# Linguaggi e paradigmi di programmazione

Luca Barra

Anno accademico 2023/2024

# Indice

# Capitolo 1

# Introduzione

## 1.1 Tipi di paradigmi di programmazione

| Paradigma | Un programma è | Esempi |
|---|---|---|
| Imperativo | Sequenza di azioni (esecuzione ⇒ nuovo stato) | C, C++, Pascal, Java, Scala, Python, Scheme |
| Object-oriented | Oggetti che comunicano (interazione ⇒ nuovo stato) | Smalltalk, C++, OCaml, Java, Scala, Python |
| Funzionale | Espressione (valutazione ⇒ risultato) | Haskell, OCaml, Scala, Erlang, Scheme |

Per il paradigma *imperativo* un programma è una sequenza di azioni che modificano lo stato attuale del calcolatore. Per il paradigma *object-oriented* un programma è un insieme di oggetti che comunicano tra loro tramite uno scambio di messaggi. Per il paradigma *funzionale* un programma è una grande espressione in cui compaiono funzioni. Un programma funzionale si *valuta*, non si esegue. Quasi tutti i linguaggi attuali sono *multi-paradigma*, ossia hanno caratteristiche di due o più paradigmi.

## 1.2 I linguaggi funzionali

### 1.2.1 Il $\lambda$-calcolo

La programmazione *funzionale* risale agli anni '30, prima della costruzione dei primi calcolatori elettronici. Il *$\lambda$-calcolo*[1] è un piccolo linguaggio che parte dall'idea di calcolare con le funzioni come si calcola con i numeri. Le uniche cose che si possono scrivere sono espressioni di funzioni (tutto è funzione). Si possono definire funzioni senza dare loro un nome, per esempio, la funzione identità (accetta un argomento x e produce x come risultato):

$$\lambda x.x$$

**Note:-**

Tutte le funzioni sono a un solo argomento (*currying*), ma si possono avere funzioni di ordine superiori usando delle funzionni come argomento. Il $\lambda$-calcolo è un modello di calcolo computazionalmente completo, ossia è abbastanza potente da esprimere tutti i programmi concepibili.

### 1.2.2 LISP, FP e ML

Negli anni '50 viene sviluppato il primo linguaggio di programmazione funzionale (in parte imperativo), *LISP*[2], in cui le liste hanno un ruolo chiave. Inizialmente era concepito per l'elaborazione non numerica in cui era possibile definire funzioni anonime con notazione prefissa (abbondante uso di parentesi). Il LISP fu il primo *garbage collector*[3].

[1] Il "$\lambda$" nasce da un errore tipografico, doveva essere un accento circonflesso sull'argomento
[2] Dalla contrazione di LISt Processor
[3] La gestione della memoria non ricade sul programmatore

Negli anni '70 John Backus, ideatore del FORTRAN, vince il premio Turing. Nella sua lezione d'onore[4] Backus critica il FORTRAN e il paradigma imperativo introducendo successivamente *FP*. In FP vengono incorporate le idee della programmazione funzionale (tuttavia non avrà mai successo).

*ML*[5] fu il primo linguaggio funzionale moderno. Esso fu concepito da Robin Milner come "spin-off" di LCF[6], ma divento un linguaggio a sè stante (OCamel si fonda su ML). Si fonda su *type inference* (il programmatore poteva non specificare i tipi) e fu il primo con *polimorfismo parametrico*. Inoltre è possibile definire tipi e viene introdotto il *pattern matching* (per analizzare le strutture dati). Infine era dotato di un sistema di moduli per scomporre programmi di grandi dimensioni.

### 1.2.3 I linguaggi Lazy e Haskell

Negli anni '80 si svilupparono i linguaggi funzionali *lazy*, in cui gli argomenti non vengono valutati se la funzione non ne ha davvero bisogno. SASL introduce le guardie e il currying. KRC introduce un precursore delle liste comprehension. Miranda introduce le sezioni (ossia funzioni in cui si applica un'operazione n-aria a un solo operando).

Negli anni '90 si organizzò un comitato per standardizzare i linguaggi lazy. *Haskell* è un linguaggio di programmazione funzionale puro in cui si usano i tipi per separare I/O, stati, etc.. Si hanno classi di tipo per implementare l'overloading. Nel 2005 si stavano raggiungendo i limiti della velocità dei processori, per cui si sviluppano architetture multi-core. Il *calcolo parallelo* è difficile a causa dei linguaggi imperativi, ma Haskell, essendo puro, può facilmente essere usato.

[4] Lezione data da un vincitore del premio Turing in cui si racconta il perchè si è vinto il premio
[5] Meta language
[6] Logic for Computable Function

# Capitolo 2

# Il $\lambda$-calcolo

## 2.1 Perchè studiarlo?

Il $\lambda$-calcolo definisce in modo preciso il processo di valutazione (riduzione) di un programma funzionale. Esso pone le basi per lo sviluppo di un sistema di tipi e il relativo algoritmo di inferenza.

### 2.1.1 Le funzioni

- **Punto di vista estensionale:** $f = \{(0,1),(1,2),(2,5),(3,10),...\}$, rappresenta un'elegante interpretazione insiemistica, ma non fornisce informazioni su come calcolare le funzioni e sul relativo costo;
- **Punto di vista intensionale:** $f(x) = x^2 + 1$, fornisce una "ricetta" per il calcolo, ma non tutte le funzioni hanno una ricetta finita. Su questo punto di vista si basa il $\lambda$-calcolo.

**Note:-**

Nel $\lambda$-calcolo i numeri non sono primitive, in Haskell sì

## 2.2 Sintassi

**Definizione 2.2.1: Variabili e sintassi**

Sia Var $= \{x, y, z, ...\}$ 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 argomento x e corpo M

**Note:-**

$(MN)$ è l'applicazione delle funzione $M$ all'argomento $N$

**Esempio 2.2.1**

- $(\lambda x.x)$;
- $((\lambda x.(xx))(\lambda y.(yy)))$;
- $(\lambda f.(\lambda x.(f(fx))))$.

