

# Metodi formali dell'informatica

Luca Barra

Anno accademico 2023/2024



# INDICE

<b>CAPITOLO 1</b>	<b>INTRODUZIONE</b>	<b>PAGINA 1</b>
1.1	Cosa sono e a cosa servono i metodi formali?	1
1.2	La riscrittura Il $\lambda$ -calcolo — 2 • Il $\lambda$ -calcolo tipato — 2	1
1.3	Il problema della verifica La semantica operativa — 2 • Floyd e Hoare — 2 • Verifica e testing — 3 • Limiti teorici — 3	2
1.4	Installare Agda	3
<b>CAPITOLO 2</b>	<b>RISCRITTURA DI TERMINI</b>	<b>PAGINA 4</b>
2.1	La logica equazionale Le variabili — 5	4
2.2	La sostituzione	6
2.3	Il matching	6
2.4	Sistemi di riscrittura	7
2.5	Logica equazionale Normalizzazione — 11	10
<b>CAPITOLO 3</b>	<b>DEDUZIONE NATURALE DI GENTZEN</b>	<b>PAGINA 12</b>
3.1	La deduzione	12
3.2	Congiunzione e implicazione	12
3.3	Vero, falso e negazione	13
3.4	Disgiunzione	13
3.5	Reduction ad absurdum	14
3.6	Quantificatori	14
<b>CAPITOLO 4</b>	<b>IL <math>\lambda</math>-CALCOLO</b>	<b>PAGINA 16</b>
4.1	Introduzione	16
4.2	Il $\lambda$ -calcolo non tipato Semantica — 16 • Numerali di Church — 18	16
4.3	Il $\lambda$ -calcolo tipato Tipi — 21	21
<b>CAPITOLO 5</b>	<b>LOGICA COSTRUTTIVA</b>	<b>PAGINA 22</b>

<b>CAPITOLO 6</b>	<b>IL LINGUAGGIO IMP</b>	<b>PAGINA 23</b>
6.1	Introduzione a IMP Le relazioni in IMP — 23 • La logica di Floyd-Hoare — 24	23
6.2	Espressioni Espressioni aritmetiche — 25 • Sostituzione — 26 • Espressioni booleane — 28	24
6.3	Semantica Big-step Comandi — 29 • Convergenza — 29 • Proprietà della convergenza — 31 • Equivalenza — 33	29
6.4	Semantica Small-step Riduzione in un passo — 34 • Chiusure — 35	34
6.5	Relazione tra semantica Big-step e semantica Small-step Da Small-step a Big-step — 36 • Da Big-step a Small-step — 37	36

<b>CAPITOLO 7</b>	<b>LOGICA DI FLOYD-HOARE</b>	<b>PAGINA 38</b>
7.1	Il sistema della logica di Hoare Regole — 39 • Regole derivate — 40	38
7.2	Esempi Assegnamento — 40 • Composizione — 41 • Selezione — 41	40
7.3	Correttezza	42
7.4	Completezza Weakest Liberal Precondition — 44	44
7.5	Verification Conditions	47



# Capitolo 1

## Introduzione

### 1.1 Cosa sono e a cosa servono i metodi formali?

I metodi formali sono un particolare tipo di *tecnica matematica* per la *specifica*, lo *sviluppo* e la *verifica* dei sistemi software e hardware. Essi includono teorie, metodi e tool che derivano dalla logica matematica:

- Calcoli logici;
- Teoria degli automi;
- Algebra dei processi;
- Algebra relazione;
- Semantica dei linguaggi di programmazione;
- Teoria dei tipi;
- Analisi statica;
- etc..

L'utilizzo dei metodi formali è poter avere uno strumento per analizzare e certificare il software:

- Verifica di SW e HW;
- Documentazione, specifica e sviluppo del software;
- Debugging;
- Monitoring;
- etc..

### 1.2 La riscrittura

La *riscrittura* parte dall'idea di trasformare in una "forma normale" delle proposizioni tramite una serie di trasformazioni (per esempio la doppia negazione che è uguale a un' affermazione o le leggi di De Morgan).

### 1.2.1 Il $\lambda$ -calcolo

Il  $\lambda$ -calcolo è un sistema per calcolare usando le funzioni.

#### Definizione 1.2.1: La sintassi del $\lambda$ -calcolo

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

dove

- $x$  è il parametro formale;
- $\lambda x.M$  è l'astrazione di un termine rispetto a una variabile;
- $M N$  è l'applicazione di  $N$  a  $M$ .

#### Note:-

Tuttavia si può anche assegnare una funzione a una funzione creando problemi, per esempio una ricorsione infinita

### 1.2.2 Il $\lambda$ -calcolo tipato

Il  $\lambda$ -calcolo *tipato* serve per risolvere il precedente problema, introducendo il concetto di tipo. Si introduce una sintassi con *tipi di base* (int, bool, etc.) e *tipi composti* (int  $\rightarrow$  bool, int  $\rightarrow$  int, etc.). Questo definisce il dominio delle funzioni ed è alla base di tutti i sistemi di tipo.

## 1.3 Il problema della verifica

**Dati:** una descrizione concreta di un sistema (es. il codice di un programma) e una *specifica* del suo comportamento o di una sua proprietà.

**Risultati:** un'evidenza del fatto che il codice soddisfa la specifica o un *controesempio*.

#### Note:-

Il problema nasce dal fatto che il programma è un oggetto formale, mentre le specifiche non lo sono sempre (per cui vanno formalizzate)

### 1.3.1 La semantica operativa

La *semantica operativa* definisce il comportamento di un programma e ne modifica il suo stato. Lo stato è un'astrazione della memoria che viene riscritta dal programma.

#### Definizione 1.3.1: La semantica operativa

Uno stato è una mappa dalle variabili ai valori:  $\sigma : Var \rightarrow Var$

$$(P, \sigma) = (P_0, \sigma_0) \rightarrow (P_1, \sigma_1) \rightarrow \dots \rightarrow (P_k, \sigma_k)$$

$P_i$  è la parte che resta da eseguire di  $P_{i-1}$ ,  $\sigma_i$  è lo stato risultante dall'esecuzione della prima istruzione di  $P_{i-1}$  nello stato  $\sigma_{i-1}$ , se  $P_k$  è vuoto allora  $\sigma_k$  è il risultato della computazione

### 1.3.2 Floyd e Hoare

Floyd introdusse il *metodo delle asserzioni* che utilizza formule logiche per arricchire il flusso di un programma. Il problema di questo approccio è che bisogna scrivere le formule e ragionarci sopra in astratto. Hoare propose un *calcolo logico* che utilizza una "pre-condizione" (ipotesi sui dati,  $\phi$ ) e una "post-condizione" (cosa calcola il programma,  $\psi$ ).

**Note:-**

$\{\phi\}P\{\psi\}$  è vera nello stato  $\sigma$  se quando  $\phi$  sia vera in  $\sigma$  e l'esecuzione di  $P$  da  $\sigma$  termini in  $\lambda'$ ,  $\psi$  è vera in  $\sigma'$

**Teorema 1.3.1 Logica di Hoare**

Se la tripla  $\{\phi\}P\{\psi\}$  è derivabile in  $HL^a$  allora è valida

$$\vdash \{\phi\}P\{\psi\} \Rightarrow \models \{\phi\}P\{\psi\}$$

dove  $\{\phi\}P\{\psi\}$  è valida se

$$\forall \sigma. \sigma \models \{\phi\}P\{\psi\}$$

---

<sup>a</sup>Hoare's logic

**1.3.3 Verifica e testing**

Il testing (verifica dinamica) indica che per un certo insieme di valori il programma è corretto. La verifica (statica) indica che il programma è corretto per qualsiasi valore. La verifica non prevede l'esecuzione del programma. Essa deve stabilire se un "contratto" è valido, ossia se le "post-condizioni" siano rispettate partendo dalle "pre-condizioni". L'*invariante di ciclo* è vero sia prima che dopo e bisogna dimostrare che sia uguale per tutte le iterazioni. In un sistema di verifica *model-based* o model checking si costruisce un modello  $M$  del sistema/protocollo e se ne specifica il comportamento con una formula temporale (LTL, CTL, ...)  $\phi$  quindi si stabilisce se  $M$  soddisfa  $\phi$ . La verifica *proof-based* o deduttiva non considera tutti gli infiniti stati ma si dimostra che la relazione di input/output è deducibile da un calcolo logico su un insieme finito.

**1.3.4 Limiti teorici**

- $FOL^1$  è corretta e completa, ma indecidibile;
- $HL$  è corretta, ma completa solo in senso debole ed è indecidibile;
- Il *teorema di Rice* indica che tutte le proprietà funzionali (che dipendono dalla semantica) sono indecidibili o triviali.

**1.4 Installare Agda**

Questa mini guida utilizza Linux, in quanto l'installazione risulta più veloce e semplice.

1. Come prima cosa bisogna installare emacs. Per fare ciò si può usare il proprio gestore di pacchetti con il terminale. Per esempio in ubuntu "sudo apt update" e "sudo apt install emacs";
2. Dopo di ch  si pu  installare Agda con il comando "sudo apt install agda";
3. Creare un file chiamato ".emacs" e copiare il seguente comando "(load-file (let ((coding-system-for-read 'utf-8)) (shell-command-to-string "agda-mode locate"))))".

**Note:-**

In alcune vecchie versioni di Ubuntu potrebbe essere necessario usare "sudo apt install agda-mode"

---

<sup>1</sup>First-order logic



## Capitolo 2

# Riscrittura di termini

### 2.1 La logica equazionale

La logica equazionale è una parte della logica in cui i termini sono delle equazioni.

**Esempio 2.1.1** (Un'equazione)

$$t = s \mid t, s$$

In cui  $t$  e  $s$  sono termini con la stessa signature

**Note:-**

$t, s \in \mathcal{T}_\Sigma$  è l'insieme di tutti i termini con signature  $\Sigma$

#### Definizione 2.1.1: Signature

Una signature  $\Sigma$  è un insieme finito di  $k$  simboli  $\{f_1, \dots, f_k\}$  e di una funzione che assegna a ciascuno di essi un'arietà<sup>a</sup>  $ar : \Sigma \rightarrow \mathbb{N}$

<sup>a</sup>A quanti operandi può essere applicato un operatore

#### Definizione 2.1.2: Insieme dei termini sulla signature $\Sigma$

Se  $f \in \Sigma$  e  $ar(f) = 0$  allora  $f \in \mathcal{T}_\Sigma$

Se  $f \in \Sigma$ ,  $ar(f) = n > 0$  e  $\{t_1, \dots, t_n\} \in \mathcal{T}_\Sigma$  allora  $f(t_1, \dots, t_n) \in \mathcal{T}_\Sigma$

La definizione precedente è induttiva, infatti dà una regola con cui è possibile generare ricorsivamente tutti i possibili termini.

**Esempio 2.1.2** (Generazione induttiva dei numeri naturali)

$$\Sigma_{nat} = \{\text{Zero}, \text{Succ}\}$$

Zero è una costante, quindi ha arietà  $ar(\text{Zero}) = 0$ , mentre l'arietà di Succ è  $ar(\text{Succ}) = 1^a$ . Per costruire l'insieme dei numeri naturali:

$$\mathcal{T}_{\Sigma_{nat}} = \{\text{Zero}, \text{Succ}(\text{Zero}), \text{Succ}(\text{Succ}(\text{Zero}), \dots)\}$$

<sup>a</sup>A ogni valore assegna il suo successore

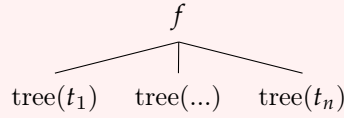
**Note:-**

Si può abbreviare, impropriamente,  $\text{Succ}(\text{Succ}(\text{Zero}))$  con  $\text{Succ}^2(\text{Zero})$

Un termine che viene definito nel precedente modo può essere visto come un albero.

