Tecniche di Programmazione Funzionale e Imperativa

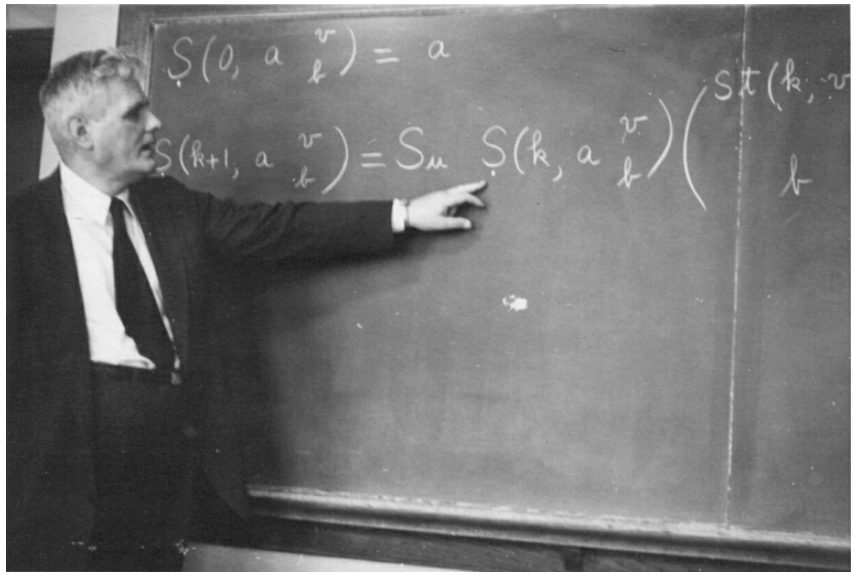
Ivano Salvo

In principio era il Lambda Calcolo
Lambda Calcolo in Haskell

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **2**, 25 febbraio 2022



Lezione 2.1:
In principio era il λ-calcolo

λ-calcolo: anima e corpo

[Alonso Church, 1931]

In **matematica** una funzione $f : A \rightarrow B$ è definita semplicemente come un sottoinsieme del prodotto cartesiano $A \times B$, con il vincolo che a ogni elemento $a \in A$ sia associato un unico elemento $f(a) \in B$ (dipendenza **funzionale**).

In analisi siamo abituati a scritture come $f(x) = e^{\sin(x)} + \log(\cos(x))$ che **esplicitano la dipendenza funzionale** tra una variabile indipendente x e una variabile dipendente y.

In origine, il λ-calcolo vuole:

1) **esplicitare** questa **dipendenza** (λ-notazione, nel nostro esempo $f = \lambda x.\ e^{\sin(x)} + \log(\cos(x))$) e

2) esplicitare il **processo di calcolo** che porta da un elemento del dominio a uno del codominio.

In altre parole il λ-calcolo è **sintassi**, quindi **calcolo meccanico** (corpo) mentre l'usuale idea matematica di funzione è prevalentemente **semantica** (anima).

λ-calcolo: sintassi e computazione

[Alonso Church, 1931]

Sintassi:

M ::=	x	variabile
	$\lambda x.\, M$	astrazione
	$(M\ N)$	applicazione

L'**astrazione** è analoga alla dipendenza di una funzione da un argomento e l'**applicazione** alla chiamata di funzione.

Computazione (**β-regola**):

> Un termine nella forma $(\lambda x.M)\ N$ si dice **β-redex** e si β-riduce al termine M' che è M **in cui tutte le occorrenze di x sono state sostituite con N**.
>
> Occorre fare attenzione ai **nomi delle variabili**! (**α-regola**) ed **evitare la cattura** di variabili libere.

Un termine senza β-redessi è detto in **forma normale** ed è un **valore**.

