Metodi formali dell'informatica

Luca Barra

Anno accademico 2023/2024

Indice

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 λ-calcolo

Il λ-*calcolo* è un sistema per calcolare usando le funzioni.

Definizione 1.2.1: La sintassi del λ-calcolo

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

dove

- x è il parametro formale;
- λx.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 λ-calcolo tipato

Il λ-*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 soddisfi 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 operazionale

La *semantica operazionale* 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 operazionale

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

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

P_i è la parte che resta da eseguire di P_{i-1}, σ_i è lo stato risultante dall'esecuzione della prima istruzione di P_{i-1} nello stato σ_{i-1}, se P_k è vuoto allora σ_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, ϕ) e una "post-condizione" (cosa calcola il programma,ψ).

Note:-

$\{\phi\}P\{\psi\}$ è vera nello stato σ se quando ϕ sia vera in σ e l'esecuzione di P da σ termni in λ', ψ è vera in σ'

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

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

[a] 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, ...)ϕ quindi si stabilisce se M soddisfa ϕ. 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 chè si può 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"

[1] 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 signatura

Note:-

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

Definizione 2.1.1: Signatura

Una signatura Σ è un insieme finito di k simboli $\{f_1, \dots, f_k\}$ e di una funzione che assegna a ciascuno di essi un'arietà[a] ar : $\Sigma \longrightarrow \mathbb{N}$

[a] A quanti operandi può essere applicato un operatore

Definizione 2.1.2: Insieme dei termini sulla signatura Σ

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:

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

[a] A ogni valore assegna il suo succssore

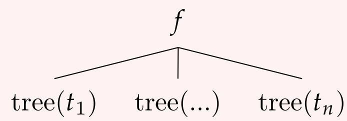
Note:-

Si può abbreviare, impropriamente, Succ(Succ(Zero)) con Succ^2(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 tree($f(t_1, \ldots, t_n)$)[a] allora si può definire l'albero sintattico



[a] ar(f) = n

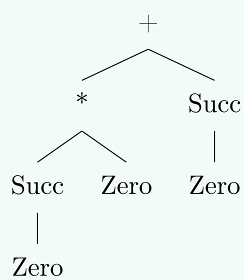
Esempio 2.1.3 (Conversione da espressione ad albero)

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

con $ar(+) = ar(*) = 2$

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

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 $X = \{x_0, x_1, \ldots\}$, l'insieme dei termini $\mathcal{T}_\Sigma(X)$ è:

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

Esempio 2.1.5 (Somma di un successore)

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

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

Definizione 2.1.5: Le variabili

In generale si possono definire le varibili come:

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

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 σ che assegna a ciascuna variabile un termine nella signatura

$$\sigma \,:\, X \longrightarrow \mathcal{T}_\Sigma$$

Definizione 2.2.2: Sostituzione generale

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

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

Note:-

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

Esempio 2.2.1 (Sostituzione)

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

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

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 σ. 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;
- $\text{match}(t, f(p_1, ..., p_n)) = \sigma_1 \cup ... \cup \sigma_2$ se:
  1. se $t \neq c$[a] alllora $t \neq g(t_1, ..., t_n)$
  2. se $t \equiv f(t_1, ..., t_n)$ allora $\text{match}(t_i, p_i) = \sigma_i$, con $i = 1, ..., n$;
  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.

[a] Costante

2.4 Sistemi di riscrittura

Definizione 2.4.1: Sistema di riscrittura

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

1. $l_i \notin X$ $(l_i \not\equiv x\ \forall x \in X)$[b];
2. $\text{var}(r_i) \subseteq \text{var}(l_i)$[c].

[a] Regole
[b] l_i non può essere una variabile
[c] Le variabili nella parte destra compaiono anche nella parte sinistra

Note:-

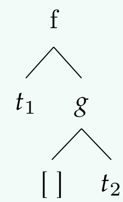
l indica il lato *sinistro* (left) della freccia
r indica il lato *destro* (right) della freccia

Definizione 2.4.2: Contesto

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

Esempio 2.4.1 (Albero di un contesto)

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

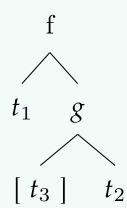


I contesti indicano che le regole di riduzioni vanno applicate in un punto preciso, sotto determinate condizioni.

