

Lambda Calcolo

| λ-calcolo

Formalismo che consente di rappresentare procedure di calcolo in termini di composizioni e applicazioni di funzioni minimali

Sintassi

Consente di scrivere espressioni che contengono solo dichiarazioni di funzioni anonime

$e ::= x$ (variabile)

x rappresenta un qualsiasi variabile

$\lambda x.e$ (Astrazione Funzionale)

$\lambda x.e$ esprime una funzione anonima con parametro formale x e corpo e (es. $\lambda x.x$ è funzione identità)

$e e$ (Applicazione)

Esempio:

Javascript: `function(x){return x}`

λ-calcolo: $\lambda x.x$

Possiamo anche applicare una funzione ad una nuova variabile utilizzando ee

Esempio

Applichiamo la funzione identità su x alla variabile y
 $(\lambda x.x)y$

Esempi di λ-espressioni sintatticamente corrette

- $(\lambda x.x)(\lambda y.y)$
- $((\lambda x.(\lambda y.(xy)))z)k$
- $((\lambda x.(\lambda y.y))z$
- $((\lambda x.(\lambda y.(xy)))(\lambda z.z))k$

Convenzioni sintattiche

① lo scopo del costruttore λ si estende il più a destra possibile ($\lambda x.\lambda y.\lambda z \equiv \lambda x.(\lambda y.(\lambda z))$)

② il costrutto di applicazione è associativo e sinistro ($xyz \equiv (xy)z$ ossia, prima si applica x a y e poi il risultato ottenuto a z)

grazie a queste regole, gli esempi di sopra diventano:

- $(\lambda x.x)(\lambda y.y)$
- $(\lambda x.\lambda y.xy)zk$
- $((\lambda x.(\lambda y.y))z$
- $((\lambda x.(\lambda y.xy))(\lambda z.z))k$

Variabili libere e legate

$(\lambda x.z)$ → variabile libera, non può essere sostituita dato che non è legata da nessun λ

↪ variabile legata al costruttore e quindi può essere sostituita

Insieme delle variabili libere

L'insieme delle variabili libere è denotato $Fv(e)$ ed è definito ricorsivamente da e

$$Fv(x) = \{x\}$$

$$Fv(\lambda x.e) = Fv(e) \setminus \{x\}$$

$$Fv(e_1, e_2) = Fv(e_1) \cup Fv(e_2)$$

Esempi:

$$\begin{aligned} & \cdot Fv((\lambda x.(\lambda y.xy))(\lambda z.z)k) \\ &= Fv((\lambda x.(\lambda y.xy))(\lambda z.z)) \cup Fv(k) \\ &= Fv(\lambda x.(\lambda y.xy)) \cup Fv(\lambda z.z) \cup \{k\} \\ &= (Fv(\lambda x.(\lambda y.xy)) \cup \emptyset) \cup \{k\} \\ &= (Fv(\lambda y.xy) \setminus \{x\}) \cup \{k\} \\ &= ((Fv(y) \setminus \{y\}) \setminus \{x\}) \cup \{k\} \\ &= (((Fv(x) \cup Fv(y)) \setminus \{y\}) \setminus \{x\}) \cup \{k\} \\ &= (((\{x\} \cup \{y\}) \setminus \{y\}) \setminus \{x\}) \cup \{k\} = (\{x\} \setminus \{x\}) \cup \{k\} = \emptyset \cup \{k\} = \{k\} \end{aligned}$$

Notioni di α -equivalenza e α -conversione

Due espressioni e_1, e_2 sono α -equivivalenti ($e_1 \equiv_\alpha e_2$) se una può essere ottenuta dall'altra rinominando una o più variabili legate, senza però modificare le variabili libere, ovvero non legate al tempo di α -equivalenza.

Dato una espressione e_1 , l'operazione di ridenominazione delle variabili legate tale che la trasforma in e_2 ($e_1 \equiv_\alpha e_2$) è detta α -conversione.

Esempi:

- $\lambda x.x \equiv_\alpha \lambda y.y$
- $\lambda x.\lambda y.xy \equiv_\alpha \lambda y.\lambda x.yx$
- $(\lambda x.x)x \equiv_\alpha (\lambda y.y)x$

Valutazione delle λ -espressioni

Processo con una serie di passi di calcolo, per arrivare ad un risultato finale. I passi saranno in sostanza tutti passi di funzione.