**Definizione 2.1.3: Associazione Termine ::= Albero**

Se si ha un termine ben definito  $\text{tree}(f(t_1, \dots, t_n))^a$  allora si può definire l'albero sintattico




---

<sup>a</sup> $\text{ar}(f) = n$

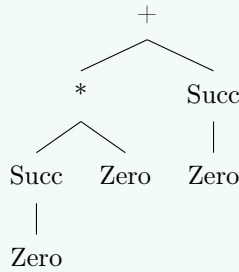
**Esempio 2.1.3** (Conversione da espressione ad albero)

$$\Sigma_{\text{arit}} = \Sigma_{\text{nat}} \cup \{+, *\}$$

con  $\text{ar}(+) = \text{ar}(*) = 2$

$$+(*(\text{Succ}(\text{Zero}), \text{Zero}), \text{Succ}(\text{Zero}))$$

corrisponde all'albero



**Note:-**

$t + s$ , in notazione infissa, corrisponde a  $+(t, s)$  in notazione polacca o prefissa

## 2.1.1 Le variabili

**Esempio 2.1.4** (Differenza di due quadrati)

$$x^2 - y^2 = (x + y) * (x - y)$$

è un esempio interessante poichè si utilizzano *variabili*, per cui per ogni possibile scelta di  $x$  e  $y$  l'equazione è vera

**Definizione 2.1.4: Insieme dei termini**

Dato un insieme infinito di variabili  $X = \{x_0, x_1, \dots\}$ , l'insieme dei termini  $\mathcal{T}_\Sigma(X)$  è:

$$\mathcal{T}_{\Sigma \cup X} \text{ se } \text{ar}(x_i) = 0, x \in \mathcal{T}_\Sigma(X) \quad \forall x \in X$$

### Esempio 2.1.5 (Somma di un successore)

$$\text{Succ}(x) + y = \text{Succ}(x + y)$$

entrambi appartengono a  $\mathcal{T}_{\Sigma \text{arit}}(\{x, y\})$

### Definizione 2.1.5: Le variabili

In generale si possono definire le variabili come:

- $\text{var}(x) = \{x\};$
- $\text{var}(f(t_1, \dots, t_n)) = \bigcup_{i=1}^n \text{var}(t_i).$

### Note:-

Negli alberi le variabili sono le *foglie*

## 2.2 La sostituzione

### Definizione 2.2.1: Sostituzione chiusa

La sostituzione chiusa è una mappa insiemistica  $\sigma$  che assegna a ciascuna variabile un termine nella signature

$$\sigma : X \rightarrow \mathcal{T}_{\Sigma}$$

### Definizione 2.2.2: Sostituzione generale

La sostituzione generale è una mappa insiemistica  $\sigma$  che assegna a ciascuna variabile un termine nella signature in cui si possono avere variabili anche nei termini che si sostituiscono

$$\sigma : X \rightarrow \mathcal{T}_{\Sigma}(X) \quad x \in X \mapsto \sigma(x) \equiv t \in \mathcal{T}_{\Sigma}(X)$$

### Note:-

$t^{\sigma}$  è il risultato della sostituzione in  $t$  di ogni  $x \in \text{var}(t)$  con  $\Sigma(x)$

### Esempio 2.2.1 (Sostituzione)

$t \equiv +(x, *(\text{Succ}(y), x))$ , con  $\sigma(x) = \text{Succ}(\text{Zero})$  e  $\sigma(y) = \text{Zero}$ , allora

$$t^{\sigma} \equiv +(\text{Succ}(\text{Zero}), *(\text{Succ}(\text{Zero}), \text{Succ}(\text{Zero})))$$

### Note:-

Quando si sostituisce manualmente si fa un passo alla volta, ma in realtà la sostituzione di una determinata variabile avviene contemporaneamente in tutta l'equazione (è simultanea)

## 2.3 Il matching

Nelle equazioni quando si applica una formula scoperta a un calcolo particolare bisogna riconoscere che un termine o un sotto-termine è un caso particolare di quella formula. Questo riconoscimento è un matching.

### Definizione 2.3.1: Matching

Dati due termini  $s, t \in \mathcal{T}_\Sigma(X)$ ,  $s$  è istanza di  $t$  se  $s \equiv t^\sigma$  per qualche  $\sigma$ . Dato ciò si può definire:

$$\text{match}(t, p) = \begin{cases} \sigma \text{ tale che } t \equiv p^\sigma \text{ se esiste} \\ \text{fail se } t \text{ non è un'istanza di } p \end{cases}$$

#### Note:-

Si utilizza il simbolo  $p$  come richiamo al fatto che nei linguaggi funzionali si usa il termine "pattern"

### Definizione 2.3.2: Algoritmo per il calcolo del matching

- $\text{match}(t, x) = \{x \mapsto t\}$  caso banale in cui si sostituisce una variabile;
- $\text{match}(t, f(p_1, \dots, p_n)) = \sigma_1 \cup \dots \cup \sigma_n$  se:
  1. se  $t \neq c^a$  allora  $t \neq g(t_1, \dots, t_n)$
  2. se  $t \equiv f(t_1, \dots, t_n)$  allora  $\text{match}(t_i, p_i) = \sigma_i$ , con  $i = 1, \dots, n$ ;
  3.  $\forall x \in \text{var}(t)$  se  $i \neq j$  allora  $\sigma_i(x) \equiv \sigma_j(x)$
- fail in tutti gli altri casi.

---

<sup>a</sup>Costante

## 2.4 Sistemi di riscrittura

### Definizione 2.4.1: Sistema di riscrittura

Fissati  $\sigma$  e  $x$ , un sistema di riscrittura  $R$  è un insieme finito di coppie<sup>a</sup>  $\{l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n\}$  in cui  $l_i, r_i \in \mathcal{T}_\Sigma(X)$ . Le coppie  $(l_i, r_i)$  devono soddisfare (per  $i = \{1, \dots, n\}$ ):

1.  $l_i \notin X$  ( $l_i \neq x \forall x \in X$ )<sup>b</sup>;
2.  $\text{var}(r_i) \subseteq \text{var}(l_i)$ <sup>c</sup>.

---

<sup>a</sup>Regole

<sup>b</sup> $l_i$  non può essere una variabile

<sup>c</sup>Le variabili nella parte destra compaiono anche nella parte sinistra

#### Note:-

$l$  indica il lato *sinistro* (left) della freccia

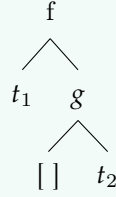
$r$  indica il lato *destro* (right) della freccia

### Definizione 2.4.2: Contesto

Un contesto  $C[\ ]$  può essere un buco  $[\ ]$ , una variabile  $x$  o un termine di arietà  $n$   $f(t_1, \dots, C[\ ], \dots, t_n)$

#### Esempio 2.4.1 (Albero di un contesto)

$f(t_1, g([\ ] t_2))$



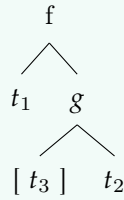
I contesti indicano che le regole di riduzioni vanno applicate in un punto preciso, sotto determinate condizioni.

#### Definizione 2.4.3: Rimpiazzo

Dato  $C[]$  e un termine  $t$ , allora  $C[t]$  si ottiene da  $C[]$  rimpiazzando l'unico buco  $[]$  (se esiste) con  $t$

#### Esempio 2.4.2 (Rimpiazzo)

$f(t_1, g([t_3] t_2))$



#### Asserzione 2.4.1

Un termine  $t$  si riduce in un solo passo a un termine  $s$  ( $t \rightarrow_R s$ ) se esiste un contesto  $C[]$ , una regola  $l \rightarrow r \in R$ , e una sostituzione  $\sigma$  tali che

$$t \equiv C[l^\sigma] \quad \wedge \quad s \equiv C[r^\sigma]$$

ossia  $t$  è un'istanza di  $l$  attraverso  $\sigma$

#### Esempio 2.4.3 (Riscrittura)

$\Sigma = \{a, f, g\}$  con  $ar(a) = 0$ ,  $ar(f) = 1$ ,  $ar(g) = 2$

Si ha il sistema di riscrittura:  $R = \{f(x) \rightarrow a, g(f(x), y) \rightarrow f(y)\}$

Si vuole riscrivere  $g(f(a), f(f(a)))$ . In questo caso si hanno quattro possibili applicazioni delle regole (due producono un risultato identico):

1.  $g(a, f(f(a)))$ 
  - (a)  $g(a, f(a))$ 
    - i.  $g(a, a)$ ;
  - (b)  $g(a, a)$ ;
2.  $g(f(a), f(a))$ 
  - (a)  $g(a, f(a))$ 
    - i.  $g(a, a)$ ;
  - (b)  $g(f(a), a)$ 
    - i.  $g(a, a)$ ;
  - (c)  $f(f(a))$

- i.  $f(a)$   
A.  $a$ ;
- 3.  $f(f(f(a)))$ 
  - (a)  $f(f(a))$ 
    - i.  $f(a)$   
A.  $a$ .
  - (b)  $f(a)$ 
    - i.  $a$ .
  - (c)  $a$ .

Ci possono essere più forme normali, in questo caso sono due:  $g(a, a)$  e  $a$ .

#### Asserzione 2.4.2

Una riduzione in un passo<sup>a</sup>  $\rightarrow R \in \mathcal{T}_\Sigma(X)^2$  è una relazione binaria per cui si può ridurre un termine in un altro

<sup>a</sup>One-step reduction

#### Corollario 2.4.1

$\xrightarrow{+} R$  rappresenta la più piccola riduzione tale che  $\rightarrow R \subseteq \xrightarrow{+} R$  e  $\xrightarrow{+} R$  sia transitiva<sup>a</sup>

<sup>a</sup>Riduzione in  $n$  passi con  $n \geq 1$

#### Corollario 2.4.2

$\xrightarrow{*} R$  rappresenta la più piccola riduzione tale che  $\rightarrow R \subseteq \xrightarrow{*} R$  e  $\xrightarrow{*} R$  sia transitiva e riflessiva<sup>a</sup>

<sup>a</sup>Riduzione in  $n$  passi con  $n \geq 0$

#### Corollario 2.4.3

$\leftrightarrow R$  rappresenta la più piccola riduzione tale che  $\rightarrow R \subseteq \leftrightarrow R$  e  $\leftrightarrow R$  sia transitiva, riflessiva e simmetrica<sup>a</sup>. Questa relazione si chiama relazione di convertibilità

<sup>a</sup>Ossia si può ridurre in ambo i sensi

#### Definizione 2.4.4: Church-Rosser

$R$  è confluyente o Church-Rosser (CR) se

$$\forall s, t, t' \quad d \xrightarrow{*}_R t \wedge s \xrightarrow{*}_R t' \Rightarrow \exists t'' \xrightarrow{*}_R t \wedge t' \xrightarrow{*}_R t''$$

#### Corollario 2.4.4

Se  $R$  è CR allora ogni  $t$  ha al più una forma normale

#### Note:-

In un  $R$  che è CR, anche se si possono fare più riduzioni differenti il ridotto finale è comune

## 2.5 Logica equazionale

### Definizione 2.5.1: Logica equazionale

Fissata una signature  $\Sigma$  e un insieme numerabile di variabili  $X$ , un'equazione è una coppia  $(s, t) \in \mathcal{T}_\Sigma(X)^2$ , scritta  $s \approx t$ .

### Corollario 2.5.1

Per un insieme di equazioni  $E = \{s_1 \approx t_1, \dots, s_n \approx t_n\} \subseteq T_\Sigma(X)^2$  (definita  $E \vdash s \approx t$ ) valgono le seguenti proprietà:

- Riflessività (*refl*):  $\frac{}{E \vdash s \approx s}$ ;
- Simmetria (*sym*):  $\frac{E \vdash s \approx t}{E \vdash t \approx s}$ ;
- Transitività (*trans*):  $\frac{E \vdash s \approx r \quad E \vdash r \approx t}{E \vdash s \approx t}$ ;
- Congruenza (*congr*):  $\frac{E \vdash s_1 \approx t_1 \quad E \vdash s_n \approx t_n}{E \vdash f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n)}$ ;
- Sostituzione (*sub*):  $\frac{E \vdash a \approx t}{E \vdash s^\sigma \approx t^\sigma}$ ;
- Uso di un'assioma (*ax*):  $ax \frac{s \approx t \in E}{E \vdash s \approx t}$ .

