, ovvero ciò che volevamo affermare.

4.3.1 Formule Derivabili

Dal concetto di *derivazione* possiamo introdurre il concetto di **formule derivabili**. Per introdurre tale concetto, però, facciamo prima una breve considerazione sulla definizione di *ragionamento* ***ipotetico***. Infatti, posso decidere di prendere un sottoinsieme $M \subseteq W$, che sarà il nostro insieme di fbf che costituiscono l'insieme di **ipotesi** del nostro ragionamento. Da qui, quindi, diamo la definizione di formula derivabile:

Definizione 4.3.2. *Una **formula*** [3] α *è **derivabile** nel sistema formale $\mathcal{D}$ a partire da un insieme di ipotesi M se è solo se esiste una $\mathcal{D} - derivazione$ a partire da M la cui ultima fbf è α. Scriveremo in questo caso che $M \vdash_{\mathcal{D}} \alpha$ e leggeremo: M **deriva** α nel sistema formale $\mathcal{D}$. Se M è vuoto scriveremo $\vdash_{\mathcal{D}} \alpha$ e leggeremo α è un **teorema** in $\mathcal{D}$*

In questa definizione, molto semplicemente, stiamo dicendo come leggere la notazione di una *formula derivabile* e, inoltre, che dato un insieme di ipotesi M, se esiste una derivazione nel sistema formale $\mathcal{D}$ che ha come **conclusione**, ovvero l'ultima formula della deduzione, α allora si scriverà $M \vdash_{\mathcal{D}} \alpha$ e si leggerà che M deriva α in $\mathcal{D}$, ovvero possiamo dire che α è derivabile (o dimostrabile) in $\mathcal{D}$ a partire da un insieme di ipotesi. Inoltre, la definizione ci dice che se l'insieme delle ipotesi $M = \varnothing$, allora diremo che α è un **teorema** in D, e, molto banalmente, potrò dedurlo in qualsiasi caso. Magari però, la sola definizione può non chiarirci completamente le idee. Vediamo, quindi, un esempio di deduzione, utilizzando come sistema di riferimento il nostro amato $\mathcal{CL}$.

Deduzione in CL

Vediamo un **esempio** di una deduzione in $\mathcal{CL}$, ed analizziamo i vari passi: dimostriamo che $\vdash_{\mathcal{CL}} (((sk)k)k) = k$, che è un *teorema* per quanto visto nella definizione 4.3.2:

1. $(((sk)k)k) = ((kk)(kk)) \qquad Axs$
2. $((kk)(kk)) = k \qquad Axk \; con \; P \equiv k \; e \; Q \equiv (kk)$
3. $(((sk)k)k) = k \qquad TRANS(1,2)$

Di questa, deduzione, possiamo dunque dire che ha 3 passi, ovvero *lunghezza* 3, e che si sviluppa nel seguente modo:

1. Nel primo passo affermiamo che $(((sk)k)k) = ((kk)(kk))$ giustificando tale affermazione con l'**assioma** Axs, che, dice che $(((sP)Q)R) = ((PR)(QR))$. In questo caso se assumiamo come $P, Q, R \equiv k$ allora possiamo affermare questo primo passo della deduzione

2. Nel secondo passo affermiamo che $((kk)(kk)) = k$ giustificando tale affermazione con l'**assioma** Axk che dice che $((kP)Q) = P$, e, come indicato nel passo stesso della deduzione, utilizzando la notazione $P \equiv k$ e $Q \equiv (kk)$.

3. La conclusione della deduzione è giustificata dalla **regola** $TRANS$, che affermava che $\frac{P=Q \quad Q=R}{P=R}$ ed ha come *premesse*, rispettivamente, la formula del passo 1 e la formula del passo 2, ovvero se pensiamo $P \equiv (((sk)k)k)$, $Q \equiv ((kk)(kk))$ ed $R \equiv k$, e che se $P = Q$ e $Q = R$ allora $P = R$ (seguendo la regola), si può affermare come conclusione della regola che $(((sk)k)k) = k$. Abbiamo quindi dimostrato tale formula è *derivabile* in $\mathcal{CL}$ e, inoltre, poichè si tratta di un *teorema*, può essere sempre derivabile.

[3] Per formula si intende fbf

4.3.2 Regole Derivabili e Ammissibili

Iniziamo, come al solito, dando la definizione di **regola derivabile**, e poi cerchiamo di spiegare quanto scritto nella definizione:

Definizione 4.3.3. *Sia $\mathcal{R}$ l'insieme di regole finitarie di un sistema formale $\mathcal{D}$. Una regola $R : \frac{\alpha_1 ... \alpha_k}{\alpha_{k+1}}$ si dice **derivabile** in $\mathcal{D}$ se e solo se per tutte le fbf $\alpha_1, ..., \alpha_k$ che soddisfano R si ha: $\alpha_1, ...\alpha_k \vdash_{\mathcal{D}} \alpha_{k+1}$*

Come al solito la definizione può sembrarci ostica poichè viene scritta in maniera formale, ma in realtà il concetto che ci sta dietro non è complesso. Per capire bene la definizione facciamo un esempio: supponiamo di avere il sistema formale $\mathcal{D} + R$, ovvero al quale abbiamo aggiunto la regola derivabile $R : \frac{\alpha_1 ... \alpha_k}{\alpha_{k+1}}$. Supponiamo che, in una derivazione da tale sistema formale, nella lista di fbf abbiamo derivato $\alpha_1, ..., \alpha_k$. Se volessimo dimostrare α_{k+1} potremmo utilizzare come giustificazione la regola R, che aveva come premesse le $\alpha_1, ..., \alpha_k$ fbf. Per definizione di *regola derivabile*, però, potremmo anche pensare che tale **regola** è **eliminabile**. Infatti se sappiamo che $\alpha_1, ...\alpha_k \vdash_{\mathcal{D}} \alpha_{k+1}$, ovvero esiste una derivazione in $\mathcal{D}$, tale che preso l'insieme di fbf $\alpha_1, ..., \alpha_k$ derivate e giustificate da assiomi o regole del sistema $\mathcal{D}$, si riesce a dimostrare α_{k+1}, allora la regola è superflua, cioè potevo derivare α_{k+1} senza utilizzare la regola, poichè abbiamo supposto all'inizio di aver derivato le ipotesi che ci servivano. La cosa importante quindi da comprendere è che le regole derivabili sono quelle **regole di inferenza inutili**, ovvero che non aumentano il numero di affermazioni che posso derivare, di conseguenza si può dire che è *inutile per l'espressività* del sistema formale. Molto spesso si tende ad aggiungere tali regole per aumentare la facilità di dimostrazione, senza dover quindi ogni volta dimostrare α_{k+1} a partire dalle $\alpha_1, .., \alpha_k$. Quindi, tornando al nostro esempio del sistema $\mathcal{D} + R$, poichè R è derivabile, posso trasformare la derivazione in $\mathcal{D} + R$ in una derivazione in $\mathcal{D}$.
Vediamo adesso la definizione di **regola ammissibile**, che può sembrare molto simile alla definizione di regola derivabile, ma in realtà sono diverse:

Definizione 4.3.4. *Data una regola R di un sistema formale $\mathcal{D}$, essa è detta **ammissibile** (o eliminabile) in $\mathcal{D}$ se e solo se da una derivazione $\vdash_{\mathcal{D} \cup \{R\}} \alpha$ segue $\vdash_{\mathcal{D}} \alpha$, dove $\mathcal{D} \cup \{R\}$ denota il sistema formale ottenuto aggiungendo a $\mathcal{D}$ la regola R.*

Quindi la relazione che vi è tra regole derivabili e ammissibili è la seguente:

$$\text{Regola Derivabile} \Rightarrow \text{Regola Ammissibile} \tag{4.4}$$

Ovvero il fatto che una regola sia derivabile *implica* il fatto che sia ammissibile, poichè la quantità di teoremi dimostrabili è la stessa. Mentre vale che:

$$\text{Regola Ammissibile} \not\Rightarrow \text{Regola Derivabile} \tag{4.5}$$

Infatti se una regola è *ammissibile* non aumenta il numero di teoremi dimostrabili, e inoltre, non può essere derivata nel sistema formale, a differenza di una regola derivabile. Ovvero utilizzando le premesse $\{\alpha_1, ..., \alpha_k\}$ della regola R ammissibile non riusciamo a derivare la sua conclusione nel sistema formale.

4.4 Caratteristiche dei Sistemi Formali

Vediamo adesso una serie di **proposizioni** che hanno lo scopo di dirci caratteristiche dei sistemi formali. Inoltre, le dimostrazioni di queste proposizioni, non sono propriamente quelle che abbiamo visto nel sistema formale. Infatti fanno uso di un *ragionamento metalogico* che usa il sistema formale della matematica, per il quale le frasi della dimostrazione stessa sostituiscono in qualche modo le fbf di un sistema formale. Vediamo quindi la prima proposizione:

Proposizione 4.1. *Se $M \vdash_{\mathcal{D}} \alpha$ allora esiste $N \subseteq M$, con N finito, per il quale si ha $N \vdash_{\mathcal{D}} \alpha$.*

Dimostrazione. $M \vdash_{\mathcal{D}} \alpha$ se e solo se esiste una sequenza $\alpha_1, ..., \alpha_n \equiv \alpha$ [4] che è una derivazione. Però, solo un numero minore o uguale a n, quindi finito, delle α_i appartiene ad M, per cui basta prendere $N \equiv M \cap \{\alpha_1, ..., \alpha_n\}$ per ottenere la tesi. □

La dimostrazione, ci dice, quindi, che se esiste una derivazione nel sistema $\mathcal{D}$ che ha α come conclusione, e *se* tale derivazione ha ipotesi, si trovano in M ($M \vdash_{\mathcal{D}} \alpha$). Se, però, immaginassimo tale insieme come un contenitore molto grande, dove oltre alla sequenza $\alpha_1, ..., \alpha_n$ di fbf ne abbiamo moltissime altre, allora banalmente potrei dire che esiste un contenitore più piccolo, ovvero l'insieme N, finito, tale che N conterrà la sequenza di $\alpha_1, ..., \alpha_n$ che ci servono per derivare α. Quindi esisterà N tale che $N \vdash_{\mathcal{D}} \alpha$ con $N \subseteq M$.

Proposizione 4.2 (Proprietà di Eliminazione dei Lemmi)**.** *Se $M \vdash_{\mathcal{D}} \alpha_1, ..., M \vdash_{\mathcal{D}} \alpha_n$ e $\{\alpha_1, ..., \alpha_n\} \vdash_{\mathcal{D}} \beta$, allora si avrà $M \vdash_{\mathcal{D}} \beta$.*

Dimostrazione. Per ipotesi so che esiste una $\mathcal{D} - derivazione\ \beta_1, ..., \beta_k, \beta$ di β a partire da $\{\alpha_1, ..., \alpha_n\}$. Se per qualche i, j si ha che $\beta_i = \alpha_j$ sostituisco β_i con la $\mathcal{D} - derivazione\ M \vdash \alpha_j$. Così facendo ottengo una $\mathcal{D} - derivazione$ con ipotesi solo in M. □

Il focus principale di questa proposizione consiste nel concetto di **scomporre il ragionamento** in piccoli ragionamenti (lemmi). Ovvero se ho che, nel sistema formale $\mathcal{D}$, esiste una derivazione che ha come insieme di ipotesi $\{\alpha_1, ..., \alpha_n\}$ che hanno come conclusione β, e posso derivare tali ipotesi tramite i vari *lemmi* $M \vdash_{\mathcal{D}} \alpha_1, ..., M \vdash_{\mathcal{D}} \alpha_n$, allora $M \vdash_{\mathcal{D}} \beta$. La dimostrazione, in questo caso, ci dice la seguente cosa: se supponiamo che abbiamo un α_4, che fa parte delle α_i fbf che dobbiamo inserire nella lista di fbf per arrivare alla conclusione β, e tale α_4 quindi è un'ipotesi, allora possiamo sostituire la derivazione che da M conclude α_4 nella lista mantenendo la proprietà di essere una derivazione corretta. Se iteriamo questo processo per tutte le α_n, allora potremo dire che β è una conclusione di M. Tale proposizione possiamo dire che *generalizza il* ***passo di computazione***, che vedremo in seguito cosa significa, e ci dice inoltre che i vari **lemmi**, ovvero piccoli teoremi atti alla dimostrazione di un teorema più grande in teoria **non sono necessari**, ma in realtà facilitano la dimostrazione del teorema stesso.

[4] In questo caso si intende che l'ultima formula della deduzione α_n corrisponde ad α

4.4.1 Consistenza e Inconsistenza

Introduciamo adesso una caratteristica molto importante di un sistema formale, che è quella di **consistenza** o **inconsistenza**:

Definizione 4.4.1. *Un sistema formale $\mathcal{D}$ è detto **consistente** se e solo se esiste una fbf α di $\mathcal{D}$ tale che $\nvdash_{\mathcal{D}} \alpha$* [5]*; se $\mathcal{D}$ non è consistente è detto **inconsistente**.*

Quello che essenzialmente ci dice questa definizione è che possiamo dire che un sistema formale è *inconsistente* quando affermo un *teorema*, ovvero una derivazione senza ipotesi, da qualsiasi fbf. E' consistente se almeno una fbf non è derivabile. Tale motivazione è ragionevole: se pensiamo a un sistema formale inconsistente, poichè posso affermare un teorema da una qualsiasi fbf, allora posso dire tutto e il contrario di tutto. Affinchè il sistema sia consistente quindi devo vedere se esiste almeno una fbf non derivabile.

4.5 Insieme delle Conseguenze, Teoria e Teoria Pura

Vediamo adesso 3 concetti molto importanti all'interno dei sistemi formali, ovvero cos'è l'**insieme delle conseguenza**, cos'è una **teoria** e cos'è una **teoria pura**.

4.5.1 Insieme delle Conseguenze

Iniziamo dando la definizione di **insieme delle conseguenze**, passando sequenzialmente alle altre 2 definizioni:

Definizione 4.5.1. *Sia Γ un insieme finito di fbf di un sistema formale $\mathcal{D}$: si dice **insieme delle conseguenze** di Γ l'insieme così costituito:*

$$Con_{\mathcal{D}}(\Gamma) = \{\alpha \in W : \ \Gamma \vdash_{\mathcal{D}} \alpha\} \tag{4.6}$$

Per capire bene cosa sia l'insieme delle conseguenze, immaginiamo come se l'insieme Γ di fbf sia l'*insieme delle nostre conoscenze attuali*. L'insieme delle conseguenze sono *tutte le conoscenze che posso ottenere ampliando i miei ragionamenti* durante il corso degli studi, ovvero tutte le α che posso derivare a partire da Γ. Quindi, le cose fondamentali che possiamo attenzionare sono che $\mathbf{\Gamma \subseteq Con_{\mathcal{D}}(\Gamma)}$, e che, l'**insieme dei teoremi** è $Con_{\mathcal{D}}(\varnothing)$. Tornando inoltre alla definizione di consistenza, possiamo dire che:

- Se $Con_{\mathcal{D}}(\Gamma) \neq W$, allora Γ **è consistente**. Quindi, affinchè Γ sia consistente, deve esistere almeno una $\alpha \in W$ tale che $\Gamma \nvdash_{\mathcal{D}} \alpha$.
- Se $Con_{\mathcal{D}}(\Gamma) = W$, allora Γ **è inconsistente**.

[5] Il simbolo $\nvdash$ indica la negazione del simbolo $\vdash$, ovvero non esiste una derivazione in $\mathcal{D}$ che abbia α come conclusione

4.5.2 Teoria e Teoria Pura

Una diretta *conseguenza* dell'insieme delle conseguenze sono i concetti di **teoria** e **teoria pura**. Vediamone quindi una definizione:

Definizione 4.5.2 (Teoria)**.** *Un insieme di fbf Γ di un sistema formale $\mathcal{D}$ è detto **teoria** in D se e solo se Γ è chiuso rispetto alla relazione $\vdash_{\mathcal{D}}$ (ovvero se e solo se $Con_{\mathcal{D}}(\Gamma) = \Gamma$, ovvero ancora se e solo se da $\Gamma \vdash_{\mathcal{D}} \alpha$ segue $\alpha \in \Gamma$).*

Se facciamo riferimento all'esempio che avevamo fatto per l'insieme delle conseguenze, in questo caso, la **teoria** è l'insieme di conoscenze che attraverso il sistema formale non posso estendere in alcun modo. Ciò significa che a qualsiasi conclusione io possa arrivare dall'insieme di ipotesi, otterrò una fbf appartenente all'insieme di ipotesi stesso. Tradotto, ciò significa che $Con_{\mathcal{D}}(\Gamma) = \Gamma$, e la definizione ci dice che Γ è chiuso rispetto alla relazione $\vdash_{\mathcal{D}}$. Tale simbolo si chiama **conseguenza logica** ed è la descrizione di una **relazione di derivabilità** tra un insieme di ipotesi e la conclusione. Se un insieme è chiuso rispetto alla relazione significa che se prendiamo la relazione $R \subseteq A \times A$, e prendiamo l'insieme $B \subseteq A$, allora se un elemento $b \in B$ *è in relazione R* con b', allora anche $b' \in B$. Nel caso di Γ, se $\Gamma \vdash_{\mathcal{D}} \alpha$ allora $\alpha \in \Gamma$.

Definizione 4.5.3 (Teoria Pura)**.** *La **teoria pura** di un sistema formale $\mathcal{D}$ è l'insieme $Con_{\mathcal{D}}(\varnothing)$.*

Molto semplicemente, quindi, la teoria pura è l'**insieme dei teoremi** del sistema formale $\mathcal{D}$, ovvero l'insieme delle conseguenze avente $\Gamma = \varnothing$. Le *osservazioni* finali che possiamo fare, che possono sembrare banali, ma in realtà sono importanti sono le seguenti:

- Una **teoria pura** è una **teoria**
- Può capitare che si abbia $\Gamma \neq \Gamma'$ e ciò nonostante che $Con_{\mathcal{D}}(\Gamma) = Con_{\mathcal{D}}(\Gamma')$.

4.6 Cenni sulla Semantica di un sistema formale

Fino ad adesso abbiamo parlato della *sintassi di un sistema formale*, ovvero come scrivere essenzialmente i simboli di un sistema formale e cosa rappresentano. In realtà c'è anche una parte di **semantica**, ovvero il *significato* che possiamo dare a un sistema formale, che però non ha una definizione precisa come per la parte sintattica. Possiamo dire, in maniera grossolana, che generalmente si basa sul **concetto di validità**, che si dirama poi in 2 proprietà:

- Proprietà di **correttezza**: un sistema formale è corretto se **tutto ciò che è derivabile è valido o vero** sulla base della semantica che abbiamo dato al sistema formale. Ovvero tutte le conclusioni a cui arrivo sono vere rispetto al mio concetto di semantica. Tale proprietà è quella che ritroviamo più spesso, ed è quella fondamentale per un sistema formale.
- Proprietà di **completezza**: un sistema formale è completo se **tutto ciò che è valido o vero è derivabile**. Tale proprietà è più rara da trovare nella

semantica di un sistema formale rispetto alla correttezza, ma è altrettanto importante.