Variabili libere (free) e legate (bound)

Definiamo le variabili libere FV(M) di un termine M per **induzione sulla struttura sintattica** dei termini:

$$\mathrm{FV}(x) = \{x\} \qquad \mathrm{FV}(\lambda x.\, M) = \mathrm{FV}(M) \setminus \{x\}$$

$$\mathrm{FV}(M\, N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$$

Ad esempio, nel termine: $x(\lambda x.(\lambda y.yx))$ ho che x è libera, mentre x e y sono legate. Osservare che **abbiamo lo stesso nome** per due variabili diverse: questo fenomeno è analogo alla questione variabili **locali/globali** in tutti i Linguaggi di Programmazione.

Il termine $x(\lambda x.(\lambda y.yx))$ peraltro è **equivalente** al termine $x(\lambda z.(\lambda y.yz))$ e a tutti i termini in cui **rinomino le variabili legate**.

⏵ Per evitare problemi, possiamo assumere tutti i **nomi** delle **variabili legate** **diversi tra loro** e **diversi** da quelli delle **variabili libere** (**Barendregt name convention**).

I termini tali che FV(M) = $\varnothing$ sono detti **combinatori**.

Associazioni & Abusi

Scriveremo sempre:

$F\ N_1 \ldots N_n$ per $(\ldots (F\ N_1) \ldots N_n)$ (**applicazione associa a sinistra**)

$\lambda\ x_1 \ldots x_n.\ M$ per $\lambda\ x_1.\lambda\ x_2. \ldots \lambda\ x_n.\ M$,
cioè: $\lambda\ x_1.\ (\lambda\ x_2.\ (\ldots \lambda\ x_n.\ M)\ldots)$ (**astrazione associa a destra**)

Formalizziamo la relazione di **sostituzione** (di **variabili libere**):

$$x[^N/_x] = N \qquad y[^N/_x] = y \quad (y \neq x)$$

$$(\lambda y.M)[^N/_x] = \lambda y.(M[^N/_x]) \qquad \text{se } y \neq x$$

$$(\lambda x.M)[^N/_x] = \lambda x.M \qquad (\text{Il } \lambda x \text{ “oscura” } x \text{ esterni})$$

$$(M\ P)[^N/_x] = (M[^N/_x]\ P[^N/_x])$$

Dopodichè la ***β*-regola** si riscrive:

$$(\lambda x.M)N \rightarrow_\beta M[^N/_x]$$

Alcuni Combinatori Famosi

[Alonso Church, 1931]

Identità: I $\equiv \lambda x.x$ $\qquad \forall M$ ho che **I** $M \equiv (\lambda x.x)M \rightarrow M$

Cancellatori:

K (o **T** per TRUE)$\equiv \lambda x.(\lambda y.x) \equiv \lambda xy.x$ $\qquad$ **K** $M\ N \rightarrow M$

O (o **F** per FALSE)$\equiv \lambda xy.y$ $\qquad$ **O** $M\ N \rightarrow N$

- **if** x **then** M **else** $N \equiv \lambda x.x\ M\ N$, infatti:

$$(\lambda x.x\ M\ N)\ \mathbf{T} \rightarrow \mathbf{T}\ M\ N \equiv (\lambda xy.x)M\ N \rightarrow M$$

$$(\lambda x.x\ M\ N)\ \mathbf{F} \rightarrow \mathbf{F}\ M\ N \equiv (\lambda xy.y)M\ N \rightarrow N$$

Compositori: S $\equiv \lambda xyz.xz(yz)$ $\qquad$ **S** $M\ N\ P \rightarrow M\ P\ (N\ P)$

Esempio: **S K K** $\approx$ **I**, cioè ha lo **stesso comportamento**, infatti:

$$\mathbf{S\ K\ K}\ M \equiv\ (\lambda xyz.xz(yz))\ \mathbf{K\ K}\ M \rightarrow$$
$$\rightarrow \mathbf{K}\ M\ (\mathbf{K}\ M) \rightarrow$$
$$\rightarrow M$$