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 ($t \longrightarrow_R s$) se esiste un contesto C[ ], una regola $l \longrightarrow r \in R$, e una sostituzione σ tali che

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

ossia t è un'istanza di l attraverso σ

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: $R = \{f(x) \longrightarrow a,\ g(f(x), y) \longrightarrow f(y)\}$
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[a] $\to R \in \mathcal{T}_\Sigma(X)^2$ è una relazione binaria per cui si può ridurre un termine in un altro

[a] One-step reduction

Corollario 2.4.1

$\xrightarrow{+} R$ rappresenta la più piccola riduzione tale che $\to R \subseteq \xrightarrow{+} R$ e $\xrightarrow{+} R$ sia transitiva[a]

[a] Riduzione in n passi con $n \geqslant 1$

Corollario 2.4.2

$\xrightarrow{*} R$ rappresenta la più piccola riduzione tale che $\to R \subseteq \xrightarrow{*} R$ e $\xrightarrow{*} R$ sia transitiva e riflessiva[a]

[a] Riduzione in n passi con $n \geqslant 0$

Corollario 2.4.3

$\xleftrightarrow{*} R$ rappresenta la più piccola riduzione tale che $\to R \subseteq \xleftrightarrow{*} R$ e $\xleftrightarrow{*} RR$ sia transitiva, riflessiva e simmetrica[a]. Questa relazione si chiama relazione di convertibilità

[a] Ossia si può ridurre in ambo i sensi

Definizione 2.4.4: Church-Rosser

R è confluente o Church-Rosser (CR) se

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

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 signatura Σ 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 = $\{s_1 \approx t_1, ..., s_n \approx t_n\} \subseteq T_\Sigma(X)^2$ (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*): $\frac{E \vdash s_1 \approx t_1 \quad E \vdash s_n \approx t_n}{E \vdash f(s_1, ..., s_n) \approx f(t_1, ..., t_n)}$;
- 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).

$$\mathrm{sym}\frac{\mathrm{trans}\dfrac{\mathrm{ax}_1\dfrac{}{\mathrm{cong}\frac{E \vdash a \approx b}{E \vdash f(a) \approx f(b)}} \qquad \mathrm{ax}_2\dfrac{}{\mathrm{sub}\frac{E \vdash f(x) \approx g(x)}{E \vdash f(b) \approx g(b)}}}{}}{}\quad \frac{E \vdash f(a) \approx g(b)}{E \vdash g(b) \approx f(a)}$$

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

Definizione 2.5.2: s $\leftrightarrow_R$ t

s $\leftrightarrow_R$ t $\overset{*}{\Longleftrightarrow}$ s $\rightarrow_R$ t $\vee$ t $\rightarrow_R$. Sia $\overset{*}{\longleftrightarrow}_R$ chiusura riflessiva e transitiva di $\leftrightarrow$

Corollario 2.5.2

Se R è CR allora

$$s \overset{*}{\longleftrightarrow} t \Leftrightarrow \exists r. s \overset{*}{\longrightarrow} r \vee t \overset{*}{\longrightarrow} r$$

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

2.5.1 Normalizzazione

Definizione 2.5.3: Normalizzazione (forte)

Fissati Σ e Q:

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

Corollario 2.5.3

Se R è CR e SN allora $s \overset{*}{\to}_R t$ è deducibile

Capitolo 3

Deduzione naturale di Gentzen

3.1 La deduzione

Definizione 3.1.1: Modus ponens (MP)

$$\frac{\phi \to \psi \qquad \phi}{\psi}\text{MP}$$

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

$$\frac{A \qquad B}{A \wedge B} \wedge \text{ I}$$

$$\frac{A \wedge B}{A} \wedge \text{ E}_1$$

$$\frac{A \wedge B}{B} \wedge \text{ E}_2$$

Definizione 3.2.2: L'implicazione

$$\begin{array}{c} [A]^i \\ \vdots \\ -i\,\dfrac{B}{A \to B} \to \text{ I} \end{array}$$

$$\frac{B \to A \qquad A}{B} \to \text{ E}$$

Note:-

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

Esempio 3.2.1

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

$$-1\,\dfrac{\dfrac{\dfrac{[A\wedge B]^1}{B}\wedge \mathrm{E}_2 \qquad \dfrac{[A\wedge B]^1}{A}\wedge \mathrm{E}_1}{B\wedge A}\wedge\ \mathrm{I}}{A\wedge B\to B\wedge A}\to\ \mathrm{I}$$

3.3 Vero, falso e negazione

Definizione 3.3.1: Vero

$$\dfrac{}{\top}\ \top\ \mathrm{I}$$

Note:-

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

Definizione 3.3.2: Falso

$$\dfrac{\bot}{A}\ \bot\ \mathrm{E}$$

Note:-

Il falso può solo essere introdotto

Definizione 3.3.3: Negazione

$$\begin{array}{c}[A]^i\\ \vdots\\ -i\,\dfrac{\bot}{\neg A}\neg\ \mathrm{I}\end{array}$$

$$\dfrac{\neg A \qquad A}{\bot}\neg\ \mathrm{E}$$

3.4 Disgiunzione

Definizione 3.4.1: Disgiunzione

$$\dfrac{A}{A\vee B}\vee\ \mathrm{I}_1$$

$$\dfrac{B}{A\vee B}\vee\ \mathrm{I}_2$$

$$-i,\ -j\,\dfrac{A\vee B \qquad C \quad C}{C}\vee\ \mathrm{E}$$

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

Esempio 3.4.1 (Legge di De Morgan)

$\vdash \neg(A \vee B) \to (\neg A \wedge \neg B)$

$$-1\,\dfrac{\dfrac{-2\,\dfrac{\dfrac{[\neg(A\vee B)]^1 \qquad \dfrac{[A]^2}{A\vee B}\vee\ \text{I}_1}{\perp}\neg\ \text{E}}{\neg A}\neg\ \text{I} \qquad \neg B}{\neg A\wedge\neg B}\wedge\ \text{I}}{\vdash \neg(A\vee B)\to(\neg A\wedge\neg B)}\to\ \text{I}$$

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

Esempio 3.4.2 (Doppia negazione)

$\vdash A \to \neg\neg A$

$$-1\,\dfrac{\dfrac{\dfrac{[\neg A]^2 \qquad [A]^1}{\perp}\neg\ \text{E}}{\neg\neg A}\neg\ \text{I}}{A\to\neg\neg A}\to\ \text{I}$$

Note:-

Nella deduzione naturale $\vdash A \to \neg\neg A$ vale (come appena dimostrato), ma $\vdash \neg\neg A \to A$ 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 $\vdash_{CL} \neg\neg A \to A$

Esempio 3.5.1

$\vdash \neg\neg A \to A$

$$-1\,\dfrac{-2\,\dfrac{\dfrac{[\neg\neg A]^1 \qquad [\neg A]^2}{\perp}}{A}RAA}{\neg\neg A\to A}\to\ \text{I}$$

3.6 Quantificatori

Note:-

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

Definizione 3.6.1: Quanrificatore universale

$$\alpha \notin FV(\Gamma)\dfrac{P(\alpha)}{\forall x\ \ P(x)}\forall\ \text{I}$$

$$\dfrac{\forall x\ \ P(x)}{P(t)}\forall\ \text{E}$$

Note:-

Γ è l'insieme delle premesse

Definizione 3.6.2: Quantificatore esistenziale

$$\frac{P(t)}{\exists\ P(x)}\exists\ \mathrm{I}$$

$$\alpha \notin FV(\Gamma) \cup FV(C)\frac{\exists x\ \ P(x) \qquad c}{c}\exists\ \mathrm{E}$$