Capitolo 5

La Logica Proposizionale

Introduciamo in tale capitolo la **logica proposizionale**, detta anche *calcolo proposizionale*. Probabilmente vi sarà capitato di imbattervi in questo tipo di logica in moltissimi casi, soprattutto nel costruire dei ragionamenti sensati. Prima di affrontare, però, la parte sintattica analizzando come è costruito il sistema formale della logica proposizionale, diamo un breve accenno alla **semantica**. Riprendendo quanto detto alla fine del capitolo 2 (sezione 4.6), dobbiamo definire il concetto di *validità* nella logica proposizionale, che non è altro che descrivere in modo formale il concetto di **verità in ogni mondo possibile**. Quindi ciò che si fa è studiare fbf costruite come **formule elementari**, ovvero che possono assumere valore di verità vero o falso, unite da **connettivi logici** e se ne studia la validità.

5.1 Il Sistema Formale P_0

Abbiamo visto un breve cenno sulla semantica della logica proposizionale che riprenderemo nelle sezioni successivi. Concentriamoci adesso sulla **parte sintattica**, ovvero cerchiamo di capire come è costruito il sistema formale P_0 della logica proposizionale.

Definizione 5.1.1. *Il **sistema formale** P_0 è il seguente sistema formale:*

- ***S**: è formato dall'unione fra un insieme numerabile di variabili proposizionali $p, q, r, s...$, l'insieme dei connettivi '$\rightarrow$' e '$\neg$' e l'insieme dei 2 simboli ausiliari '(' e ')'.*
- ***W**: l'insieme delle fbf è così formato:*
  1. *ogni variabile proposizionale è una fbf*
  2. *se α e β sono fbf, allora anche $(\alpha \rightarrow \beta)$ e $(\neg\alpha)$ sono fbf.*
  3. *nient'altro è una fbf*
- ***Ax**: i seguenti sono schemi di assioma:*

  - $\alpha \to (\beta \to \alpha)$ (Ak)
  - $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\beta \to \gamma))$. (As)
  - $(\neg\beta \to \neg\alpha) \to ((\neg\beta \to \alpha) \to \beta)$ $(\neg A)$
- $\mathcal{R} = \{\mathbf{MP}\}$ *dove* MP *(modus ponens) è la regola* $\frac{\alpha \quad \alpha \to \beta}{\beta}$

Vediamo quindi di analizzare tale definizione, capendo quindi come è formato il sistema formale P_0 dal punto di vista sintattico:

- L'insieme S è l'insieme numerabile che rappresenta l'**alfabeto**, ed è composto da un insieme di lettere $p, q, r, s...$ che vengono chiamate *variabili proposizionali*, i simboli '$\to$' e '$\neg$', che sono i connettivi logici, che vediamo da questo punto di vista come simboli qualsiasi, e i simboli '(', ')'.

- L'insieme delle **fbf**, come visto nella definizione di sistema formale (definizione 4.2.1), è *definito* ***induttivamente***, ovvero presenta un caso base e un passo induttivo che definiscono il *programma per costruire qualsiasi fbf*. Essenzialmente tale programma ci dice che ogni variabile proposizionale $p, q, r, s, ...$ sono fbf (caso base). Prese α, β fbf [1], allora anche $(\alpha \to \beta)$ e $(\neg\alpha)$ sono fbf (passo induttivo) e nient'altro è una fbf. Quindi una formula del tipo $p \to q \to s$ non è una fbf del sistema P_0, infatti non rispecchia il fatto di avere le parentesi agli estremi come descritto nella definizione induttiva. Se la trasformassimo in $(p \to (q \to s))$, questa è una fbf del sistema P_0.

- L'insieme degli **assiomi** è composto dai 3 **schemi di assioma** elencati nella definizione. La cosa importante da sottolineare in questo caso è che tali schemi possono rappresentare un numero eventualmente infinito di assiomi, dove α, β, γ sono qualsiasi fbf costruita attraverso la definizione induttiva. Quindi ad esempio la formula: $(p \to ((\neg t) \to p))$ è un assioma Ak, considerando $p \equiv \alpha$ e $\neg t \equiv \beta$.

- L'insieme di regole, nel caso del sistema P_0, è formato da una singola regola. Tale regola viene anche chiamata *modus ponens*, ed è una delle regole che più utilizziamo all'interno dei nostri ragionamenti. Essa è formata dalla tripla $(\alpha, \alpha \to \beta, \beta)$ e dice che se nella nostra derivazione abbiamo affermato α, e abbiamo affermato anche $\alpha \to \beta$, allora possiamo concludere β.

5.1.1 I connettivi logici AND e OR

Qualcuno, nel leggere la definizione del sistema formale P_0, magari si può essere chiesto *perche mancano i connettivi AND e OR*? La risposta è che **non sono strettamente indispensabili**. Infatti le formule $\alpha \vee \beta, \alpha \wedge \beta$ *possono essere derivate* nel sistema formale P_0 con i connettivi logici da noi definiti, quindi sono **uguali dal punto di vista sintattico** della *derivabilità*, e sono **uguali dal punto di vista semantico**. In particolare vale la seguente *definizione esplicita*:

[1] da non confondere con le variabili proposizionali poichè si tratta infatti di *metavariabili* ovvero variabili che possono contenere una qualsiasi fbf

- $\alpha \vee \beta \equiv ((\neg\alpha) \to \beta)$
- $\alpha \wedge \beta \equiv \neg((\neg\alpha) \vee (\neg\beta))$, dove il simbolo '$\vee$' può essere eliminato derivando la formula.

La domanda che può sorgere spontaneamente è: *perchè non utilizziamo tutti i simboli a nostra disposizione*? Ciò che vogliamo fare è di non rendere il sistema formale troppo complesso dal punto di vista sintattico, creando quindi moltissime regole, fbf, schemi di assioma, che possono essere rappresentati anche con i simboli essenziali. In particolare potremmo sostituire l'*implicazione*, ovvero il simbolo freccia, con l'and e l'or, ma vedremo che nel nostro obbiettivo di dimostrare la corrispondenza tra derivazioni e programmi vedremo come sarà più utile l'implicazione.

5.2 Il Teorema di Deduzione

Enunciamo adesso un teorema fondamentale della logica proposizionale che è il **teorema di deduzione di Herbrand**.

Teorema 5.2.1. *Dato un insieme di fbf Γ in P_0 si ha:*

$$\Gamma \vdash_{P_0} \alpha \to \beta \iff \Gamma, \alpha \vdash_{P_0} \beta \tag{5.1}$$

In questo caso abbiamo una doppia implicazione, che ha il significato di '*se e solo se*', ovvero o sono entrambe le affermazioni vere o entrambe false. Tale teorema è fondamentale poichè da un interpretazione al simbolo '$\to$' e '$\vdash$': infatti possiamo interpretare $\Gamma \vdash \alpha \to \beta$ come: **se** ho ipotesi Γ e derivo $\alpha \to \beta$, **allora** posso concludere β se tra le ipotesi abbiamo l'insieme $\Gamma \cup \{\alpha\}$ e trattandosi di una doppia implicazione vale anche il viceversa, ovvero se ho come insieme di ipotesi $\Gamma \cup \{\alpha\}$ e derivo β, allora posso affermare anche, a partire da ipotesi Γ, posso concludere $\alpha \to \beta$. Ciò ci permette di rappresentare all'interno di un sistema formale il **concetto di derivabilità**, rifacendoci all'implicazione. Se pensiamo, infatti, al concetto di implicazione *se...allora*, una derivazione non rappresenta che una serie di implicazioni per il quale possiamo concludere un qualcosa *se* questo qualcosa viene giustificato. Tale teorema quindi ci permette di affrontare le derivazioni in maniera molto più semplice in P_0. Per esempio dimostriamo $\vdash_{P_0} \alpha \to \alpha$: utilizziamo il teorema di deduzione e deriviamo $\{\alpha\} \vdash_{P_0} \alpha$:

1. α Hyp

Poichè abbiamo derivato in P_0 $\{\alpha\} \vdash_{P_0} \alpha$, allora per il *teorema di deduzione* possiamo concludere anche $\vdash_{P_0} \alpha \to \alpha$. Se non avessimo utilizzato quindi tale teorema, avremmo fatto tale derivazione in molti più passi.

5.2.1 Cenni sull'induzione matematica e sull'induzione completa

Dimostreremo nella prossima sezione il teorema di deduzione (teorema 5.2.1) per **induzione**. Per dare un cenno possiamo dire che l'induzione non è nient'altro che

una **regola di inferenza** che utilizziamo molto spesso nelle derivazioni che facciamo in matematica riguardanti moltissime proprietà. E si descrive formalmente come, presa una proprietà $\phi(n)$ sui numeri naturali, allora:

$$\frac{\phi(0) \qquad \forall x > 0 \ \phi(x) \rightarrow \phi(x+1)}{\forall n \in \mathbb{N} \ \phi(n)}$$

(5.2)

Quindi essenzialmente se dimostriamo un **caso base** $\phi(0)$, e nel **passo induttivo**, assumendo come *ipotesi induttiva* $\phi(x)$, quindi assumendola per vera, dimostriamo $\phi(x+1)$ allora possiamo concludere che $\forall n \in \mathbb{N}$ vale $\phi(n)$ Si può dire inoltre che tale regola non è derivabile, infatti alcune dimostrazioni non potrebbero essere fatte senza induzione. A volte, però, l'induzione matematica risulta molto complicata da usare, per questo viene anche utilizzata la regola di inferenza dell'**induzione completa**, che formalmente è descritta come:

$$\frac{\phi(0). \qquad \forall y: \ (0 < y < x)\phi(y) \rightarrow \phi(x)}{\forall n \in \mathbb{N}.\phi(n)} \tag{5.3}$$

In questo caso abbiamo un *ipotesi induttiva molto più forte*. Infatti diciamo nel caso dell'induzione completa che se dimostro il caso base $\phi(0)$ e nel passo induttivo assumo come ipotesi induttiva un qualsiasi caso $\phi(y)$ tale che $0 < y < x$, ovvero y più corto di $\phi(x)$, e dimostro $\phi(x)$, allora posso concludere che $\phi(n)$ vale per tutti gli n naturali. L'induzione completa è molto utile nelle **dimostrazioni per induzione sulla lunghezza della dimostrazione**. Poichè, infatti, l'induzione si applica ai naturali ma noi ci serviamo di questa regola per le deduzioni, lo utilizziamo sulla lunghezza delle deduzioni. Presa D come derivazione possibile, possiamo affermare, in effetti, che $|D| = n$ con $n \in \mathbb{N}$, dove n è il numero di passi di dimostrazione, ergo la sua lunghezza, che è *quantificabile con i numeri naturali* e, di conseguenza, si presta all'applicazione della regola di induzione completa.

Esempio dell'applicazione dell'Induzione Completa

Vediamo quindi di applicare il principio o regola di induzione completa per una dimostrazione sulla lunghezza della derivazioni. In particolare vogliamo dimostrare una *versione più debole del teorema di correttezza* (sezione 5.4.2), ovvero:

$$\text{se } \vdash_{P_0} \alpha \text{ allora } \vDash_{P_0} \alpha \tag{5.4}$$

Tale affermazione può essere letta come: *se α è un teorema in P_0, allora α è anche una tautologia*. Vedremo in seguito il concetto di tautologia analizzando la semantica. Adesso soffermiamoci sulla dimostrazione: infatti il modo equivalente per scrivere la frase sopra è: *se α è derivabile con una deduzione di lunghezza $n \geq 1$ senza usare ipotesi, allora α è una tautologia*. Vedendola in questo modo, possiamo dimostrare questa affermazione per induzione sulla lunghezza delle derivazioni:

- **Caso Base** $\phi(1)$: dobbiamo dimostrare $\phi(1)$, infatti non esiste una derivazione di lunghezza 0. In particolare dobbiamo far vedere che α è derivabile senza ipotesi in P_0 utilizzando una derivazione di lunghezza 1, allora α è una tautologia. In questo caso la derivazione sarà della forma:

  $$1.\ \alpha \qquad (Ax)$$

  dove Ax rappresenta uno degli schemi di assioma di P_0. Infatti non può essere un ipotesi poichè vogliamo dimostrare che α è derivabile in P_0 senza ipotesi. Inoltre non può essere la conclusione della regola *modus ponens*, poichè non ci sono premesse che possano concludere α, poichè la derivazione è lunga 1. Quindi per $\phi(1)$ il teorema è dimostrato

- **Passo induttivo** $(\forall 1 \leq y < n.\phi(y)) \Rightarrow \phi(n)$: in questo caso la nostra *ipotesi induttiva* è che la deduzione vale per tutte le deduzioni più corte di n, ovvero per tutte le deduzioni di lunghezza y tale che $1 \leq y < n$. Dobbiamo far vedere, quindi, che se assumiamo l'ipotesi induttiva come vera allora vale $\phi(n)$. Ricordiamo che far vedere che vale $\phi(n)$ equivale a far vedere che se esiste una derivazione di lunghezza n di α allora α è una tautologia. Quello che ci aspettiamo dunque è una derivazione del tipo:

  $$\begin{array}{lll} 1. & \ldots & \\ \vdots & \vdots & \\ n. & \alpha & (X) \end{array} \tag{5.5}$$

  dove se $(X) = Ax$, ovvero è uno degli assiomi di P_0, abbiamo finito poichè come visto nel caso base gli assiomi sono tautologie. Inoltre (X) non può essere ipotesi poichè stiamo considerando una derivazione di α che non utilizza ipotesi. L'unica possibilità che ci rimane è che $(X) = (MP(i,j))$ per qualche $1 < i, j < n$. La derivazione in questo caso diventa del tipo:

  $$\begin{array}{lll} 1. & \ldots & \\ \vdots & \vdots & \\ i. & \delta & \\ \vdots & \vdots & \\ j. & \delta \to \alpha & \\ \vdots & \vdots & \\ n. & \alpha & (MP(i,j)) \end{array} \tag{5.6}$$

  Soffermandoci sulla derivazione lunga $i < n$, possiamo dire per ipotesi induttiva che poichè ϕ vale per tutte le derivazioni più corte di n, allora vale anche per i. Questo ci dice che poichè δ è la conclusione di una derivazione di lunghezza $i < n$, allora δ è una tautologia. Allo stesso modo, per ipotesi induttiva, possiamo dire che poichè $j < n$ allora possiamo affermare $\phi(j)$, ovvero per una derivazione di lunghezza j che conclude con $\delta \to \alpha$, allora

$\delta \to \alpha$ è una tautologia. A questo punto, facendo riferimento alla proposizione 5.2, che è una proprietà che ci dice che una tautologia può essere derivata solo da premesse che sono tautologie, ovvero ci assicura che se le premesse della regola MP sono tautologie, allora lo è anche la sua conclusione, allora se δ e $\delta \to \alpha$ sono tautologie, allora anche α è una tautologia.

5.2.2 Dimostrazione Del Teorema di Herbrand

Una volta compresa la regola di inferenza di *induzione completa*, che ci servirà per dimostrare il teorema di deduzione, possiamo procedere con la dimostrazione:

Dimostrazione del Teorema di Deduzione. Dobbiamo dimostrare una *doppia implicazione*, infatti il teorema dice che:

$$\Gamma \vdash_{P_0} \alpha \to \beta \Leftrightarrow \Gamma, \alpha \vdash_{P_0} \beta \tag{5.7}$$

Poichè, dunque, si tratta di una doppia implicazione, dimostriamo il teorema in 2 parti:

- **Condizione necessaria** ($\Rightarrow$): Dobbiamo dimostrare l'implicazione $\Gamma \vdash_{P_0} \alpha \to \beta \Rightarrow \Gamma, \alpha \vdash_{P_0} \beta$. Vediamo di costruire dunque una deduzione che abbia *ipotesi* in $\Gamma \cup \{\alpha\}$, e che concluda β, sapendo per ipotesi che $\Gamma \vdash_{P_0} \alpha \to \beta$:

$$\begin{array}{ccc} 1. & \alpha & (Hyp) \\ \vdots & \vdots & \\ k. & \alpha \to \beta & \text{Espansione di } \Gamma \vdash \alpha \to \beta \\ k+1. & \beta & MP(1,k) \end{array} \tag{5.8}$$

  Essenzialmente, quindi, abbiamo affermato α giustificandola come *ipotesi*, infatti l'insieme di ipotesi è $\Gamma \cup \{\alpha\}$. Alla posizione k, abbiamo affermato $\alpha \to \beta$: possiamo immaginare la giustificazione *espansione di* $\Gamma \vdash \alpha \to \beta$, come se avessimo inserito l'intera derivazione a partire da ipotesi in Γ che concludeva $\alpha \to \beta$, che sapevamo esistere per ipotesi. Di conseguenza abbiamo dimostrato che $\Gamma \vdash_{P_0} \alpha \to \beta \Rightarrow \Gamma, \alpha \vdash_{P_0} \beta$.

- **Condizione Sufficiente** ($\Leftarrow$): Dobbiamo dimostrare l'implicazione $\Gamma, \alpha \vdash \beta \Rightarrow \Gamma \vdash \alpha \to \beta$. Dimostriamo questa implicazione con la nostra bellissima regola di **induzione completa**. Vogliamo quindi dimostrare quest'implicazione per *induzione sulla lunghezza delle derivazioni*, in particolare dobbiamo dimostrare $\phi(n)$, che coincide in questo caso con la scrittura:

  Per ogni $n > 1$, se $\Gamma, \alpha \vdash \beta$ con una **derivazione lunga** n allora possiamo derivare $\Gamma \vdash \alpha \to \beta$

  Per dimostrare $\phi(n)$, allora dovremo costruire la nostra regola di induzione completa in questo modo:

$$\frac{\phi(1) \qquad (1 < y < n.\phi(y)) \Rightarrow \phi(n)}{\forall n \geq 1.\phi(x)} \tag{5.9}$$

Vediamo quindi di andare per gradi dimostrando le premesse della regola di induzione, ovvero il *caso base* e il *passo induttivo*:

- **Caso Base** ($\phi(1)$): In questo caso vogliamo far vedere che con una derivazione di lunghezza 1 di $\Gamma, \alpha \vdash \beta$, riusciamo a costruire una derivazione $\Gamma \vdash \alpha \to \beta$. Se $n = 1$ allora abbiamo che la derivazione sarà del tipo:

$$1. \quad \beta \qquad () \tag{5.10}$$

  Dobbiamo inserire dunque una giustificazione tra le parentesi per affermare β. Se abbiamo dunque che la lunghezza della derivazione è 1, la giustificazione per affermare β è che sia **un assioma oppure un ipotesi**, oppure che $\beta \equiv \alpha$, ovvero β *coincide* con α. Infatti non può essere la regola di MP poichè non vi sono le premesse che possano concludere β con una lunghezza di derivazione 1, come visto nella sezione precedente. Allora se β è un ipotesi oppure un assioma, la derivazione che possiamo costruire è la seguente:

$$\begin{array}{lcc} 1. & \beta \to (\alpha \to \beta) & (Ak) \\ 2. & \beta & Hyp \text{ o } Ax \\ 3. & \alpha \to \beta & MP(1,2) \end{array} \tag{5.11}$$

  Quindi $\alpha \to \beta$ è derivabile con deduzione di lunghezza 1 di $\Gamma, \alpha \vdash \beta$ utilizzando come giustificazione che sia un ipotesi o un assioma. Se invece come giustificazione abbiamo che $\beta \equiv \alpha$, allora possiamo utilizzare un teorema di P_0, che dice $\vdash_{P_0} \alpha \to \alpha$, il che implica la tesi.