### Note:-

Sopra la linea sono poste le premesse e sotto la linea sono poste le conclusioni

### Esempio 2.5.1 (Logica equazionale)

Sapendo che  $E = \{a \approx b, f(x) \approx g(x)\}$ , dimostriamo che  $E \vdash g(b) \approx f(a)$ . Ci sono due metodi per risolvere il problema:

- Si combinano le regole partendo dalle ipotesi (metodo sintetico), ma richiede intuito ed è spesso troppo complicato;
- Si parte dalla tesi (metodo analitico).

$$\text{trans} \frac{\text{ax1} \frac{}{E \vdash a \approx b} \text{congr} \frac{}{E \vdash f(a) \approx f(b)} \quad \text{ax2} \frac{}{E \vdash f(x) \approx g(x)} \text{sub} \frac{}{E \vdash f(b) \approx g(b)}}{\text{sym} \frac{E \vdash f(a) \approx g(b)}{E \vdash g(b) \approx f(a)}}$$

che si può riscrivere come  $\text{sym}(\text{trans}(\text{congr}(\text{ax1}), \text{sub}(\text{ax2}))) : E \vdash g(b) \approx f(a)$

### Definizione 2.5.2: $s \leftrightarrow_R t$

$s \leftrightarrow_R t \stackrel{*}{\Leftrightarrow} s \rightarrow_R t \vee t \rightarrow_R s$ . Sia  $\stackrel{*}{\Leftrightarrow}_R$  chiusura riflessiva e transitiva di  $\leftrightarrow$

### Corollario 2.5.2

Se  $R$  è CR allora

$$s \stackrel{*}{\Leftrightarrow} t \Leftrightarrow \exists r. s \stackrel{*}{\rightarrow} r \vee t \stackrel{*}{\rightarrow} r$$

$$(\rightarrow) \quad s \equiv t_0 \leftarrow t_1 \leftarrow \dots \leftarrow t_k \equiv t$$

### 2.5.1 Normalizzazione

#### Definizione 2.5.3: Normalizzazione (forte)

Fissati  $\Sigma$  e  $Q$ :

- $t$  è in forma normale se  $\nexists t'. t \rightarrow_R t'$ ;
- $R$  è fortemente normalizzante se non esistono riduzioni infinite:  $t \equiv t_0 \rightarrow_R t_1 \rightarrow_R \dots$  (SN)

#### Corollario 2.5.3

Se  $R$  è CR e SN allora  $s \rightarrow_R^* t$  è deducibile



## Capitolo 3

# Deduzione naturale di Gentzen

### 3.1 La deduzione

#### Definizione 3.1.1: Modus ponens (MP)

$$\frac{\phi \rightarrow \psi \quad \phi}{\psi} \text{MP}$$

Nella logica definita da Gentzen non si utilizzano assiomi, ma soltanto due tipi di regole:

- l'*introduzione*: ossia come viene definito un connettivo;
- l'*eliminazione*: ossia come si usa un connettivo nelle ipotesi.

### 3.2 Congiunzione e implicazione

#### Definizione 3.2.1: La congiunzione

$$\frac{A \quad B}{A \wedge B} \wedge \text{ I}$$
$$\frac{A \wedge B}{A} \wedge \text{ E}_1$$
$$\frac{A \wedge B}{B} \wedge \text{ E}_2$$

#### Definizione 3.2.2: L'implicazione

$$[A]^i$$
$$\vdots$$
$$\vdots$$
$$\vdots$$
$$-i \frac{B}{A \rightarrow B} \rightarrow \text{ I}$$
$$\frac{B \rightarrow A \quad A}{B} \rightarrow \text{ E}$$

#### Note:-

Con  $[A]^i$  si indica la "scarica" di ipotesi  $A$

### Esempio 3.2.1

$\vdash (A \wedge B) \rightarrow (B \wedge A)$

$$-1 \frac{\frac{\frac{[A \wedge B]^1}{B} \wedge E_2 \quad \frac{[A \wedge B]^1}{A} \wedge E_1}{B \wedge A} \wedge I}{A \wedge B \rightarrow B \wedge A} \rightarrow I$$

## 3.3 Vero, falso e negazione

### Definizione 3.3.1: Vero

$$\overline{T} \quad T \quad I$$

#### Note:-

Il vero può solo essere introdotto, ma non serve a dedurre altro

### Definizione 3.3.2: Falso

$$\frac{\perp}{A} \quad \perp \quad E$$

#### Note:-

Il falso può solo essere introdotto

### Definizione 3.3.3: Negazione

$$\begin{array}{c} [A]^i \\ \vdots \\ \vdots \\ \vdots \\ -i \frac{\perp}{\neg A} \neg I \\ \neg A \quad A \\ \hline \perp \neg E \end{array}$$

## 3.4 Disgiunzione

### Definizione 3.4.1: Disgiunzione

$$\begin{array}{c} \frac{A}{A \vee B} \vee I_1 \\ \frac{B}{A \vee B} \vee I_2 \\ -i, -j \frac{A \vee B \quad C \quad C}{C} \vee E \end{array}$$

Il primo C è dedotto da  $[A]^i$ , il secondo da  $[B]^j$  (entrambi "scaricati")

**Esempio 3.4.1** (Legge di De Morgan) $\vdash \neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$ 

$$\begin{array}{c}
\frac{\frac{\frac{[\neg(A \vee B)]^1}{\perp} \quad \frac{[A]^2}{A \vee B} \vee I_1}{\neg A} \neg E \quad \neg B \wedge I}{\neg A \wedge \neg B} \wedge I \\
-1 \frac{}{\vdash \neg(A \vee B) \rightarrow (\neg A \wedge \neg B)} \rightarrow I
\end{array}$$

Per la parte " $\neg B$ " si effettua un procedimento analogo

**Esempio 3.4.2** (Doppia negazione) $\vdash A \rightarrow \neg\neg A$ 

$$\begin{array}{c}
\frac{[\neg A]^2 \quad [A]^1}{\perp} \neg E \quad \neg I}{A \rightarrow \neg\neg A} \rightarrow I \\
-1 \frac{}{} \rightarrow I
\end{array}$$

**Note:-**

Nella deduzione naturale  $\vdash A \rightarrow \neg\neg A$  vale (come appena dimostrato), ma  $\vdash \neg\neg A \rightarrow A$  non vale

### 3.5 Reduction ad absurdum

**Definizione 3.5.1: Reduction ad absurdum (RAA)**

Per dimostrare  $A$  deriviamo l'assurdo  $\perp$  dalla sua negazione  $\neg A$

**Note:-**

RAA non è una regola "costruttiva" bensì classica (CL), per cui si può dimostrare  $\vdash_{CL} \neg\neg A \rightarrow A$

**Esempio 3.5.1** $\vdash \neg\neg A \rightarrow A$ 

$$\begin{array}{c}
\frac{[\neg\neg A]^1 \quad [\neg A]^2}{\perp} \neg E \quad RAA \\
-1 \frac{}{\neg\neg A \rightarrow A} \rightarrow I
\end{array}$$

### 3.6 Quantificatori

**Note:-**

La logica che fa uso dei quantificatori si dice "del prim'ordine" se i quantificatori si possono usare solo su variabili

**Definizione 3.6.1: Quantificatore universale**

$$\begin{array}{c}
\alpha \notin FV(\Gamma) \quad \frac{P(\alpha)}{\forall x P(x)} \forall I \\
\frac{\forall x P(x) \quad P(t)}{P(t)} \forall E
\end{array}$$

**Note:-**

$\Gamma$  è l'insieme delle premesse

**Definizione 3.6.2: Quantificatore esistenziale**

$$\frac{P(t)}{\exists P(x)} \exists \text{ I}$$
$$\alpha \notin FV(\Gamma) \cup FV(C) \frac{\exists x P(x) \quad c}{c} \exists \text{ E}$$

# Capitolo 4

## Il $\lambda$ -calcolo

### Note:-

Questo capitolo è concettualmente affine al capitolo "Il  $\lambda$ -calcolo" negli appunti del corso di "Linguaggi e paradigmi di programmazione"

### 4.1 Introduzione

Il  $\lambda$ -calcolo fu introdotto nel 1933 da Alonzo Church. Con questo calcolo, Church, cercò di formalizzare la nozione di funzione calcolabile.

### Note:-

Non tutte le funzioni sono calcolabili. Alcuni dei motivi per cui è vero ciò sono spiegati nel corso di "Calcolabilità e complessità"

#### Definizione 4.1.1

Sia  $\text{Var} = \{x, y, z, \dots\}$  un insieme finito di variabili, la sintassi è la seguente:

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

### Note:-

$\lambda x.M$  è un'astrazione o funzione con parametro formale  $x$  e corpo  $M$

### Note:-

$(MN)$  è l'applicazione della funzione  $M$  al parametro attuale  $N$

### 4.2 Il $\lambda$ -calcolo non tipato

#### 4.2.1 Semantica

### Note:-

Applicare una funzione  $\lambda x.M$  a un argomento  $N$  significa valutare il corpo della funzione ( $M$ ) in cui ogni occorrenza libera dell'argomento ( $x$ ) è stata sostituita da  $N$

#### Definizione 4.2.1: Insieme delle variabili libere

L'insieme delle variabili libere di un termine  $M$ , denotato come  $fv(M)$ , è definito induttivamente sulla struttura di  $M$  come segue:

$$fv(x) = \{x\} \quad fv(\lambda x.M) = fv(M) \setminus \{x\} \quad fv(MN) = fv(M) \cup fv(N)$$

**Definizione 4.2.2: Sostituzione**

- $x\{N/y\} = \begin{cases} N & \text{se } x = y \\ x & \text{se } x \neq y \end{cases}$
- $(M_1M_2)\{N/y\} = M_1\{N/y\}M_2\{N/y\};$
- $(\lambda x.M)\{N/y\} = \begin{cases} \lambda x.M & \text{se } x = y \\ \lambda x.M\{N/y\} & \text{se } x \neq y \text{ e } x \notin fv(N) \\ \lambda z.M\{z/x\}\{N/y\} & \text{se } x \neq y \text{ e } x \in fv(N) \text{ e } z \in Var - (fv(M) \cup fv(N)) \end{cases}$

**Definizione 4.2.3:  $\alpha$ -equivalenza**

L' $\alpha$ -equivalenza  $\Leftrightarrow_\alpha$  è la congruenza tra  $\lambda$ -espressioni tale che, se  $y \notin fv(M)$ , allora  $\lambda x.M \Leftrightarrow_\alpha \lambda y.M\{y/x\}$

**Note:-**

$y \notin fv(M)$  serve a evitare che una variabile libera in  $M$  venga catturata dalla congruenza

**Definizione 4.2.4:  $\beta$ -riduzione**

La  $\beta$ -riduzione è la relazione tra  $\lambda$ -espressioni tale che:

- $(\lambda x.M)N \rightarrow_\beta M\{N/y\};$
- se  $M \rightarrow_\beta M'$  allora  $MN \rightarrow_\beta M'N;$
- se  $M \rightarrow_\beta M'$  allora  $MN \rightarrow_\beta NM';$
- se  $M \rightarrow_\beta M'$  allora  $MN \rightarrow_\beta \lambda x.M'.$

**Note:-**

Nella  $\beta$ -riduzione:

$$(\lambda x.M)N \rightarrow_\beta M\{N/y\}$$

$(\lambda x.M)N$  è un  $\beta$ -redex<sup>a</sup>.

$M\{N/y\}$  è il suo ridotto.

Ci possono essere più modi di ridurre la stessa  $\lambda$ -espressione. La riduzione di un  $\beta$ -redex può creare altri  $\beta$ -redex. La riduzione di un  $\beta$ -redex può cancellare altri  $\beta$ -redex. La riduzione può non terminare.

<sup>a</sup>REDucible EXpression

**Definizione 4.2.5: Church-Rosser nel  $\lambda$ -calcolo**

$R$  è confluyente o Church-Rosser (CR) se

$$\forall M, N, L. M \xrightarrow{*} N \wedge M \xrightarrow{*} L \Rightarrow \exists P. M \xrightarrow{*} P \wedge L \xrightarrow{*} P$$

**Corollario 4.2.1**

Se  $=_\beta$  è la chiusura simmetrica di  $\rightarrow$  allora  $M =_\beta N \Rightarrow \exists L. M \xrightarrow{*} L \wedge N \xrightarrow{*} L$

#### Definizione 4.2.6: Booleani

Si possono definire  $\text{true} \equiv \lambda x \ y. x^a$  e  $\text{false} \equiv \lambda x \ y. y^b$   
 Partendo da ciò:  $\text{if-then-else} \equiv \lambda x \ y \ z. x \ y \ z$

<sup>a</sup>Combinatore K

<sup>b</sup>Combinatore O

#### Note:-

Questa scrittura è basata sulla logica combinatorica, ma non è esattamente lo stesso nel  $\lambda$ -calcolo. Per essere precisi: tutti i modelli del  $\lambda$ -calcolo sono modelli della logica combinatorica, ma non il viceversa

#### Esempio 4.2.1

- if-then-else  $\text{true} \ M \ N \rightarrow_{\beta} \text{true} \ M \ N \rightarrow_{\beta} M$ ;
- if-then-else  $\text{false} \ M \ N \rightarrow_{\beta} \text{false} \ M \ N \rightarrow_{\beta} N$ ;

### 4.2.2 Numerali di Church

#### Definizione 4.2.7: Numerale di Church

$$\underline{n} \equiv \lambda x \ y. x(\dots(x \ y)\dots)$$

La  $y$  si comporta come lo zero, mentre la  $x$  come il successore

#### Esempio 4.2.2

$$\underline{0} \equiv \lambda x \ y. y$$

$$\underline{2} \equiv \lambda x \ y. x(x \ y)$$

$$\underline{3} \equiv \lambda x \ y. x(x(x \ y))$$

#### Note:-

In ogni numerale sono presenti  $n \ x$  dove  $n$  rappresenta il "numero" in decimale