Altri Combinatori Famosi

Tuple: La tupla $(M_1, M_2, \ldots, M_k)$ si può rappresentare con il termine $\lambda x.xM_1M_2\ldots M_k$ (notare la somiglianza con l'**if-then-else**)

I proiettori hanno la forma $\lambda x.x\pi^k_j$ dove $\pi^k_j \equiv \lambda x_1\ldots x_k.\ x_j$. Infatti:

$$\lambda x.x\pi^k_j\ (\lambda x.xM_1M_2\ldots M_k) \rightarrow (\lambda x.xM_1M_2\ldots M_k)\ \pi^k_j \rightarrow$$
$$\rightarrow \pi^k_j M_1M_2\ldots M_k \rightarrow M_j$$

Duplicatori: $\boldsymbol{\omega} \equiv \lambda x.xx$ (noto come **autoapplicazione**) $\boldsymbol{\omega}_3 \equiv \lambda x.xxx$

Il termine $\boldsymbol{\Omega} \equiv \boldsymbol{\omega}\ \boldsymbol{\omega}$ rapresenta la **funzione indefinita**: un programma che non termina mai, senza fare niente. Infatti:

$$\boldsymbol{\Omega} \equiv (\lambda x.xx)\ \boldsymbol{\omega} \rightarrow \boldsymbol{\omega}\ \boldsymbol{\omega} \equiv \boldsymbol{\Omega}\ \text{(forever!)}$$

Anche il termine $\boldsymbol{\Omega}_3 \equiv \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3$ rappresenta una computazione non-terminante, ma si comporta piuttosto come un "**allocatore di memoria**", infatti, continua a "**crescere**" **indefinitamente**:

$$\boldsymbol{\Omega}_3 \equiv (\lambda x.xxx)\ \boldsymbol{\omega}_3 \rightarrow \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3 \equiv \boldsymbol{\Omega}_3\ \boldsymbol{\omega}_3 \rightarrow \boldsymbol{\Omega}_3\ \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3 \rightarrow \ldots$$

Numeri, Iteratori e Ricorsori

[Church/Kleene, 1936]

Usando l'idea di duplicazione, in λ-calcolo è facile scrivere **ricorsori** o **iteratori**, per esempio i **numerali di Church**:

$$\underline{n} \equiv \lambda sz.s(s(\ldots(sz)\ldots)) \qquad [n \text{ applicazioni}]$$

Esempio $\underline{0} \equiv \lambda sz.z \equiv \mathbf{O} \equiv \mathbf{F}$ $\qquad \underline{3} \equiv \lambda sz.s(s(sz)))$

succ $\equiv \lambda xsz.s(xsz)$ $\qquad$ ma anche succ' $\equiv \lambda xsz.xs(s\ z)$

da cui: succ $\underline{3} \equiv (\lambda xsz.s(xsz))\ \underline{3} \to \lambda sz.s(\underline{3}sz) \equiv \lambda sz.s(\lambda xy.x(x(xy))\ s\ z)$

$\to \lambda sz.s(s(s(sz))) \equiv \underline{4}$

Attenzione! Le s e z dentro $\underline{3}$ sono **variabili diverse** dalle s e z esterne di succ). Le variabili legate da un λ si **devono rinominare** per evitare confusioni di nomi.

➧ Che funzione è: $\lambda xy.x$ succ y? $\underline{n}$ succ $\underline{m}$ riduce a $\underline{n+m}$, perché itera n volte l'operazione di successore su $\underline{m}$.

Morale: I **numerali di Church** rappresentano i **numeri naturali**, ma sono al tempo stesso iteratori (cioè permettono di definire funzioni per **iterazione** e **ricorsione primitiva**)

Programmare in λ-calcolo

[Kleene, 1936]

Come scrivere la funzione **predecessore**?