- **Passo Induttivo** ($(1 < y < n.\phi(y)) \Rightarrow \phi(n)$): in questo caso assumiamo come **ipotesi induttiva** $\phi(y)$ che per tutte le *derivazioni più corte* di n di $\Gamma, \alpha \vdash \beta$ lunghe $y < n$, ovvero tutte le derivazioni che partendo da ipotesi $\Gamma \cup \alpha$ concludono β, allora assumiamo che si possa derivare $\Gamma \vdash \alpha \to \beta$. Vogliamo dimostrare che, se assumiamo per vera l'ipotesi induttiva $\phi(y)$, allora possiamo dimostrare $\phi(n)$. Proviamo quindi a farci un idea di come dovrà essere la deduzione che concluderà con β:

$$\begin{array}{lcc} 1. & \beta_1 & \dots \\ 2. & \beta_2 & \dots \\ \vdots & \vdots & \vdots \\ n. & \beta_n \equiv \beta & () \end{array} \tag{5.12}$$

  Vogliamo capire dunque quale giustificazione si può mettere tra le parentesi per poter affermare β all'n-esima affermazione. Abbiamo 2 casi:

  1. Se come giustificazione di β vi è un **ipotesi** o un **assioma**, possiamo rifarci al *caso base*.
  2. Se come giustificazione di β, utilizziamo la regola *modus ponens* come $MP(i, j)$, allora proviamo a costruire una derivazione utiliz-

zando la nostra ipotesi induttiva:

$$\begin{array}{lll} 1. & \beta_1 & \dots \\ 2. & \beta_2 & \dots \\ \vdots & \vdots & \vdots \\ i. & \beta_i \equiv \beta_j \to \beta_n & \\ \vdots & \vdots & \vdots \\ j. & \beta_j & \\ \vdots & \vdots & \vdots \\ n. & \beta_n \equiv \beta & (MP(i,j)) \end{array} \tag{5.13}$$

Concentriamoci sulla deduzione fino ad i, che è una deduzione che conclude $\beta_j \to \beta_n$ a partire da ipotesi in Γ, α, altrimenti non potremmo affermarla. Allo stesso modo la deduzione fino a j è una deduzione che a partire da ipotesi in Γ, α conclude β_j. Ma poichè notiamo che $i, j < n$ allora vale l'**ipotesi induttiva** per cui valgono $\phi(i), \phi(j)$, ovvero:

* $\phi(i)$: $\Gamma, \alpha \vdash \beta_j \to \beta_n \Rightarrow \Gamma \vdash \alpha \to (\beta_j \to \beta_n)$
* $\phi(j)$: $\Gamma, \alpha \vdash \beta_j \Rightarrow \Gamma \vdash \alpha \to \beta_j$

Sulla base di queste formule che abbiamo potuto affermare grazie all'ipotesi induttiva, si costruisce la deduzione finale per *modus ponens* che conclude $\alpha \to \beta$:

$$\begin{array}{lll} 1. & (\alpha \to (\beta_j \to \beta_n)) \to ((\alpha \to \beta_j) \to (\alpha \to \beta_n)) & (As) \\ \vdots & \vdots & \vdots \\ k. & \alpha \to (\beta_j \to \beta_n) & \text{espansione di } \Gamma \vdash \alpha \to (\beta_j \to \beta_n) \\ k+1. & (\alpha \to \beta_j) \to (\alpha \to \beta_n) & (MP(1,k)) \\ \vdots & \vdots & \vdots \\ h. & \alpha \to \beta_j & \text{espansione di } \Gamma \vdash \alpha \to \beta_j \\ h+1. & \alpha \to \beta_n \equiv \alpha \to \beta & MP(k+1,h) \end{array} \tag{5.14}$$

Quindi la scrittura "*espansione di*", significa che stiamo affermando che esiste la derivazione $\Gamma \vdash \alpha \to \beta_j$ ecc. supponendo vera l'ipotesi induttiva. Quindi dove ci sono i puntini verticali dobbiamo immaginarci di sostituire l'intera derivazione

Infine, quindi, poichè abbiamo dimostrato il **caso base** $\phi(0)$, e il **passo induttivo** $\phi(y) \Rightarrow \phi(n)$, allora possiamo concludere per la regola di inferenza dell'induzione completa che $\forall n > 1.\phi(n)$, ovvero *Per ogni* $n > 1$, se $\Gamma, \alpha \vdash \beta$ con una **derivazione lunga** n allora possiamo derivare $\Gamma \vdash \alpha \to \beta$. Quindi, ciò implica che abbiamo dimostrato la condizione sufficiente:

$$\Gamma, \alpha \vdash_{P_0} \beta \Rightarrow \Gamma \vdash_{P_0} \alpha \to \beta \tag{5.15}$$

□

5.2.3 Cenni sulla corrispondenza Dimostrazioni-Programmi

Abbiamo accennato a inizio di questa parte che il nostro intento era quello di far risaltare quella che è la **corrispondenza dimostrazioni-programmi**. Con la dimostrazione del *teorema di deduzione* abbiamo già visto un programma, anche senza rendercene conto. Prendiamo quindi la regola di induzione matematica e pensiamola come una *funzione*:

$$\frac{\phi(0). \qquad \forall y: \ (0 < y < x)\phi(y) \to \phi(x)}{\forall n \in \mathbb{N}.\phi(n)} \tag{5.16}$$

In questo caso la descrizione della funzione è tale che per ogni n passato come parametro, possiamo restituire $\phi(n)$ attraverso un semplice processo. Supponiamo quindi di voler sapere $\phi(7)$. Il programma si comporterebbe in modo tale che:

$$\begin{array}{c} \phi(0) \Rightarrow \phi(1) \\ \phi(1) \Rightarrow \phi(2) \\ \phi(2) \Rightarrow \phi(3) \\ \vdots \\ \phi(6) \Rightarrow \phi(7) \end{array} \tag{5.17}$$

Essenzialmente il programma basa il suo funzionamento sapendo calcolare il *caso base* $\phi(0)$, e sapendo calcolare $\phi(n+1)$ se so $\phi(n)$. Questo che abbiamo appena descritto non è altro che un meccanismo di **ricorsione**, ovvero la descrizione di un **processo computazionale**. Ciò vale anche per la dimostrazione del teorema di Herbrand, che data una dimostrazione $\Gamma, \alpha \vdash \beta$ restituisce una dimostrazione $\Gamma \vdash \alpha \to \beta$. Torneremo su questo argomento, una volta che avremo ancora più strumenti per affrontarlo.

5.3 Caratteristiche di P_0

Vediamo adesso alcune proposizioni e teoremi che caratterizzano P_0. Vediamo la prima proposizione, che ci servirà per spiegare un concetto molto importante:

Proposizione 5.1. $\alpha \to \beta \vdash_{P_0} \neg\beta \to \neg\alpha$

Per il *teorema di deduzione* possiamo dire che $\vdash_{P_0} (\alpha \to \beta) \to (\neg\beta \to \neg\alpha)$ è un *teorema* di P_0. Inoltre le 2 fbf della proposizione sono *equivalenti nella dimostrabilità*, ovvero se assumo una posso dimostrare l'altra e viceversa. Se notate bene, tale proposizione è quella su cui si basa il principio di **dimostrazione per assurdo**. Se, infatti, assumiamo α come ipotesi e β come tesi; se nego la tesi e dimostro l'implicazione $\neg\beta \to \neg\alpha$, è equivalente a dire che $\alpha \to \beta$. Tale proposizione potrebbe essere usata come *giustificazione* in altre dimostrazioni. In realtà noi sappiamo che sarebbe illecito usare una proposizione come giustificazione. Infatti in una *derivazione* possiamo usare solo *assiomi*, *conclusioni di regole di inferenza* le cui premesse erano state affermate in precedenza oppure *ipotesi*. In realtà per la proposizione 4.2, ovvero la *proprietà di eliminazione dei lemmi*, possiamo dire che

sostituendo le varie dimostrazioni della determinata proposizione alla fine arrivo alla conclusione che cercavo. Però è importante comprendere che anche **verificare la correttezza di una dimostrazione** è molto importante.

5.3.1 Consistenza e Contraddittorietà in P_0

Vediamo adesso un teorema e a seguire dei suoi corollari che trattano di consistenza e inconsistenza o contraddittorietà di P_0:

Teorema 5.3.1. *Un insieme Γ di fbf di P_0 si dice **contraddittorio** (equivalente a inconsistente, ovvero $Con_{P_0}(\Gamma) = W$) se e solo se esiste una fbf α di P_0 tale che $\Gamma \vdash_{P_0} \alpha$ e $\Gamma \vdash_{P_0} \neg\alpha$*

Dimostrazione. Poichè abbiamo enunciato un *se e solo se* dobbiamo dimostrare una doppia implicazione:

- ($\Rightarrow$): in questo caso la dimmostrazione è ovvia, infatti se per ipotesi abbiamo che $Con_{P_0}(\Gamma) = W$, allora possiamo dimostrare $\Gamma \vdash_{P_0} \beta$, **per ogni** $\beta \in W$, quindi possiamo derivare anche α e $\neg\alpha$
- ($\Leftarrow$): in questo caso abbiamo come ipotesi che $\Gamma \vdash_{P_0} \alpha$ e $\Gamma \vdash_{P_0} \neg\alpha$ e vogliamo dimostrare che $\Gamma \vdash_{P_0} \beta$:
  - Esiste una derivazione in P_0 tale che $\vdash_{P_0} \alpha \to (\neg\alpha \to \beta)$ è un teorema di P_0. Per il **teorema di deduzione** possiamo dire quindi che $\alpha, \neg\alpha \vdash_{P_0} \beta$.
  - Poichè per ipotesi sapevamo che $\Gamma \vdash_{P_0} \alpha$ e $\Gamma \vdash_{P_0} \neg\alpha$, allora per la proprietà 4.2 di eliminazione dei lemmi che se $\Gamma \vdash_{P_0} \alpha$ e $\Gamma \vdash_{P_0} \neg\alpha$ e $\alpha, \neg\alpha \vdash_{P_0} \beta$ allora $\Gamma \vdash_{P_0} \beta$ per ogni β, quindi Γ è **contraddittorio**.

□

Quindi abbiamo introdotto il concetto di *contraddittorietà*, che è equivalente a quello di inconsistenza, e abbiamo dimostrato che se in P_0 dimostriamo qualcosa (α) e il suo contrario ($\neg\alpha$), allora l'insieme di fbf è inconsistente in quanto possiamo dimostrare tutto e il contrario di tutto. Vediamo adesso un corollario di questo teorema che però prende come insieme di riferimento l'intero sistema P_0:

Corollario 5.3.1. *P_0 è **inconsistente**, o contraddittorio, se e solo se esiste una fbf α di P_0 tale che $\vdash_{P_0} \alpha$ e $\vdash_{P_0} \neg\alpha$*

In tal caso la dimostrazione è analoga al teorema 5.3.1, basta porre $\Gamma = \varnothing$. Equivalentemente possiamo dire, grazie alla proposizione 5.1, che $\neg\beta \to \neg\alpha$, ovvero *se*, presa qualsiasi fbf, lei e la sua negazione non sono teoremi di P_0, quindi non sono *tautologie* (le vedremo nella prossima sezione), *allora* P_0 è consistente.
Un altro teorema riguardante la consistenza molto importante è il seguente:

Teorema 5.3.2. *Sia Γ un insieme **consistente** di P_0 e sia α tale che $\Gamma \nvdash_{P_0} \alpha$. Allora $\Gamma \cup \{\neg\alpha\}$ è consistente.*

Dimostrazione. Utilizziamo la *dimostrazione per assurdo*, per cui supponiamo la tesi falsa e cerchiamo di arrivare a negare l'ipotesi per dimostrare la contraddizione (proposizione 5.1):

- Supponiamo, quindi, che $\Gamma \cup \{\neg\alpha\}$ è *inconsistente*.
- Se per ipotesi l'insieme $\Gamma \cup \{\neg\alpha\}$ è inconsistente, allora posso derivare tutto, quindi posso dire che $\Gamma, \neg\alpha \vdash_{P_0} \alpha$.
- Utilizzando il **teorema di deduzione** possiamo dire che se $\Gamma, \neg\alpha \vdash_{P_0} \alpha$ allora $\Gamma \vdash_{P_0} \alpha \rightarrow \neg\alpha$.
- Esiste una derivazione di P_0 tale che $\alpha \rightarrow \neg\alpha \vdash_{P_0} \alpha$. Quindi se $\Gamma \vdash_{P_0} \alpha \rightarrow \neg\alpha$ e $\alpha \rightarrow \neg\alpha \vdash_{P_0} \alpha$, allora per la proprietà 4.2, possiamo dire che $\Gamma \vdash_{P_0} \alpha$. Per ipotesi però avevamo detto che $\Gamma \nvdash_{P_0} \alpha$. Di conseguenza abbiamo trovato una *contraddizione*, quindi il teorema vale.

□

Questo teorema, quindi, detto in maniera grossolana, ci permette di allargare l'insieme di ipotesi la cui derivabilità da Γ non è possibile, in modo da ottenere un nuovo insieme che rimanga consistente. Un corollario importante del teorema è il seguente:

Corollario 5.3.2. *Se* $\Gamma \cup \{\alpha\}$ *è* ***contraddittorio*** *allora* $\Gamma \vdash_{P_0} \neg\alpha$

Tale corollario è una delle proprietà fondamentali della sintassi di P_0 utilizzati in linguaggi di programmazione logica come il *prolog*.

5.4 Semantica di P_0

Abbiamo accennato a inizio capitolo alla **semantica** del calcolo proposizionale. Cerchiamo adesso di definire quei concetti in maniera formale. Per prima cosa, quindi, dobbiamo definire il **concetto di validità**: in particolare per P_0 vale che *una fbf è* ***valida*** *in* P_0 *se è* ***vera in tutti i mondi possibili***. In tal caso dobbiamo definire dunque 3 concetti:

- Concetto di **verità**: formalmente, possiamo dire che tale concetto è dato da un **insieme di 2 elementi distinguibili** l'uno dall'altro. Tipicamente è l'insieme $\{1, 0\}$, dove con 1 identifichiamo il vero, con 0 il falso.
- Concetto di **mondo possibile**: se prendiamo la struttura di una fbf composta e andiamo a scomporla noteremo che il *valore di verità* di tali fbf dipenderà esclusivamente dalle **variabili proposizionali**, poichè sono quelle formule elementari che possono assumere un valore di verità vero o falso. In questo senso, possiamo pensare al concetto di mondo possibile come **un qualcosa che fornisce un valore di verità alle formule elementari**. Definito formalmente, il mondo possibile è definito dalla seguente funzione:

$$B : \ VarProp \rightarrow \{0, 1\} \tag{5.18}$$

I mondi possibili sono, dunque, tutte le funzioni che hanno per dominio le variabili proposizionali e per codominio l'insieme $\{0,1\}$, ovvero le funzioni grazie al quale vengono mappate le variabili proposizionali in $\{0,1\}$. Il *mondo possibile* viene chiamato, nel calcolo proposizionale, **assegnamento proposizionale**.

- Concetto di **verità in un mondo possibile**: tale concetto potrebbe essere formalizzato attraverso una funzione che preso un assegnamento proposizionale, ovvero un mondo possibile, e un fbf, mi dice se quest'ultima è vera o falsa: $F: \ AP \times W \to \{0,1\}$. Nel calcolo proposizionale, in realtà, si utilizza una funzione $\bar{B}$ analoga a F, che è:

$$\bar{B}: \ W \to \{0,1\} \tag{5.19}$$

Quindi possiamo dire che confrontando F con $\bar{B}$, che $F(B,\alpha) = \bar{B}(\alpha)$. Traduciamo formalmente, utilizzando la funzione $\bar{B}$ quindi, come una fbf può essere vera:

  - $\bar{B}(p) = B(p)$, per ogni variabile proposizionale;
  - $\bar{B}(\alpha \to \beta) =$
$$\begin{cases} 0 \Leftrightarrow \bar{B}(\alpha) = 1 \ e \ \bar{B}(\beta) = 0 \\ 1 \ altrimenti \end{cases} \tag{5.20}$$
  - $\bar{B}(\neg\alpha) = 1 - \bar{B}(\alpha)$

5.4.1 Tautologia, Formula Soddisfacibile e Contradditoria

Nella semantica di P_0 dei concetti fondamentali sono quelli di **tautologia**, **formula soddisfacibile** e **formula contradditoria**. Diamone quindi una definizione formale, utilizzando gli strumenti introdotti in questa sezione:

Definizione 5.4.1. *Una fbf di P_0 è:*

- *Una **tautologia** se per ogni assegnamento proposizionale B, $\bar{B}(\alpha) = 1$*
- ***Soddisfacibile** se esiste un assegnamento proposizionale B tale che $\bar{B}(\alpha) = 1$*
- ***Contradditoria** se non è soddisfacibile*

Da tale definizione possiamo dedurre la seguente proposizione:

Proposizione 5.2. *Se α e $\alpha \to \beta$ sono tautologie, allora anche β è una tautologia.*

Dimostrazione. La dimostrazione in questo caso è ovvia dalla definizione di $\bar{B}$. Se infatti $\bar{\alpha} = 1$, ovvero è vera in tutti i mondi possibili, e anche $\alpha \bar{\to} \beta = 1$, allora β non potrà mai assumere valore falso poiche se $\bar{\alpha} = 1$ e $\alpha \bar{\to} \beta = 1$, allora $\bar{\beta}$ per l'implicazione non può assumere il valore 0, quindi falso, poichè altrimenti l'implicazione diventerebbe falsa e, di conseguenza $\bar{B} = 1$, ovvero β è anch'essa una **tautologia**. □

Un *lemma* importante, che abbiamo già usato nella definizione di verità in un mondo possibile è il seguente:

Lemma 5.4.1. *Data una fbf α di P_0, $\bar{B}(\alpha)$ dipende esclusivamente dal valore assegnato da B alle **variabili proposizionali** presenti in α.*

Dimostrazione. Nella definizione di $\bar{B}$, abbiamo detto che il valore di verità delle fbf composte dipende dalle sottoformule che si riconducono alla verità delle variabili proposizionali. Infatti, nella definizione, abbiamo detto, induttivamente, che $\bar{B}(p) = B(p)$ per ogni variabile proposizionale. □

Da questo lemma ricaviamo un *teorema* molto importante:

Teorema 5.4.1. *Data una fbf α di P_0, è **decidibile** se α è una tautologia.*

Per questo teorema è importante fare una breve considerazione: abbiamo già detto nel capitolo precedente che *decidibile* indica che è presente una **procedura effettiva**, o algoritmo, per dire se quell'elemento appartiene all'insieme o meno. Nel caso della logica proposizionale, tale algoritmo siamo abituati a pensarlo con la **tabella di verità**. Ma, per la definzione di *tautologia*, se volessimo dimostrare il teorema, dovremmo controllare *tutti i mondi possibili*, che sono infiniti. Sappiamo, però, per il lemma 5.4.1, che il valore di verità di una fbf dipende solo dalle sue variabili proposizionali. Di conseguenza ciò che si fa è suddividere la tabella di verità in **partizioni** rispetto al valore che tutti i mondi possibili di una partizione danno alle variabili proposizionali. In particolare, poichè i valori possibili sono 2 ($\{0, 1\}$), le partizioni, che sono individuate dalle righe della tabella, sono 2^n, dove n rappresenta il numero di variabili proposizionali che ci interessano.
Passiamo adesso alla definizione fondamentale di **conseguenza logica**:

Definizione 5.4.2 (**Conseguenza tautologica**)**.** *Data una fbf α di P_0 e un insieme Γ di fbf di P_0, si dice che α è **conseguenza tautologica** di Γ se per ogni $\beta \in \Gamma$, se $\bar{B}(\beta) = 1$ allora $\bar{B}(\alpha) = 1$. In questo caso si scriverà: $\Gamma \vDash \alpha$.*

Con tale definizione abbiamo introdotto la **relazione di conseguenza logica**, che è il *corrispettivo semantico* della relazione di derivabilità '$\vdash$'. Quindi diciamo, con la definizione 5.4.2, che α è la conseguenza tautologica di Γ se è vera in tutti i mondi nel quale tutte le fbf Γ sono vere.