#### Definizione 4.2.8: Successore di un numerale

$$\text{succ} \ n =_{\beta} n + 1 \equiv \lambda x \ y. x(x(\dots(x \ y)\dots))$$

$$\underline{n} \ x \ y =_{\beta} x(\dots(x \ y)\dots)$$

Dunque  $\text{succ} \equiv \lambda z \ x \ y. x(z \ y \ x)$

#### Esempio 4.2.3

$$\text{succ} \ \underline{2} = \lambda x \ y. x(\underline{2} \ x \ y)$$

$$= \lambda x \ y. x(x(x \ y)) \text{ , perchè } \underline{2} \ x \ y = x(x \ y)$$

$$= \underline{3}$$

#### Definizione 4.2.9: Somma

$$\text{add } \underline{n} \ \underline{m} = \underline{n + m}$$

$$\underline{n + m} = \text{succ}^n \ \underline{m} \equiv \text{succ}(\dots(\text{succ } \underline{m})\dots)$$

$$= \underline{n} \ \text{succ } \underline{m}$$

Allora  $\text{add} \equiv \lambda x \ y. x \ \text{succ } y$

#### Note:-

Nello stesso modo si può definire mult come iterazione di add

#### Definizione 4.2.10: Test per zero

$$\text{is-zero } 0 = \text{true}$$

$$\text{is-zero } n + 1 = \text{false}$$

Allora  $\text{is-zero} \equiv \lambda n. n(\lambda z. \text{false}) \ \text{true}$

#### Esempio 4.2.4

$$\underline{0} \ x \ y = y \quad y \equiv \text{true}$$

$$\underline{1} \ x \ y = x \ y \quad x \equiv \lambda z. \text{false}$$

$$\underline{2} \ x \ y = x(x \ y) \quad x \equiv \lambda z. \text{false}$$



### Definizione 4.2.11: Ricorsione

$$\begin{cases} \text{fact } \underline{0} = \underline{1} \\ \text{fact } \underline{n+1} = \underline{\text{mult}}(n+1)(\text{fact } \underline{n}) \end{cases}$$

Supponiamo di aver definito  $\underline{\text{pred}}$  tale che:

- $\underline{\text{pred}} \ \underline{0} = \underline{0}$ ;
- $\underline{\text{pred}} \ \underline{n+1} = \underline{n}$ .

$$F \ \underline{\text{fact}} \ \underline{n} = \text{if-then-else} \ (\underline{\text{is-zero}} \ \underline{n}) \ \underline{1} \ (\underline{\text{mult}} \ \underline{n} \ (\underline{\text{fact}} \ (\underline{\text{pred}} \ \underline{n})))$$

$$F \equiv \lambda f \ x. \text{if-then-else} \dots (f \ (\underline{\text{pred}} \ \underline{n})) \dots$$

Si suppone l'esistenza di una funzione  $\underline{\text{fix}} \ F = F \ (\text{fix } F)^a$ , allora:

$$\underline{\text{fact}} \equiv \underline{\text{fix}} \ F \text{ allora } F \ \underline{\text{fact}} = \underline{\text{fact}}$$

$$\begin{aligned} \underline{\text{fact}} \ \underline{n} &= F \ \underline{\text{fact}} \ \underline{n} = \dots \underline{\text{fact}} \ (\underline{\text{pred}} \ \underline{n}) \\ &= \dots (F \ \underline{\text{fact}}) (\underline{\text{pred}} \ \underline{n}) \end{aligned}$$

<sup>a</sup>Punto fisso

#### Note:-

Le funzioni ricorsive sono comunque calcolabili a patto che siano composte da funzioni calcolabili

### Teorema 4.2.1 Teorema del punto fisso

$$\forall F \ \exists X. F \ X = X$$

**Proof:** Leggiamo l'equazione alla rovescia, quindi:

$$X = F \ X$$

Proviamo che  $X = W \ W$ , allora:

$$W \ W = F \ (W \ W)$$

Allora  $W \equiv \lambda w. F \ (w \ w)$  risolve la seconda equazione e dunque, anche la prima. ☺

### Definizione 4.2.12: Operatore a punto fisso (Y)

$$\underline{\text{fix}} \equiv \lambda f. (\lambda n. f \ (x \ x)) (\lambda x. f \ (x \ x))$$

$$\text{Allora } \underline{\text{fix}} \ F = (\lambda n. F \ (x \ x)) (\lambda x. F \ (x \ x)) = F \ ((\lambda x. F \ (x \ x)) (\lambda x. F \ (x \ x))) = F(\underline{\text{fix}} \ F)$$

#### Note:-

Il  $\lambda$ -calcolo non tipato puro non è SN

### Teorema 4.2.2 Teorema di Kleensn

Per ogni funzione calcolabile parziale esiste  $F \in A$  tale che:

$$f \ (n_1, \dots, n_k) \simeq m \Leftrightarrow F \ n_1 \dots n_k \rightarrow_{\beta} \underline{n}$$

Dove  $f(n^{\rightarrow}) \simeq m$  significa che  $f(n^{\rightarrow})$  è definita uguale a  $(n^{\rightarrow} = n_1, \dots, n_k)$

## 4.3 Il $\lambda$ -calcolo tipato

### 4.3.1 Tipi

#### Question 1

Come si interpreta un termine  $X \rightarrow X$ ?

**Risposta:** nel  $\lambda$ -calcolo non tipato si può anche scrivere una cosa come l'autoapplicazione. Ma in generale una funzione non dovrebbe appartenere al proprio dominio.

#### Esempio 4.3.1

Se il primo  $X \in A \rightarrow A$  e il secondo  $X \rightarrow A$  non esiste alcun  $A \neq \{*\}$  tale che  $A \simeq A \rightarrow A$  in  $\text{Set}^a$

<sup>a</sup>Categoria degli insiemi

#### Definizione 4.3.1: Tipi semplici

$$A, B ::= \alpha \mid A \rightarrow B$$

dove  $\alpha \in \{\text{bool}, \text{nat}, \dots\}$  è atomico fissate l'interpretazione  $[\alpha]$  (es.  $[\text{nat}] = \mathbb{N}$ )

$$[A \rightarrow B] = [B]^{[A]}$$

dove il dominio è  $[A]$  e il codominio è  $[B]$

#### Definizione 4.3.2: Sistema di tipo

$$\Gamma \vdash M : A \text{ "M ha tipo A in } \Gamma"$$

#### Definizione 4.3.3: Contesto

Un contesto è un insieme finito di giudizi di tipo  $(x_i : A_i)$ :

$$\Gamma = x_1 : A_1, \dots, x_n : A_n, \text{ con } x_i \neq x_j \text{ se } i \neq j$$

#### Corollario 4.3.1

Valgono le seguenti proprietà:

- $\text{ax}_{\Gamma, x:A \vdash x:A}$ ;
- $\rightarrow E \frac{\Gamma \vdash M:A \rightarrow B \quad \Gamma \vdash N:A}{\Gamma \vdash M \ N:B}$ ;
- $\rightarrow I \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x:A. M:A \rightarrow B}$

Dove  $\Gamma, x \in A = \Gamma \cup \{x : A\}$  e  $x \notin \text{Dom}(\Gamma)$

## Capitolo 5

# Logica costruttiva

# Capitolo 6

## Il linguaggio IMP

Per studiare il problema della verifica in programmi imperativi si utilizzerà un piccolo linguaggio di programmazione chiamato *IMP*<sup>1</sup>.

### 6.1 Introduzione a IMP

#### Definizione 6.1.1: Comandi di IMP

Un programma, in IMP, è un comando con la seguente sintassi:

$$\text{Com} \in c, c' ::= \text{SKIP} \mid x := a \mid c :: c' \mid \text{IF } b \text{ THEN } c \text{ ELSE } c' \mid \text{WHILE } b \text{ DO } c$$

#### Note:-

La sintassi è simile al Pascal o al C, ma:

- $\Rightarrow \text{SKIP}$ : termina l'esecuzione senza effetti collaterali;
- $\Rightarrow c :: c'$ : la composizione (in AGDA è  $c ; c'$ ).

#### Corollario 6.1.1 Stati di un programma IMP

Un comando, in IMP, è una trasformazione della memoria. Uno *stato della memoria* (o stato) è una mappatura del tipo  $s : \text{State}$  con  $\text{State} = \text{Varname} \rightarrow \text{Val}$  ossia l'assegnazione di un valore (Val) a ogni variabile (Varname).

#### 6.1.1 Le relazioni in IMP

In IMP esistono due possibili relazioni:

- Big-step**:  $((c, s)) \Rightarrow t$ , dove  $((\_, \_)) \Rightarrow \_ \subseteq (\text{Com} \times \text{State}) \times \text{State}$ ;
- Small-step**:  $((c, s)) \rightarrow ((c', t))$ , dove  $((\_, \_)) \rightarrow ((\_, \_)) \subseteq (\text{Com} \times \text{State}) \times (\text{Com} \times \text{State})$ .

#### Teorema 6.1.1 Equivalenza di Big-step e Small-step

Big-step e Small-step sono legate dalla seguente relazione:

$$\forall c \ s \ t. ((c, s)) \Rightarrow t \iff ((c, s)) \rightarrow^* ((\text{SKIP}, t))$$

#### Note:-

Dove  $\rightarrow^*$  è la relazione meno riflessiva e transitiva che includa  $\rightarrow$ .

<sup>1</sup>A volte viene chiamato "while".

## 6.1.2 La logica di Floyd-Hoare

Per compiere la verifica formale di programmi sono necessarie le *specificazioni*. In questo corso si utilizzano le *asserzioni*.

### Definizione 6.1.2: Asserzioni

Un'asserzione ( $P : \text{Assn}$ ), dove  $\text{Assn} = \text{State} \rightarrow \text{Set}$ , è un predicato di stati.

### Corollario 6.1.2 Pre-condizioni e Post-condizioni

Un paio di asserzioni  $P$  e  $Q$  sono pre-condizioni e post-condizioni di un programma  $c$  nella tripla  $[P] c [Q]$ .

#### Note:-

Nei libri di testo le pre-condizioni e le post-condizioni sono segnate come  $\{P\} c \{Q\}$ , ma questa notazione **non** è permessa da AGDA.

### Teorema 6.1.2 Correttezza parziale

Una tripla  $[P] c [Q]$  è *valida* ( $\models [P] c [Q]$ ) se per ogni stato  $s$  e  $t$  se  $P \ s$  e  $((c, s)) \Rightarrow t$  allora  $Q \ t$ .

In simboli:

$$\forall s \ t. P \ s \wedge ((c, s)) \Rightarrow t \implies Q \ t$$

#### Note:-

Questa correttezza è solo parziale, perchè le pre-condizioni non sono richieste per dire che il programma  $c$  termini partendo da uno stato  $s$

## 6.2 Espressioni

In questa sezione si introducono le espressioni *aritmetiche* ( $\text{Aexp}$ ) e le espressioni *booleane* ( $\text{Bexpr}$ ).

### Definizione 6.2.1: Variabili

Prendiamo  $\{X_0, X_1, \dots\}$  come insieme numerabile di variabili. In AGDA formalizziamo  $X_i$  con  $\text{Vn } i$  ossia la variabile il cui nome ha indice  $i$  ( $i \in \text{Index}^a$ ).

---

<sup>a</sup> $\text{Index}$  è  $\mathbb{N}$ .

```
Index = N
data Vname : Set where
  Vn : Index → Vname
```

#### Note:-

Negli esempi presentati assumeremo  $X = \text{Vn } 0$ ,  $Y = \text{Vn } 1$  e  $Z = \text{Vn } 2$ .

### Definizione 6.2.2: Confronto

Per confrontare due variabili definiamo la funzione  $x =_{\text{Vn}} y$  che compara due nomi e restituisce **true** se sono gli stessi, **false** altrimenti. Questa funzione dipende a sua volta da un'altra funzione  $x =_{\mathbb{N}} y$  per controllare che due  $\mathbb{N}$  siano uguali.

```

_=N_ : N → N → Bool
zero =N zero = true
zero =N succ m = false
succ n =N zero = false
succ n =N succ m = n =N m

_=Vn_ : (x y : Vname) → Bool
Vn i =Vn Vn j = i =N j

```

### 6.2.1 Espressioni aritmetiche

#### Definizione 6.2.3: Aexp

Si può definire la sintassi delle *espressioni aritmetiche* (Aexp) con la grammatica:

$$\text{Aexp} \in a, a' ::= N \ n \mid V \ vn \mid Plus \ a \ a'$$

Dove  $n \in \text{Nat}$  e  $vn \in \text{Vname}$ .

```

data Aexp : Set where
  N : N → Aexp           -- numerals
  V : Vname → Aexp       -- variables
  Plus : Aexp → Aexp → Aexp -- sum

```

#### Esempio 6.2.1 ( $X + (1 + Y)$ )

```

aexp0 : Aexp
aexp0 = Plus (V X) (Plus (N 1) (V Y))

```

#### Definizione 6.2.4: Stato

Uno *stato* è una mappatura dai nomi delle variabili ai loro valori:

$$\Rightarrow \text{Val} = \mathbb{N};$$

$$\Rightarrow \text{State} = \text{Vname} \rightarrow \text{Val}.$$

Il significato di stato è un'astrazione della memoria finita di un computer.