Capitolo 4

Il λ-calcolo non tipato

Note:-
Questo capitolo è concettualmente affine al capitolo "Il λ-calcolo" negli appunti del corso di "Linguaggi e paradigmi di programmazione"

4.1 Introduzione

Il λ-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 Var $= \{x, y, z, ...\}$ un insieme finito di variabili, la sintassi è la seguente:

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

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

Note:-
(MN) è l'applicazione delle funzione M al parametro attuale N

4.2 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:

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

Definizione 4.2.2: Sostituzione

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

Definizione 4.2.3: α-equivalenza

L'α-equivalenza $\Leftrightarrow_\alpha$ è la congruenza tra λ-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: β-riduzione

La β-riduzione è la relazione tra λ-espressioni tale che:

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

Note:-

Nella β-riduzione:

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

$(\lambda x.M)N$ è un β-*redex*[a].
$M\{N/y\}$ è il suo *ridotto*.
Ci possono essere più modi di ridurre la stessa λ-espressione. La riduzione di un β-redex può creare altri β-redex. La riduzione di un β-redex può cancellare altri β-redex. La riduzione può non terminare.

[a] REDucible EXpression

Definizione 4.2.5: Church-Rosser nel λ-calcolo

R è confluente 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 aimmetrica di $\xrightarrow{*}$ allora $M =_\beta N \Rightarrow \exists L.M \xrightarrow{*} L \wedge N \xrightarrow{*} L$

Definizione 4.2.6: Booleani

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

[a]Combinatore K
[b]Combinatore O

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

Esempio 4.2.1

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

4.3 Numerali di Church

Definizione 4.3.1: Numerale di Church

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

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

Esempio 4.3.1

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

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

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

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

Definizione 4.3.2: Successore di un numerale

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

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

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

Esempio 4.3.2

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

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

$$= \underline{3}$$

Definizione 4.3.3: Somma

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

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

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

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

Note:-

Nello stesso modo si può definire $\underline{\text{mult}}$ come iterazione di $\underline{\text{add}}$

Definizione 4.3.4: Test per zero

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

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

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

Esempio 4.3.3

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

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

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

Definizione 4.3.5: 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:

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

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

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

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

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

$$\underline{\text{fact}}\ \underline{\text{n}} = \text{F } \underline{\text{fact}}\ \ n = ...\underline{\text{fact}}\ (\underline{\text{pred}}\ \ \underline{n})$$

$$= ......(\text{F } \underline{\text{fact}})(\underline{\text{pred}}\ \ \underline{n})$$

[a] Punto fisso

Note:-

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

Teorema 4.3.1 Teorema del punto fisso

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

Proof: Leggiamo l'equaziobe 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 4.3.6: Operatore a punto fisso (Y)

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

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

Note:-

Il λ-calcolo non tipato puro non è SN

Teorema 4.3.2 Teorema di Kleensn

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

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

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

Capitolo 5

Il λ-calcolo tipato

5.1 Tipi

Question 1

Come si interpreta un termine $X\ X$?

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

Esempio 5.1.1

Se il primo $X \in A \to A$ e il secondo $X \to A$ non esiste alcun $A \neq \{*\}$ tale che $A \simeq A \to A$ in Set[a]

[a] Categoria degli insiemi

Definizione 5.1.1: Tipi semplici

$$A,\ B ::= \alpha | A \to B$$

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

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

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

Definizione 5.1.2: Sistema di tipo

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

Definizione 5.1.3: Contesto

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

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

Corollario 5.1.1

Valgono le seguenti proprietà:

- ax$\overline{\Gamma,\ x{:}A \vdash x{:}A}$;

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

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

Capitolo 6

Logica costruttiva

Capitolo 7

Il linguaggio IMP

Per studiare il problema della verifica in programmi imperativi si utilizzerà un piccolo linguaggio di programmazione chiamato *IMP*[1].