5.4.2 Teorema di Correttezza e Completezza

Una volta introdotta la relazione di *conseguenza logica*, vediamo il **teorema di correttezza e completezza**, che in qualche modo lega l'aspetto semantico e sintattico di P_0:

Teorema 5.4.2 (Teorema di Correttezza e Completezza)**.** *Siano Γ un insieme di fbf di P_0 e α una fbf di P_0, si ha che:*

$$\Gamma \vDash \alpha \Leftrightarrow \Gamma \vdash \alpha \tag{5.21}$$

Quindi si può dire che P_0 è **corretto** rispetto alla semantica che abbiamo dato ($\Leftarrow$), e che è **completo** rispetto alla sua sintassi ($\Rightarrow$). Questo significa che tutto ciò che è *derivabile* è anche vero, e ciò che è vero e derivabile. Inoltre possiamo dire che se $\vdash_{P_0} \alpha$, allora $\vDash_{P_0} \alpha$ è una **tautologia**, e viceversa se $\vDash_{P_0} \alpha$, allora $\vdash_{P_0} \alpha$ è un **teorema**. Il teorema di correttezza e completezza ci permette quindi di sfruttare concetti sintattici e semantici allo stesso tempo, così da avere più strumenti a disposizione per le dimostrazioni. Per cui se dimostriamo che una cosa è vera tramite la tabella di verità che partizioniamo in 2^n righe, allora questa sarà derivabile e viceversa. Da tale teorema si ricava un importante *corollario*:

Corollario 5.4.1. P_0 *è* ***consistente***

Dimostrazione. Se prendiamo una formula p, tale che non sia vera in tutti i mondi possibili, allora essa non sarà una tautologia, ovvero $\nvDash p$. Per il teorema di correttezza e completezza, possiamo quindi affermare che:

$$\nvDash p \Leftrightarrow \nvdash p \tag{5.22}$$

ovvero se p non è una tautologia, allora p non è derivabile e, di conseguenza, per la definizione di consistenza, P_0 è **consistente** □

5.5 La Deduzione Naturale per la Logica Proposizionale

Nei capitoli 4 e 5 abbiamo visto la rappresentazione dei sistemi formali in uno stile detto *à la Hilbert*. Tali sistemi formali vengono anche detti **sistemi assiomatici** (capiremo il perchè). In realtà esistono tantissimi stili di rappresentazione di un sistema formale. In tale sezione ci soffermeremo sul *sistema formale della logica proposizionale* in stile **deduzione naturale**, che è del tutto equivalente a P_0 dal punto di vista della *derivabilità*, ovvero dato un insieme Γ di ipotesi, il suo *insieme delle conseguenze* (capitolo 4, sezione 4.5.1) è lo stesso sia nel sistema assiomatico che in quello in deduzione naturale. Affrontare la logica proposizionale in deduzione naturale, però, ci permetterà di **associare** meglio **il concetto di dimostrazione e programmi**.

5.5.1 Sistema formale in Deduzione Naturale

Per la deduzione naturale aggiungiamo nell'alfabeto i simboli '$\vee$' e '$\wedge$' (*and* e *or*). Inoltre abbiamo:

- **fbf**: identiche a P_0. Aggiungiamo che se α è una fbf e β è una fbf, allora $\alpha \vee \beta$ e $\alpha \wedge \beta$ sono anch'esse fbf
- $\mathbf{Ax}$: Non ci sono assiomi. Alcuni degli assiomi che conosciamo sono contenuti all'interno delle regole.
- **Regole**: non ci sono relazioni come quelle viste nei sistemi *à la Hilbert*.

- **Deduzioni**: sono **alberi**. Alla **radice** abbiamo la conclusione della deduzione. Al di sopra abbiamo le premesse, che vengono chiamate **foglie**. La sintassi di tali alberi e la medesima di quella vista per le regole di inferenza, ovvero se abbiamo che con premesse α, β concludiamo γ scriveremo:

$$\frac{\alpha \qquad \beta}{\gamma}$$

  Approfondiremo comunque il discorso vedendo degli esempi di deduzione.

5.5.2 Regole del Sistema Formale in Deduzione Naturale

Le **regole**, in deduzione naturale, si classificano per **regole di introduzione** e **regole di eliminazione** che hanno lo scopo, rispettivamente, di introdurre o eliminare il connettivo logico che si sta utilizzando in base appunto alla regola utilizzata. Per questo si utilizza la notazione $(...I)$ per dire che è una regola di introduzione, mentre $(...E)$ per dire che è una regola di eliminazione. Un altra notazione che utilizzeremo è $[\alpha]^n$, per il quale intendiamo dire che **scarichiamo l'ipotesi n-esima**, ovvero non consideriamo più come ipotesi la fbf tra parentesi quadre, capiremo meglio il significato quando incontreremo tale notazione. Vediamo, quindi, di vedere e analizzare queste regole:

- **Congiunzione** $(\wedge)$:
  - **Introduzione**: La regola di introduzione della congiunzione è la seguente:

$$\frac{\alpha \qquad \beta}{\alpha \wedge \beta}\,(\wedge I)$$

    In questo caso stiamo dicendo che se nel nostro albero affermiamo una fbf α e una fbf β, allora possiamo concludere introducendo la congiunzione di $\alpha \wedge \beta$.
  - **Eliminazione**: Le regole di eliminazione della congiunzione, entrambe equivalenti, sono le seguenti:

$$\frac{\alpha \wedge \beta}{\alpha}\,(\wedge E)$$

$$\frac{\alpha \wedge \beta}{\beta}\,(\wedge E)$$

    In questo caso stiamo dicendo che se affermiamo la congiunzione di 2 fbf come ipotesi, allora possiamo concludere una delle 2, eliminando la congiunzione.

- **Disgiunzione** $(\vee)$:
  - **Introduzione**: le 2 regole equivalenti di introduzione della disgiunzione sono le seguenti:

$$\frac{\alpha}{\alpha \vee \beta}\ (\vee I)$$

$$\frac{\beta}{\alpha \vee \beta}\ (\vee I)$$

Possiamo dire per questa regola che se affermiamo α o β, allora possiamo introdurre la disgiunzione $\alpha \vee \beta$.

- **Eliminazione**: La regola di eliminazione della disgiunzione è la seguente:

$$\frac{\begin{array}{ccc} & [\alpha]^1 & [\gamma]^2 \\ \alpha \vee \gamma & \beta & \beta \end{array}}{\beta}\ (\vee \text{ E})\ (1)(2)$$

Vediamo in questo caso che se riusciamo attraverso un albero di derivazione a partire da α a concludere β, e se riusciamo attraverso un albero di derivazione a partire da γ a concludere β, allora possiamo scaricare le ipotesi α, γ. In conclusione se affermiamo $\alpha \vee \gamma$, allora possiamo concludere β, eliminando la disgiunzione. Il senso è che se io riesco a concludere la stessa fbf da 2 ipotesi diverse, allora possiamo dire che, affermando la disgiunzione possiamo concludere con la fbf che avevamo affermato dalle 2 ipotesi iniziali, scaricando le 2 ipotesi stesse.

- **Implicazione** ($\rightarrow$):
  - **Introduzione**: La regola di introduzione dell'implicazione è la seguente:

$$\frac{\begin{array}{c}[\alpha]^1 \\ \beta \end{array}}{\alpha \rightarrow \beta}\ (\rightarrow \text{I})(1)$$

Tale regola, se ci fate caso, è la *trasformazione in regola di inferenza del teorema di deduzione* (teorema 5.2.1). Infatti, con tale regola stiamo dicendo:

$$\alpha \vdash \beta \Rightarrow \vdash \alpha \rightarrow \beta \tag{5.23}$$

ovvero se attraverso un albero di derivazione a partire da α riesco a derivare β, allora posso scaricare l'ipotesi α e concludere $\alpha \rightarrow \beta$ senza ipotesi.

  - **Eliminazione**: La regola di eliminazione dell'implicazione ("*modus ponens*") è la seguente:

$$\frac{\alpha \qquad \alpha \rightarrow \beta}{\beta}\ (\rightarrow \text{E})$$

Per ovvi motivi viene chiamata *modus ponens* poichè è la stessa regola vista nel sistema P_0.

- **Negazione** ($\neg$): Prima di introdurre le regole che riguardano la negazione, introduciamo il simbolo '$\perp$', che viene detto *bottom*, ed è la rappresentazione in simbolo del **concetto dell'assurdo**. Si tratta, cioè, di una fbf che ha semantica $\bar{B}(\perp) = 0$, ovvero è falso in tutti i mondi possibili. Può essere quindi visto come la fbf $\alpha \wedge \neg\alpha$. Nel caso della negazione, quindi, non vedremo solo le regole di introduzione ed eliminazione, ma anche regole che riguardano tale simbolo e che sono legate a doppio filo con il concetto di negazione:

  - **Regola del Falso**: Tale regola, riguardante proprio il simbolo $\perp$, afferma:

    $$\frac{\perp}{\alpha}\,(\perp)$$

    Molto semplicemente se affermiamo l'assurdo, allora possiamo affermare qualsiasi fbf. E' la stessa cosa di scrivere che:

    $$\alpha, \neg\alpha \vdash \delta \quad \forall \delta \in W \tag{5.24}$$

    dove W è l'insieme di fbf di P_0.

  - **Introduzione ed Eliminazione della negazione**: La regola di introduzione della negazione è la seguente:

    $$\frac{\begin{array}{c}[\alpha]^1 \\ \perp\end{array}}{\neg\alpha}\,(\neg\ \mathsf{I})$$

    Quindi se attraverso un albero di derivazione a partire da α riusciamo a derivare l'assurdo, allora possiamo concludere $\neg\alpha$. Abbiamo visto tale regola sottoforma di teorema in P_0 (corollario 5.3.2). Infatti, dato un insieme di ipotesi contraddittorio $\Gamma \cup \{\alpha\}$ allora possiamo dire che:

    $$\Gamma \cup \{\alpha\} \Rightarrow \Gamma \vdash \neg\alpha \tag{5.25}$$

    Per quanto riguarda invece l'eliminazione della negazione la regola è la seguente:

    $$\frac{\alpha \qquad \neg\alpha}{\perp}\,(\neg\ \mathsf{E})$$

    Se, quindi, affermiamo qualcosa e il suo contrario allora possiamo affermare l'assurdo.

  - ***Reductio Ad Absurdum***: Tale regola è la seguente:

    $$\frac{\begin{array}{c}[\neg\alpha]^1 \\ \perp\end{array}}{\alpha}\,(\mathsf{RAA})(1)$$

    Il nome *reductio ad absurdum* è molto indicativo sull'utilizzo della regola, infatti è la regola che utilizziamo per le *dimostrazioni per assurdo*: ovvero se attraverso un albero di derivazione a partire da $\neg\alpha$ (negazione della tesi) riusciamo ad affermare l'assurdo (contraddizione), allora possiamo concludere α scaricando l'ipotesi $\neg\alpha$

Vediamo quindi un esempio di deduzione utilizzando le regole appena viste:

Esempio 5.5.1. *Dimostrare in deduzione naturale:*

$$\vdash (\neg\neg\alpha) \to \alpha \tag{5.26}$$

Risoluzione. In questo caso dobbiamo trovare una delle tantissime possibili derivazioni di $\vdash (\neg\neg\alpha) \to \alpha$, senza avere quindi alcuna ipotesi. Una delle possibili derivazioni è la seguente:

$$\dfrac{\dfrac{\dfrac{[\neg\alpha]^1 \qquad [\neg\neg\alpha]^2}{\bot}(\neg\ \mathsf{E})}{\alpha}(\mathsf{RAA})(1)}{(\neg\neg\alpha) \to \alpha}(\to \mathsf{I})(2)$$

In questo caso quindi avevamo l'obiettivo di scaricare tutte le ipotesi iniziali. Abbiamo quindi usato la regola di eliminazione della negazione per affermare l'assurdo. Utilizzando poi la *reductio ad absurdum* abbiamo affermato α scaricando $\neg\alpha$, e infine attraverso l'introduzione dell'implicazione abbiamo affermato $(\neg\neg\alpha) \to \alpha$ scaricando l'ipotesi $\neg\neg\alpha$ e ottenendo quindi la dimostrazione. □

Capitolo 6

La Logica dei Predicati del primo ordine

Una logica più potente dal punto di vista espressivo, rispetto alla logica proposizionale, è la **logica dei predicati del primo ordine**. Infatti, in tale logica introduciamo un elemento che mancava nella logica proposizionale, ovvero le *relazioni*.

6.1 Sistema Formale della Logica dei Predicati

Partiamo dunque dal presupposto che i *giudizi*, nel sistema formale della logica dei predicati, sono definiti come **insiemi di individui** e **relazioni fra individui**. Dobbiamo però formalizzare cosa si intende per *individuo*, ovvero definire il **sistema formale** della logica dei predicati:

- **alfabeto** S: l'alfabeto del sistema formale è composto da:
  - Un insieme di *simboli* $\{(,), \neg, \rightarrow\}$
  - I quantificatori universale ed esistenziale $\{\forall, \exists\} \in S$
  - Un **insieme dei simboli di funzione**, ovvero l'insieme di simboli *rappresentante gli insiemi di individui* del tipo [1]:

$$\{0^0, S^1\} \tag{6.1}$$

    dove l'esponente rappresenta l'**arietà del simbolo di funzione**. Ad esempio, quindi, il simbolo '0', rappresenta una *funzione di arietà 0*, ovvero con 0 argomenti e, dunque, una *funzione costante*. Il simbolo S rappresenta una *funzione di arietà 1*, ovvero con un argomento. Le espressioni che rappresentano gli individui sono detti **termini** e si costruiscono nel seguente modo (prendiamo come esempio i simboli di funzione precedentemente visti):

[1] I simboli di funzione non sono univoci, ma vengono decisi in base alla circostanza per cui ci servono per un determinato sistema formale

* 0 è un simbolo di arietà 0, e rappresenta un termine
* $S(t)$ è una funzione di arietà 1, e rappresenta un termine del tipo $S(0), S(S(0))...$
* $x, y, z, ..$ sono *variabili* che rappresentano *generici individui* e rappresentano un termine

In generale, quindi, possiamo dire che:

* L'insieme $x, y, z, ..$ di variabili individuali appartiene all'insieme di termini
* Se abbiamo che $t_1, ..., t_n$ appartengono all'insieme di termini, ed un simbolo f^n di funzione che appartiene all'insieme di simboli di funzione, allora possiamo dire che $f(t_1, ..., t_n)$ è un termine

– Affinchè si possano fare affermazioni sugli individui, definiamo un insieme di *relazioni fra individui*, ovvero affermazioni elementari sugli individui, detti **insieme di simboli di predicato**, o relazione, descritti nel seguente modo:

$$\{<^2, p^1\} \tag{6.2}$$

dove gli esponenti rappresentano l'arietà della relazione. L'insieme

$$\Sigma = \{0^0, S^1\}, \{<^2, p^1\} \tag{6.3}$$

è detto **segnatura**

- L'insieme W delle fbf, data una segnatura del tipo $\Sigma = \{0^0, S^1\}, \{<^2, p^1\}$, è costruito nel seguente modo:

  – Le più semplici fbf definite sulla segnatura Σ sono ad esempio le seguenti:

  * $p(S(S(0))) \in W$
  * $< (0, S(x)) \in W$
  * $< (S(x), S(S(S(x))) \in W$

  Possiamo quindi dire in generale che se $t_1, ..., t_n$ sono termini e P^n è un simbolo di predicato con arità n-aria di Σ, allora $P(t_1, t_2, ..., t_n) \in W$

  – Se $\alpha \in W$, allora $(\neg\alpha) \in W$

  – Se $\alpha, \beta \in W$, allora $(\alpha \to \beta) \in W$

  – Se $\alpha \in W$, $\forall x\alpha \in W$, dove $\forall x$ significa che la relazione vale per ogni generico individuo x

  – Se $\alpha \in W$, $\exists x\alpha \in W$, dove $\exists x$ significa che la relazione vale per almeno un generico individuo x.

6.2 Deduzione Naturale per la Logica dei Predicati

In questo corso andremo ad analizzare semplicemente lo **stile in deduzione naturale** della logica dei predicati. Di conseguenza, le regole non le vediamo come

relazioni n-arie, ma le suddividiamo in *regole di introduzione* e *regole di eliminazione*, come abbiamo fatto per la logica proposizionale (sezione 5.5). Possiamo dire, innanzitutto, che valgono tutte le regole che abbiamo visto per la logica proposizionale. Dobbiamo introdurre in questo caso delle regole di introduzione ed eliminazione per i *quantificatori* $\forall, \exists$:

- Regole per il **quantificatore universale**:
  - **Eliminazione** di $\forall$:

$$\frac{\forall x A}{A[t/x]} \forall \text{ E}$$

    Il significato di tale regola è che se posso affermare A per qualunque individuo generico x, allora posso affermare A per qualsiasi individuo specifico *sostituito* all'individuo generico. In particolare qui è importante capire il concetto di sostituzione, rappresentato nella regola come $A[t/x]$, dove tale scrittura può essere interpretato come *nella fbf A sostituisco tutte le **variabili libere** x con lo specifico termine t*. Il concetto di variabile libera e legata è spiegato più approfonditamente nella sezione 7.3.3 per il modello computazionale del λ-calcolo. Ciò che possiamo dire in questo caso è che se immaginiamo che A sia lo *scope* o *raggio di azione* della variabile x quantificata universalmente, allora diremo che tutte le variabili x sono legate dagli operatori $\forall, \exists$. Ad esempio quindi nel termine $\forall x.P(x)$, la variabile x all'interno di P è legata e può essere anche *rinominata* $\forall y.P(y)$ (concetto di α-conversione, sezione 7.3.4). Se invece prendessimo il termine $P(x)$, considerandolo senza la quantificazione, allora possiamo dire che in questo caso x è libera. L'utilizzo di tali concetti poi si applica alla regola di eliminazione di $\forall$, ad esempio:

$$\frac{\forall x.P(x)}{P(x)[S(0)/x] = P(S(0))} \forall \text{ E}$$

  - **Introduzione di** $\forall$: la regola di introduzione del simbolo di quantificatore universale è la seguente:

$$\frac{A}{\forall x A} \forall \text{ I}$$

    Tale regola, però, *non può essere sempre applicata*. La condizione affinché possa essere applicata è che le ipotesi non scaricate che ho utilizzato per derivare A non contengano la variabile x libera. Un esempio di corretta applicazione della regola può essere la seguente:

$$\frac{\dfrac{[P(x)]}{P(x) \to P(x)} \to \text{I}}{\forall x.(P(x) \to P(x))} \forall \text{ I}$$

In questo caso l'unica ipotesi che avevamo è stata scaricata, nonostante in quel caso la variabile x fosse libera e, di conseguenza, possiamo utilizzare la regola di introduzione di $\forall$ che appunto ci dice che se nel ragionamento non utilizziamo l'individuo x, allora possiamo affermare che il giudizio che abbiamo concluso vale per ogni x.