#### Note:-

Usando questa definizione di stato (che è totale) non si avrà a che fare con funzioni parziali o con il costruttore Maybe.

#### Definizione 6.2.5: Aggiornamento

L'*aggiornamento dello stato* è un cambiamento del significato delle singole variabili.

Per formalizzare: l'operatore  $s [x ::= v]$  restituisce lo stato che si comporta come  $s$ , ma quando è applicato a  $X$  lo trasforma in  $Y$ .

```

_ [_ ::= _ ] : State → Vname → Val → State
(s [ x ::= v ]) y = if x = Vn y then v else s y

```

### Esempio 6.2.2 (Stati)

```

st0 : State
st0 = λ x → 0

st1 : State
st1 = st0 [ X ::= 1 ]

st2 : State
st2 = st1 [ Y ::= 2 ] -- equivalently: st2 = (st0 [ X ::= 1 ]) [ Y ::= 2 ]

```

#### Definizione 6.2.6: Aval

La funzione `aval` è un'interpretazione di `Aexpr` utilizzando gli stati.

```

aval : Aexp → State → Val
aval (N n) s = n
aval (V vn) s = s vn
aval (Plus a1 a2) s = aval a1 s + aval a2 s

```

- Il caso `N` `n` non dipende dallo stato, ma restituisce solo `n`;
- Il caso `V` `vn` restituisce il valore dello stato `s` quando applicato a `vn`<sup>2</sup>;
- Il caso `Plus` `a1 a2` restituisce la somma aritmetica della valutazione ricorsiva su `a1` e `a2`.

## 6.2.2 Sostituzione

#### Definizione 6.2.7: Sostituzione

La *sostituzione* consiste nel rimpiazzare ogni occorrenza di una variabile `x` in un'espressione `a` con un'espressione `a'`.

```

_ [_ / _ ] : Aexp → Aexp → Vname → Aexp
N n [ a' / x ] = N n
V y [ a' / x ] with x = Vn y
... | true = a'
... | false = V y
Plus a1 a2 [ a' / x ] = Plus (a1 [ a' / x ]) (a2 [ a' / x ])

```

<sup>2</sup>Ovvero il suo valore salvato in memoria, come nei registri in Assembly.

**Esempio 6.2.3**  $((X + (1 + Y)) \mid (Z + 3) / X)$ 

aexp1 : Aexp

aexp1 = aexp0 [ Plus (V Z) (N 3) / X ]

**Lemma 6.2.1** Sostituzione

Sostituendo  $x$  con  $a'$  in  $a$  e valutando il risultato si ottiene lo stesso stato  $s$  che si otterrebbe valutando  $x$  nello stato  $s \mid x ::= (\text{aval } a' \ s)$ , ossia lo stato in cui il valore di  $x$  è stato aggiornato con il valore di  $a'$ .

```

lemma-subst-aexp : ∀ (a a' : Aexp) (x : Vname) (s : State) →
    aval (a [ a' / x ]) s = aval a (s [ x ::= (aval a' s) ])

lemma-subst-aexp (N n) a' x s =
  begin
    aval ((N n) [ a' / x ]) s =()      -- by definition of substitution
    aval (N n) s                      =()      -- by definition of aval
    n                                  =()      -- by definition of aval
    aval (N n) (s [ x ::= (aval a' s) ])
  end
lemma-subst-aexp (V y) a' x s with x =Vn y
... | true  = refl
... | false = refl
lemma-subst-aexp (Plus a1 a2) a' x s =
  begin
    aval (a1 [ a' / x ]) s + aval (a2 [ a' / x ]) s = ( cong2 _+_ h1 h2 )
                                                    -- by the ind. hyp. h1, h2
    aval a1 s' + aval a2 s'
  end
where
  s' : State
  s' = (s [ x ::= aval a' s ])

  h1 : aval (a1 [ a' / x ]) s = aval a1 (s [ x ::= (aval a' s) ])
  h1 = lemma-subst-aexp a1 a' x s

  h2 : aval (a2 [ a' / x ]) s = aval a2 (s [ x ::= (aval a' s) ])
  h2 = lemma-subst-aexp a2 a' x s

```

**Step della prova:**

1. Per prima cosa si fa induzione su  $a$ ;
2. Il caso  $a = N \ n$ : banale, perchè non può comparire la  $x$  essendo  $n$  un numerale;
3. Il caso  $a = V \ y$ : viene risolto mediante l'utilizzo del costrutto `with`;
4. Il caso  $a = Plus \ a1 \ a2$ : si utilizzano le ipotesi induttive perchè ne è la diretta conseguenza.



### 6.2.3 Espressioni booleane

#### Definizione 6.2.8: Bexp

Si può definire la sintassi delle *espressioni aritmetiche* (Aexp) con la grammatica:

$$\text{Bexp} \in b, b' ::= B \text{ bc} \mid \text{Less } a \ a' \mid \text{Not } b \mid \text{And } b \ b'$$

Dove  $\text{bc} \in \text{Bool}$  e  $a, a' \in \text{Aexp}$ .

```
data Bexp : Set where
  B : Bool → Bexp           -- boolean constants
  Less : Aexp → Aexp → Bexp -- less than
  Not : Bexp → Bexp         -- negation
  And : Bexp → Bexp → Bexp  -- conjunction
```

#### Esempio 6.2.4 (Alcuni esempi)

```
bexp1 : Bexp
bexp1 = Not (Less (V X) (N 1))
```

```
bexp2 : Bexp
bexp2 = And bexp1 (Less (N 0) (V Y))
```

#### Definizione 6.2.9: Confronto

La valutazione delle espressioni booleane dipende dalla valutazione delle espressioni aritmetiche e quindi, indirettamente, dallo stato.

```
_<N_ : N → N → Bool           -- n <N m = true if n < m
zero <N zero   = false         -- in the ordinary ordering of N
zero <N succ n = true
succ n <N zero  = false
succ n <N succ m = n <N m

bval : Bexp → State → Bool
bval (B x) s      = x
bval (Less x y) s = (aval x s) <N (aval y s)
bval (Not b) s    = not (bval b s)
bval (And b1 b2) s = bval b1 s && bval b2 s
```

## 6.3 Semantica Big-step

Tra le due possibili semantiche operazionali la Big-step è un approccio astratto basato sulla nozione di *convergenza*.

### 6.3.1 Comandi

#### Definizione 6.3.1: Comandi

La sintassi dei *comandi* si basa sulla grammatica:

$$\text{Com} \in c, c' ::= \text{SKIP} \mid x := a \mid c :: c' \mid \text{IF } b \text{ THEN } c \text{ ELSE } c' \mid \text{WHILE } b \text{ DO } c$$

Dove  $x \in \text{Vname}$ ,  $a \in \text{Aexp}$  e  $b \in \text{Bexp}$ .

```
data Com : Set where
  SKIP      : Com                                -- inaction
  _:=_      : Vname → Aexp → Com                -- assignment
  _::_      : Com → Com → Com                   -- sequence
  IF_THEN_ELSE_ : Bexp → Com → Com → Com       -- conditional
  WHILE_DO_   : Bexp → Com → Com                -- iteration
```

### 6.3.2 Convergenza

#### Definizione 6.3.2: Predicato di convergenza

La relazione  $\langle\langle c, s \rangle\rangle \Rightarrow t$  significa che l'esecuzione di  $c$ , quando inizia in  $s$ , termina in  $t$ .

#### Note:-

Questo in generale può richiedere una serie di step che sono racchiusi in un unico Big-step.

#### Corollario 6.3.1 Configurazioni

Chiamiamo *configurazioni* ogni coppia  $\langle\langle c, s \rangle\rangle$  comando-stato.

```
data Config : Set where
  ((_,_)) : Com → State → Config
```

### Definizione 6.3.3: Relazione

Si definisce la relazione  $\Rightarrow$  tra **Config** e **State** per creare un sistema formale.

```
data  $\Rightarrow$  : Config  $\rightarrow$  State  $\rightarrow$  Set where

Skip :  $\forall \{s\}$ 
   $\rightarrow$  (( SKIP , s ))  $\Rightarrow$  s

Loc :  $\forall \{x \ a \ s\}$ 
   $\rightarrow$  (( x := a , s ))  $\Rightarrow$  (s [ x ::= aval a s ])

Comp :  $\forall \{c_1 \ c_2 \ s_1 \ s_2 \ s_3\}$ 
   $\rightarrow$  (( c1 , s1 ))  $\Rightarrow$  s2
   $\rightarrow$  (( c2 , s2 ))  $\Rightarrow$  s3
   $\rightarrow$  (( c1 :: c2 , s1 ))  $\Rightarrow$  s3

IfTrue :  $\forall \{c_1 \ c_2 \ b \ s \ t\}$ 
   $\rightarrow$  bval b s = true
   $\rightarrow$  (( c1 , s ))  $\Rightarrow$  t
   $\rightarrow$  (( IF b THEN c1 ELSE c2 , s ))  $\Rightarrow$  t

IfFalse :  $\forall \{c_1 \ c_2 \ b \ s \ t\}$ 
   $\rightarrow$  bval b s = false
   $\rightarrow$  (( c2 , s ))  $\Rightarrow$  t
   $\rightarrow$  (( IF b THEN c1 ELSE c2 , s ))  $\Rightarrow$  t

WhileFalse :  $\forall \{c \ b \ s\}$ 
   $\rightarrow$  bval b s = false
   $\rightarrow$  (( WHILE b DO c , s ))  $\Rightarrow$  s

WhileTrue :  $\forall \{c \ b \ s_1 \ s_2 \ s_3\}$ 
   $\rightarrow$  bval b s1 = true
   $\rightarrow$  (( c , s1 ))  $\Rightarrow$  s2
   $\rightarrow$  (( WHILE b DO c , s2 ))  $\Rightarrow$  s3
   $\rightarrow$  (( WHILE b DO c , s1 ))  $\Rightarrow$  s3

infix 10  $\Rightarrow$ 
```

### 6.3.3 Proprietà della convergenza

#### Teorema 6.3.1 Non trivialità

Esiste almeno un comando che non produce nessuno stato finale come risultato della sua esecuzione.

##### Note:-

L'esempio più naturale è *WHILE* B true *DO* c.

```
lemma-while-true : ∀ {c : Com} {s t : State} →  
  ¬ ( ( ( WHILE B true DO c , s ) ) ⇒ t )  
  
lemma-while-true (WhileTrue x hyp1 hyp2) = lemma-while-true hyp2
```

- La prova di questo lemma è per contraddizione;
- $\text{hyp1} = \langle\langle c, s \rangle\rangle \Rightarrow s_2$ ;
- $\text{hyp2} = \langle\langle \text{WHILE } B \text{ true DO } c, s_2 \rangle\rangle \Rightarrow t$ .

##### Note:-

Non è una Reductio Ad Absurdum, ma una semplice prova per contraddizione.

#### Teorema 6.3.2 Determinismo

Ogni volta che  $\langle\langle c, s \rangle\rangle \Rightarrow t$  è derivabile per qualche  $\langle\langle c, s \rangle\rangle \in \text{Config}$  e  $t \in \text{State}$ , lo stato  $t$  è unico.

##### Note:-

Per provare questo teorema abbiamo bisogno di due lemmi.

#### Lemma 6.3.1 Una cosa o è vera o è falsa

```
true-neq-false : ¬ (true = false)  
true-neq-false = λ ()
```

#### Lemma 6.3.2 Il vero è diverso dal falso

```
lemma-true-neq-false : ∀ {A : Set} → true = false → A  
lemma-true-neq-false x = ex-falso (true-neq-false x)
```

```

theorem-deterministic :  $\forall \{c : \text{Com}\} \{s \ t \ t' : \text{State}\} \rightarrow$ 
     $((c, s) \Rightarrow t \rightarrow ((c, s) \Rightarrow t' \rightarrow t = t')$ 

theorem-deterministic Skip Skip = refl
theorem-deterministic Loc Loc = refl
theorem-deterministic (Comp hyp1 hyp3) (Comp hyp2 hyp4)
    rewrite theorem-deterministic hyp1 hyp2 |
        theorem-deterministic hyp3 hyp4 = refl
theorem-deterministic (IfTrue x hyp1) (IfTrue y hyp2)
    rewrite theorem-deterministic hyp1 hyp2 = refl
theorem-deterministic (IfTrue x hyp1) (IfFalse y hyp2)
    = lemma-true-neq-false abs
    where
        abs : true = false
        abs = tran (symm x) y
theorem-deterministic (IfFalse x hyp1) (IfTrue y hyp2)
    = lemma-true-neq-false abs
    where
        abs : true = false
        abs = tran (symm y) x
theorem-deterministic (IfFalse x hyp1) (IfFalse y hyp2)
    rewrite theorem-deterministic hyp1 hyp2 = refl
theorem-deterministic (WhileFalse x) (WhileFalse y) = refl
theorem-deterministic (WhileFalse x) (WhileTrue y hyp2 hyp3)
    = lemma-true-neq-false abs
    where
        abs : true = false
        abs = tran (symm y) x
theorem-deterministic (WhileTrue x hyp1 hyp3) (WhileFalse y)
    = lemma-true-neq-false abs
    where
        abs : true = false
        abs = tran (symm x) y
theorem-deterministic (WhileTrue x hyp1 hyp3) (WhileTrue y hyp2 hyp4)
    rewrite theorem-deterministic hyp1 hyp2 |
        theorem-deterministic hyp3 hyp4 = refl

```

**Note:-**

La prova consiste semplicemente in due induzioni simultanee sulle ipotesi  $((c, s) \Rightarrow t$  e  $((c, s) \Rightarrow t'$ , usando la tattica *rewrite*. I due lemmi dimostrati in precedenza sono utili per gestire i casi impossibili riducendoli all'assurdo (ex-falso).