β -riduzione: Operazione di applicazione funzionale del λ -calcolo.

redex: porzione di λ -espressione in cui viene applicata la β -riduzione e viene sostituita dal risultato, ovvero non contiene più funzioni pronte per essere applicate.

Applicare una β -riduzione

Si prende l'argomento e lo sostituisce sintatticamente a tutte le occorrenze del parametro formale che si trovano nel corpo della funzione.

$(\lambda x.x)z \rightsquigarrow$ Argomento della funzione λ
↳ variabile restituita in output
↳ variabile presa in input

Esempi:

$$\begin{aligned} (\lambda x.x)y &\rightarrow y \\ (\lambda x.x)(\lambda y.y)z &\rightarrow (\lambda y.y)z \rightarrow z \end{aligned}$$

Strutture di un redux

Possiamo vedere un redux come una struttura $(\lambda x.e)e_2$, questa struttura deve essere individuata così da poter applicare la β -riduzione.

Nell'individuare un redux dobbiamo stare attenti a non avere parentesi aperte o chiuse tra la λ -estrazione $(\lambda x.e)$ e l'argomento e_2 .

Quando ci sono redux uno dentro l'altro, abbiamo più modi per risolvere.

Capture avoid substitution

$$\left. \begin{array}{l} x := e_2 \\ y := e_2 \end{array} \right\} = e_2 \quad \left. \begin{array}{l} \text{Se applichiamo la sostituzione } \{x := e_2\} \text{ ad un'espressione costituita da} \\ \text{solo una variabile, la sostituzione avrà luogo solo se tale variabile è proprio } x \end{array} \right\}$$

$$(e'e'')\{x := e_2\} = (e'\{x := e_2\})(e''\{x := e_2\}) \quad \left. \begin{array}{l} \Rightarrow \text{Se l'espressione } e \text{ a cui applichiamo la sostituzione } e' \text{ di tipo } e'e'' \text{ allora la sostituzione dovrà propagarsi} \\ \text{riconcavamente sulle due sottoespressioni} \end{array} \right.$$

$$(\lambda y.e)\{x := e_2\} = \lambda y.(e\{x := e_2\}) \quad \left. \begin{array}{l} \text{se } x \neq y \wedge y \notin Fv(e) \\ \Rightarrow \text{il parametro formale } y \text{ non appartiene alle variabili libere di } e_2 \text{ e quindi in questo} \\ \text{caso si può procedere con la sostituzione nel corpo della } \lambda\text{-estrazione} \end{array} \right.$$

$$(\lambda x.e)\{x := e_2\} = \lambda z.((e\{y := e_2\})\{x := e_2\}) \quad \left. \begin{array}{l} \text{se } x \neq y, y \in Fv(e) \text{ e } z \text{ fresca} \\ \Rightarrow \text{se } y \text{ è sia un parametro formale che una variabile libera di } e_2 \\ \text{allora si opera in doppia sostituzione, ossia, prima di applicare la sostituzione } \{x := e_2\} \\ \text{nella corpo della } \lambda\text{-estrazione, ve rinominato il parametro formale } y \\ \text{utilizzando una variabile } z \text{ detta } \text{fresca}, \text{ ossia che non compare da} \\ \text{nessun altro punto delle } \lambda\text{-espressione.} \end{array} \right.$$

Ottenendo una espressione α -equividente alla precedente.

Esempi:

- $((\lambda x.yx)w)\{x := \lambda k.kx\} = (\lambda x.yx)w$

In questo esempio la sostituzione non ha luogo in quanto x non è libero

- $((\lambda x.yx)w)\{y := \lambda k.kx\} \equiv_a ((\lambda z.yz)w)\{y := \lambda k.kx\} = \lambda z(\lambda k.kx)z w$

L'espressione $\lambda k.kx$ contiene la variabile libera x che rischia di essere catturata, per questo, prima di operare è necessario rinominare il parametro. Formale x usando una variabile fresca z

- $((\lambda x.yx)w)\{w := \lambda k.kx\} = (\lambda x.yx)(\lambda k.kx)$

In questo esempio, la sostituzione avviene senza problemi

β -riduzione

Describe un passo di calcolo di una λ -espressione e ci fa evitare problemi con le variabili libere