### 2.2.1 Convenzioni sintattiche

Si può limitare l'uso delle parentesi:

- le parentesi più esterne sono omesse $MN = (MN)$;
- il corpo delle associazioni si estende il più possibile a destra $\lambda x.xx = (\lambda x(xx))$;
- l'applicazione è associativa a sinistra $M_1M_2M_3 = ((M_1M_2)M_3)$.

## 2.3 Semantica

**Note:-**

Diciamo che un occorrenza di $x$ in $M$ è legata se compare in sotto-termine della forma $\lambda x.N$ in $M$, altrimenti è libera

**Esempio 2.3.1**

- $\lambda x.x$ non ha nessuna variabile libera;
- x y z ha solo variabili libere;
- $(\lambda x.xy)x$ in cui x occorre sia libera che legata.

**Definizione 2.3.1: 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 2.3.2: Sostituzione**

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

**Note:-**

$M\{N/y\}$ è ottenuta sostituendo le occorrenze libere di $y$ in $M$ con $N$. Serve per evitare la cattura delle variabili libere di N per non alterarne il senso

**Esempio 2.3.2**

- $\lambda x.x\{y/x\} = \lambda x.x$ non avviene nessuna sostituzione, perchè non ci sono variabili libere;
- $((\lambda x.x)x)\{y/x\} = (\lambda x.x)y$ sono state sostituite le occorrenze libere;
- $(\lambda z.x)\{y/x\} = \lambda z.y$ $y \neq z$ quindi non viene catturata;
- $(\lambda y.xy)\{y/x\} = \lambda z.yz$ y verrebbe catturata, quindi viene cambiata con z;
- $(\lambda x.y)\{\lambda x.x/y\} = \lambda x.\lambda x.x$ non c'è cattura;

- $(\lambda x.y)\{\lambda z.x/y\} = \lambda u.\lambda z.x$ perchè x verrebbe catturata.

**Note:-**

Il significato di una funzione è indipendente dal nome dell'argomento

### 2.3.1 $\alpha$-conversione

**Definizione 2.3.3: $\alpha$-conversione**

L'$\alpha$-conversione $\Leftrightarrow_\alpha$ 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 conversione

### 2.3.2 $\beta$-riduzione

**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 2.3.4: $\beta$-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*[a].
$M\{N/y\}$ 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.

[a] REDucible EXpression

**Note:-**

Ci sono esempi di espressioni che rappresentano la stessa espressione, ma che non possono essere ridotti. Per esempio:

$$(\lambda x.Mx)N \rightarrow_\beta (Mx)\{N/x\} = M\{N/x\}N = MN$$

M e $\lambda x.Mx$ producono lo stesso risultato se applicate allo stesso argomento, tuttavia nessuna delle due è $\beta$-riducibile nell'altra.

### 2.3.3 $\eta$-riduzione

**Definizione 2.3.5: $\eta$-riduzione**

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

- se x $\notin$ fv(M) allora $\lambda x.Mx \to_\eta M$;
- se $M \to_\eta M'$ allora $MN \to_\eta M'N$;
- se $M \to_\eta M'$ allora $MN \to_\eta NM'$;
- se $M \to_\eta M'$ allora $MN \to_\eta \lambda x.M'$.

**Definizione 2.3.6: Riduzioni singole e multiple**

Si scrive $\to$ per indicare l'unione $\to_\beta \cup \to_\eta$ e $\Rightarrow$ per indicare la chiusura riflessiva e transitiva di $\to$. Ovvero $\Rightarrow$ è la più piccola relazione tale che:

- $M \Rightarrow M$;
- se $M \to N$ allora $M \Rightarrow N$;
- se $M \Rightarrow M'$ e $M' \Rightarrow N$ allora $M \Rightarrow N$.

**Definizione 2.3.7: Convertibilità**

Scriviamo $M \leftrightarrow N$ se $M \to N$ oppure $N \to M$ e scriviamo $\Leftrightarrow$ per la chiusura riflessiva e transitiva di $\leftrightarrow$. Diciamo che M e N sono convertibili se $M \Leftrightarrow N$.

## 2.4 Confluenze e strategie di riduzione

**Teorema 2.4.1** Confluenza

Se $M \Rightarrow N_1$ e $M \Rightarrow N_2$ allora esiste $N$ tale che $N_1 \Rightarrow N$ e $N_2 \Rightarrow N$

**Note:-**

Ossia, indipendentemente dalla strada presa, si giunge allo stesso risultato

**Definizione 2.4.1: Forma normale**

Diciamo che $M$ è in forma normale se non può più essere ridotto, ovvero se non esiste $N$ tale che $M \to N$. In tal caso scriviamo $M \not\to$

**Corollario 2.4.1**

La forma normale di M, se esiste, è unica (a meno di $\alpha$-conversioni)

**Esempio 2.4.1** (Dimostrazione della forma normale)

Supponiamo che $M$ abbia due forme normali $N_1$ e $N_2$, ovvero $M \Rightarrow N_1 \not\to$ e $M \Rightarrow N_2 \not\to$. Per il teorema di confluenza esiste $N$ tale che $N_1 \Rightarrow N$ e $N_2 \Rightarrow N$. Siccome $N_1$ e $N_2$ sono in forma normale, deve essere $N_1 = N = N_2$ (a meno di $\alpha$-conversioni)

### 2.4.1 Ordine applicativo

**Definizione 2.4.2: Ordine applicativo**

Nell'ordine applicativo si sceglie il redesso più interno e più a sinistra

**Note:-**

Viene utilizzato dai linguaggi zelanti o eager

### 2.4.2 Ordine normale

**Definizione 2.4.3: Ordine normale**

Nell'ordine normale si sceglie il redesso più esterno e più a destra

**Note:-**

Viene utilizzato dai linguaggi pigri o lazy

### 2.4.3 Normalizzazione

**Teorema 2.4.2** Normalizzazione

Se $M \Leftrightarrow N$ e $N$ è in forma normale, allora c'è una riduzione in ordine normale $M \Rightarrow N$

**Note:-**

Questa proprietà non vale per l'ordine applicativo

# Capitolo 3

# Programmare nel $\lambda$-calcolo

## 3.1 Semantica

**Definizione 3.1.1: Valori booleani**

- TRUE $= \lambda x.\lambda y.x$;
- FALSE $= \lambda x.\lambda y.y$;
- IF $= \lambda z.z$

**Proposizione 3.1.1**

- IF TRUE $MN \Leftrightarrow M$;
- IF TRUE $MN \Leftrightarrow N$.