### 6.3.4 Equivalenza

#### Definizione 6.3.4: Equivalenza

Due comandi  $c, c' \in \text{Com}$  sono equivalenti per ogni  $s \in \text{State}$  delle computazioni  $\langle\langle c, s \rangle\rangle$  e  $\langle\langle c', s \rangle\rangle$  non convergono o  $\langle\langle c, s \rangle\rangle \Rightarrow t$  e  $\langle\langle c', s \rangle\rangle \Rightarrow t$  per ogni  $t \in \text{State}$ .

```
_~_ : Com → Com → Set      -- the symbol ~ is written \sim

c ~ c' = ∀{s t} → ( ⟨⟨ c , s ⟩⟩ = t → ⟨⟨ c' , s ⟩⟩ = t ) ∧
                  ( ⟨⟨ c' , s ⟩⟩ = t → ⟨⟨ c , s ⟩⟩ = t )

infixl 19 _~_
```

#### Note:-

L'equivalenza tra i comandi è utilizzata per ottimizzazioni.

#### Esempio 6.3.1 (IF)

In questo esempio l'IF può essere rimosso perchè sia che la condizione sia vera sia che sia falsa eseguirà sempre lo stesso comando.

```
lemma-if-c-c : ∀ (b : Bexp) (c : Com) →
  IF b THEN c ELSE c ~ c

lemma-if-c-c b c {s} {t} = only-if-part , if-part
  where
    only-if-part : ⟨⟨ IF b THEN c ELSE c , s ⟩⟩ = t → ⟨⟨ c , s ⟩⟩ = t
    only-if-part (IfTrue x hyp) = hyp
    only-if-part (IfFalse x hyp) = hyp

    if-part : ⟨⟨ c , s ⟩⟩ = t → ⟨⟨ IF b THEN c ELSE c , s ⟩⟩ = t
    if-part hyp with lemma-bval-tot b s
    ... | inl x = IfTrue x hyp
    ... | inr x = IfFalse x hyp
```

lemma-bval-tot è un lemma per cui la valutazione di un'espressione booleana restituisce o true o false.

## 6.4 Semantica Small-step

Un approccio alternativo alle semantiche operazionali è quello di descrivere la computazione come l'esecuzione di una serie di step.

### 6.4.1 Riduzione in un passo

#### Definizione 6.4.1: Relazione di riduzione in un passo

La relazione  $((c, s)) \rightarrow ((c', s'))$  modella l'esecuzione del comando "più a sinistra" in  $c$  iniziando da  $s$ , producendo la nuova configurazione  $((c', s'))$  dove  $c'$  (*continuazione*) è ciò che resta da eseguire di  $c$  e  $s'$  è il nuovo stato prodotto. La relazione  $\rightarrow$  è chiamata *riduzione in un passo*.

```
data _→_ : Config → Config → Set where -- the symbol → is written \→

Loc : ∀{x a s}
  → (( x := a , s )) → (( SKIP , s [ x ::= aval a s ] ))

Comp1 : ∀{c s}
  → (( SKIP :: c , s )) → (( c , s ))

Comp2 : ∀{c1 c1' c2 s s'}
  → (( c1 , s )) → (( c1' , s' ))
  → (( c1 :: c2 , s )) → (( c1' :: c2 , s' ))

IfTrue : ∀{b s c1 c2}
  → bval b s = true
  → (( IF b THEN c1 ELSE c2 , s )) → (( c1 , s ))

IfFalse : ∀{b s c1 c2}
  → bval b s = false
  → (( IF b THEN c1 ELSE c2 , s )) → (( c2 , s ))

While : ∀{b c s}
  → (( WHILE b DO c , s )) → (( IF b THEN (c :: (WHILE b DO c)) ELSE SKIP , s ))
```

- *SKIP* è il comando terminale, quindi non riduce a niente e tutti i comandi che lo raggiungono sono terminati;
- *Comp<sub>1</sub>* indica che il primo comando è terminato e quindi l'esecuzione continua con il prossimo;
- *Comp<sub>2</sub>* indica che il primo comando si può ridurre a un comando diverso da SKIP;
- *IfTrue* e *IfFalse* sono banali, perché IfTrue esegue il ramo THEN e IfFalse esegue il ramo ELSE;
- *While* si comporta come in un generico linguaggio di programmazione in cui controlla (mediante If) a ogni iterazione. Se è true continua, mentre se è false diventa SKIP (termina).

## 6.4.2 Chiusure

### Definizione 6.4.2: Riduzione in più passi

La *riduzione in più passi* (o riduzione) è la chiusura transitiva e riflessiva della riduzione in un passo.

```
data _→*_ : Config → Config → Set where

  →*-refl : ∀ {c s} → ((c , s) →* (c , s)) -- reflexivity
  →*-incl : ∀ {c1 s1 c2 s2 c3 s3} →          -- including →
    ((c1 , s1) → (c2 , s2)) →
    ((c2 , s2) →* (c3 , s3)) →
    ((c1 , s1) →* (c3 , s3))
```

- La regola  $\rightarrow^*-refl$  postula la riflessività;
- La regola  $\rightarrow^*-incl$  concatena la riduzione in un passo alla riduzione in più passi<sup>3</sup>.

#### Note:-

Da queste si deriva la regola di transitività.

```
→*-tran : ∀ {c1 s1 c2 s2 c3 s3} →
  ((c1 , s1) →* (c2 , s2)) →
  ((c2 , s2) →* (c3 , s3)) →
  ((c1 , s1) →* (c3 , s3))

→*-tran →*-refl hyp2 = hyp2
→*-tran (→*-incl x hyp1) hyp2 = →*-incl x (→*-tran hyp1 hyp2)
```

In AGDA andremo a utilizzare le seguenti macro.

```
((_,_) ■ : ∀ c s → ((c , s) →* (c , s))
((c , s) ■ = →*-refl

(⟦_,_⟧ →(⟦_)_ : ∀ c s {c' c'' s' s''} →
  ((c , s) → (c' , s')) →
  ((c' , s') →* (c'' , s'')) →
  ((c , s) →* (c'' , s''))
(⟦c , s⟧ →( x ) y = →*-incl x y

(⟦_,_⟧ →*(⟦_)_ : ∀ c s {c' c'' s' s''} →
  ((c , s) →* (c' , s')) →
  ((c' , s') →* (c'' , s'')) →
  ((c , s) →* (c'' , s''))
(⟦c , s⟧ →*( x ) y = →*-tran x y
```

<sup>3</sup>Ricorda il cons nelle liste.



## 6.5 Relazione tra semantica Big-step e semantica Small-step

### Teorema 6.5.1 Equivalenza tra Big-step e Small-step

Dati qualsiasi:

- $c \in \text{Com.}$
- $s, t \in \text{State.}$

$\langle\langle c, s \rangle\rangle \Rightarrow t$  se e solo se  $\langle\langle c, s \rangle\rangle \longrightarrow^* \langle\langle \text{SKIP}, t \rangle\rangle$

### 6.5.1 Da Small-step a Big-step

#### Lemma 6.5.1 Small-Big

Se una configurazione riduce in un passo a un'altra che converge in uno stato allora la configurazione iniziale converge a quello stato.

```
lemma-small-big :  $\forall \{c1\ s1\ c2\ s2\ t\} \rightarrow$   
   $\langle\langle c1, s1 \rangle\rangle \rightarrow \langle\langle c2, s2 \rangle\rangle \rightarrow$   
   $\langle\langle c2, s2 \rangle\rangle = t \rightarrow$   
   $\langle\langle c1, s1 \rangle\rangle = t$   
  
lemma-small-big Loc Skip = Loc  
lemma-small-big Comp1 hyp2 = Comp Skip hyp2  
lemma-small-big (Comp2 hyp1) (Comp hyp2 hyp3) = Comp indHyp hyp3  
  where  
    indHyp = lemma-small-big hyp1 hyp2  
lemma-small-big (IfTrue x) hyp2 = IfTrue x hyp2  
lemma-small-big (IfFalse x) hyp2 = IfFalse x hyp2  
lemma-small-big While (IfTrue x (Comp hyp2 hyp3)) =  
  WhileTrue x hyp2 hyp3  
lemma-small-big While (IfFalse x Skip) = WhileFalse x  
  
theorem-small-big :  $\forall \{c\ s\ t\} \rightarrow$   
   $\langle\langle c, s \rangle\rangle \longrightarrow^* \langle\langle \text{SKIP}, t \rangle\rangle \rightarrow \langle\langle c, s \rangle\rangle = t$   
  
theorem-small-big  $\longrightarrow^*$ -refl = Skip  
theorem-small-big ( $\longrightarrow^*$ -incl x hyp) = lemma-small-big x indHyp  
  where  
    indHyp = theorem-small-big hyp
```

#### Note:-

Il teorema Small-Big è una semplice induzione sulla definizione di  $\longrightarrow^*$ .

## 6.5.2 Da Big-step a Small-step

### Lemma 6.5.2 Big-Small

Si estende la regola  $\text{Comp}_2$  a  $\longrightarrow^*$  nella definizione di  $\longrightarrow$ .

```
lemma-big-small :  $\forall \{c \ c' \ c'' \ s \ s'\} \rightarrow$ 
   $((c, s) \longrightarrow^* (c', s')) \rightarrow$ 
   $((c :: c', s) \longrightarrow^* (c' :: c'', s'))$ 

lemma-big-small  $\longrightarrow^*$ -refl =  $\longrightarrow^*$ -refl
lemma-big-small ( $\longrightarrow^*$ -incl x hyp) =
   $\longrightarrow^*$ -incl (Comp2 x) (lemma-big-small hyp)
```

#### Note:-

Il teorema Big-Small fa induzione su  $((c, s)) \Rightarrow t$

```
theorem-big-small :  $\forall \{c \ s \ t\} \rightarrow$ 
   $((c, s) \Rightarrow t \rightarrow ((c, s) \longrightarrow^* (\text{SKIP}, t)))$ 

theorem-big-small (Skip {s}) =
   $((\text{SKIP}, s) \blacksquare)$ 
theorem-big-small (Loc {x} {a} {s}) =
   $((x := a, s) \longrightarrow (\text{Loc}))$ 
   $((\text{SKIP}, s [x ::= \text{aval } a \ s]) \blacksquare)$ 
theorem-big-small (Comp {c1} {c2} {s1} {s2} {s3} hyp1 hyp2) =
   $((c_1 :: c_2, s_1) \longrightarrow^* (\text{lemma-big-small } (\text{theorem-big-small } \text{hyp1})))$ 
   $((\text{SKIP} :: c_2, s_2) \longrightarrow (\text{Comp}_1))$ 
   $((c_2, s_2) \longrightarrow^* (\text{theorem-big-small } \text{hyp2}))$ 
   $((\text{SKIP}, s_3) \blacksquare)$ 
theorem-big-small (IfTrue {c1} {c2} {b} {s} {t} x hyp) =
   $((\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \longrightarrow (\text{IfTrue } x))$ 
   $((c_1, s) \longrightarrow^* (\text{theorem-big-small } \text{hyp}))$ 
   $((\text{SKIP}, t) \blacksquare)$ 
theorem-big-small (IfFalse {c1} {c2} {b} {s} {t} x hyp) =
   $((\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, s) \longrightarrow (\text{IfFalse } x))$ 
   $((c_2, s) \longrightarrow^* (\text{theorem-big-small } \text{hyp}))$ 
   $((\text{SKIP}, t) \blacksquare)$ 
theorem-big-small (WhileFalse {c} {b} {s} x) =
   $((\text{WHILE } b \text{ DO } c, s) \longrightarrow (\text{While}))$ 
   $((\text{IF } b \text{ THEN } c :: (\text{WHILE } b \text{ DO } c) \text{ ELSE } \text{SKIP}, s) \longrightarrow (\text{IfFalse } x))$ 
   $((\text{SKIP}, s) \blacksquare)$ 
theorem-big-small (WhileTrue {b} {c} {s} {r} {t} x hyp hyp1) =
   $((\text{WHILE } b \text{ DO } c, s) \not\rightarrow (\text{While}))$ 
   $((\text{IF } b \text{ THEN } c :: (\text{WHILE } b \text{ DO } c) \text{ ELSE } \text{SKIP}, s) \not\rightarrow (\text{IfTrue } x))$ 
   $((c :: (\text{WHILE } b \text{ DO } c), s) \not\rightarrow^* (\text{lemma-big-small } (\text{theorem-big-small } \text{hyp})))$ 
   $((\text{SKIP} :: (\text{WHILE } b \text{ DO } c), r) \not\rightarrow (\text{Comp}_1))$ 
   $((\text{WHILE } b \text{ DO } c, r) \not\rightarrow^* (\text{theorem-big-small } \text{hyp}_1))$ 
   $((\text{SKIP}, t) \blacksquare)$ 
```

#### Note:-

Le due semantiche sono semi-decidibili, ma non decidibili secondo la teoria della computabilità.