- Identificato un **redex** $(\lambda x.e)e_2$ all'interno dell'espressione, questo viene processato con le seguenti regole: $(\lambda x.e)e_2 \rightarrow e_1\{x := e_2\}$
il risultato ottenuto sostituisce il redex, mentre tutto il resto rimane invariato

esempio: $(\lambda x.\lambda y.xy)y$

- senza β -riduzione:** Sostituendo l'argomento y al parametro Formale x , ottenendo $\lambda y.y y$ ma in questo modo modificiamo una variabile libera, e ciò non si può

- con β -riduzione:**
 - Applichiamo l'argomento y alla funzione $(\lambda x.\lambda y.xy)$. poiché y è legato nell'ambiente esterno, rinominiamo le variabili legate all'interno in modo da non avere conflitti: $(\lambda x.\lambda z.xz)y$
 - Applichiamo y a $(\lambda x.\lambda z.xz)$ ossia sostituendo ogni occorrenza di x con y ottenendo $\lambda z.yz$
 - Mostriamo l'equivalenza fra le due funzioni: $(\lambda x.\lambda y.xy)y \equiv_a (\lambda x.\lambda z.xz)y \rightarrow \lambda z.yz$

La valutazione completa di una λ -espressione è una sequenza di passi di applicazione delle β -riduzione che porta ad un'espressione non ulteriormente riducibile, ovvero una **forma normale beta**, che rappresentano di fatto i valori che si possono ottenere come risultato

esempi di valori: $\lambda x.x$, $\lambda x.xy$, x , xy

Si può dire quindi, grazie ai prime due esempi, che le funzioni sono valori

β -equivalezza

Due λ -espressioni e_1, e_2 si dicono β -equivarianti: $e_1 \equiv_\beta e_2$ se vale una delle seguenti condizioni: (\Rightarrow , indica una serie di β -riduzioni consecutive)

- $e_1 \equiv_\beta e_2$
- $e_1 \Rightarrow e'_1 \equiv_a e_2$ oppure $e_2 \Rightarrow e'_2 \equiv_a e_1$
- $e_1 \Rightarrow e'_1 \wedge e_2 \Rightarrow e'_2$ con $e'_1 \equiv_a e'_2$

In sostanza, due espressioni sono β -equivarianti se e solo se porterebbero allo stesso risultato

esempi:

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

$$(\lambda x.\lambda y.xy)(\lambda x.x)z \equiv_\beta (\lambda k.k)z$$

Teorema di Church-Rosser

date due λ -espressioni e , se esistono due sequenze di β -riduzioni diverse che consentono di ridurre e in e_1 o in e_2 non β -equivarianti, allora esiste anche una λ -espressione e' tale che entrambe e_1 ed e_2 potranno essere ridotte in e' (o a qualcosa di non β -equividente)

il teorema quindi implica che qualunque scelta si operi nell'applicare la β -riduzione, le espressioni ottenute convergeranno verso lo stesso risultato

Confluenza: diversi flussi di esecuzione vengono a confluire nello stesso risultato finale

Oss. la valutazione di una λ -espressione può non terminare ($\underline{sL} = (\lambda x.xx)(\lambda x.xx)$) e questa non terminazione può essere anche causata da una sbagliata scelta di redex

λ -calcolo e programmazione funzionale

Uno degli elementi chiave della programmazione funzionale è il fatto che le funzioni sono valori e come tali possono essere il risultato di una computazione o anche passate come argomenti ad altre funzioni.

Funzioni Higher-order

Sono funzioni che accettano una o più funzioni come argomenti e/o restituisce una funzione come risultato.

```
function applica_due_volti(Funzione, valore)
    return Funzione(Funzione(valore))
```

Apply = $(\lambda f. \lambda x. f x)$ la funzione Apply prende una funzione f , un argomento x e poi applica f ad x .

poiché il λ -calcolo prevede estrazioni funzionali con un solo parametro, si può evitare ciò combinando le estrazioni funzionali cioè $\lambda x. \lambda y ...$

Funzioni Curried

Sono funzioni che accettano uno o più argomenti in modo parziale restituendo una funzione che accetta gli argomenti rimanenti o restituisce il risultato. In pratica le tecniche di currying permette di trasformare una funzione che accetta più argomenti in una sequenza di funzioni omnidate, ognuna delle quali accetta un singolo argomento.

```
function somma(x,y)
    return x+y
```

```
Function somma(x)
    return function(y)
        return x + y
```

Ricorsione: il combinatore Y

Nel λ -calcolo dato che le funzioni non hanno nomi non è possibile richiamarle. Questo si può bypassare grazie al combinatore Y definito come segue: $Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$

data una qualsiasi λ -espressione F il combinatore Y soddisfa la seguente proprietà del punto fisso: $YF \equiv_p F(YF)$