**Definizione 3.1.2: Operatori logici**

- AND $= \lambda x.\lambda y.$IF $xy$ FALSE;
- OR $= \lambda x.\lambda y.$IF $x$ TRUE $y$;
- NOT $= \lambda x.$IF $x$ FALSE TRUE.

**Definizione 3.1.3: Coppie**

- PAIR $= \lambda x.\lambda y.\lambda z.zxy$;
- FST $= \lambda p.p$ TRUE;
- SND $= \lambda p.p$ FALSE.

**Proposizione 3.1.2**

- FST (PAIR $MN$) $\Leftrightarrow M$;
- SND (PAIR $MN$) $\Leftrightarrow N$.

**Definizione 3.1.4: Numeri naturali (codifica di Church)**

Dato $k \in \mathbb{N}$ scriviamo $M^k N$ per $M(M(...(MN)))$ (k volte). In particolare: $M^0 N = N$.

- $\underline{n} = \lambda f.\lambda x.f^n x$ (codifica del numero naturale n);
- SUCC $= \lambda a.\lambda f.\lambda x.af(fx)$ (funzione successore).

**Note:-**

Un numero naturale è un iteratore che itera una certa funzione n volte su un certo elemento

## 3.2 Operazioni sui numeri naturali

**Definizione 3.2.1: ADD, MUl e EXP**

- ADD $= \lambda a.\lambda b.b\text{SUC}a$;
- MUL $= \lambda a.\lambda b.b(\text{ADD}a)\underline{0}$;
- EXP $= \lambda a.\lambda b.b(\text{MUL}a)\underline{1}$.

**Proposizione 3.2.1**

- ADD $\underline{mn} \Leftrightarrow \underline{m+n}$;
- MUL $\underline{mn} \Leftrightarrow \underline{m*n}$;
- EXP $\underline{mn} \Leftrightarrow \underline{m^n}$.

**Definizione 3.2.2: NEXT e PRED**

- NEXT $= \lambda p.$PAIR (SND $p$)(SUCC (SND $p$));
- PRED $= \lambda a.$FST ($a$ NEXT (PAIR $\underline{0}$ $\underline{0}$)).

**Proposizione 3.2.2**

- NEXT (PAIR $\underline{m}$ e $\underline{n}$) $\Leftrightarrow$ PAIR $\underline{n}$ $\underline{n+1}$;
- PRED $\underline{n} \Leftrightarrow \begin{cases} \underline{0} & \text{se } n = 0 \\ \underline{n-1} & \text{se } n > 0 \end{cases}$

**Definizione 3.2.3: ISZERO**

- ISZERO $= \lambda a.a(\lambda x.\text{FALSE TRUE})$.

**Proposizione 3.2.3**

- ISZERO $\underline{n} \Leftrightarrow \begin{cases} \text{TRUE} & \text{se } n = 0 \\ \text{FALSE} & \text{se } n \neq 0 \end{cases}$

**Definizione 3.2.4: Punto fisso**

$x$ è un punto fisso di $F$ se $x = F(X)$

**Definizione 3.2.5: Operatore di un punto fisso**

FIX $= \lambda f.(f(x\ x))(\lambda x.f(x\ x))$

**Proposizione 3.2.4**

FIX $M \Leftrightarrow M$(FIX $M$)

**Note:-**

FIX è utilizzato per definire funzioni ricorsive

**Definizione 3.2.6: FACT**

FACT = FIX AUX, dove AUX $=\lambda f.\lambda a.$IF (ISZERO $a$) $\underline{1}$ (MUL $a$ (f (PRED a)))

**Note:-**

AUX "butta via" il punto fisso quando $a$ è zero (caso base), restituendo 1

**Proposizione 3.2.5**

FACT $\Leftrightarrow \lambda a.$IF (ISZERO $a$) $\underline{1}$ (MUL $a$ (FACT (PRED a)))

## 3.3 Sintassi delle $\lambda$-espressioni con booleani

**Definizione 3.3.1: Estensione della sintassi**

**Espressioni** $M, N$ ::==

- x (variabile);
- c (costante);
- $\lambda x.M$ (astrazione);
- $M\ N$ (applicazione);
- if $M\ N_1\ N_2$ (condizionale).

**Costanti** c $\in$ {False, True}.
**Riduzioni per if**

- if true $M\ N \rightarrow M$;
- if false $M\ N \rightarrow N$.

**Note:-**

Quest'estensione introduce possibili errori. Per catturarli si introduce il concetto di *tipo*

**Definizione 3.3.2: Sintassi dei tipi**

**Tipi** $t$, $s$ ::==

- Bool (booleani);
- $t \to s$ (funzioni).

**Note:-**

Per cui esistono due tipi: Bool e freccia ($\to$)

**Definizione 3.3.3: Contesto**

Un contesto $\Gamma$ è una funzione parziale da variabili a tipi

**Proposizione 3.3.1**

- Scriviamo $dom(\Gamma)$ per il dominio di $\Gamma$;
- Scriviamo $x : t$ per il contesto $\Gamma$ tale che $dom(\Gamma) = \{x\}$ e $\Gamma(x) = t$;
- Scriviamo $\Gamma, \Gamma'$ per l'unione di $\Gamma$ e $\Gamma'$ quando $do(\Gamma) = \{x\} \cap dom(\Gamma') = \emptyset$

### 3.3.1 Regole di tipo

**Definizione 3.3.4: Regole di tipo**

Le regole di tipo hanno la forma:

$$\frac{\text{Premessa}_1 \ \ldots \ \text{Premessa}_n}{\text{Conclusione}}$$

Se le premesse sono vere, allora la conclusione è vera

**Note:-**

Una regola senza premesse è detta *assioma* ed è vero

**Definizione 3.3.5: Regole di ripo per il $\lambda$-calcolo con costanti**

- [t-var] = $\frac{}{\Gamma, x:t \vdash x:t}$;
- [t-bool] = $\frac{}{\Gamma \vdash c:\ \text{Bool}}$;
- [t-lam] = $\frac{\Gamma, x:t \vdash M:s}{\Gamma \vdash \lambda x.M : t \to s}$;
- [t-if] = $\frac{\Gamma \vdash M:\ \text{Bool} \quad \Gamma \vdash N_1 : t \quad \Gamma \vdash N_2 : t}{\Gamma \vdash \text{if } M\ N_1\ N_2 : t}$;
- t-app = $\frac{\Gamma \vdash M : t \to s \quad \Gamma N : t}{\Gamma M\ N : s}$.