- Regole per il **quantificatore esistenziale**:
  - **Introduzione di** $\exists$:

$$\frac{A[t/x]}{\exists x A}\ \exists\ \mathsf{I}$$

    In tal caso, quindi, vogliamo dire che se abbiamo una derivazione che conclude con un individuo al quale abbiamo sostituito uno specifico termine al posto di x, allora possiamo dire che esisterà almeno un individuo x per cui valga l'affermazione A.
  - **Eliminazione di** $\exists$:

$$\frac{\begin{array}{c}[A]_n \\ | \\ B\end{array} \qquad \exists x A}{B}\ \exists\ \mathsf{E}\ (\mathsf{n})$$

    Anche in tal caso, per poter applicare tale regola deve essere soddisfatta una condizione, che è quella per cui x non sia libera né in B ne in nessun altra affermazione non scaricata nella sottoderivazione di B, eccetto A. Quindi anche in questo caso non è possibile utilizzare x libera nel ragionamento che ci porta a concludere B, altrimenti non potremmo eliminare la quantificazione di x con B.

6.3 Semantica per la Logica dei Predicati

Una volta definito il sistema formale per la logica dei predicati, e una volta definite le regole nello stile in deduzione naturale, vediamo la **semantica per la logica dei predicati**. Come per la logica proposizionale, abbiamo una nozione di *validità delle fbf*, e diciamo che una fbf è valida quando è *vera in tutti i mondi possibili*. Dobbiamo dunque definire il concetto di *mondo possibile* e il concetto di *essere vero in un mondo possibile*

6.3.1 Struttura o Modello

Il concetto di *mondo possibile*, che dobbiamo definire rispetto a una segnatura Σ viene formalizzato tramite il concetto di **struttura** o **modello** . La definiamo nel seguente modo:

$$\mathcal{A}_\Sigma =< A, F, P > \tag{6.4}$$

ovvero una *tripla* dove:

- A rappresenta l'*insieme di individui* del mondo, e viene spesso indicato con il simbolo $|\mathcal{A}_\Sigma|$.
- P rappresenta una *famiglia di relazioni*, dove per ogni P^n simbolo di predicato appartenente alla segnatura Σ è definita nell'insieme P una funzione $\tilde{P}^n$ che è definita nel seguente modo:

$$\tilde{P}^n : \ A^n \longmapsto \{0,1\} \tag{6.5}$$

  ovvero associa all'insieme di tutti i possibili individui A^n il valore di verità 0 o 1. In parole povere, stiamo assegnando una semantica a tutte le possibili relazioni, ovvero ai simboli di predicato appartenenti alla segnatura. Se per esempio avessimo nella nostra segnatura un simbolo di predicato $R = \{<^2\}$, la funzione $\tilde{R} \in P$ potrebbe associare, ad esempio, all'individuo $< (0,1)$ il valore 1, se intendessimo il simbolo $<$ come relazione "*essere strettamente minore di*"

- F rappresenta la *famiglia di funzioni*, ovvero che preso ogni f^n simbolo di funzione appartenente alla segnatura Σ, definiamo una funzione $\tilde{f}^n$ che è definita nel seguente modo:

$$\tilde{f}^n : \ A^n \longmapsto A \tag{6.6}$$

  ovvero associa all'insieme di possibili individui A^n un individuo A. In parole povere, in questo caso stiamo associando una semantica ai simboli di funzione. Se, per esempio, avessimo S^1 come simbolo di funzione appartenente alla nostra segnatura Σ al quale vogliamo associare la funzione successore, allora possiamo dire che la funzione $\tilde{S}$ associerà ad esempio al termine $S(x)$:

$$\tilde{S} : \ S(x) \longmapsto S(S(x)) \tag{6.7}$$

  ovvero un individuo che rappresenta il successore di $S(x)$

Possiamo dunque vedere il nostro mondo possibile come struttura $\mathcal{A}_\Sigma =< A, F, P >$.

6.3.2 Interpretazione su una Struttura

Una volta visto che il concetto di mondo possibile è riconducibile al concetto di *struttura su una segnatura* Σ, dobbiamo comprendere il concetto di *verità rispetto al mondo possibile*. Prima, introduciamo una nuova funzione, che è la seguente:

$$[[t]]^{\mathcal{A}_\Sigma} \tag{6.8}$$

Tale funzione è quella funzione che associa un valore di verità 0 o 1 al termine t rispetto al mondo possibile $\mathcal{A}_\Sigma$, e viene detta **interpretazione** di formula vera o falsa nel mondo possibile e viene definita sulla *struttura delle fbf*. Prima di vedere come è definita, bisogna porsi un importante questione. Infatti abbiamo visto che le nostre fbf possono contenere *variabili individuali libere*. In tal caso *qual*

è l'individuo che associamo a tali variabili, essendo generiche? Esiste una funzione p, detta **ambiente**, definita per una segnatura Σ nel seguente modo:

$$p: \text{ Variabili Individuali} \longmapsto |A_\Sigma| \tag{6.9}$$

ovvero associa ad ogni variabile individuale libera un individuo della struttura $\mathcal{A}_\Sigma$.

Interpretazione delle fbf

Una volta definito l'insieme di ambienti $ENV^{\mathcal{A}_\Sigma}$, come tutti i possibili ambienti su una struttura$\mathcal{A}_\Sigma$, la definizione di **interpretazione per una fbf su una struttura** $\mathcal{A}_\Sigma$ è la funzione:

$$[[]]^{\mathcal{A}_\Sigma}: \ fbf \times ENV \to \{0,1\} \tag{6.10}$$

ed è definita sulla *struttura delle fbf*, ovvero considerando ogni caso possibile:

- Per quanto riguarda le fbf elementari, ovvero le *relazioni fra individui*, diciamo che il loro valore di verità dipende dall'ambiente p, in caso fossero presenti all'interno dei termini variabili individuali libere, e in particolare dall'individuo che associa p a ogni termine della fbf. Formalmente scriviamo:

  $$[[P^n(t_1,...,t_n)]]p = \tilde{P}^n([[t_1]]p,...,[[t_n]]p) \tag{6.11}$$

  ovvero per ogni possibile relazione P^n, la sua interpretazione dipende dal valore vero o falso associato dalla funzione $\tilde{P}^n$ e dall'ambiente che associa ogni variabile individuale a un determinato individuo

- $[[\neg\alpha]]p = 1$ se e solo se $[[\alpha]]p = 0$

- $[[\alpha \to \beta]] = 1$ se e solo se da $[[\alpha]]p = 1$ segue che $[[\beta]]p = 1$.

- $[[\forall x\alpha]] = 1$ se per ogni $a \in |\mathcal{A}_\Sigma|$ abbiamo che $[[\alpha]]p_x^a = 1$ dove $p_x^a \in ENV$ è la funzione così definita:

  $$p_x^a = \begin{cases} p(y) & \text{se } xy \\ a & \text{altrimenti} \end{cases} \tag{6.12}$$

  ovvero p_x^a è un *ambiente modificato*, ovvero una funzione che associa alla variabile individuale x l'individuo a.

6.3.3 Validità delle fbf per la logica dei predicati

Una volta definita la definizione di **interpretazione di formula ben formata**, possiamo dare le seguenti definizioni. Data una fbf α e $\mathcal{A}_\Sigma$ struttura, diremo che:

- α è **soddisfacibile** in $\mathcal{A}_\Sigma$ se e solo se esiste $p \in ENV$ per cui si abbia $[[\alpha]]p = 1$

- α è **vera** in $\mathcal{A}_\Sigma$ se e solo se per ogni $p \in ENV$ si ha $[[\alpha]]p = 1$ e scriveremo $\mathcal{A}_\Sigma \vDash \alpha$, ovvero α è vera nel mondo possibile $\mathcal{A}_\Sigma$.

Inoltre possiamo dire che presa una qualsiasi fbf α:

- α è *soddisfacibile* se e solo se esiste una struttura $\mathcal{A}_\Sigma$ in cui α è soddisfacibile
- α è **valida** se e solo se per ogni struttura $\mathcal{A}_\Sigma$ si ha che $\mathcal{A}_\Sigma \vDash \alpha$, ovvero α è vera in ogni mondo possibile.
- α è **contraddittoria** se e solo se non è soddisfacibile.

Un ultima importante definizione è quella di **conseguenza logica**. Preso un insieme Γ di fbf e una fbf α, diciamo che α è *conseguenza logica* di Γ se e solo se per ogni struttura $\mathcal{A}_\Sigma$ e per ogni ambiente $p \in ENV$, da $[[\Gamma]]p = 1$ segue che $[[\alpha]]p = 1$, e scriveremo che $\Gamma \vDash \alpha$.

6.3.4 Correttezza e Completezza

Come per la logica proposizionale, anche per la logica dei predicati vale il **teorema di correttezza e completezza** per cui:

$$\Gamma \vdash \alpha \Longleftrightarrow \Gamma \vDash \alpha \tag{6.13}$$

In particolare vale il seguente corollario, per cui:

$$\nvdash \alpha \Leftrightarrow \nvDash \alpha \tag{6.14}$$

ovvero se α non è un teorema, allora α non è vera in tutti i mondi possibili, ovvero esisterà sicuramente un mondo possibile, cioè una struttura, in cui α è falsa. Inoltre, un importante osservazione che si può fare è che la relazione di conseguenza logica **non è decidibile** a differenza della relazione di conseguenza tautologica, nel quale esiste una procedura effettiva, ovvero quella delle tabelle di verità.

Parte III

Modelli Computazionali

Capitolo 7

Programmazione Funzionale e λ-calcolo

Nella parte riguardante la *logica* (parte II) abbiamo accennato diverse volte al fatto che vi è una *corrispondenza intrinseca tra* ***deduzioni*** *e* ***programmi***. Inoltre, nella sezione 5.5, abbiamo introdotto lo stile in deduzione naturale per la logica proposizionale che ci aiuterà in tale scopo. Adesso, però, concentriamoci sul descrivere formalmente cos'è un **programma** e cosa siano i **modelli computazionali**.

7.1 Computazione e Modelli Computazionali

Da informatici, quando ci viene chiesto di descrivere formalmente la definizione di algoritmo, diciamo sempre che si tratta di una *serie di passi* atti alla risoluzione di un problema. Addentrandoci, però, nel significato intrinseco della parola *computer*, possiamo comprendere che ciò che fa la nostra macchina è qualcosa che rimandi a una **computazione**, ovvero i programmi o algoritmi sono costituiti da una serie di **passi di computazione**. Ma quindi, cosa significa *computare* o, utilizzando un sinonimo più comprensibile ai più, *calcolare*? Per dare una definizione *generale*, ma non generica, possiamo dire che computare significa

trasformare un informazione da **forma implicita** a **forma esplicita**

Facciamo un esempio: l'espressione

$$13 + 33 \tag{7.1}$$

rappresenta una *sequenza di simboli che in* ***modo implicito*** *significano* 46. Se applichiamo a questa sequenza di simboli l'algoritmo di somma, ovvero, trasformiamo l'informazione $13 + 33$ in

$$46 \tag{7.2}$$

allora abbiamo trasformato l'informazione in **forma esplicita**, applicando senza saperlo prima di questa lettura una **computazione**.

Modelli Computazionali

Alla base dei linguaggi di programmazione, poi, vi sono i **modelli computazionali**, che sono:

formalismi matematici *che descrivono in modo preciso il concetto di* ***computazione***

Di modelli computazionali, quindi, ce ne sono tanti per formalizzare in tanti modi il concetto di computazione che stanno alla base delle *classi di linguaggi di programmazione*:

- **Linguaggi Imperativi**: tale classe di linguaggi di programmazione, che è quella che sicuramente conoscete tutti si basa sul modello computazionale delle **macchine di Turing** ed ha come concetti fondamentali quelli di:
  - *variabile*
  - *assegnamento*
  - *iterazione*
  - *esecuzione*
  - *istruzione*
- **Linguaggi Funzionali**: probabilmente questa classe di linguaggi di programmazione sarà nuova ai più, ed è un tipo di programmazione basata sul modello computazionale del λ-calcolo che è un modello utilizzato per affrontare problemi strettamente logici. I linguaggi funzionali si basano sui concetti di:
  - *espressioni*
  - **variabili** (in senso matematico): possiamo intendere come variabile $f(x)$, oppure $x + 1$, quindi in maniera nettamente differente dai linguaggi imperativi
  - *valutazione* (differente da esecuzione): per esempio se ho l'espressione $3 + 2$, in questo caso valuto tale espressione che mi porta a restituire 5
  - *ricorsione*: possiamo dire che corrisponde a quella che è l'iterazione per i linguaggi imperativi, ovvero senza ricorsione non si potrebbe fare alcun programma utile. Il concetto di ricorsione è strettamente legato, inoltre, con il concetto matematico di **induzione**. Per cui dimostrare una proprietà di un programma funzionale risulta molto facile in quanto basta applicare l'induzione.

Ovviamente oltre a questi ci sono tantissimi altri modelli computazionali e sono inoltre tutti **equivalenti**, ovvero ciò che faccio in un modello posso farlo anche in un altro.

7.2 Introduzione alla Programmazione Funzionale

In tale sezione analizziamo quindi il paradigma di **programmazione funzionale**, che può sembrarci ostico all'apparenza, ma che in realtà è utile per avere un punto di vista diverso sulla programmazione, soprattutto perchè negli ultimi tempi i *linguaggi funzionali* stanno prendendo sempre più piede anche nel mondo dell'IT.
In programmazione funzionale, ciò che facciamo è

definire funzioni

Funzione Inversa Vediamo un esempio di una definizione di funzione. Definiamo, quindi, la funzione che restituisce l'**inversa di una stringa**

Definizione. La funzione è la seguente:

$$\tilde{x} = \begin{cases} \varepsilon & \text{se } x = \varepsilon \\ \tilde{y} & \text{se } x = ay \text{ con } a \in \Sigma \end{cases} \tag{7.3}$$

□

Ciò che abbiamo fatto è definire una funzione, che abbiamo simbolicamente indicato $\tilde{x}$, dove al posto del simbolo posso inserire qualsiasi stringa, ovvero si tratta della nostra **variabile**. Ciò significa che l'unica funzione sulle stringhe che messa al posto di tilde mi da quell'output è la funzione inversa. Si tratta quindi di un *programma per calcolare la funzione inversa*. Applicando il programma della definizione posso trovare l'inversa di qualsiasi stringa:

Esempio 7.2.1. *Calcoliamo, per esempio, la stringa inversa di* cdb*:*

$$\begin{gathered} \tilde{cdb} \\ \tilde{d}bc \\ \tilde{b}dc \\ \tilde{\varepsilon}bdc \\ bdc \end{gathered} \tag{7.4}$$

Abbiamo quindi eseguito una computazione, ovvero abbiamo trasformato un informazione da una forma implicita a una forma esplicita.

7.2.1 Definire le funzioni in Haskell

Uno dei linguaggi funzionali tra i più utilizzati è *Haskell*, che utilizzeremo con lo scopo di analizzare i suoi elementi di base per poi comprendere il modello computazionale sul quale sono basati i linguaggi funzionali, ovvero il λ-calcolo. Partiamo dal presupposto che quando definiamo una funzione nei linguaggi funzionale, tale definizione contiene anche il **metodo di calcolo**, come anche abbiamo visto per la stringa inversa. Ad esempio la funzione matematica:

$$f(x) = 5 + x \tag{7.5}$$

è una funzione che ci dice come deve essere calcolata. Infatti, se consideriamo la x come parametro formale, e volessimo calcolare la funzione per il valore 3, allora basterà sostituire il **parametro attuale** 3 con il **parametro formale** x e *valutare la funzione*. In Haskell la funzione vista si scrive con la seguente sintassi:

```
f x = x+5
```

dove stiamo dicendo che la funzione che ha nome f, al parametro formale x *associa* l'espressione $x + 5$. Tale concetto matematicamente si esprime come:

$$x \mapsto x + 5 \tag{7.6}$$

ovvero definiamo l'**associazione** ($\mapsto$) tra x e $x+5$. Quindi quando abbiamo definito la funzione, la corrispondente funzione può essere definita in Haskell anche come:

$$f = x \longmapsto x + 5 \tag{7.7}$$

Tale associazione in Haskell viene anche scritta con la seguente sintassi:

```
\x->x+5
```

In tal caso tale funzione viene detta **anonima** poichè non stiamo dando nessun nome alla funzione.

Esempio 7.2.2. *Vediamo per esempio di definire in Haskell una funzione che restituisce la stringa inversa:*

```
inv = \y -> if(null y) then y
                       else(inv(tail y))++[head y]
```

Per capire bene quanto scritto nell'esempio, abbiamo definito alcune **funzioni predefinite** di Haskell:

- **null** restituisce vero se la stringa è vuota
- **tail** prende in input una stringa e restituisce una stringa uguale ma senza il primo elemento
- **head** prende una stringa in input e restituisce il primo carattere
- $++$ prende due stringhe come argomenti e restituisce la loro concatenazione

Inoltre possiamo dire che tutto ciò che viene scritto tra parentesi quadre [] è una stringa. Le *if...then...else* ci permettono di costruire **espressioni condizionali**, allo stesso modo di come avviene per i linguaggi operativi. Come invece abbiamo già detto, il significato di $inv =$ è interpretabile come inv è il *nome della funzione*... e la simbologia :

```
\y -> ....
```

possiamo interpretarla, come già detto, come la funzione che prende in input un argomento y restituisce l'espressione presente dopo '->' nel quale si sostituisce il parametro attuale con quello formale.

7.2.2 Elementi di Base della Programmazione Funzionale

Sulla base di ciò che abbiamo detto, possiamo fare un importante considerazione su quali siano gli **elementi di base della programmazione funzionale**:

- Per definire una funzione, che è ciò che si fa in programmazione funzionale, ci serve costruire a monte *espressioni matematiche*, che sono costituite da elementi come **costanti**, **variabili** (nel senso matematico) e **funzioni predefinite**. A partire da questi elementi costruiamo l'espressione utilizzando l'**applicazione funzionale**. Nell'esempio 7.2.2 abbiamo, per esempio utilizzato questi elementi, come le funzioni predefinite, per costruire l'espressione dopo il simbolo '->'

- E' fondamentale poter costruire *funzioni anonime*. Infatti, a partire dalle nostre espressioni matematiche, che costituiscono il *corpo* della funzione, costruiamo la nostra funzione anonima a partire da una variabile, che sarà il nostro parametro formale, al quale sostituiamo l'intera espressione. Nell'esempio 7.2.2, avevamo la seguente funzione anonima che definiva la funzione *inv*:

  ```
  \y -> if(null y) then y
                   else(inv(tail y))++[head y]
  ```

  L'operatore che permette di definire funzioni anonime '\->' è chiamato operatore di **astrazione funzionale**

- Un altro punto fondamentale è quello di **associare** l'espressione che rappresenta una funzione anonima con **un nome**, utilizzando l'operatore '='. Nell'esempio 7.2.2 avevamo associato all'espressione che restituisce la stringa inversa il nome *inv*. In realtà possiamo associare un nome non solo a funzioni anonime ma anche a qualsiasi espressione, come ad esempio:

  ```
  pippo = (2*3)+4
  ```

  Se chiediamo ad Haskell di valutare tale espressione, ci restituirà sempre il valore 10

Abbiamo quindi definito le *componenti essenziali o atomiche* della programmazione funzionale. Se pensiamo che il nostro obiettivo sarà quello di studiare il modello computazionale del λ-calcolo, esso sarà un formalismo matematico basato, anche se pur in maniera più astratta, su tali concetti essenziali.