7.1 Introduzione a IMP

Definizione 7.1.1: Comandi di IMP

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

Com ∈ c, c' ::= *SKIP* | x := a | c :: c' | *IF* b *THEN* c *ELSE* c' | *WHILE* b *DO* c

Note:-

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

⇒ *SKIP*: termina l'esecuzione senza effetti collaterali;

⇒ c :: c' : la composizione (in AGDA è c ; c').

Corollario 7.1.1 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 : State` con `State = Varname` $\longrightarrow$ `Val` ossia l'assegnazione di un valore (`Val`) a ogni variabile (`Varname`).

7.1.1 Le relazioni in IMP

In IMP esistono due possibili relazioni:

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

Teorema 7.1.1 Equivalenza di Big-step e Small-step

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

$$\forall c\ s\ t\ .\ (\!(c, s)\!) \Rightarrow t \Longleftrightarrow (\!(c, s)\!) \longrightarrow^* (\!(\mathit{SKIP}, t)\!)$$

Note:-

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

[1] A volte viene chiamato "while".

7.1.2 La logica di Floyd-Hoare

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

Definizione 7.1.2: Asserzioni

Un'*asserzione* (`P : Assn`), dove `Assn = State → Set`, è un predicato di stati.

Corollario 7.1.2 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 7.1.2 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:

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

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

7.2 Espressioni

In questa sezione si introducono le espressioni *aritmetiche* (`Aexp`) e le espressioni *booleane* (`Bexpr`).

Definizione 7.2.1: Variabili

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

[a] `Index` è ℕ.

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

Note:-

Negli esempi presentati assumeremo X = `Vn 0`, Y = `Vn 1` e Z = `Vn 2`.

Definizione 7.2.2: Confronto

Per confrontare due variabili definiamo la funzione `x =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 =ℕ y` per controllare che due 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
```

7.2.1 Espressioni aritmetiche

Definizione 7.2.3: Aexp

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

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

Dove n ∈ `Nat` e vn ∈ `Vname`.

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

Esempio 7.2.1 (X + (1 + Y))

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

Definizione 7.2.4: Stato

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

⇒ `Val` = ℕ;

⇒ `State` = `Vname` ⟶ `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 7.2.5: 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 7.2.2 (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 7.2.6: 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[2];
- Il caso *Plus* a1 a2 restituisce la somma aritmetica della valutazione ricorsiva su a1 e a2.

7.2.2 Sostituzione

Definizione 7.2.7: Sostituzione

La *sostituzione* consiste nel rimpiazzare ogni occorrenza di una varibile 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 ])
```

[2]Ovvero il suo valore salvato in memoria, come nei registri in Assembly.

Esempio 7.2.3 ((X + (1 + Y)) [(Z + 3) / X])

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

Lenma 7.2.1 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 [x ::= (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.

7.2.3 Espressioni booleane

Definizione 7.2.8: Bexp

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

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

Dove bc ∈ `Bool` e a, a' ∈ `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 7.2.4 (Alcuni esempi)

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

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

Definizione 7.2.9: Confronto

La valutazione delle espessioni 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
```

7.3 Semantica Big-step

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

7.3.1 Comandi

Definizione 7.3.1: Comandi

La sintassi dei *comandi* si basa sulla grammatica:

`Com` ∈ c, c' ::= *SKIP* | x := a | c :: c' | *IF* b *THEN* c *ELSE* c' | *WHILE* b *DO* c

Dove x ∈ `Vname`, a ∈ `Aexp` e b ∈ `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
```

7.3.2 Convergenza

Definizione 7.3.2: Predicato di convergenza

La relazione $(\!(c, s)\!) \Rightarrow t$ 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 7.3.1 Configurazioni
Chiamiamo *configurazioni* ogni coppia $(\!(c, s)\!)$ comando-stato.

```
data Config : Set where
     ⦅_,_⦆  : Com → State → Config
```