Idea: iterare n volte un termine P che trasforma la coppia $(\underline{m}, \underline{n})$ in $(\underline{n}, \underline{n+1})$, partendo dalla coppia $(\underline{0}, \underline{0})$. Occorre "**perdere un giro**" $(\underline{0}, \underline{0}) \to (\underline{0}, \underline{1}) \to (\underline{1}, \underline{2}) \to (\underline{2}, \underline{3}) \to \ldots \to (\underline{n-1}, \underline{n})$.

$$P = \lambda x.x(\lambda y_1 y_2 z.\ z y_2\ (\mathsf{succ}\ y_2))$$

Infatti: $P(\lambda w.w\underline{m}\ \underline{n}) \to (\lambda w.w\underline{m}\ \underline{n})\ (\lambda y_1 y_2 z.\ z y_2\ (\mathsf{succ}\ y_1)) \to$

$$\to (\lambda y_1 y_2 z.\ z y_2\ (\mathsf{succ}\ y_2))\ \underline{m}\ \underline{n} \to \lambda z.\ z\ \underline{n}\ (\underline{\mathsf{succ}}\ \underline{n}) \equiv (\underline{n}, \underline{n+1})$$

per cui: $\underline{n}\ P\ (\underline{0}, \underline{0}) \to (\underline{n-1}, \underline{n})$.

Esercizi: provare a definire prodotti, esponenziali, fattoriali, fibonacci ☺ e concluderne la seguente affermazione:

Tesi di Church-Turing: *Ogni funzione calcolabile è λ-definibile.*

Fu **originariamente formulata da Alonzo Church**, il giorno che il suo allievo Kleene scoprì il **predecessore** (mentre era dal dentista, pare)! Poi riadattata alle Macchine di Turing.

Combinatore di Punto fisso

[Curry, Turing 1936]

L'iterazione sui naturali (e la ricorsione primitiva) non sono in realtà sufficienti a rappresentare tutte le funzioni computabili: occorre un **principio di ricorsione più generale**.

Teorema: Nel λ-calcolo esiste un termine $\mathbf{Y}$ tale che per ogni altro termine M, $\mathbf{Y}\ M \rightarrow M\ (\mathbf{Y}\ M)$.

Dim: Consideriamo $\theta \equiv \lambda xy.y(xxy)$ e definiamo $\mathbf{Y}_T = \theta\theta$.
Abbiamo che:

$$\mathbf{Y}_T\ M \equiv (\theta\theta)M \equiv ((\lambda xy.\underline{y}(xxy)\ \theta)\ M \rightarrow (\lambda y.\underline{y}(\theta\theta y))\ M$$
$$\rightarrow M\ (\theta\theta M) \equiv M\ (\mathbf{Y}_T\ M) \qquad \square$$

$\mathbf{Y}_T$ è il **combinatore di punto fisso di Turing**. Ne esistono (infiniti) altri. Tra i più noti, il **combinatore paradossale di Curry**:

$$\mathbf{Y}_C \equiv \lambda f.\ (\lambda\, x.\underline{f}\,(x\ x))\ (\lambda\, x.\underline{f}\,(x\ x))$$

Esempio: $\mathbf{Y}_T\, \boldsymbol{I} \equiv (\theta\theta)\ \boldsymbol{I} \equiv ((\lambda xy.\underline{y}(xxy)\ \theta)\ \boldsymbol{I} \rightarrow (\lambda y.\underline{y}(\theta\theta y))\ \boldsymbol{I}$
$\rightarrow \boldsymbol{I}\ (\theta\theta\ \boldsymbol{I}) \equiv \theta\theta\ \boldsymbol{I} \equiv \mathbf{Y}_T\boldsymbol{I}$, mentre $\mathbf{Y}_C\, \boldsymbol{I} \rightarrow (\lambda\, x.\ \underline{\boldsymbol{I}(x\ x)})\ (\lambda\, x.\underline{\boldsymbol{I}(x\ x)}) \rightarrow$
$\rightarrow (\lambda\, x.\ x\ x)\ (\lambda\, x.x\ x) \equiv \boldsymbol{\omega}\ \boldsymbol{\omega} \equiv \boldsymbol{\Omega}$

Ricorsione generale in λ-calcolo

[Church/Kleene, 1936]

Sia f una funzione definita per ricorsione $f = M[f]$, dove $M[f]$ è un termine che contiene la variabile libera f.