**Note:-**

Le regole vanno lette dalla conclusione fino alle premesse

**Lenma 3.3.1** Subject reduction

Se $\Gamma \vdash M : t$ e $M \to N$, allora $\Gamma \vdash N : t$

**Definizione 3.3.6: Valore**

Diciamo che $M$ è un *valore* se $M$ è una costante o un'astrazione

**Note:-**

Un valore è un risultato

**Teorema 3.3.1** Progresso

Se $\vdash M : t$ e $M \Rightarrow N \nrightarrow$ allora $N$ è un valore

**Note:-**

Il teorema del progresso enuncia che se $M$ si riduce a $N$ in un certo numero di passi e $N$ non si può ridurre, allora $N$ è ben tipato

# Capitolo 4

# Algoritmo di inferenza

## 4.1 Sommario

**Definizione 4.1.1: Fasi dell'algoritmo**

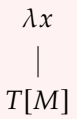
1. Si costrusce l'albero sintattico della $\lambda$-espressione;
2. Si annota l'albero e si generano i vincoli:
   - variabile di tipo: tipo sconosciuto;
   - espressione di tipo: tipo (parzialmente) sconosciuto;
   - vincolo: relazione di uguaglianza tra tipi espressa nelle regole di tipo.
3. Si risolvono i vincoli:
   - Si deve determinare se il sistema ammette almeno una soluzione;
   - Si deve calcolare la soluzione più generale da cui derivare tutte le altre.

### 4.1.1 Costruzione dell'albero sintattico

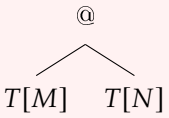
- Si rappresenta la $\lambda$-espressione come un albero;
- I nodi interni e le foglie sono opportunamente etichettati;
- Indichieamo con T[M] l'albero corrispondente alla $\lambda$-espressione.

**Definizione 4.1.2: Albero sintattico**

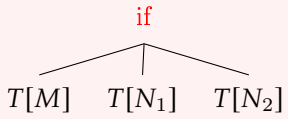
- $T[x] = x$ (foglia);
- $T[c] = c$ (foglia);
- $T[\lambda x.M] =$




- $T[M\ N] =$

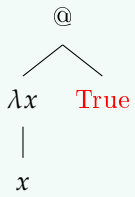



- $T[\text{if } M\ N_1\ N_2] =$

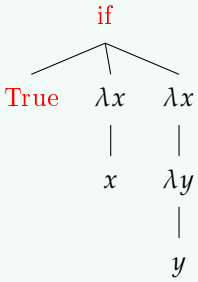



**Esempio 4.1.1**

- $(\lambda x.y)$ True




- if True $(\lambda x.x)$ $(\lambda x.\lambda y.y)$




### 4.1.2 Annotazione dell'albero e generazione dei vincoli

- Ogni nodo dell'albero viene annotato con un'espressione di tipo;
- Si procede in modo bottom-up, dalle foglie verso la radice.

**Definizione 4.1.3: Sintassi delle espressioni di tipo**

Sia $TVar = \{\alpha, \beta, \gamma, ...\}$ un insieme infinito di variabili di tipo e $\alpha$ un tipo sconosciuto[a], le espressioni di tipo $\tau, \sigma$ ::=

$\alpha$ (variabile di tipo)

Bool (booleani)

$\tau \to \sigma$ (funzioni)

[a] Ancora da determinare

**Note:-**

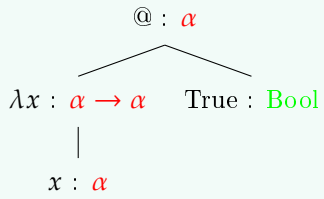
Un vincolo è una coppia di espressioni di tipo scritta come $\tau = \sigma$

| Albero | Note |
|---|---|
| $x : \alpha$ | $\alpha$ è nuova, avendo cura di usare la stessa $\alpha$ per tutte le occorrenze della stessa $x$ |
| $c : \texttt{Bool}$ | Altre costanti richiederanno tipi diversi |
| $\lambda x : \alpha \to \tau$ <br> \| <br> $T : \tau$ | $\alpha$ è la variabile di tipo usata per annotare $x$ in $T$ se $x$ compare in $T$ o è nuova altrimenti |
| $@ : \alpha$ <br> $T_1 : \tau$ $\quad$ $T_2 : \sigma$ | $\alpha$ è nuova <br> Generare il vincolo $\tau = \sigma \to \alpha$ |
| $@ : \tau_2$ <br> $T_1 : \tau_1 \to \tau_2$ $\quad$ $T_2 : \sigma$ | Ottimizzazione facoltativa del caso precedente che evita l'introduzione di una nuova $\alpha$ <br> Generare il vincolo $\tau_1 = \sigma$ |
| $\texttt{if} : \tau_2$ <br> $T_1 : \tau_1$ $\quad$ $T_2 : \tau_2$ $\quad$ $T_3 : \tau_3$ | Generare il vincolo $\tau_1 = \texttt{Bool}$ <br> Generare il vincolo $\tau_2 = \tau_3$ |