Come utilizzare il combinatore per funzioni ricorsive

- Si definisce una λ -espressione e nel cui interno si usa un nome f per effettuare le chiamate ricorsive. In sostanza, e rappresenta il corpo della funzione ricorsiva f che si intende definire.
- Si definisce la λ -espressione $G = \lambda f. e$
- Si definisce $F = YG$

Esempio:

Consideriamo l'espressione $e = \lambda y. (f y) y$ che rappresenta una funzione che applica ricorsivamente f al suo parametro y aggiungendo un'istanza di y in fondo.

- Definiamo la λ -estrazione:
 $G = \lambda f. e$
- Applichiamo il combinatore Y :
 $F = YG$
- Applicando F ad un argomento K , le sue esecuzioni sarà le seguenti:

$$Y G K$$

$$\Rightarrow G((\lambda x. G(xx))(\lambda x. G(xx))K)$$

$$\rightarrow \exists p ((YG)y)yK$$

$$\rightarrow (YG)KK$$

$$\rightarrow \dots$$

```
const Y = f => (x => x(x))(x => f(y => x(x)(y)))
const factorial = f => (x => (x==1 ? 1 : x * f(x-1)))
const ris = Y(factorial)(10)
```

Representazione dati

Letterali booleani

$$\text{True} = \lambda t. \lambda f. t$$

$$\text{False} = \lambda t. \lambda f. f$$

Letterali numerici

Il numero n è codificato come la λ -espressione C_n definita come segue:

$$C_n = \lambda z. \lambda s. s(s(\dots(s(z))\dots))$$
 dove s è ripetuto n volte

Il parametro z rappresenta lo zero

Il parametro s rappresenta la funzione successore. Quindi $C_0 = \lambda z. \lambda s. z$, $C_1 = \lambda z. \lambda s. s(z)$, ecc...

Controllo del flusso di esecuzione

Espressioni condizionali

$$\text{IF} = \lambda c. \lambda \text{then}. \lambda \text{else}. c \text{ then else}$$
 (il parametro c è la condizione booleana valutata)

Esempio: IF true e₁ e₂

$$\begin{aligned}
 \text{IF true } e_1, e_2 &= (\lambda c. \lambda \text{then}. \lambda \text{else}. c \text{ then else}) \text{true } e_1, e_2 \\
 &\rightarrow (\lambda \text{then}. \lambda \text{else}. \text{true} \text{ then else}) e_1, e_2 \\
 &\rightarrow (\lambda \text{else}. \text{true } e_1, \text{ else}) e_2 \\
 &\rightarrow \text{true } e_1, e_2 \\
 &\rightarrow (\lambda t. \lambda f. t) e_1, e_2 \\
 &\rightarrow (\lambda f. e) e_2 \\
 &\rightarrow e_1
 \end{aligned}$$

Operazioni sui numeri

$$\text{Plus} = \lambda m. \lambda n. \lambda z. \lambda s. (m(n \in s))s$$

$$\text{Times} = \lambda m. \lambda n. (m C_0 (\text{Plus } n))$$

Cell-by-name (lazy)

L'argomento viene passato così come scritto, applicando quindi la funzione più esterna. Valutazione più complicate ma offre vantaggi prestazionali.

Cell-by-value (eager)

L'argomento viene valutato per primo e poi si applica la funzione più esterna al risultato ottenuto. Valutazione più semplice.

Regole di inferenza

β -riduzione standard

$$(\lambda x. e_1)e_2 \rightarrow e_1\{x := e_2\}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2} \quad \frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}$$

Esempio: $(\lambda x. xx)((\lambda y.yyy)k)$

$$\frac{(\lambda y.yyy)k \rightarrow kkk}{(\lambda x. xx)((\lambda y.yyy)k) \rightarrow (\lambda x. xx)(kkk)}$$

Queste tre regole di inferenza consentono di ridurre sotto-espressioni senza vincolare le scelte su quale processare.

Cell-by-name

$$(\lambda x. e_1)e_2 \rightarrow e_1\{x := e_2\} \quad \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

Cell-by-value

$$(\lambda x. e_1)v \rightarrow e_1\{x := v\} \quad \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2}$$

$\hookrightarrow \lambda$ -espressione non ulteriormente riducibile (ossia un valore)

CONTROLLO DEI TIPI

Cose sono i tipi

Il tipo è un attributo di dato che descrive come il linguaggio di programmazione permette di usare quel particolare dato. I linguaggi di programmazione prevedono i tipi di dati di base come: interi, numeri in virgola mobile, caratteri, booleans.