7.2.3 Computazione nei Linguaggi Funzionali

Abbiamo analizzato le componenti essenziali di un linguaggio funzionale, ovvero quegli elementi di base senza il quale non potrebbero essere creati algoritmi o programmi funzionali. Nel nostro obiettivo di comprendere a fondo il modello computazionale del λ-calcolo, dobbiamo però introdurre ancora il **concetto di computazione nei linguaggi funzionali**, che essenzialmente è quello di *prendere un espressione e valutarla in* ***passi discreti di trasformazione***, ovvero come avevamo

detto inizialmente trasformare un'informazione da una forma implicita a una esplicita. La domanda che ci poniamo quindi è: *Come trasformiamo le espressioni*? Essenzialmente abbiamo 3 tipi diversi di *passi di computazione*, che sono:

1. Sostituire un nome con l'espressione associata ad esso. Per esempio, con la funzione *inv*, al posto di inv, mettiamo la funzione associata al nome *inv*.

2. Sostituire il parametro formale con il parametro attuale della funzione anonima all'interno della funzione stessa. Se per esempio volessimo trovare l'inversa di "cbd", allora si sostituirà al posto delle y, che rappresentano il parametro formale, il parametro attuale "cbd"

3. Calcolare una funzione predefinita al quale abbiamo applicato un valore "esplicito". Ad esempio si valuteranno tutte le funzioni come *null cbd, tail cbd, ...*

Strategie di Valutazione

Nella computazione, quindi, tutto si riduce all'individuare delle sottoespressioni dall'espressione principale e poi valutarle. La domanda che ci si pone, quindi, di conseguenza è: *quale tra le varie espressioni viene valutata per prima*? La risposta è che vi è un algoritmo che permette di dire quale espressione valutare prima, e tale politica viene attuata tramite le **strategie di valutazione**. Haskell, per esempio, utilizza una strategia *lazy*, ovvero che decide di valutare l'espressione necessaria per arrivare al valore finale. In un linguaggio imperativo, per esempio, è molto più complesso riuscire a comprendere il risultato finale di una computazione: infatti vi sono dei *side effects* dovuti al fatto che la computazione può dare diversi risultati in base a dove la funzione si trova nel programma, a quante volte viene eseguita o dal flusso del programma stesso. In programmazione funzionale, invece, poichè definiamo *funzioni matematiche*, il risultato che otteniamo è sempre lo stesso indipendentemente da dove si trova l'espressione o da quante volte viene valutata. Questa proprietà dei linguaggi funzionali si chiama *referential transparency* e il fatto che i linguaggi imperativi non presentino tale proprietà rende difficile anche dimostrare proprietà dei programmi stessi.

7.2.4 Funzioni di Ordine Superiore

Un ulteriore vantaggio che i linguaggi funzionali ci offrono rispetto ai linguaggi imperativi è quello di poter *definire funzioni che hanno come argomento altre funzioni o che restituiscono altre funzioni*, che vengono chiamate **funzioni di ordine superiore**. Un banale esempio può essere la seguente funzione:

```
atzero f = f 0
```

La funzione che ha nome *atzero* prende come argomento una funzione f e restituisce $f(0)$. La possibilità di definire questo tipo di funzioni che non è possibile nei linguaggi imperativi, rende i *linguaggi funzionali molto* ***espressivi***. Infatti, i programmi

che scriviamo nei linguaggi funzionali si scrivono in pochissime righe di codice. Inoltre, *dimostrare una proprietà di un programma in un linguaggio funzionale equivale a* ***dimostrare definizioni matematiche***, il ché è molto più semplice rispetto al dimostrare proprietà di linguaggi imperativi, che richiedono strumenti matematici molto più complessi. Vediamo quindi un esempio di funzione di ordine superiore.

7.2.5 Curryficazione

Un concetto fondamentale in programmazione funzionale è quello di **curryficazione**, che si basa essenzialmente sul

associare a funzioni con n argomenti una funzione f_c

tale che

$$f_c : \ A_1 \to (A_2 \to (\dots (A_n \to B) \dots)) \tag{7.8}$$

dove f_c viene detta *versione curryficata della funzione*. Possiamo dire che ogni volta che utilizziamo una funzione, possiamo utilizzare la sua versione curryficata, che mi permette di fare le stesse cose che avrei fatto con una funzione che prende in input n argomenti e mi restituisce un risultato, col vantaggio però di avere una **maggiore espressività**, data dal fatto che si prende un solo argomento in input, e inoltre che mi permette di definire più funzioni. Proprio per i maggiori vantaggi che apportano le versioni curryficate di una funzione, quando definiamo una funzione in Haskell stiamo definendo in realtà automaticamente la sua versione curryficata.

Esempio 7.2.3. *Definiamo la versione currificata della seguente funzione:*

```
doublesum x y = 2*(x+y)
```

Proof. La versione currificata equivalente ala funzione *doublesum* è la seguente:

```
doublesum = \x -> \y -> 2*(x+y)
```

□

7.3 Il modello computazionale del λ-calcolo

Una volta che abbiamo compreso quali sono gli elementi costitutivi della programmazione funzionale, allora sicuramente poichè quest'ultima è basata sul modello computazionale del λ-calcolo, abbiamo gli strumenti necessari per affrontare quest'ultimo. Abbiamo detto che i concetti di base della programmazione funzionale per definire le funzioni sono i seguenti:

- **variabili**
- **valori di base**
- **funzioni di base** o predefinite

- **applicazione funzionale**, ovvero lo scrivere l'espressione corrispondente alla funzione data
- **astrazione funzionale**, ovvero il meccanismo per definire funzioni anonime
- **associare nomi a espressioni** per definire funzioni ricorsive

Oltre a definire tali elementi necessari per definire funzioni, abbiamo visto *come viene valutata un espressione*, ovvero come avviene la *computazione* e avevamo visto che avevamo tre tipi di trasformazione:

- $\rightsquigarrow_A$: *sostituire un nome ad un espressione*
- $\rightsquigarrow_B$: *sostituire il parametro formale con quello attuale*
- $\rightsquigarrow_C$: *restituisce il valore della funzione predefinita sull'espressione*

Dopo aver fatto questo riassunto, possiamo dire che il λ-calcolo sarà un formalismo matematico che rappresenta in modo preciso il concetto di computazione alla base dei linguaggi funzionali, ovvero è la formalizzazione degli elementi necessari della programmazione funzionale.

7.3.1 Elementi Necessari e Computazione nel λ-calcolo

Tutti quegli elementi che ci sembravano elementari o indispensabili in programmazione funzionale, in realtà, vedremo adesso, che non lo sono. Il λ-calcolo è infatti un modello computazionale estremamente elementare che si basa su questi 3 semplici concetti:

- **variabili**
- **applicazione funzionale**
- **astrazione funzionale**

Inoltre anche la computazione nel λ-calcolo sarà definita dalla trasformazione che avevamo etichettato come $\rightsquigarrow_B$, infatti per quanto riguarda la trasformazione $\rightsquigarrow_A$ non abbiamo nomi da sostiture e per quanto riguarda la trasformazione $\rightsquigarrow_C$ non abbiamo funzioni predefinite.

7.3.2 Insieme dei λ-termini

Tutto ciò che abbiamo detto a parole deve essere formalizzato. Essenzialmente per formalizzare il concetto di dato o di funzione utilizziamo l'insieme Λ, che è detto **insieme dei λ-termini**. I *termini*, infatti, sono la formalizzazione sia di dato che di *definizione di funzione*. Abbiamo visto infatti, che grazie all'utilizzo della curryficazione per le funzioni di ordine superiore, non vi è alcuna differenza tra dato e definizione di funzione.

Definizione 7.3.1 (Insieme dei λ-termini)**.** *Preso un insieme infinito e numerabile di variabili*

$$\{x, y, z, t, x', ...\} \tag{7.9}$$

abbiamo che:

- *Qualsiasi* ***variabile*** $x \in \Lambda$
- *Se* $M, N \in \Lambda$ *allora* $(M \cdot N) \in \Lambda$
- *Se* $M \in \Lambda$ *allora* $\lambda x.M \in \Lambda$

Nella definizione abbiamo essenzialmente definito formalmente gli elementi costitutivi del λ-calcolo. In particolare abbiamo detto che:

- Dati 2 termini qualsiasi, la loro applicazione $M \cdot N$ è anch'essa un λ-termine. Per semplicità, inoltre, di solito si scrive l'applicazione come MN, ovvero omettendo il simbolo di applicazione '$\cdot$'
- Definito λ come operatore che prende un argomento, ovvero la variabile x, e restituisce un λ-termine M, abbiamo definito quindi il concetto di **astrazione funzionale**, ovvero la definizione di funzioni anonime. Come possiamo notare, inoltre, abbiamo un solo argomento. Infatti sappiamo che grazie alla *curryficazione* tutte le funzioni n-arie possono essere trasformate in funzioni con *arità 1*, ovvero funzioni unarie che prendono un solo argomento

Nonostante possa sembrarci molto strano gli elementi essenziali del λ-calcolo sono solo questi. Quello su cui ci concentreremo adesso è su come avviene la computazione per il λ-calcolo.

Convenzioni sulle Parentesi

Prima di addentrarci nel mondo della computazione del λ-calcolo, vediamo alcune convenzioni che sono utilizzate nel λ-calcolo per abbreviare le scritture utilizzando meno parentesi possibili:

$$\begin{array}{c}(\lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.M))) \text{ è abbreviato in } (\lambda x_1 x_2 \ldots x_n.M) \\ (\ldots((M_1 M_2)M_3)\ldots M_n) \text{ è abbreviato in } (M_1 M_2 M_3 \ldots M_n)\end{array} \tag{7.10}$$

7.3.3 Variabili Legate e Variabili Libere

Prima di formalizzare il concetto di computazione nel λ-calcolo, dobbiamo prima saper distinguere il significato di **variabile legata**, o *bound variable*, e **variabile libera**, o *free variable*. Possiamo dire che data la seguente *astrazione*:

$$\lambda x.P \tag{7.11}$$

dove P è detto ***raggio di azione*** [1] *dell'astrazione*, abbiamo che:

[1] In inglese *scope*

- Se la variabile x compare nel raggio di azione dell'astrazione P, allora è **legata**. Se abbiamo la seguente astrazione:

$$\lambda x.((yx)z) \tag{7.12}$$

  In tale esempio vediamo che compare la x nel raggio di azione dell'astrazione, per cui diciamo che nel termine yx, la variabile x è legata a λx. Essenzialmente, una variabile legata può essere vista anche come il punto in cui, data un applicazione di un parametro formale finisce l'astrazione, ovvero una sorta di *segnaposto*. Se infatti applicassimo la funzione anonima come $(x.((yx)z))3$, avremmo che il segnaposto della variabile verrebbe sostituito con $(3y)z$.

- Se la variabile x non compare nel raggio dell'astrazione P, allora è **libera**. In questo caso, quindi, tali variabili non serviranno da segnaposto ma rappresentano cose non meglio definite. Nell'esempio visto in precedenza, le variabili y, z erano per esempio libere.

La definizione formale è data per induzione:

Definizione 7.3.2. *Definiamo $BV(M)$ l'insieme di **variabili legate** di un termine M nel seguente modo:*

- $BV(x) = \varnothing$, *dove $M = x$ è una variabile*
- $BV(PQ) = BV(P) \cup BV(Q)$, *dove $M = PQ$ è un applicazione*
- $BV(\lambda x.P) = \{x\} \cup BV(P)$, *dove $M = \lambda x.P$ è un astrazione*

Definizione 7.3.3. *Definiamo $FV(M)$ l'insieme di **variabili libere** di un termine M nel seguente modo:*

- $FV(x) = \{x\}$, *dove $M = x$ è una variabile*
- $FV(PQ) = FV(P) \cup FV(Q)$, *dove $M = PQ$ è un applicazione*
- $FV(\lambda x.P) = FV(P) \setminus \{x\}$, *dove $M = \lambda x.P$ è un astrazione*

7.3.4 Computazione nel λ-calcolo

Dobbiamo adesso formalizzare il concetto di **computazione** nel λ-calcolo. Abbiamo già visto che l'unico passo di trasformazione che presenta il λ-calcolo è il passo $\rightsquigarrow_B$, ovvero quello in cui

*trasformiamo la funzione anonima in una funzione nel quale **sostituiamo** un parametro con quello attuale*

Un esempio di questa trasformazione può essere il seguente:

$$(\lambda x.(t(xy)))(\lambda z.z) \rightsquigarrow t((\lambda z.z)y) \tag{7.13}$$

Per formalizzare il concetto di computazione, dobbiamo definire la *relazione di trasformazione di β-***riduzione**:

Definizione 7.3.4 (β-riduzione)**.** *Preso un termine nella forma $(\lambda x.M)N$, chiamato **redex** e detto $M[N/x]$ il suo **contractum**, allora la **relazione di β-riduzione** è così definita:*

$$(\lambda x.M)N \rightarrow_\beta M[N/x] \subseteq \Lambda \times \Lambda \tag{7.14}$$

ovvero, se è presente un redex, il corpo della funzione è in relazione di β-riduzione con il suo contractum.

Tale relazione binaria fra λ-termini formalizza il concetto di computazione nel λ-calcolo. Essenzialmente, la simbologia $M[N/x]$ indica l'operazione di **sostituzione** del redex con il suo corrispettivo contractum. Dobbiamo quindi formalizzare questo concetto di *operazione di sostituzione*.

α-conversione

Per definire la sostituzione, dobbiamo sapere prima cosa sia l'α-**conversione** e quando 2 termini si dicono α-*convertibili*. Diciamo che vogliamo formalizzare tale considerazione: prendiamo il seguente λ-termine

$$(\lambda x.yx) \tag{7.15}$$

che è un λ-termine che preso un argomento x restituisce l'applicazione di y a x. Prendiamo un secondo λ-termine

$$(\lambda z.yz) \tag{7.16}$$

che è un λ-termine che preso un argomento z restituisce l'applicazione di y a z. Come avrete sicuramente notato, essenzialmente tali termini sono esattamente la stessa cosa a meno del nome che diamo alle variabili legate. Se chiamiamo M, M' tali λ-termini possiamo affermare:

$$M =_\alpha M' \tag{7.17}$$

ovvero M e M' sono α-convertibili. Tale concetto di α-conversione ci serve per definire in modo preciso il concetto di sostituzione e, di conseguenza, di β-riduzione.

Sostituzione

Formalizziamo quindi l'operazione di **sostituzione**

$$M[N/x] \tag{7.18}$$

dove abbiamo detto che il nostro scopo è quello di *sostituire, nel termine M ogni occorrenza **libera** di x con il termine N*. Dobbiamo però dare una definizione precisa e formale di tale concetto. Per quanto riguarda la sostituzione utilizziamo una definizione per **induzione strutturale**, ovvero consideriamo tutti i possibili $\lambda - termini$.

Definizione 7.3.5. *1. Se M è una **variabile** $(M = x)$ allora:*

$$\begin{cases} x[L/x] \equiv L \\ y[L/x] \equiv y \end{cases} \tag{7.19}$$

*2. Se M è un **applicazione** $(M = PQ)$ allora:*

$$(PQ)[L/x] \equiv P[L/x]Q[L/x] \tag{7.20}$$

*3. Se M è un **astrazione** $(M = \lambda x.N)$ allora:*

$$\begin{cases} (\lambda x.N)[L/x] \equiv \lambda x.N \\ (\lambda y.N)[L/x] \equiv \lambda x.(N[L/x]) \end{cases} \tag{7.21}$$

Da tale definizione di sostituzione nasce un problema che può essere visto tramite il seguente esempio. Vogliamo applicare la seguente sostituzione ai rispettivi termini:

$$\begin{cases} (\lambda x.yx)[(xv)/y] \\ (\lambda w.yw)[(xv)/y] \end{cases} \tag{7.22}$$

Possiamo notare che i 2 termini su cui vogliamo applicare la sostituzioni sono α-*convertibili*, infatti sono in realtà lo stesso termine a meno del nome delle variabili legate. Applichiamo quindi adesso la definizione di sostituzione, che è un *algoritmo ricorsivo* che effettua, dato un termine, una *sottoinduzione*:

$$\begin{cases} (\lambda x.yx)[(xv)/y] \equiv (\lambda x.((xv)x)) \\ (\lambda w.yw)[(xv)/y] \equiv (\lambda w.((xv)w)) \end{cases} \tag{7.23}$$

Notiamo che abbiamo ottenuto un risultato completamente diverso, infatti nel termine $(\lambda x.((xv)x))$, la variabile x è diventata legata. In realtà avremmo dovuto ottenere lo stesso risultato per entrambi i termini, visto che sono α-convertibili. In tal senso bisogna modificare la definizione di sostituzione in modo tale che anche dopo la sostituzione le variabili libere rimangano tali. La definizione, dunque, si modifica in tal modo:

$$(\lambda y.M)[L/x] \equiv \lambda y.(M[L/x]) \Rightarrow FV(L) \cap (\lambda y.M) = \varnothing \tag{7.24}$$

In tal caso, indicando con il termine $FV(L)$ le variabili libere del termine L, diciamo che la condizione per la quale può essere effettuata una sostituzione è che non ci devono essere variabili libere di L nel termine $(\lambda y.M)$. Data tale definizione, può sembrarci che la sostituzione non sia sempre possibile da fare; in realtà grazie all'α-conversione riesco a soddisfare la condizione rinominando le variabili interessate. Facciamo un esempio:

$$\lambda y.y(\lambda z.z)[(\lambda z.y)/x] =_\alpha \lambda v.v(\lambda w.w)[(\lambda z.y)/x] \equiv \lambda v.(v(\lambda w.(\lambda z.y)w)) \tag{7.25}$$

Normalizzazione

Nella relazione di β-riduzione il nostro scopo è quello di **normalizzare** i termini. Per definizione,

*un termine si dice in **forma normale** quando non contiene redex*

Quindi, se la vogliamo vedere dal punto di vista computazionale, possiamo dire che la forma normale di un termine rappresenta il suo *valore esplicito*. In particolare, quindi, *normalizzare un termine* significa *applicare una riduzione fino a che non otteniamo una forma normale*, se esiste. Quindi si tratta di una

trasformazione o riduzione in 0 o più passi del termine

Solitamente, se esiste la normalizzazione di un termine M nella sua forma normale Q, viene indicata con:

$$M \ -\!-\!>> \ Q \tag{7.26}$$

Alcuni termini, infatti, non hanno forma normale, per esempio:

$$(\lambda x.xx)(\lambda x.xx) \tag{7.27}$$

Importanti Risultati della teoria di β-riduzione

Dei teoremi fondamentali come risultati della teoria di β-riduzione sono il **teorema di confluenza** e il *lemma* di **unicità delle forme normali**. Il teorema di confluenza ha il seguente enunciato:

Teorema 7.3.1 (Teorema Di Confluenza)**.** *Preso un termine M tale che:*

$$M \ -\!-\!>> \ P \text{ e } M \ -\!-\!>> \ Q \tag{7.28}$$

allora esisterà sicuramente un termine R tale che:

$$P \ -\!-\!>> \ R \text{ e } Q \ -\!-\!>> \ R \tag{7.29}$$

Essenzialmente, quindi, il significato del teorema è che se con una riduzione in 0 o più passi di un termine M, arrivo a un termine P e a un altro termine Q, allora posso *confluire* sia da P che da Q a un termine R.
Il corollario che nasce da questo teorema è quello sull'*unicità delle forme normali*, che dice:

Corollario 7.3.1. *Se un termine ha una forma normale, allora essa è **unica**.*

7.3.5 Tipi di Dato nel λ-calcolo

Dopo aver analizzato come è composto il λ-calcolo, e come avvengono le computazioni nello stesso, ci chiediamo: *riusciamo a rappresentare nel λ-calcolo tutte le computazioni effettive*? Ovvero, tale formalismo mi da la *garanzia di rappresentare qualsiasi algoritmo*? La risposta è che tale garanzia non è fornita da alcun formalismo, ovvero da nessun modello computazionale. L'unica garanzia che abbiamo,

è che **le computazioni** che possono essere rappresentate **in altri modelli computazionali**, possono essere **rappresentate anche nel λ-calcolo**. Poichè, quindi, il λ-calcolo ha la stessa rappresentabilità degli algoritmi degli altri modelli, allora è possibile rappresentare anche nel λ-calcolo tutti gli *algoritmi effettivi*. Poichè, quindi, abbiamo detto che nel λ-calcolo non vi è distinzione fra *dato* e *programma*, o definizione di funzione, possiamo trovare dei modi per rappresentare nel λ-calcolo dei tipi di dato come i *boolean*, i *numeri naturali*, le *liste*...