**Esempio 4.1.2**

$\lambda x.x$ True

Albero annotato



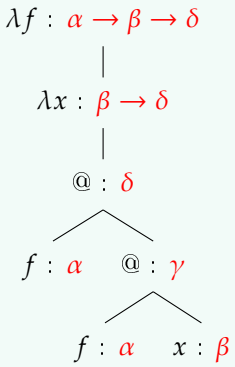

Vincoli generati

$\alpha = \text{Bool}$

**Esempio 4.1.3**

$\lambda f.\lambda x.f\ (f\ x)$

Albero annotato




Vincoli generati

$\alpha = \beta \to \gamma$

$\alpha = \gamma \to \delta$

**Note:-**

Nell'ultimo esempio entrambe le $f$ hanno la stessa annotazione

### 4.1.3 Sistemi di equazioni e soluzioni

**Definizione 4.1.4: Sostituzione**

Una sostituzione $\theta$ è una funzione da variabili di tipo a espressioni di tipo. Scriviamo $\theta(\tau)$ per l'espressione ottenuta da $\tau$ sostituendo ogni $\alpha$ con $\theta(\alpha)$

**Definizione 4.1.5: Soluzione**

Dato un sistema di vincoli $\{\tau_i = \sigma_i\}_{1 \leqslant i \leqslant n}$ e una sostituzione $\theta$ diciamo che $\theta$ è soluzione (o unificatore) del sistema se $\theta(\tau_i) = \theta(\sigma_i)$ per ogni $1 \leqslant i \leqslant n$. Diciamo inoltre che $\theta$ è l'unificatore più generale del sistema se ogni soluzione del sistema è ottenibile componendo $\theta$ con un'altra sostituzione

| Se c'è un vincolo | e | allora |
|---|---|---|
| $\tau = \tau$ | – | eliminare il vincolo |
| $\tau = \alpha$ | $\tau$ non è una variabile | rimpiazzare il vincolo con $\alpha = \tau$ |
| $\tau \to \tau' = \sigma \to \sigma'$ | – | rimpiazzare il vincolo con $\tau = \sigma$ e $\tau' = \sigma'$ |
| $\tau \to \sigma =$ `Bool` o `Bool` $= \tau \to \sigma$ | – | ☠ l'algoritmo fallisce (type error) |
| $\alpha = \tau$ | $\alpha \neq \tau$ ma $\alpha$ compare in $\tau$ | ☠ l'algoritmo fallisce (occur check) |
| $\alpha = \tau$ | $\alpha$ non compare in $\tau$ $\alpha$ compare altrove | sostituire $\alpha$ con $\tau$ in tutti gli altri vincoli ($\alpha = \tau$ rimane) |

**Note:-**

L'ordine in cui si applicano le trasformazioni non è importante. Se non si può applicare nessuna trasformazione l'algoritmo ha avuto successo

**Proposizione 4.1.1**

- In un numero finito di passi l'algoritmo ha successo o fallisce;
- Se l'algoritmo fallisce allora il sistema è insoddisfacibile;
- Se l'algoritmo ha successo:
  - il sistema ha forma $\{\alpha_i = p_i\}_{1 \leqslant i \leqslant m}$ in cui ciascuna $\alpha_i$ compare una sola volta nel sistema;
  - la sostituzione $\theta = \{\alpha_i \to p_i\}_{1 \leqslant i \leqslant m}$ è l'unificatore più generale del sistema iniziale, in particolare $\sigma(\tau_i) = \theta(\sigma_i))$ per ogni $\leqslant i \leqslant m$.

**Esempio 4.1.4**

$$\begin{array}{l} \alpha = \beta \to \gamma \\ \alpha = \gamma \to \delta \end{array}$$

$$\begin{array}{l} \alpha = \beta \to \gamma \\ \beta \to \gamma = \gamma \to \delta \end{array}$$

$$\begin{array}{l} \alpha = \beta \to \gamma \\ \beta = \gamma \\ \gamma = \delta \end{array}$$

$$\begin{array}{l} \alpha = \gamma \to \gamma \\ \beta = \gamma \\ \gamma = \delta \end{array}$$

$$\begin{array}{l} \alpha = \delta \to \delta \\ \beta = \delta \\ \gamma = \delta \end{array}$$

$$\begin{array}{l} \alpha = \beta \to \beta \\ \alpha = \texttt{Bool} \to \gamma \end{array}$$

$$\begin{array}{l} \alpha = \beta \to \beta \\ \beta \to \beta = \texttt{Bool} \to \gamma \end{array}$$

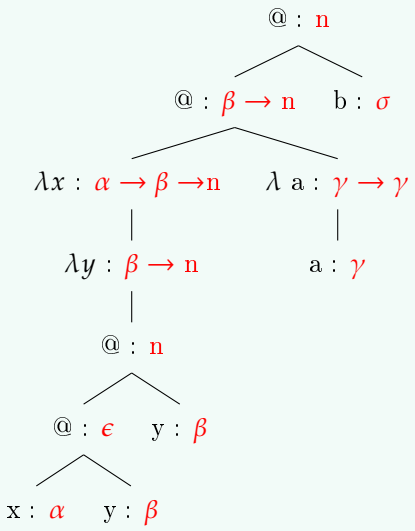
$$\begin{array}{l} \alpha = \beta \to \beta \\ \beta = \texttt{Bool} \\ \beta = \gamma \end{array}$$

$$\begin{array}{l} \alpha = \texttt{Bool} \to \texttt{Bool} \\ \beta = \texttt{Bool} \\ \texttt{Bool} = \gamma \end{array}$$

$$\begin{array}{l} \alpha = \texttt{Bool} \to \texttt{Bool} \\ \beta = \texttt{Bool} \\ \gamma = \texttt{Bool} \end{array}$$

**Esempio 4.1.5** (Esercizio)

$(\lambda x.\lambda y.x\ \ y\ \ y)(\lambda a.a)b)$




$\alpha = \beta \to \epsilon$
$\epsilon = \beta \to \text{n}$
$\alpha = \gamma \to \gamma$
$\beta = \sigma$