La funzione definita dall'equazione ricorsiva $f = M[f]$ sarà semplicemente $\mathbf{Y}_T\ \lambda f.\ M[f]$.

Esempio: prendiamo il fattoriale.

$$f\ n = \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ f\ (n\text{-}1)$$

Prima traduco **la parte destra** in λ-calcolo, usando le codifiche viste per **if-then-else**, numerali di Church etc. e diventa:

$$\lambda z.\ z\ \underline{1}\ (f\ (\underline{\mathsf{pred}}\ n))$$

dopodiché astraggo su f ed n e applico un operatore di punto fisso e ottengo la rappresentazione in λ-calcolo:

$$\mathbf{Y}_T\ (\lambda f.\ \lambda n.\ \lambda z.\ z\ \underline{1}\ (f\ (\underline{\mathsf{pred}}\ n)))$$

Lezione 2.2:
λ-calcolo in

La funzione identità e il suo tipo

In Haskell abbiamo due modi di definire funzioni: uno lo abbiamo già visto. Rivediamolo con l'identità: assegnamo un nome a un'espressione.

L'altro è usando le **lambda-espressioni**.

`i` e `i'` sono l'identità **su tutti i tipi**.

Non sorprende che sia l'identità anche sulle funzioni!

Ovviamente: **`i . i = i`** e anche **`f . i = f = i . f`**

```
i x = x   -- definiamo l'identità
:type i
i :: t -> t

i' = \x -> x  -- ≡ λx.x
:t i'
i' :: t -> t
```

```
>i 42
42
>i 'a'
'a'
>i [1,2,4]
[1,2,4]
```

```
>let j = i i
>j 42
42
```

Polimorfismo e Type Inference

Nella definizione della funzione identità il **programmatore non scrive nessuna informazione** di tipo.

Il compilatore calcola il **tipo principale** (**type inference**): tutti gli altri tipi corretti per la funzione identità sono istanze del tipo principale, cioè possono essere ottenuti dal tipo principale **sostituendo variabili di tipo con tipi**, ad esempio:

`int->int` oppure `(int -> t) -> (int -> t)`

Quindi abbiamo definito la funzione identità su **tutti i tipi**.

Haskell è **statically type-safe**: se un programma è tipato correttamente nessun errore di tipo si verifica durante l'esecuzione.

Riflessione: l'identità è l'unica funzione di tipo $\forall$ **a. a** $\rightarrow$ **a** (provare a giustificare questa affermazione)

Regole base di di Tipaggio

Vedremo in seguito come il type-checker **indovina** (!?) il **tipo principale** dei programmi Haskell. Ora vediamo come si controllano i tipi dei termini.

Chiameremo θ una **sostituzione**, cioè una funzione a **dominio finito** che **associa variabili di tipo a tipi.**

Esempio: Prendiamo la sostituzione θ definita da $\theta(a)=b\to c$. L'effetto di applicare θ al tipo $a\to c$ sarà il tipo: $\theta(a)\to \theta(c) \equiv (b\to c)\to c$, perché le sostituzioni si comportano come l'identità fuori dal loro dominio finito, e quindi $\theta(c)=c$.

Le sostituzioni compongono, come tutte le funzioni.

Esempio: Prendiamo ora θ', definita da $\theta'(c)=$`char` avrò che $(\theta' \circ \theta)(a \to c) = \theta'(\theta(a \to c)) = \theta'((b\to c)\to c)=(b\to$`char`$)\to$`char`.

Se un tipo $\sigma = \theta(\tau)$ per una qualche sostituzione θ, diremo che τ è **più generale** di σ. Due tipi σ e τ sono **unificabili**, se esiste una sostituzione θ tale che $\theta(\sigma) = \theta(\tau)$.