Definizione 7.3.3: Relazione

Si definisce la relazione ⇒ tra `Config` e `State` per creare un sistema formale.

```
data _⇒_ : Config → State → Set  where

    Skip : ∀ {s}
          ───────────────
          → ⟨ SKIP , s ⟩  ⇒ s

    Loc : ∀{x a s}
         ────────────────────────────────────
         → ⟨ x := a , s ⟩  ⇒ (s [ x ::= aval a s ])

    Comp : ∀{c₁ c₂ s₁ s₂ s₃}
          → ⟨ c₁ , s₁ ⟩  ⇒ s₂
          → ⟨ c₂ , s₂ ⟩  ⇒ s₃
          ───────────────────
          → ⟨ c₁ :: c₂ , s₁ ⟩  ⇒ s₃

    IfTrue : ∀{c₁ c₂ b s t}
            → bval b s ≡ true
            → ⟨ c₁ , s ⟩  ⇒ t
            ─────────────────────────────
            → ⟨ IF b THEN c₁ ELSE c₂ , s ⟩  ⇒ t

    IfFalse : ∀{c₁ c₂ b s t}
             → bval b s ≡ false
             → ⟨ c₂ , s ⟩  ⇒ t
             ─────────────────────────────
             → ⟨ IF b THEN c₁ ELSE c₂ , s ⟩  ⇒ t

    WhileFalse : ∀{c b s}
                → bval b s ≡ false
                ──────────────────────
                → ⟨ WHILE b DO c , s ⟩  ⇒ s

    WhileTrue  : ∀{c b s₁ s₂ s₃}
                → bval b s₁ ≡ true
                → ⟨ c , s₁ ⟩  ⇒ s₂
                → ⟨ WHILE b DO c , s₂ ⟩  ⇒ s₃
                ───────────────────────
                → ⟨ WHILE b DO c , s₁ ⟩  ⇒ s₃

infix 10 _⇒_
```

7.3.3 Proprietà della convergenza

Teorema 7.3.1 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;
- hyp1 $= \langle\!\langle c, s \rangle\!\rangle \Rightarrow s_2$;
- hyp2 $= \langle\!\langle \text{WHILE B true DO c}, s_2 \rangle\!\rangle \Rightarrow t$.

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

Teorema 7.3.2 Determinismo
Ogni volta che $\langle\!\langle c, s \rangle\!\rangle \Rightarrow t$ è derivabile per qualche $\langle\!\langle c, s \rangle\!\rangle \in$ `Config` e $t \in$ `State`, lo stato t è unico.

Note:-
Per provare questo teorema abbiamo bisogno di due lemmi.

Lenma 7.3.1 Una cosa o è vera o è falsa

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

Lenma 7.3.2 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 : ∀ {c : Com} {s t t' : State} →
                          ⦅ c , s ⦆ ⇒ t → ⦅ c , s ⦆ ⇒ t' → 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).

7.3.4 Equivalenza

Definizione 7.3.4: Equivalenza

Due comandi c, c' ∈ `Com` sono equivalenti per ogni s ∈ `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 ∈ `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 7.3.1 (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.

7.4 Semantica Small-step

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

7.4.1 Riduzione in un passo

Definizione 7.4.1: Relazione di riduzione in un passo

La relazione $\llparenthesis c, s \rrparenthesis \longrightarrow \llparenthesis c', s' \rrparenthesis$ modella l'esecuzione del comando "più a sinistra" in c iniziando da s, producendo la nuova configurazione $\llparenthesis c', s' \rrparenthesis$ dove c' (*continuazione*) è ciò che resta da eseguire di c e s' è il nuovo stato prodotto. La relazione $\longrightarrow$ 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_1 indica che il primo comando è terminato e quindi l'esecuzione continua con il prossimo;
- Comp_2 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).

7.4.2 Chiusure

Definizione 7.4.2: 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 ⟶ *-refl* postula la riflessività;
- La regola ⟶ *-incl* concatena la riduzione in un passo alla riduzione in più passi[3].

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
```

[3] Ricorda il cons nelle liste.

7.5 Relazione tra semantica Big-step e semantica Small-step

Capitolo 8

Logica di Floyd-Hoare