Risoluzione:

$$\begin{cases} \alpha = \beta \to \epsilon \\ \epsilon = \beta \to \mathrm{n} \\ \alpha = \gamma \to \gamma \\ \beta = \sigma \end{cases}$$

$$\begin{cases} \alpha = \beta \to \epsilon \\ \epsilon = \beta \to \mathrm{n} \\ \beta \to \epsilon = \gamma \to \gamma \\ \beta = \sigma \end{cases}$$

$$\begin{cases} \alpha = \beta \to \epsilon \\ \epsilon = \beta \to \mathrm{n} \\ \beta = \gamma \\ \epsilon = \gamma \\ \beta = \sigma \end{cases}$$

$$\begin{cases} \alpha = \gamma \to \epsilon \\ \epsilon = \gamma \to \mathrm{n} \\ \beta = \gamma \\ \epsilon = \gamma \\ \gamma = \sigma \end{cases}$$

$$\begin{cases} \alpha = \gamma \to \gamma \to \mathrm{n} \\ \epsilon = \gamma \to \mathrm{n} \\ \beta = \gamma \\ \gamma \to \mathrm{n} = \gamma \\ \gamma = \sigma \end{cases}$$

$$\begin{cases} \alpha = \gamma \to \gamma \to \mathrm{n} \\ \epsilon = \gamma \to \mathrm{n} \\ \beta = \gamma \\ \gamma = \gamma \to \mathrm{n} \\ \gamma = \sigma \end{cases}$$

**! OCCUR CHECK FAIL !**

## 4.2 Estensioni dell'algoritmo

**Definizione 4.2.1: Numeri interi**

Le costanti includono i numeri interi

$$c \in \{\mathrm{False}, \mathrm{True}, \ 0, \ 1, ...\}$$

Le espressioni di tipo sono arricchite con il tipo Int

**Note:-**

Se c'è un vincolo $\tau \to \sigma =$ Int o Int $=$ Bool o Bool $=$ Int l'algoritmo fallisce (TYPE ERROR)

**Definizione 4.2.2: Liste**

Le costanti includono i costruttori canonici

$$c \in \{..., \ [\,], \ (:)\}$$

Ogni occorrenza di un costruttore fa uso di nuove variabili di tipo

**Note:-**

- Se c'è un vincolo $[\tau] = [\sigma]$ rimpiazzarlo con $\tau = \sigma$;
- Se c'è un vincolo $[\tau] =$ Bool o Bool $= [\tau]$ o $[\tau] = \sigma_1 \to \sigma_2$ o ... l'algoritmo fallisce (TYPE ERROR)

**Definizione 4.2.3: Funzioni di libreria**

Le costanti includono le funzioni di libreria

$$c \in \{..., \ \mathrm{id}, \ \mathrm{head}, \ \mathrm{tail}, ...\}$$

Ogni occorrenza di una funzione di libreria fa uso di nuove variabili di tipo

**Definizione 4.2.4: Definizioni ricorsive**

$$f = M$$

dove $f$ può comparire in $M$. Il nome $f$ è trattato come ogni altra variabile. Alla fine dell'annotazione si genera il vincolo $\alpha = \tau$ dove $\alpha$ è la variabile di tipo associata a $f$ e $\tau$ è l'annotazione di $M$

# Capitolo 5

# Dimostrazione di proprietà di funzioni

**Domanda 1**

Date una funzione e una specifica di ciò che la funzione dovrebbe calcolare, la funzione è corretta rispetto alla specifica?

**Domanda 2**

Date due funzioni (una corretta ma inefficiente e una efficiente ma più complessa da capire), sono equivalenti?

**Test:**

- facile (soprattutto se il linguaggio non è imperativo);
- non è esaustivo (possono esserci casi non considerati).

**Dimostrazione:**

- difficile (soprattutto se il linguaggio è imperativo);
- è esaustiva.

## 5.1 Dimostrazioni su numeri interi

```
1 foo :: Int -> Int -> Int -> Int
2 foo x y z | y < x       = foo y x z
3           | z < y       = foo x z y
4           | otherwise = z
```

**Domanda 3**

$\forall\ x,\ y,\ z = \max\{x,\ y,\ z\}$ è una proprietà vera?

Si possono fare dei test, ma saranno sempre in numero finito. Per cui è più conveniente cercare una dimostrazione.

### 5.1.1 Dimostrazione per casi

**Note:-**

I numeri sono infinite, per cui non si può verificare il comportamento per ogni combinazione. Ma l'ordine tra i numeri è totale, per cui possiamo considerare un numero finito di casi che è esaustivo

1. $x \leq y \leq z \Rightarrow \texttt{foo}\ x\ y\ z = z$
2. $x \leq z < y \Rightarrow \texttt{foo}\ x\ y\ z = \texttt{foo}\ x\ z\ y = y$
3. $y < x \leq z \Rightarrow \texttt{foo}\ x\ y\ z = \texttt{foo}\ y\ x\ z = z$
4. $y \leq z < x \Rightarrow \texttt{foo}\ x\ y\ z = \texttt{foo}\ y\ x\ z = \texttt{foo}\ y\ z\ x = x$
5. $z < x \leq y \Rightarrow \texttt{foo}\ x\ y\ z = \texttt{foo}\ x\ z\ y = \texttt{foo}\ z\ x\ y = y$
6. $z < y < x \Rightarrow \texttt{foo}\ x\ y\ z = \texttt{foo}\ y\ x\ z = \texttt{foo}\ y\ z\ x = \texttt{foo}\ z\ y\ x = x$

**Note:-**

Il codice è tutto ciò che serve per ragionare sui casi

### 5.1.2 Induzione

**Note:-**

L'approccio precedente è attuabile solo se il numero dei casi da prendere in considerazione è finito. Il principio di induzione permette di dimostrare una proprietà su un insieme infinito di casi

**Definizione 5.1.1: Il principio di induzione**

Data una proprietà $P(n)$ dei numeri naturali, se:

- $P(0)$;
- $P(n)$ implica $P(n+1)$ per ogni $n \in \mathbb{N}$.

allora $P(n)$ per ogni $n \in \mathbb{N}$

```
1 exp :: Int -> Int -> Int
2 exp _ 0 = 1
3 exp x n = x * exp x (n - 1)
```

**Esempio 5.1.1** (Esponenziale)

Si vuole dimostrare che:

1. $\forall\ x,\ m \geqslant 0,\ n \geqslant 0 : \exp\ x\ (m+n) = \exp\ x\ m * \exp x|\ :\ n$;
2. $\forall\ x,\ n : \exp\ (x * x)\ n = \exp\ x\ n * \exp\ x\ n$.