## Capitolo 7

# Logica di Floyd-Hoare

Lo scopo della *verifica dei programmi* è il controllo della soddisfaccibilità delle specifiche, ossia che il programma sia *corretto* e *compilante*. Negli anni '60 Floyd propose di inserire delle formule logiche nei flow-chart dei programmi e Hoare introdusse le *triple*<sup>1</sup>.

```
Assn = State → Set  
  
data Triple : Set1 where  
  [_]_[] : Assn → Com → Assn → Triple
```

### Note:-

Il tipo  $\text{Set}_1$  è l'*universo* più piccolo tale che  $\text{Set} = \text{Set}_0 : \text{Set}_1$ .

## 7.1 Il sistema della logica di Hoare

### Definizione 7.1.1: Logica di Hoare

La logica di Hoare è un'assiomatizzazione della nozione di correttezza parziale. Consiste di un sistema formale di regole di inferenza i cui giudizi sono triple più formule logiche.

#### Corollario 7.1.1 Asserzione

Un'asserzione è un predicato dello stato.

#### Corollario 7.1.2 Tripla

Una tripla  $\{P\} c \{Q\}$  significa che: quando in un certo stato vale la *precondizione*  $P$  e l'esecuzione del *comando*  $c$  in quello stato termina viene prodotto un nuovo stato che soddisfa la *postcondizione*  $Q$ .  
In modo formale: se  $P \ s$  e  $((c, s)) \rightarrow t$  allora  $Q \ t$ .

### Question 2

Quando un'asserzione è valida?

**Risposta:** un'asserzione è valida quando è vera in tutti gli stati.

<sup>1</sup>Accennate nel capitolo precedente.

### 7.1.1 Regole

```

data ⊢_ : Triple → Set₁ where

  H-Skip : ∀ {P}
    → ⊢ [ P ] SKIP [ P ]

  H-Loc : ∀ {P a x}
    → ⊢ [ (λ s → P (s [ x ::= aval a s ])) ] (x := a) [ P ]

  H-Comp : ∀ {P Q R c₁ c₂}
    → ⊢ [ P ] c₁ [ Q ]
    → ⊢ [ Q ] c₂ [ R ]
    → ⊢ [ P ] c₁ :: c₂ [ R ]

  H-If : ∀ {P b c₁ Q c₂}
    → ⊢ [ (λ s → P s ∧ bval b s = true) ] c₁ [ Q ]
    → ⊢ [ (λ s → P s ∧ bval b s = false) ] c₂ [ Q ]
    → ⊢ [ P ] (IF b THEN c₁ ELSE c₂) [ Q ]

  H-While : ∀ {P b c}
    → ⊢ [ (λ s → P s ∧ bval b s = true) ] c [ P ]
    → ⊢ [ P ] (WHILE b DO c) [ (λ s → P s ∧ bval b s = false) ]

  H-Conseq : ∀ {P Q P' Q' : Assn} {c}
    → (∀ s → P' s → P s)
    → ⊢ [ P ] c [ Q ]
    → (∀ s → Q s → Q' s)
    → ⊢ [ P' ] c [ Q' ]

```

#### Definizione 7.1.2: Regole

- ⇒ H-Skip: è un'assioma. P è sia preconditione che postcondizione;
- ⇒ H-Loc: l'assegnazione, anche questo è un'assioma. Nella logica di Hoare si ragiona al rovescio, dalla postcondizione alla preconditione. Si vuole garantire la postcondizione P, per cui prima deve valere la preconditione P, ma nello stato precedente;
- ⇒ H-Comp: si utilizza la transitività avendo un predicato intermedio tra i due comandi;
- ⇒ H-If: IF THEN ELSE esegue un comando diverso a seconda della preconditione, ossia se il booleano è vero o falso, ma alla fine la postcondizione è sempre Q;
- ⇒ H-While: al termine del while la guardia deve essere diventata falsa. Per cui nella preconditione vale sia P che la guardia b;
- ⇒ H-Conseq: P è un *indebolimento* (implicato da un certo P') e Q è un *rafforzamento* (implica un certo Q').

#### Note:-

Per via della nostra definizione di asserzione ci sono alcune differenze con le regole originali.

## 7.1.2 Regole derivate

```

H-Str :  ∀ {P Q P' : Assn} {c}
        → (∀ s → P' s → P s)
        → ⊢ [ P ] c [ Q ]
        ───────────────────
        → ⊢ [ P' ] c [ Q ]

H-Str {P}{Q}{P'}{c} hyp1 hyp2 =
    H-Conseq {P}{Q}{P'}{c} hyp1 hyp2 (λ s → λ x → x)

H-Weak :  ∀ {P Q Q' : Assn} {c}
        → ⊢ [ P ] c [ Q ]
        → (∀ s → Q s → Q' s)
        ───────────────────
        → ⊢ [ P ] c [ Q' ]

H-Weak {P}{Q}{Q'}{c} hyp1 hyp2 =
    H-Conseq {P}{Q}{P'}{c} (λ s → λ x → x) hyp1 hyp2

H-While' :  ∀ {P b c Q}
        → ⊢ [ (λ s → P s ∧ bval b s = true) ] c [ P ]
        → (∀ s → (P s ∧ bval b s = false) → Q s)
        → ⊢ [ P ] (WHILE b DO c) [ Q ]

H-While' hyp1 hyp2 = H-Weak (H-While hyp1) hyp2

```

### Definizione 7.1.3: Regole derivate

- ⇒ H-Str: specializzazione della conseguenza in cui la preconditione viene rafforzata;
- ⇒ H-Weak: specializzazione della conseguenza in cui la postcondizione viene indebolita;
- ⇒ H-While': è necessaria alla verifica semi-automatica dei programmi. Si sfrutta la regola per il while in combinazione con l'indebolimento.

## 7.2 Esempi

### 7.2.1 Assegnamento

```

_==_ : Aexp → Aexp → Assn

```

```

a == ' a ' = λ s → aval a s == aval a' s

```

```

pr0-0 : ⊢ [ V X == ' N 1 ]
        Z := V X
        [ V Z == ' N 1 ]

```

```

pr0-0 = H-Loc {V Z == ' N 1} {V X} {Z}

```

```

pr0-1 : ⊢ [ V Z == ' N 1 ]
        Y := V Z
        [ V Y == ' N 1 ]

```

```

pr0-1 = H-Loc {V Y == ' N 1} {V Z} {Y}

```

#### Note:-

La pre-condizione  $X = 1$  è il risultato della sostituzione di  $Z$  con  $X$  nella post-condizione  $Z = 1$ . Dato che il comando termina sempre e assegna il valore di  $X$  a  $Z$  la tripla è valida (secondo il nostro formalismo *H-Loc*).

## 7.2.2 Composizione

```
pr0-2 : ⊢ [ V X = ' N 1 ]
          (Z := V X) :: (Y := V Z)
          [ V Y = ' N 1 ]

pr0-2 = H-Comp {V X = ' N 1}
              {V Z = ' N 1}
              {V Y = ' N 1}
              {Z := V X}
              {Y := V Z}
              pr0-0 pr0-1
```

### Note:-

Basta applicare la regola *H-Comp* a pr0-0 e pr0-1.

## 7.2.3 Selezione

Consideriamo  $\{T\} \text{ IF } X < Y \text{ THEN } Z := Y \text{ ELSE } Z := X \{Z = \max(X,Y)\} \{T\} \text{ IF } X < Y \text{ THEN } Z := Y \text{ ELSE } Z := X \{Z = \max(X,Y)\}$  ( $T$  è sempre vera, triviale). Per dimostrarla dobbiamo prima dimostrare:

```
H1 ⊢ [ (λ s → T' s ∧ (bval (Less (V X) (V Y)) s = true)) ]
      Z := V Y
      [ max' (V X) (V Y) (V Z) ]

H2 ⊢ [ (λ s → T' s ∧ (bval (Less (V X) (V Y)) s = false)) ]
      Z := V X
      [ max' (V X) (V Y) (V Z) ]
```

```
T' : Assn
T' s = T

max' : Aexp → Aexp → Aexp → Assn
max' a1 a2 a3 = λ s → max (aval a1 s) (aval a2 s) = aval a3 s
```

### Note:-

max è  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$  ed è definita nella libreria Nat.agda.

```
pr1-1 : ⊢ [ max' (V X) (V Y) (V Y) ]
          Z := V Y
          [ max' (V X) (V Y) (V Z) ]

pr1-1 = H-Loc {max' (V X) (V Y) (V Z)} {V Y} {Z}
```

```
less-max : ∀(n m : ℕ) → n < N m = true → max n m = m

less-max zero m hyp = refl
less-max (succ n) (succ m) hyp = cong succ (less-max n m hyp)

less-max' : ∀(a a' : Aexp) → (s : State) →
  bval (Less a a') s = true → max' a a' a' s

less-max' a a' s hyp = less-max (aval a s) (aval a' s) hyp

pr1-2 : ∀ s → (T' s ∧ (bval (Less (V X) (V Y)) s = true)) →
  max' (V X) (V Y) (V Y) s

pr1-2 s (x , y) = less-max' (V X) (V Y) s y
```

### Note:-

Ora si può provare H1 mediante la regola di rafforzamento (*H-STR*).

```

H1 : ⊢ [ (λ s → T' s ∧ (bval (Less (V X) (V Y)) s = true)) ]
      Z := V Y
      [ max' (V X) (V Y) (V Z) ]

H1 = H-Str pr1-2 pr1-1

```

**Note:-**

In maniera simile possiamo provare H2.

```

geq-max : ∀(n m : ℕ) → n <N m = false → max n m = n

geq-max zero zero hyp = refl
geq-max (succ n) zero hyp = refl
geq-max (succ n) (succ m) hyp = cong succ (geq-max n m hyp)

geq-max' : ∀(a a' : Aexp) → (s : State) →
  bval (Less a a') s = false → max' a a' a s

geq-max' a a' s hyp = geq-max (aval a s) (aval a' s) hyp

pr1-3 : ⊢ [ max' (V X) (V Y) (V X) ]
      Z := V X
      [ max' (V X) (V Y) (V Z) ]

pr1-3 = H-Loc {max' (V X) (V Y) (V Z)} {V X} {Z}

pr1-4 : ∀ s → (T' s ∧ (bval (Less (V X) (V Y)) s = false)) →
  max' (V X) (V Y) (V X) s

pr1-4 s (x , y) = geq-max' (V X) (V Y) s y

H2 : ⊢ [ (λ s → T' s ∧ (bval (Less (V X) (V Y)) s = false)) ]
      Z := V X
      [ max' (V X) (V Y) (V Z) ]

H2 = H-Str pr1-4 pr1-3

```

**Note:-**

Infine si applica *H-IF* alle ipotesi.

```

pr1-5 : ⊢ [ T' ]
      IF (Less (V X) (V Y)) THEN (Z := V Y) ELSE (Z := V X)
      [ max' (V X) (V Y) (V Z) ]

pr1-5 = H-If H1 H2

```

## 7.3 Correttezza

### Question 3

Qual è la differenza tra *vero* e *valido*?