Booleani

Nella rappresentazione dei booleani, il nostro obiettivo è **rappresentare vero e falso**, ovvero un *insieme di 2 elementi distinguibili l'uno dall'altro*. Per cui possiamo dare la seguente definizione:

$$\begin{aligned} T &= \lambda xy.x \\ F &= \lambda xy.y \end{aligned} \tag{7.30}$$

Essenzialmente, quindi, definiamo *true* la funzione che restituisce il primo elemento, mentre *false* la funzione che restituisce il secondo. Una volta definiti i *booleani*, possiamo definire operazioni come l'**and**, **or** ecc. Ad esempio l'operazione **and** è definita come $\lambda ab.ab\ F$. Il costrutto *if*, ad esempio, è definito come:

$$\textbf{if\_then\_else} = \lambda x.x \tag{7.31}$$

ovvero l'*algoritmo identità* per il quale:

- if_then_else $TMN- >>_\beta M$
- if_then_else $FMN- >>_\beta N$

Numeri Naturali

Per quanto riguarda i **numeri naturali**, esistono moltissimi modi di rappresentare tale dato. Il più diffuso è quello dei **numerali di Church**, che rappresentano un λ-termine o algoritmo, che dato uno '0' e la *funzione successore* ci restituiscono il numero cercato. La convenzione vuole che per ogni numero naturale n, il corrispondente numerale di church è indicato con $\bar{n}$, in cui la funzione successore è rappresentata come $\bar{n} = \lambda fx.f^n x$. Quindi, per esempio, abbiamo:

$$\begin{aligned} \bar{0} &= \lambda fx.x \\ \bar{1} &= \lambda fx.fx \\ \bar{2} &= \lambda fx.f(f(x)) \end{aligned} \tag{7.32}$$

Quindi, presa la funzione successore f e preso il nostro x, ovvero il simbolico '0', riusciamo a rappresentare i naturali tramite tale funzione. Tramite tali strumenti possiamo rappresentare anche tutte le operazioni matematiche, come per esempio l'*addizione* oppure la *moltiplicazione*:

$$\begin{aligned} \textbf{add} &= \lambda nmfx.nf(mfx) \\ \textbf{mult} &= \lambda nmf.n(mf)x \end{aligned} \tag{7.33}$$

7.3.6 Punti fissi e Funzioni Ricorsive

Abbiamo visto che, uno degli elementi necessari per programmare in Haskell poteva essere la **ricorsione**. Per esempio definivamo la *funzione fattoriale* in questo modo:

```
fact = \n -> if(n==0) then 1 else n*(fact(n-1))
```

Proviamo quindi a trovare il λ-termine per rappresentare il fattoriale:

$$\lambda n.if(iszero\ n)(\lambda fx.fx)(mult\ n(fact\ n-1)) \tag{7.34}$$

In questo caso abbiamo utilizzato una funzione $iszero$ che ritorna true se l'argomento è 0, altrimenti false. Inoltre abbiamo utilizzato la funzione $mult$ per il prodotto, e il termine $\lambda fx.fx = \bar{1}$. Si pone adesso il problema che $fact$ deve essere rimpiazzato da qualcos'altro. Ovvero per fact devo cercare un λ-termine che abbia **significato computazionale uguale** all'espressione stessa.

β-conversione

Prima di vedere come possiamo trovare dei termini con lo stesso significato computazionale, bisogna introdurre il concetto di β-**conversione**. Diciamo che, dati 2 termini M, M' essi si dicono β-convertibili, scrivendo

$$M =_\beta M' \tag{7.35}$$

se si ottengono l'uno dall'altro tramite β-*riduzioni* oppure β-*espansioni* (ovvero l'opposto della β-riduzione). In pratica 2 termini sono β-convertibili se hanno la *stessa informazione computazionale*, ovvero **rappresentano lo stesso algoritmo**. Quindi, poichè la definizione di algoritmo *dipende dal modello computazionale* (e molto spesso quindi si da una definizione poco precisa), nel λ-calcolo si basa sulla β-conversione.

Operatori di Punto Fisso

Tornando al fattoriale, dobbiamo quindi trovare un termine β-convertibile con *fact*. In matematica diremmo che dobbiamo trovare un **punto fisso della funzione**: si dice che data una funzione $g: A \to A$ e preso un punto $a \in A$, esso si dice punto fisso della funzione g se $g(a) = a$. Per quanto riguarda i λ-termini diciamo quindi che il termine N è un punto fisso di F se

$$N =_\beta FN \tag{7.36}$$

ovvero è un λ-termine che applicato a un qualcosa mi da un valore che è esattamente quello dell'*argomento della funzione*. Inoltre vi è un teorema che ci garantisce che nel λ-calcolo **ogni termine ha un punto fisso**, ed è di conseguenza β-convertibile. Per calcolare tali punti fissi, si utilizzano operatori di punto fisso Y tali quindi che:

$$(YP) =_\beta P(YP) \tag{7.37}$$

Tornando quindi alla nostra funzione fattoriale, possiamo riscrivere la funzione come:

$$fact = \lambda f.\lambda x.if(x = 0)\ then\ 1\ else\ (x * f(x-1))fact \tag{7.38}$$

Se pensiamo quindi che $P = \lambda f.\lambda x.if(x = 0)\ then\ 1\ else\ (x * f(x-1))$, possiamo dire che $fact$ è il punto fisso di P, ovvero Possiamo quindi dire che:

$$fact =_\beta Pfact \tag{7.39}$$

ovvero il punto fisso $fact$ applicato al termine P mi da il valore dell'argomento della funzione P, ovvero $fact$.

7.4 Corrispondenza Deduzioni-Programmi

Una volta affrontato lo studio del modello computazionale del λ-calcolo, adesso abbiamo degli strumenti che riescono a darci un idea della **corrispondenza fra deduzioni e programmi**, che è anche uno degli obiettivi di tale corso. Il nostro scopo è adesso quindi di far notare come vi sia un *contenuto computazionale in tutte le dimostrazioni*.

7.4.1 Deduzione Naturale e λ-calcolo

Per evidenziare la corrispondenza tra deduzioni e programmi, prendiamo quindi in considerazione le **deduzioni della logica proposizionale in deduzione naturale** e i **programmi come termini del λ-calcolo**, che sappiamo essere effettivamente dei programmi. In particolare, consideriamo, per semplicità, solo il frammento dell'implicazione per le deduzioni, con le sue rispettive regole. L'associazione fra un termine M del λ-calcolo e una deduzione della fbf σ come:

$$M : \sigma \tag{7.40}$$

Computazione e Semplificazione

Far vedere che esiste un associazioni e programmi significa anche confrontare il concetto di **computazione**, che nel λ-calcolo coincide con la relazione di β-riduzione, con il concetto di **semplificazione**, infatti stiamo trasformando un informazione "semplificandola" dalla sua forma implicita a quella esplicita. Nelle deduzioni esiste tale concetto di *semplificazione*. Un esempio di deduzione che può essere semplificata è il seguente: Se avessimo una deduzione che introduce ed elimina la stessa regola, allora si tratterà di un *ragionamento superfluo*, ovvero che non aggiunge niente alle conoscenze che già avevo.

7.4.2 Associazione di deduzioni e programmi

Una volta compresi gli elementi costitutivi di deduzioni e programmi, definiamo la loro **associazione**, ovvero la costruzione di una deduzione con la costruzione di un λ-termine in tal modo:

- Ad ogni **ipotesi** associamo una **variabile** del λ-calcolo.
- Ogni volta che utilizziamo la regola $(\to E)$, ovvero l'**eliminazione dell'implicazione**, associamo l'**applicazione** dei 2 termini che sono premesse della regola. Quindi, presi i termini M, N come premesse della regola, possiamo vedere tale associazione come:

$$\frac{M:\ \alpha \to \beta \qquad N:\alpha}{(MN):\beta}$$

- Ogni volta che utilizziamo la regola $(\to I)$, ovvero l'**introduzione dell'implicazione**, associamo l'**astrazione** del termine associato alla premessa della regola rispetto alla variabile corrispondente alle ipotesi (o alla singola ipotesi) che vengono scaricate. Possiamo vedere tale associazione come:

$$\frac{\begin{array}{c}[x:\ \alpha]\\ \vdots \\ M:\ \beta\end{array}}{\lambda x.M:\ \alpha \to \beta}$$

Grazie a tali associazioni, possiamo prendere qualsiasi deduzione e seguendo passo passo tali istruzioni costruire il *programma* corrispondente, ovvero il λ-termine corrispondente con poca fatica.

7.4.3 Comportamento di Deduzioni e Programmi

Ciò che abbiamo fatto, quindi, è associare quindi la costruzione di una deduzione alla costruzione di un programma. Ciò, però, non basta per dimostrare che i ragionamenti logici siano effettivamente programmi: infatti le deduzioni dovrebbero **comportarsi allo stesso modo dei programmi**. Adesso diamo una spiegazione proprio per questo fatto. Prendiamo la seguente deduzione:

$$\frac{\dfrac{\begin{array}{c}[\alpha]\\ \vdots \\ \beta\end{array}}{\alpha \to \beta}\,(\to I) \qquad \begin{array}{c}\vdots \\ \alpha\end{array}}{\beta}\,(\to E)$$

In tale deduzione possiamo notare che stiamo facendo dei *giri di ragionamento inutile*, infatti stiamo introducendo un implicazione per poi eliminarla. Per eliminare tale ragionamento inutile, possiamo utilizzare l'albero di deduzione che conclude con α, senza avere la necessità di scaricarla come ipotesi, avendo una deduzione del tipo:

$$\begin{array}{c}\vdots \\ \alpha \\ \vdots \\ \beta\end{array}$$

Abbiamo quindi fornito una **versione semplificata della deduzione**. Vediamo adesso associando i λ-termini alla deduzione, a cosa corrisponde dal punto di vista computazionale sia la versione non semplificata che quella semplificata della precedente deduzione:

$$\dfrac{\dfrac{\begin{array}{c}[x:\ \alpha]\\ \vdots\\ M:\ \beta\end{array}}{\lambda x.M:\ \alpha \to \beta}\,(\to I) \qquad \begin{array}{c}\vdots\\ N:\ \alpha\end{array}}{(\lambda x.M)N:\ \beta}\,(\to E)$$

Notiamo che la conclusione di tale deduzione è un **redex**. Analizzando la versione semplificata, ovvero sostituendo al posto delle ipotesi l'albero di deduzione che conclude con α, che corrisponde a sostituire il corrispondente λ-termine N al posto della variabile x associata all'ipotesi che viene scaricata, il risultato è il seguente:

$$\begin{array}{c}\vdots\\ N:\ \alpha\\ \vdots\\ M[N/x]:\ \beta\end{array}$$

Il risultato straordinario è che la **semplificazione della deduzione** corrisponde alla **trasformazione di β-riduzione**, ovvero la nozione di computazione nel λ-calcolo. Per cui possiamo affermare che il concetto di computazione come *trasformazione dell'informazione da una forma implicita a una esplicita*, corrisponde, dal punto di vista logico, alla *semplificazione di ragionamenti deduttivi*. Quindi possiamo finalmente affermare che i programmi e le computazioni da una parte, e le dimostrazioni e le semplificazioni di esse dall'altra sono *intrinsecamente la stessa cosa*. Ovviamente questa è solo un'idea della corrispondenza deduzioni-programmi, infatti per sviscerare bene tale argomento si richiederebbe un intero corso e, sfortunatamente, non vi è tempo.

Parte IV

Semantica Formale e Macchine Astratte

Capitolo 8

Semantica Formale dei Linguaggi di Programmazione

Quando un nuovo linguaggio di programmazione viene creato la prima cosa che si fa è *definirne la **sintassi***. Questo equivale a *descrivere un linguaggio formale*, ovvero un sottinsieme $L \subseteq \Sigma^*$ che definisce tutte le stringhe corrette per la sintassi di quel linguaggio. L'altra componente fondamentale in un linguaggio di programmazione è la sua **semantica**. Infatti, quando un programma viene compilato non solo si verifica la correttezza sintattica, ma un compilatore ben costruito dovrebbe sapere precisamente quale sia la **semantica formale del linguaggio**, che ci porta i vantaggi di avere un linguaggio che sia poco ambiguo e che non sia interpretabile. Fornire una semantica formale, quindi, comporta dare una serie di definizioni matematiche che definiscono i significati di ogni istruzione che scriviamo in un determinato linguaggio di programmazione.

Approcci per descrivere la semantica formale di un linguaggio Per descrivere la semantica formale di un linguaggio di programmazione esistono principalmente tre tipi di approcci:

- **Semantica denotazionale**: questo approccio consiste nel descrivere la semantica dei programmi tramite funzioni matematiche, viste come relazioni di input-output
- **Semantica Assiomatica**: descrive la semantica definendo un sistema formale le quali fbf sono del tipo:
$$\{A\}P\{B\} \tag{8.1}$$
dove $\{A\}$ raprresenta un affermazione sul contenuto della memoria a inizio programma, P rappresenta l'esecuzione del programma vero e proprio e $\{B\}$ rappresenta il contenuto della memoria al termine del programma. Tale semantica viene descritta in sistemi formali come quelli della *logica di Hoare*, *separation logic* ecc.

- **Semantica Operazionale**: tale semantica ha lo scopo di definire un sistema formale che descrive in modo preciso il *passo di computazione* quando eseguiamo un programma la cui semantica stiamo definendo su una *macchina astratta*

La *semantica operazionale* è il tipo di semantica che utilizzeremo per descrivere un piccolo linguggio, ovvero il *linguaggio while*, che ci servirà come esempio.

8.1 Semantica Formale del linguaggio while

Per descrivere al meglio i concetti di cui prima abbiamo discusso prendiamo come linguaggio d'esempio il **linguaggio while**, che è un semplice *linguaggio imperativo*. Prima di descriverne la semantica formale, però, dobbiamo descriverne la *sintassi*, ovvero definire il sottoinsieme di Σ^* che corrisponde alle sequenze di caratteri sintatticamente corretti. A questo scopo utilizziamo la *grammatica* con il quale definiamo il linguaggio, che ci dice come scrivere un programma sintatticamente corretto. La grammatica che genera il linguaggio è la seguente:

$$
\begin{array}{rcl}
< digit > & ::= & 0|1|2|...|9 \\
< variabile > & ::= & \mathbf{X} < indice > \\
< indice > & ::= & < digit > | < indice >< digit > \\
< assegnazione > & ::= & < variabile >:= \mathbf{0} | < variabile >:= \textbf{succ}(< variabile >) | < variabile >:= \textbf{pr} \\
< test > & ::= & < variabile > \neq < variabile > \\
< istruzione > & ::= & < assegnazione > | \textbf{while} < test > \textbf{do} < istruzione > | \textbf{while} < test > \textbf{do} < \\
< sequenza > & ::= & < istruzione > | < istruzione >; < sequenza > \\
< programma > & ::= & \textbf{begin end} | \textbf{begin} < sequenza > \textbf{end}
\end{array}
\tag{8.2}
$$

Essenzialmente, quindi, tale grammatica ci dice come costruire stringhe sintatticamente corrette nel linguaggio while e, le parole in grassetto rappresentano l'alfabeto Σ sul quale il linguaggio while è costruito. Per esempio ci dice come costruire un digit, ovvero una cifra e, di conseguenza un indice, che rappresenterà un numero ecc. Inoltre possiamo interpretare il simbolo '|' come *alternativa* nella generazione di un determinato costrutto. Infine possiamo notare che tale linguaggio accetta come tipo di dato esclusivamente numeri interi, infatti, nella grammatica non vi è nessun indicazione sulla generazione di *booleani*, *stringhe* ecc.

8.1.1 Sistema Formale del linguaggio while

Una volta definita la sintassi formalmente, descrivendo la grammatica che genera tale linguaggio. Noi in questo caso utilizzeremo lo stile della *semantica operazionale strutturata*. A tal scopo è necessario definire un **sistema formale** che corrisponde alla definizione di semantica operazionale per tale linguaggio. In particolare dobbiamo definire l'insieme fbf del nostro sistema formale e darne una *nozione di validità*, ovvero il significato che assumono. Le fbf vengono definite come segue:

$$\vec{a}, P \Downarrow \vec{b} \tag{8.3}$$

dove $\vec{a}, \vec{b}$ rappresentano **vettori di numeri interi**, P rappresenta il **programma** e il simbolo '$\Downarrow$' è chiamato **simbolo di terminazione**. Per dare una nozione di validità utilizziamo un esempio:

$$(3,2), \textbf{begin} \ldots \textbf{end} \Downarrow (4,2) \tag{8.4}$$

Tale fbf ha il significato seguente: se eseguiamo il programma ***begin*** $\ldots$ ***end*** partendo da uno **stato delle variabili** definito dal vettore $(3,2)$, ovvero prima di eseguire il programma le variabili contengono i valori del vettore $\vec{a}$, una volta eseguito il programma ottengo un nuovo stato delle variabili $(4,2)$. Ciò significa che $\vec{a}$ rappresenta il contenuto delle variabili prima dell'esecuzione del programma. Una volta finita l'esecuzione del programma, indicata con il simbolo di terminazione $\Downarrow$, allora il loro contenuto sarà definito nel vettore $\vec{b}$.