**Prima dimostrazione:**

$$P(m) \stackrel{\text{def}}{=} \forall x, n \geq 0 : \exp x\ (m+n) = \exp x\ m * \exp x\ n$$

$P(0)$ — lato sinistro

$$\begin{aligned} &\exp x\ (0+n) \\ &\quad = \exp x\ n && \text{(proprietà di } +\text{)} \end{aligned}$$

$P(0)$ — lato destro

$$\begin{aligned} &\exp x\ 0 * \exp x\ n \\ &\quad = 1 * \exp x\ n && (\texttt{exp.1}) \\ &\quad = \exp x\ n && \text{(proprietà di } *\text{)} \end{aligned}$$

$P(m-1) \Rightarrow P(m)$ per ogni $m > 0$ — lato sinistro

$$\begin{aligned} &\exp x\ (m+n) \\ &\quad = x * \exp x\ ((m+n)-1) && (\texttt{exp.2}) \\ &\quad = x * \exp x\ ((m-1)+n) && \text{(proprietà di } + \text{ e } -\text{)} \\ &\quad = x * (\exp x\ (m-1) * \exp x\ n) && \text{(ipotesi induttiva)} \end{aligned}$$

$P(m-1) \Rightarrow P(m)$ per ogni $m > 0$ — lato destro

$$\begin{aligned} &\exp x\ m \times \exp x\ n \\ &\quad = (x * \exp x\ (m-1)) * \exp x\ n && (\texttt{exp.2}) \\ &\quad = x * (\exp x\ (m-1) * \exp x\ n) && \text{(proprietà di } *\text{)} \end{aligned}$$

**Seconda dimostrazione:**

$$P(n) \stackrel{\text{def}}{=} \forall x : \exp\ (x*x)\ n = \exp x\ n * \exp x\ n$$

$P(0)$ — lato sinistro

$$\exp\ (x*x)\ 0 = 1 \qquad (\texttt{exp.1})$$

$P(0)$ — lato destro

$$\begin{aligned} &\exp x\ 0 * \exp x\ 0 \\ &\quad = 1 * 1 && (\texttt{exp.1}) \\ &\quad = 1 && \text{(proprietà di } *\text{)} \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato sinistro

$$\begin{aligned} &\exp\ (x*x)\ n \\ &\quad = (x*x) * \exp\ (x*x)\ (n-1) && (\texttt{exp.2}) \\ &\quad = (x*x) * (\exp x\ (n-1) * \exp x\ (n-1)) && \text{(ipotesi induttiva)} \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato destro

$$\begin{aligned} &\exp x\ n * \exp x\ n \\ &\quad = (x * \exp x\ (n-1)) * (x * \exp x\ (n-1)) && (\texttt{exp.2}) \\ &\quad = (x*x) * (\exp x\ (n-1) * \exp x\ (n-1)) && \text{(proprietà di } *\text{)} \end{aligned}$$

**Definizione 5.1.2: Principio di induzione forte**

Data una proprietà $P(n)$ dei numeri naturali, se:

- $(\forall\ m < n : P(m)) \Rightarrow P(n)$, per ogni $n \in \mathbb{N}$.

allora $P(n)$ per ogni $n \in \mathbb{N}$

**Note:-**

Il principio di induzione forte è equivalente al principio di induzione

## 5.2 Dimostrazioni sulle liste

**Definizione 5.2.1: Principio di induzione sulle liste finite**

Data una proprietà $P(xs)$ delle liste, se:

- $P([])$;
- $P(xs)$ implica $P(x\ :\ xs)$ per ogni $x$ e $xs$.

Allora $P(xs)$ per ogni lista finita $xs$.

**Esempio 5.2.1** (Length)

$$P(n) \stackrel{\text{def}}{=} \forall x : \exp\ (x * x)\ n = \exp\ x\ n * \exp\ x\ n$$

$P(0)$ — lato sinistro

$$\exp\ (x * x)\ 0 = 1 \qquad (\exp.1)$$

$P(0)$ — lato destro

$$\begin{aligned} &\exp\ x\ 0 * \exp\ x\ 0 \\ &\quad = 1 * 1 && (\exp.1) \\ &\quad = 1 && (\text{proprietà di } *) \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato sinistro

$$\begin{aligned} &\exp\ (x * x)\ n \\ &\quad = (x * x) * \exp\ (x * x)\ (n-1) && (\exp.2) \\ &\quad = (x * x) * (\exp\ x\ (n-1) * \exp\ x\ (n-1)) && (\text{ipotesi induttiva}) \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato destro

$$\begin{aligned} &\exp\ x\ n * \exp\ x\ n \\ &\quad = (x * \exp\ x\ (n-1)) * (x * \exp\ x\ (n-1)) && (\exp.2) \\ &\quad = (x * x) * (\exp\ x\ (n-1) * \exp\ x\ (n-1)) && (\text{proprietà di } *) \end{aligned}$$

**Note:-**

Può capitare, durante una dimostrazione, di osservare passaggi intuitivi di cui non si ha una prova. Per cui si possono assumere e poi dimostrare in seguito

## 5.3 Dimostrazioni sugli alberi

**Definizione 5.3.1: Principio di induzione sugli alberi**

Data una proprietà $P(t)$, se:

- $P(Leaf)$;
- $P(t_1) \wedge P(t_2)$ implica $P(Branch\ t_1 t_2)$ per ogni $x$, $t_1$ e $t_2$.

Allora $P(t)$ per ogni albero (finito) $t$.