**Risposta:** in logica, vero si riferisce a un determinato modello con una determinata interpretazione, valido si riferisce a tutti i modelli. Nel contesto di questo corso il concetto di modello è sostituito dal concetto di stato, per cui una tripla è valida se è vera in tutti gli stati.

```

lemma-Hoare-inv :  $\forall \{P : \text{Assn}\} \{s \ b \ c \ t\} \rightarrow$ 
  ( $\forall \{s' \ t'\} \rightarrow (P \ s' \wedge \text{bval } b \ s' = \text{true}) \rightarrow$ 
    ( $\llbracket c \ , \ s' \rrbracket = t'$ 
       $\rightarrow P \ t'$ )  $\rightarrow$ 
     $P \ s \rightarrow$ 
      ( $\llbracket \text{WHILE } b \text{ DO } c \ , \ s \rrbracket = t \rightarrow$ 
         $P \ t$ )

lemma-Hoare-inv {P} {s} {b} {c} {s'} hyp1 hyp2 (WhileFalse x) = hyp2
lemma-Hoare-inv {P} {s} {b} {c} {t} hyp1 hyp2
  (WhileTrue {c} {b} {s} {s'} {s''} x hyp3 hyp4) = claim2
where
  claim1 = hyp1 {s} {s'} (hyp2 , x) hyp3
  claim2 = lemma-Hoare-inv {P} {s'} {b} {c} {t}
    (hyp1 {s} {s'}) claim1 hyp4

lemma-Hoare-loop-exit :  $\forall \{b \ c \ s \ t\}$ 
   $\rightarrow (\llbracket \text{WHILE } b \text{ DO } c \ , \ s \rrbracket = t \rightarrow \text{bval } b \ t = \text{false})$ 

lemma-Hoare-loop-exit (WhileFalse x) = x
lemma-Hoare-loop-exit (WhileTrue x hyp1 hyp2) =
  lemma-Hoare-loop-exit hyp2

```

### Definizione 7.3.1: Cicli

- $\Rightarrow$  lemma-Hoare-inv: stabilisce che quando  $P$  è vera e  $b$  anche (in  $s'$ ) se  $c$  eseguito in  $s'$  produce  $t'$  e  $P$  è vera in  $t'$  allora la tripla  $[P] \text{ WHILE } b \text{ DO } c [P]$  è valida;
- $\Rightarrow$  lemma-Hoare-loop-exit: gestisce l'uscita dal loop ossia la parte relativa alla valutazione di  $b$  come False al termine dell'esecuzione.

```

lemma-Hoare-sound :  $\forall \{P \ c \ Q \ s \ t\} \rightarrow$ 
   $\vdash [P] \ c [Q] \rightarrow$ 
   $P \ s \rightarrow$ 
  ( $\llbracket c \ , \ s \rrbracket = t \rightarrow$ 
     $Q \ t$ )

```

### Definizione 7.3.2: Lemma-Hoare-sound

- $\Rightarrow$  caso SKIP: banalità, utilizza  $P \ s$ ;
- $\Rightarrow$  caso LOC: banalità, utilizza  $P \ s$ ;
- $\Rightarrow$  caso COMP: si applica l'ipotesi induttiva sia al primo elemento del COMP che al secondo (passandogli la prima ipotesi);
- $\Rightarrow$  caso IF True: si applica l'ipotesi induttiva;
- $\Rightarrow$  caso IF False: stesso procedimento usato per il true;
- $\Rightarrow$  caso WHILE: si utilizza il lemma-Hoare-inv applicato all'ipotesi induttiva e il lemma-Hoare-exit applicato al comando;
- $\Rightarrow$  caso CONSEQ: semplice applicazione dell'ipotesi induttiva.



```
theorem-Hoare-sound :  $\forall \{P \ c \ Q\} \rightarrow$   

 $\vdash [P] \ c \ [Q] \rightarrow \models [P] \ c \ [Q]$   

theorem-Hoare-sound hyp = lemma-Hoare-sound hyp
```

### Definizione 7.3.3: Teorema-Hoare-sound

Semplicemente si generalizza il lemma.

## 7.4 Completezza

La completezza è l'implicazione inversa della completezza.

### Definizione 7.4.1: Completezza

La completezza riguarda la capacità del sistema di catturare le cose vere. Se  $\{P\} \ c \ \{Q\}$  è valida allora è derivabile.

#### Note:-

Se il teorema di completezza fosse dimostrabile si avrebbe un'assiomatizzazione corretta dell'aritmetica, ma ciò andrebbe in contrasto con il primo teorema d'incompletezza di Gödel. Tuttavia, in IMP è dimostrabile perché non è un sistema formale (per via della regola H-Conseq).

### 7.4.1 Weakest Liberal Precondition

```
wp : Com  $\rightarrow$  Assn  $\rightarrow$  Assn  

wp c Q s =  $\forall t \rightarrow ((c, s) = t \rightarrow Q \ t)$ 
```

### Definizione 7.4.2: Weakest Liberal Precondition

Le Weakest Liberal Precondition prendono un comando, un'asserzione e restituiscono un'asserzione. L'idea è: dato un comando e data una postcondizione si vuole risalire alla preconditione più debole. Questo perché altrimenti non si potrebbe fare direttamente induzione.

#### Corollario 7.4.1 Condizione debole

Una condizione è più debole rispetto agli altri se è valida in un numero maggiore di stati.

#### Note:-

La condizione più debole è quella che vale in qualsiasi stato (e. g.  $0 == 0$ ).

```
Fact :  $\forall \{P \ c \ Q\} \rightarrow \models [P] \ c \ [Q] \rightarrow (\forall s \rightarrow P \ s \rightarrow wp \ c \ Q \ s)$   

Fact {P} {c} {Q} = only-if  

where  

only-if : ( $\{s \ t : State\} \rightarrow P \ s \rightarrow ((c, s) = t \rightarrow Q \ t) \rightarrow$   

 $((s : State) \rightarrow P \ s \rightarrow wp \ c \ Q \ s)$ )  

only-if hyp1 s hyp2 t = hyp1 hyp2
```

### Definizione 7.4.3: Fatto

Se  $\{P\} \ c \ \{Q\}$  è valido e ogni stato  $s$  soddisfa  $P$  allora è vero il  $wlp \ c \ Q \ s$ .

```

wp-lemma :  $\forall c \{Q : \text{Assn}\} \rightarrow \vdash [\text{wp } c \ Q] \ c \ [Q]$ 

wp-lemma SKIP {Q} = H-Str left-premise H-Skip
  where
    left-premise :  $\forall s \rightarrow \text{wp } \text{SKIP} \ Q \ s \rightarrow Q \ s$ 
    left-premise s hyp = hyp s Skip

-- hyp :  $\text{wp } \text{SKIP} \ Q \ s \equiv \forall t \rightarrow ((\text{SKIP}, s) = t \rightarrow Q \ t)$ 
-- hyp s :  $((\text{SKIP}, s) = s \rightarrow Q \ s)$ 
-- Skip :  $((\text{SKIP}, s) = s)$ 
-- hyp s Skip :  $Q \ s$ 

wp-lemma (x := a) {Q} = H-Str left-premise H-Loc
  where
    left-premise :  $\forall s \rightarrow \text{wp } (x := a) \ Q \ s \rightarrow Q \ (s \ [x ::= \text{aval } a \ s])$ 
    left-premise s hyp = hyp (s [x ::= aval a s]) Loc

-- hyp :  $\text{wp } (x := a) \ Q \ s \equiv \forall t \rightarrow ((x := a, s) = t \rightarrow Q \ t)$ 
-- hyp (s [x ::= aval a s]) :  $((x := a, s) = (s \ [x ::= \text{aval } a \ s]) \rightarrow Q \ (s \ [x ::= \text{aval } a \ s]))$ 
-- Loc :  $((x := a, s) = (s \ [x ::= \text{aval } a \ s]))$ 
-- hyp (s [x ::= aval a s]) Loc :  $Q \ (s \ [x ::= \text{aval } a \ s])$ 

wp-lemma (c1 :: c2) {Q} = H-Str left-premise (H-Comp IH1 IH2)
  where
    IH1 :  $\vdash [\text{wp } c1 \ (\text{wp } c2 \ Q)] \ c1 \ [\text{wp } c2 \ Q]$ 
    IH1 = wp-lemma c1 {wp c2 Q}

    IH2 :  $\vdash [\text{wp } c2 \ Q] \ c2 \ [Q]$ 
    IH2 = wp-lemma c2 {Q}

    left-premise :  $\forall s \rightarrow \text{wp } (c1 :: c2) \ Q \ s \rightarrow \text{wp } c1 \ (\text{wp } c2 \ Q) \ s$ 
    left-premise s hyp1 r hyp2 t hyp3 = hyp1 t (Comp hyp2 hyp3)

```

#### Definizione 7.4.4: Wlp-lemma - parte I

- ⇒ caso SKIP: la premessa da cui si parte consiste nella dimostrazione che  $\text{wlp } \text{SKIP} \ Q \ s$  implichi  $Q \ t$ . Per completare la dimostrazione si usa il rafforzamento con la premessa e la regola di Hoare per SKIP (H-Skip);
- ⇒ caso LOC: il procedimento è analogo a quello per SKIP;
- ⇒ caso COMP: si utilizza una composizione di funzioni.

```

wp-lemma (IF b THEN c1 ELSE c2) {Q} = H-If true-premise false-premise
where
  P = wp (IF b THEN c1 ELSE c2) Q

  IH1 : ⊢ [ wp c1 Q ] c1 [ Q ]
  IH1 = wp-lemma c1 {Q}

  left-premise-true : ∀ s → P s ∧ bval b s = true → wp c1 Q s
  left-premise-true s (Ps , b=true) t hyp = Ps t (IfTrue b=true hyp)

-- Ps t : P s t ≡ (( IF b THEN c1 ELSE c2 , s )) ⇒ t → Q t
-- hyp : (( c1 , s )) ⇒ t
-- b=true : bval b s = true
-- IfTrue b=true hyp : (( IF b THEN c1 ELSE c2 , s )) ⇒ t

  true-premise : ⊢ [ (λ s → P s ∧ bval b s = true) ] c1 [ Q ]
  true-premise = H-Str left-premise-true IH1

  IH2 : ⊢ [ wp c2 Q ] c2 [ Q ]
  IH2 = wp-lemma c2 {Q}

  left-premise-false : ∀ s → P s ∧ bval b s = false → wp c2 Q s
  left-premise-false s (Ps , b=false) t hyp = Ps t (IfFalse b=false hyp)

-- Ps t : (( IF b THEN c1 ELSE c2 , s )) ⇒ t → Q t
-- hyp : (( c2 , s )) ⇒ t
-- b=false : bval b s = false
-- IfFalse b=false hyp : (( IF b THEN c1 ELSE c2 , s )) ⇒ t

  false-premise : ⊢ [ (λ s → P s ∧ bval b s = false) ] c2 [ Q ]
  false-premise = H-Str left-premise-false IH2

```

#### Definizione 7.4.5: Wlp-lemma - parte II

```

wp-lemma (WHILE b DO c) {Q} = H-Weak claim2 weak-premise
  where
    P = wp (WHILE b DO c) Q

    IH :  $\vdash [wp\ c\ P] \ c\ [P]$ 
    IH = wp-lemma c {P}

    str-premise :  $\forall s \rightarrow P\ s \wedge bval\ b\ s = true \rightarrow wp\ c\ P\ s$ 
    str-premise s (Ps , b=true) t hyp r w =
      Ps r (WhileTrue b=true hyp w)

-- Ps r :  $\langle\langle\ WHILE\ b\ DO\ c , s \rangle\rangle \Rightarrow r \rightarrow Q\ r$ 
-- w :  $\langle\langle\ WHILE\ b\ DO\ c , t \rangle\rangle \Rightarrow r$ 
-- hyp :  $\langle\langle\ c , s \rangle\rangle \Rightarrow t$ 
-- b=true :  $bval\ b\ s = true$ 
-- WhileTrue b=true hyp w :  $\langle\langle\ WHILE\ b\ DO\ c , s \rangle\rangle \Rightarrow r$ 
-- Goal:  $Q\ r$ 

    claim1 :  $\vdash [(\lambda s \rightarrow P\ s \wedge bval\ b\ s = true)] \ c\ [P]$ 
    claim1 = H-Str str-premise IH

    claim2 :  $\vdash [P] \ WHILE\ b\ DO\ c\ [(\lambda s \rightarrow P\ s \wedge bval\ b\ s = false)]$ 
    claim2 = H-While claim1

    weak-premise :  $\forall s \rightarrow P\ s \wedge bval\ b\ s = false \rightarrow Q\ s$ 
    weak-premise s (Ps , b=false) = Ps s (WhileFalse b=false)

-- Ps s :  $\langle\langle\ WHILE\ b\ DO\ c , s \rangle\rangle \Rightarrow s \rightarrow Q\ s$ 
-- b=false :  $bval\ b\ s = false$ 
-- WhileFalse b=false :  $\langle\langle\ WHILE\ b\ DO\ c , s \rangle\rangle \Rightarrow s$ 

```

#### Definizione 7.4.6: Wlp-lemma - parte III

## 7.5 Verification Conditions

#### Definizione 7.5.1: Verification Conditions