Assiomi e Regole del sistema formale

Le sole fbf non possono essere però considerate come strumento per definire la corretta semantica formale di un linguaggio di programmazione. Abbiamo infatti bisogno di regole e assiomi che ci permettano di derivare *fbf* che siano semanticamente corrette nel linguaggio while. Vediamo quindi tali assiomi e regole:

- Il primo assioma, che riguarda un programma con sequenza vuota di istruzioni è il seguente:

$$\frac{}{\vec{a}, \textbf{begin end} \Downarrow \vec{a}}$$

  Essendo assioma, possiamo dire che posso dire, senza affermare alcuna premessa, che dato un vettore $\vec{a}$ come stato iniziale del programma, ed eseguito il programma ***begin end***, tale programma non modifica lo stato delle variabili, che dunque alla fine della computazione rimarrà $\vec{a}$. Essenzialmente, quindi, tale assioma descrive il programma che non fa nulla

- Adesso vediamo una regola che riguarda *programmi con sequenze di istruzioni*:

$$\frac{\vec{a}, \textbf{begin } \delta_1 \textbf{ end} \Downarrow \vec{a'} \qquad \vec{a'}, \textbf{begin } \delta_2; \ldots; \delta_n \textbf{ end} \Downarrow \vec{b}}{\vec{a}, \textbf{begin } \delta_1; \ldots; \delta_n \textbf{ end} \Downarrow \vec{b}}$$

  A differenza di come facevamo nei sistemi formali, in tal caso conviene leggere le regole dal basso verso l'alto. In tal caso stiamo dicendo che possiamo concludere che un programma che ha stato iniziale delle variabili $\vec{a}$, che esegue una sequenza di istruzioni $\delta_1; \ldots; \delta_n$ termina con uno stato $\vec{b}$ se il programma che ha stato iniziale $\vec{a}$ ed esegue l'istruzione δ_1 conclude con uno stato $\vec{a'}$ e, il programma che ha stato iniziale $\vec{a'}$ e che esegue la sequenza di istruzioni $\delta_2; \ldots; \delta_n$ termina con lo stato $\vec{b}$. Detta in termini più semplici tale regola ci dice *l'esecuzione di una sequenza di istruzioni corrisponde ad eseguire le istruzioni una dopo l'altra*.

- Vediamo adesso una serie di assiomi che riguardano *programmi con una singola istruzione*:
  - Il primo assioma è il seguente:

    $$(a_1, ..., a_i, a_{i+1}, ..., a_k), \textbf{begin Xi } := \textbf{0 end} \Downarrow (a_1, ..., 0, a_{i+1}, ..., a_k)$$

    Il significato di questa regola è dunque che se partiamo da uno stato delle variabili $\vec{a} = (a_1, ..., a_i, a_{i+1}, ..., a_k)$ ed eseguiamo il programma composto dalla singola istruzione **begin Xi** $:=$ **0 end**, avremo uno stato finale $\vec{b} = (a_1, ..., 0, a_{i+1}, ..., a_k)$. Detto in termini più semplici stiamo eseguendo il programma che assegna 0 a una variabile, per cui lo stato finale delle variabili sarà uguale tranne per la i-esima variabile al quale assegniamo il valore 0.
  - Il secondo assioma è il seguente:

    $$(a_1, ..., a_i, a_{i+1}, ..., a_k), \textbf{begin Xi } := \textbf{succ(Xj)} \Downarrow (a_1, ..., a_{j+1}, a_{i+1}, ..., a_k)$$

    In tal caso possiamo vedere tale assioma che se abbiamo uno stato delle variabili $(a_1, ..., a_i, a_{i+1}, ..., a_k)$ ed eseguito il programma composto dalla singola istruzione **begin Xi** $:=$ **succ(Xj)**, avremo uno stato finale delle variabili $(a_1, ..., a_{j+1}, a_{i+1}, ..., a_k)$. Essenzialmente quindi diciamo che eseguendo questa singola istruzione in un programma nel quale eseguiamo l'istruzione che incrementa la j-esima variabile di 1, e assegna tale valore alla i-esima.

- Vediamo adesso una regola riguardante l'istruzione ***while***<*test*>***do***<*istruzione*>. In tal caso imponiamo che la condizione che deve essere soddisfatta affinchè possa essere applicata la regola è che $\mathbf{a_i} \neq \mathbf{a_j}$. La regola è la seguente:

  $$\frac{\vec{a}, \textbf{begin } \vec{\delta} \textbf{ end} \Downarrow \vec{b} \qquad \vec{b} \textbf{ begin while Xi} \neq \textbf{ Xj do } \vec{\delta} \textbf{ end} \Downarrow \vec{c}}{\vec{a} \textbf{ begin while Xi} \neq \textbf{ Xj do } \vec{\delta} \textbf{ end} \Downarrow \vec{c}}$$

  Tale regola ci dice quindi che possiamo affermare che il programma **begin while Xi** $\neq$ **Xj do**$\vec{\delta}$ **end** dove $\vec{\delta}$ rappresenta una *sequenza di istruzioni* che inizia con uno stato $\vec{a}$, termina con uno stato $\vec{c}$, solo se il programma **begin** $\vec{\delta}$ **end** che ha uno stato iniziale delle variabili $\vec{a}$ termina con uno stato delle variabili $\vec{b}$, e il programma **begin while Xi** $\neq$ **Xj do**$\vec{\delta}$ **end** che comincia con uno stato delle variabili $\vec{b}$ termina con $\vec{c}$. Tutto ciò ovviamente se e solo se è soddisfatta la condizione per cui $a_i \neq a_j$ Essenzialmente tale regola ci da la semantica formale del while per il linguaggio. Ovvero fino a quando il test è soddisfatto vengono eseguite le istruzioni presenti dopo il ***do***..., ovvero la sequenza $\vec{\delta}$.

- Vediamo adesso un assioma concernente sempre la singola istruzione ***while***...***do***... nel caso in cui il test non è soddisfatto. La condizione per cui può essere applicato tale assioma è che $\mathbf{a_i} = \mathbf{a_j}$. L'assioma è il seguente:

$$\frac{}{\vec{a}\ \textbf{begin while Xi} \neq\ \textbf{Xj do}\ \vec{\delta}\ \textbf{end} \Downarrow \vec{a}}$$

In tal caso quindi, poichè abbiamo la condizione $a_i = a_j$ il test risulta falso. Proprio per questo il programma restituisce lo stesso stato, ovvero non viene modificato nulla proprio perchè il test non è soddisfatto.

Con tali assiomi e regole abbiamo quindi descritto quello che fa una macchina astratta quando eseguo dei programmi a cui voglio dare tale semantica. Utilizzando inoltre tali assiomi e regola posso dimostrare formalmente proprietà dei programmi che eseguo. In tal caso dovremo derivare la fbf senza ipotesi corrispondente al programma la cui proprietà vogliamo dimostrare a partire da assiomi o regole le cui premesse sono state affermate in precedenza.

Programmi che non Terminano

Abbiamo fino ad ora definito l'esecuzione di programmi che terminano. Ma come vediamo se *un programma non termina*? Molto semplicemente, se non riesco a derivare la fbf che rappresenta il mio programma come teorema, allora il programma non terminerà, ovvero andrà in un loop infinito.

Capitolo 9

Macchine Astratte

Introduciamo adesso il concetto di **macchina astratta**, che è quello strumento utile a noi informatici per descrivere e maneggiare sistemi di calcolo complessi. Proprio perchè tali sistemi sono *complessi*, occorre descriverli in *livelli di astrazione*. Possiamo fare un parallelismo col corpo umano, che è un sistema talmente complesso da non poter essere descritto così facilmente. Infatti suddividiamo la descrizione di apparati, organi, tessuti, cellule... La macchina astratta, quindi, ha proprio lo scopo di descrivere astrattamente una macchina che esegue dei programmi scritti in un certo linguaggio. Essenzialmente, quindi, rappresentano un *modo di descrivere un linguaggio di programmazione*. Noi ci concentreremo sulle *macchine astratte imperative*, ovvero quelle macchine astratte sul quale girano programmi scritti con linguaggi imperativi.

9.1 Il concetto di Macchina Astratta

Andiamo adesso a formalizzare quanto detto nell'introduzione dando una **definizione di macchina astratta imperativa**, che formalmente è:

un insieme di strutture dati ed algoritmi, o procedure effettive, in grado di memorizzare ed eseguire programmi

Proprio perchè stiamo dando una descrizione di un linguaggio di programmazione, ogni linguaggio ha la sua macchina astratta, ma, come abbiamo detto nell'introduzione, noi ci concentreremo sulle macchine astratte imperative. Tali macchine sono costituite dalle seguenti componenti principali:

- Una **memoria**, ovvero un insieme di strutture dati (vettori, liste...) atte a "*memorizzare*" istruzioni e dati. Abbiamo visto ad esempio nella sezione riguardante il *linguaggio while* (sezione 8.1) che indicavamo la memoria con dei vettori indicizzati.
- Un insieme di **operazioni primitive**, ovvero un insieme di *algoritmi* per calcolare le funzioni che sono indicate nel linguaggio

- Un insieme di *algoritmi* e *strutture dati* per gestire il **flusso di controllo**, ovvero algoritmi per gestire l'esecuzione delle istruzioni di un programma nell'ordine corretto. Abbiamo visto nel linguaggio while che la sua semantica era definita in modo tale che eseguire una sequenza di istruzioni $\delta_1, ..., \delta_n$ corrispondeva ad eseguire le istruzioni una per volta a partire da δ_1 finendo con δ_n. La struttura dati che in questo caso ci aiuta a mantenere il giusto ordine di esecuzioni delle istruzioni sarà un *puntatore* all'istruzione successiva, ovvero quello che all'interno del processore è chiamato *registro PC*.

- Un insieme di *algoritmi* e *strutture dati* che gestiscono il **controllo del trasferimento dei dati**, ovvero *algoritmi* e *strutture dati* che recuperano gli operandi e li memorizzano i risultati delle varie istruzioni. Nel linguaggio while per esempio leggevamo i valori che ci interessavano da un vettore.

- Un insieme di *algoritmi* e *strutture dati* per la **gestione della memoria**

- Un **interprete**: si tratta di un unico algoritmo che da alla macchina astratta la capacità di eseguire programmi. Infatti tale componente *coordina il funzionamento delle altre componenti* (su macchine astratte imperative). Il coordinamento di tutte le componenti avviene tramite l'esecuzione di un algoritmo che esegue un *ciclo composto da una serie di fasi*, chiamato **ciclo fetch-execute** che si ferma con una particolare istruzione primitiva detta *halt*. Il ciclo è composto dalle seguenti fasi:

  1. Inizialmente avviene l'**acquisizione della prossima istruzione da eseguire**, mediante l'utilizzo delle strutture dati e algoritmi atti a governare il *controllo di sequenza*. Tale fase è chiamata *fetch*

  2. A tal punto avviene la **decodifica dell'istruzione** tramite gli algoritmi e strutture dati per il *controllo sul trasferimento dati* che acquisiscono gli eventuali operandi

  3. A questo punto viene **eseguita l' operazione primitiva** e l'eventuale risultato viene memorizzato. Se l'operazione eseguita non è quella che fa arrestare l'interprete, il ciclo ricomincia.

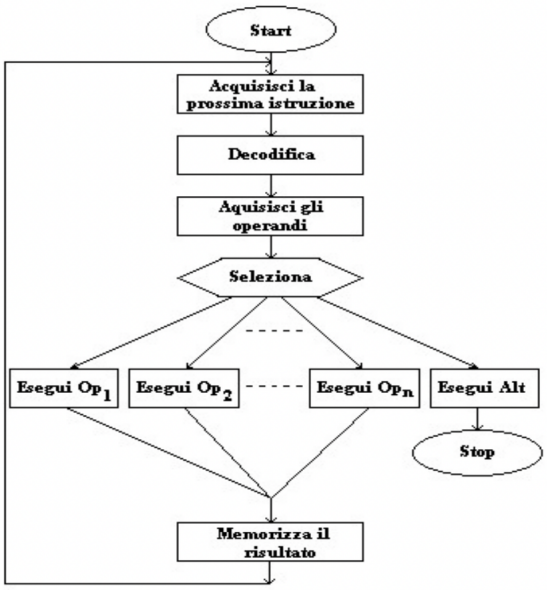


Figure 9.1: Il ciclo Fetch-Execute

9.1.1 Linguaggi di Programmazione e Macchine Astratte

Come abbiamo già accennato, possiamo dire che una **macchina astratta**, che per definizione esegue programmi memorizzati al suo interno, è sempre associata a un **linguaggio di programmazione**. In particolare il linguaggio che è associato alla macchina, è il linguaggio con cui si esprimono tutti i programmi interpretabili dalla macchina. Possiamo inoltre dire che i programmi scritti in un determinato linguaggio associato alla macchina, non sono altro che dati primitivi su cui lavora l'interprete della macchina stessa. Possiamo quindi convincerci che dato un linguaggio di programmazione è definita la sua macchina astratta associata.

9.2 Realizzazione di Macchine Astratte

Abbiamo trattato le macchine astratte dicendo che sono un insieme di *algoritmi* e *strutture dati* che sono *astratte*. Ma come faccio a **implementare una macchina astratta**? Ovvero come faccio ad eseguire i programmi scritti nel linguaggio associato alla macchina? E' possibile utilizzare 3 tecniche:

- realizzazione **hardware**
- realizzazione **interpretativa**
- realizzazione **compilativa**

9.2.1 Realizzazione Hardware

Possiamo affermare che ogni struttura dati e algoritmo che ci venga in mente può essere realizzato tramite una *serie di circuiti*. E' comprensibile, però, che solitamente questo tipo di realizzazione di macchine astratte è di solito utilizzato per macchine astratte molto semplice, come ad esempio la macchina astratta associata al linguaggio macchina. Infatti, implementare la macchina astratta per linguaggi complessi come *C++*, *Java* o altri linguaggi di alto livello, è molto complesso da realizzare in hardware proprio per la poca flessibilità di quest'ultimo e per gli elevati costi di progettazione.

9.2.2 Realizzazione Interpretativa

La **realizzazione interpretativa** di una macchina astratta è un implementazione non hardware. Inoltre tale tecnica richiede l'esistenza di una macchina astratta già implementata. Se consideriamo MA_1 la macchina astratta già implementata al quale è associato il linguaggio L_1; e consideriamo MA_2 la macchina astratta da implementare al quale associamo il linguaggio L_2; allora possiamo dire che la realizzazione interpretativa di MA_2 consiste nell'implementare le sue strutture dati tramite strutture dati di MA_1, e implementare i suoi algoritmi tramite programmi scritti in L_1, ovvero il linguaggio associato alla macchina già implementata MA_1. Se quindi per esempio dobbiamo implementare la struttura dati *lista* in MA_2, e, nella macchina già implementata è presente la struttura dati *vettore*, allora dovrò implementare la lista tramite vettori utilizzando, per esempio, il puntatore all'elemento successivo. La stessa cosa vale per l'implementazione degli algoritmi: se per esempio vogliamo implementare tra gli algoritmi di MA_2, un algoritmo che calcoli il determinante di una matrice, allora utilizzerò un programma scritto in L_1, ovvero il linguaggio associato a MA_1 che calcoli il determinante.

9.2.3 Realizzazione Compilativa

Anche la **realizzazione compilativa** di una macchina astratta, come la realizzazione interpretativa, è una realizzazione non hardware che richiede l'esistenza di una macchina astratta già implementata. Se consideriamo MA_1 la macchina astratta già implementata al quale è associato il linguaggio L_1; e consideriamo MA_2 la macchina astratta da implementare al quale associamo il linguaggio L_2; allora possiamo dire che la realizzazione compilativa di MA_2, consiste nello scrivere un *programma* ***compilatore*** che traduca un programma scritto in L_2, in un programma semanticamente uguale sulla relazione di input-output nel linguaggio L_1, ovvero quello associato alla macchina già implementata. Ciò significa che dato un input uguale per entrambi i linguaggi, avremo lo stesso output.

9.3 Macchine Intermedie e Struttura a Livelli dei computer moderni

Molto spesso, quando deve essere realizzata una macchina molto complessa vi è un problema tra il livello di espressività molto alto che vogliamo dare a tale macchina, e il livello di espressività della macchina già implementata che abbiamo a disposizione. Tale ***semantic gap***, ovvero tale differenza di espressività, comporta il fatto che macchine molto complesse non vengano realizzate a partire da macchine astratte elementari già implementate, ma siano il risultato di un *estensione di macchine intermedie*, che hanno un semantic gap minore con la nostra macchina da realizzare. Quindi, proprio per questo motivo, è solito che nell'implementazione di una nuova macchina sia il risultato di una **gerarchie di macchine astratte**, in cui la nuova macchina astratta è, praticamente, la macchina astratta precedente, però con una piccola estensione, che viene realizzata su compilazione o interpretazione di macchine che stanno al di sotto in tale gerarchia. L'esempio più immediato che possiamo dare per il sistema a livelli, è quello presente nel nostro computer, che è benomale un sistema come nella seguente figura:

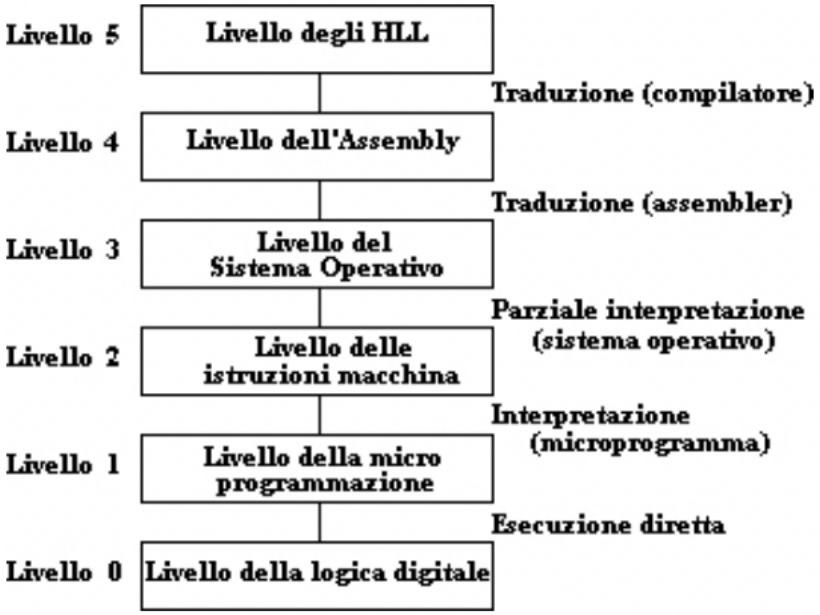


Figure 9.2: Un esempio di gerarchia di macchine astratte

In tal senso possiamo vedere anche l'attività di programmazione come un implementazione di nuovi livelli di calcolo, ovvero un estensione di una macchina astratta. Infatti quando programmiamo, implementiamo una nuova operazione della macchina astratta per via interpretativa al quale aggiungiamo quella determinata funzione che

abbiamo programmato.

Acronyms

ASFD Automi a Stati Finiti Deterministici. 19, 20, 22–25, 27, 29–31

ASFND Automi a Stati Finiti Non Deterministici. 3, 26–31

fbf Formule ben Formate. 49, 74–77, 80, 110, 111, 113

Bibliography

[1] Giorgio Ausiello, Fabrizio d'Amore, Giorgio Gambosi, and Luigi Laura. *Linguaggi modelli complessità*. F. Angeli, 2003.