- Un tipo limita i valori che un'espressione può assumere. Inoltre definisce le operazioni che possono essere effettuate sui dati e il modo in cui i valori di quel tipo possono essere memorizzati.

Sistemi di tipi

Metodo sintetico, effettivo per dimostrare l'essenza di comportamenti anomali del programma strutturando le operazioni del programma in base ai tipi di valori che calcolano.

Metodo sintetico: Le strutture sintetiche guida il metodo di analisi del comportamento del programma.

Effettivo: Si può definire un algoritmo che controlla i vincoli sui tipi e implementarlo in un compilatore o un interpretante.

Strutturale: i tipi assegnati alle componenti di un programma sono calcolati in modo complessionale, ossia che il tipo di un'espressione dipende dai tipi delle sue sottospressioni.

- Un sistema dei tipi associa tipi ai valori calcolati.
- Esaminando il flusso dei valori calcolati, il sistema dei tipi tenta di dimostrare che non avvengano errori di tipo.
- Il sistema stesso determina che cosa costituisce un errore di tipo, garantendo che le operazioni lavorino solo per i tipi per i quali sono stati realizzati.

Controllo dei tipi

Statico: A tempo di compilazione e trova gli errori a tempo di esecuzione, non degrada le prestazioni.

Dinamico: A tempo di esecuzione.

Type checker: verifica che le intenzioni del programmatore siano rispettate.

Sintassi

Espressioni: $E ::= \text{true} \mid \text{false} \mid NV \mid \text{IF } E \text{ then } E_1 \text{ else } E_2 \mid \text{succ } E \mid \text{pred } E \mid \text{isZero } E$

Valori: $\text{true} \mid \text{false} \mid NV$

Valori numerici: $NV ::= 0 \mid 1 \mid 2 \mid \dots$

Tipi per espressioni: $T ::= \text{Bool} \mid \text{Nat}$

Regole di valutazione

$\text{if true then } E_1 \text{ else } E_2 \rightarrow E_1$	IF-TRUE	$\text{if false then } E_1 \text{ else } E_2 \rightarrow E_2$	IF-FALSE
$\frac{E \rightarrow E'}{\text{if } E \text{ then } E_1 \text{ else } E_2 \rightarrow \text{if } E' \text{ then } E_1 \text{ else } E_2}$	IF-COND	$\frac{E \rightarrow E' \quad \text{succ } E \rightarrow \text{succ } E'}{\text{succ } E \rightarrow \text{succ } E'}$	$\frac{m = n + 1}{\text{succ } n \rightarrow m}$
$\frac{E \rightarrow E' \quad \text{pred } E \rightarrow \text{pred } E'}{\text{pred } E \rightarrow \text{pred } E'}$	$\frac{n > 0 \quad m = n - 1}{\text{pred } n \rightarrow m}$	$\frac{\text{pred } 0 \rightarrow 0}{E \rightarrow E'}$	$\frac{\text{isZero } 0 \rightarrow \text{true}}{E \rightarrow E'}$
		$\frac{}{\text{isZero } E \rightarrow \text{isZero } E'}$	$\frac{isZero \ 0 \rightarrow \ true \quad n > 0}{isZero \ n \rightarrow false}$
		$\frac{0: \text{Nat} \quad 1: \text{Nat}}{\text{isZero } 0: \text{Bool}}$	$\frac{2: \text{Nat} \quad pred \ 1: \text{Nat}}{\text{if isZero } 0 \text{ then } 2 \text{ else pred } 1: \text{Nat}}$

Algoritmo di type checking

```

typeof(E) = if (E = true) then Bool
else if (E = false) then Bool
else if (E = if E1 then E2 else E3) then {
    T1 = typeof(E1); T2 = typeof(E2); T3 = typeof(E3)
    if (T1 = Bool and T2=T3) then T2
    else "type error"
else if (E = n) then Nat
else if (E = succ E1) then {
    T1 = typeof(E1)
    if (T1 = Nat) then Nat else "type error"
}
else if (E = pred E1) then {
    T1 = typeof(E1)
    if (T1 = Nat) then Nat else "type error"
}
else if (E = iszero E1) then {
    T1 = typeof(E1)
    if T1 = Bool then Bool else "type error"
}