**Esempio 5.3.1** (Trees)

$$P(n) \stackrel{\text{def}}{=} \forall x : \exp\ (x * x)\ n = \exp\ x\ n * \exp\ x\ n$$

$P(0)$ — lato sinistro

$$\exp\ (x * x)\ 0 = 1 \qquad (\exp.1)$$

$P(0)$ — lato destro

$$\begin{aligned} &\exp\ x\ 0 * \exp\ x\ 0 \\ &= 1 * 1 && (\exp.1) \\ &= 1 && (\text{proprietà di } *) \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato sinistro

$$\begin{aligned} &\exp\ (x * x)\ n \\ &= (x * x) * \exp\ (x * x)\ (n-1) && (\exp.2) \\ &= (x * x) * (\exp\ x\ (n-1) * \exp\ x\ (n-1)) && (\text{ipotesi induttiva}) \end{aligned}$$

$P(n-1) \Rightarrow P(n)$ per ogni $n > 0$ — lato destro

$$\begin{aligned} &\exp\ x\ n * \exp\ x\ n \\ &= (x * \exp\ x\ (n-1)) * (x * \exp\ x\ (n-1)) && (\exp.2) \\ &= (x * x) * (\exp\ x\ (n-1) * \exp\ x\ (n-1)) && (\text{proprietà di } *) \end{aligned}$$

## 5.4 Dimostrazioni sulle funzioni di ordine superiore

**Definizione 5.4.1: Principio di estensionabilità**

Due funzioni $f$ e $g$ sono uguali se producono lo stesso risultato quando sono applicate allo stesso argomento. Formalmente:

$$(\forall x\ :\ f\ g = g\ x) \Leftrightarrow f = g$$

**Note:-**

È il principio che giustifica la regola di $\eta$-riduzione.

**Corollario 5.4.1** proprietà della composizione funzionale

$$\forall f : f \;.\; \mathtt{id} = f$$
$$\begin{aligned}
&(f \;.\; \mathtt{id})\; x \\
&\quad = f\; (\mathtt{id}\; x) && (.) \\
&\quad = f\; x && (\mathtt{id})
\end{aligned}$$

$$\forall f, g, h : f \;.\; (g \;.\; h) = (f \;.\; g) \;.\; h \qquad \text{lato sinistro}$$
$$\begin{aligned}
&(f \;.\; (g \;.\; h))\; x \\
&\quad = f\; ((g \;.\; h)\; x) && (.) \\
&\quad = f\; (g\; (h\; x)) && (.)
\end{aligned}$$

$$\forall f, g, h : f \;.\; (g \;.\; h) = (f \;.\; g) \;.\; h \qquad \text{lato destro}$$
$$\begin{aligned}
&((f \;.\; g) \;.\; h)\; x \\
&\quad = (f \;.\; g)\; (h\; x) && (.) \\
&\quad = f\; (g\; (h\; x)) && (.)
\end{aligned}$$

**Teorema 5.4.1** Legge di fusione

Se:

- $f\; a = b$;
- $f\; (g\; x\; y) = h\; x\; (f\; y)$;

allora

- $f$ . foldr $g\; a$ = foldr $h\; b$.

# Capitolo 6

# Java 8

A un certo punto, nella storia di Java, si è deciso di effettuare un "*restiling*" delle collection[1]. Per cui si sono introdotti:

- i metodi di default;
- lambda-espressioni: rendono il codice più compatto, modulare e leggibile.

## 6.1 Lambda-espressioni

**Definizione 6.1.1: Lambda-espressione**

Una lambda-espressione è una funzione anonima che può essere passata come parametro ad un metodo o ad un costruttore. La sintassi è la seguente:

- (parameters) $\rightarrow$ expression;
- (parameters) $\rightarrow$ { statements; }.

**Corollario 6.1.1** Identità

L'Identità $\lambda x : \text{int}.x$ è:

- $(x : \text{int}) \rightarrow x$;
- $(x : \text{int}) \rightarrow \{\text{return } x;\}$.

**Note:-**

In alcuni casi si può omettere il tipo dei parametri, in quanto Java è in grado di inferirlo (raramente)

**Domanda 4**

Qual è il tipo dell'Identità?

**Definizione 6.1.2: Interfacce funzionali**

In Java 8 sono state introdotte le interfacce funzionali, ovvero interfacce che hanno un solo metodo astratto. Esse rappresentano il tipo delle lambda-espressioni. Per esempio, l'Identità ha tipo Int che ritorna Int (quindi Int $\rightarrow$ Int, in Haskell)

[1] Implementazioni delle strutture dati più comuni, come liste, insiemi, mappe, etc.

**Note:-**

Esistono interfacce funzionali già pronto, come per esempio *Predicate*, *Consumer*, *Function*, etc.

**Definizione 6.1.3: Method reference**

Un method reference è un modo per riferirsi a un metodo. Aumenta la leggibilità del codice. La sintassi è la seguente: Type::methodName

## 6.2 Compatibilità e metodi di default

**Definizione 6.2.1: Binary compatibility**

Due versioni di una libreria sono binariamente compatibili se i file binari continuano a funzionare anche con la nuova versione

**Note:-**

Se si modifica un software bisogna tener conto della backward compatibility, ovvero la compatibilità con le versioni precedenti

**Definizione 6.2.2: Source compatibility**

Due versioni di una libreria sono sorgente compatibili se un programma esistente continua a funzionare anche dopo essere stato ricompilato con la nuova versione

**Definizione 6.2.3: Behavioural compatibility**

Due versioni di una libreria sono behavioural compatibili se un programma esistente continua a produrre lo stesso output con lo stesso input

**Note:-**

Per mantenere la source compatibility tra Java 7 e Java 8, si è deciso di introdurre i metodi di default

**Definizione 6.2.4: Metodi di default**

I metodi di default sono metodi che possono essere implementati all'interno di un'interfaccia. Essi permettono di aggiungere nuove funzionalità alle interfacce senza rompere la compatibilità con le versioni precedenti. Per fare ciò vengono aggiunti metodi opzionali (che possono essere implementati o meno). Se non vengono implementati, viene usata l'implementazione di default

**Note:-**

I metodi di default vengono usati per modificare molte interfacce di Java collections per rendere più semplice l'uso delle lambda-espressioni e mantenere la backward compatibility

## 6.3 L'interfaccia Function

**Definizione 6.3.1: Function⟨T, R⟩**

L'interfaccia Function⟨T, R⟩ è un'interfaccia funzionale che rappresenta una funzione che prende un parametro di tipo T e ritorna un parametro di tipo R. Essa ha un solo metodo astratto:

- R apply(T t).

Oltre a questo presenta il metodo di default:

- Function⟨T, R⟩ andThen(Function⟨? super R, ? extends V⟩ after) che ritorna una funzione che prima applica la funzione corrente e poi la funzione after.