```

<pre> typeof(E) = if (E = true) then Bool else if (E = false) then Bool else if (E = if E1 then E2 else E3) then { T1 = typeof(E1); T2 = typeof(E2); T3 = typeof(E3) if (T1 = Bool and T2=T3) then T2 else "type error" else if (E = n) then Nat else if (E = succ E1) then { T1 = typeof(E1) if (T1 = Nat) then Nat else "type error" } else if (E = pred E1) then { T1 = typeof(E1) if (T1 = Nat) then Nat else "type error" } else if (E = iszero E1) then { T1 = typeof(E1) if T1 = Bool then Bool else "type error" } </pre>	<table border="0"> <tr> <td><i>true : Bool</i></td> <td><i>false : Bool</i></td> </tr> <tr> <td><i>E: Bool, E1: T, E2: T</i></td> <td><i>if E then E1 else E2: T</i></td> </tr> <tr> <td><i>E: Nat</i></td> <td><i>n: Nat</i></td> </tr> <tr> <td><i>succ E: Nat</i></td> <td></td> </tr> <tr> <td><i>E: Nat</i></td> <td><i>pred E: Nat</i></td> </tr> <tr> <td><i>E: Nat</i></td> <td><i>isZero E: Bool</i></td> </tr> </table>	<i>true : Bool</i>	<i>false : Bool</i>	<i>E: Bool, E1: T, E2: T</i>	<i>if E then E1 else E2: T</i>	<i>E: Nat</i>	<i>n: Nat</i>	<i>succ E: Nat</i>		<i>E: Nat</i>	<i>pred E: Nat</i>	<i>E: Nat</i>	<i>isZero E: Bool</i>
<i>true : Bool</i>	<i>false : Bool</i>												
<i>E: Bool, E1: T, E2: T</i>	<i>if E then E1 else E2: T</i>												
<i>E: Nat</i>	<i>n: Nat</i>												
<i>succ E: Nat</i>													
<i>E: Nat</i>	<i>pred E: Nat</i>												
<i>E: Nat</i>	<i>isZero E: Bool</i>												

Type Safety

Le correttezze del sistema di tipo e' espresso formalmente dalle seguenti proprietà:

Progresso: Una espressione ben tipata non si blocca a run-time

- Se $E:T$ allora E e' un valore oppure $E \rightarrow E'$ per una qualche espressione E'

Conservazione: i tipi sono preservati dalle regole di esecuzione

- Se $E:T$ e $E \rightarrow E'$ allora $E':T$

Queste due proprietà, messe insieme, garantiscono che l'intera esecuzione dell'espressione non si blocca a run-time

Definizioni per il λ -calcolo tipato

Sintassi	$V ::= true \mid false \mid fun\ x:\tau = e$
	$e ::= x \mid fun\ x:\tau = e \mid Apply(e,e) \mid true \mid false$
Regole di controllo dei tipi	
$\Gamma \vdash true : Bool$	$\Gamma \vdash false : Bool$
	$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$
$\Gamma \vdash E : Bool \quad \Gamma \vdash E1 : \tau \quad \Gamma \vdash E2 : \tau$	$\frac{}{\Gamma \vdash if\ E\ then\ E1\ else\ E2 : \tau}$
$\frac{\Gamma, x : \tau \vdash e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash fun\ x : \tau_1 = e : \tau_1 \rightarrow \tau_2}$	
$\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1$	$\frac{}{\Gamma \vdash Apply(e_1, e_2) : \tau_2}$

Tipi	$\tau ::= Bool \mid \tau \rightarrow \tau$
Semantica (valutazione)	
$if\ true\ then\ e_1\ else\ e_2 \rightarrow e_1$	
$if\ false\ then\ e_1\ else\ e_2 \rightarrow e_2$	$e_1 \rightarrow e_4$
$if\ e_1\ then\ e_2\ else\ e_3 \rightarrow if\ e_4\ then\ e_2\ else\ e_3$	
$Apply(fun\ x:\tau = e_1, v) \rightarrow e_1\{x := v\}$	
	$e_1 \rightarrow e'$
$Apply(e_1, e_2) \rightarrow Apply(e', e_2)$	
	$e_2 \rightarrow e'$
$Apply(e_1, e_2) \rightarrow Apply(v, e_2)$	