Regole base di di Tipaggio

Avremo bisogno di un ambiente $\Gamma: Var \rightarrow Types$ per dare un tipo alle **variabile libere** di un termine.

I programmi sono **termini chiusi**, ma avremo in generale bisogno di tipare sotto-termini con variabili libere.

Deriveremo **giudizi** (=tipaggi) nella forma $\Gamma \vdash M : \tau$ che significa "con le assunzioni di tipo in Γ sui nomi liberi di M, tipo M con il tipo Γ". Ecco le regole:

$$\frac{x:\alpha \in \Gamma}{\Gamma \vdash x:\alpha}(VAR) \qquad \frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x.M:\sigma \longrightarrow \tau}(ABS)$$

$$\frac{\Gamma \vdash M:\sigma \longrightarrow \tau \quad \Gamma \vdash N:\rho \quad \exists\theta.\theta(\boldsymbol{\sigma'}) = \theta(\boldsymbol{\rho'})}{\Gamma \vdash M\ N:\ \theta(\tau)}(\text{APP})$$

Nella regola (APP) σ' e ρ' sono ottenuti da σ e ρ rinominando **eventuali variabili quantificate** universalmente: questo permette di essere più liberi nei tipaggi.

Esempi di di Tipaggio

- Posso applicare l'identità **I** di tipo $\alpha \to \alpha$ a un intero, perché c'è la sostituzione $\theta(\alpha)$ = `Integer`, che unifica il tipo del dominio di **I** (che è α) con il tipo del numero (che è `Integer`).

- Posso tipare l'applicazione dell'identità a sé stessa **I I**, perché:

1) prima rendo generiche le variabili: la prima **I** viene tipata con $\alpha' \to \alpha'$ e la seconda con $\alpha'' \to \alpha''$

2) c'è la sostituzione $\theta(\alpha') = \beta \to \beta$, e $\theta(\alpha'') = \beta$. Quindi la prima **I** viene tipata (viene **specializzato** il tipo) con $(\beta \to \beta) \to (\beta \to \beta)$ e la seconda con $\beta \to \beta$.

3) A questo punto, il dominio della prima, $(\beta \to \beta)$, coincide con il tipo della seconda, $\beta \to \beta$ e posso tipare l'applicazione.

Osservazione: noi useremo solo termini chiusi, ma durante la derivazione di tipo potrei dover considerare **sottotermini aperti** (per tipare $\lambda x.M$ devo tipare M in cui x occorre libera)

Altre piccole funzioni famose…

Facciamo altri esempi, mostrando gli altri combinatore base del lambda calcolo in Haskell.

Cominciamo con il combinatore **K**, come funzione e lambda-espressione.

Oppure il combinatore **S**.

Anche qui potremo scrivere una legge algebrica: `s k k = i`

Chi è `c`? Guardate il tipo!

```
c = \x y z = x (y z)
c ::(b -> c) -> (a -> b) -> a -> c
```

```
k x y = x -- proiettore primo arg.
:type k
k :: t1 -> t2 -> t1

k' = \x y -> x   -- ≡ λxy.x
:t k'
k' :: t1 -> t2 -> t1
```

```
s x y z = x z (y z)
s' = \x y z = x z (y z)
:t s
s :: (t2 -> t1 -> t) -> (t2 -> t1)
   -> t2 -> t
> s k k 42
42
```

Esempio (difficile) di Tipaggio

Vediamo come si tipa `s k k` (termine Haskell) equivalente al λ-termine **S K K**: GHCi ci dice (usiamo notazione matematica):

$$\mathbf{S} :: (t_2 \to t_1 \to t) \to (t_2 \to t_1) \to (t_2 \to t) \text{ e } \mathbf{K}_1 :: t_1 \to t_2 \to t_1$$

Rinominiamo tutte le **variabili quantificate** in modo da renderle **tutte diverse tra loro**, per non confondere i tipi di **S** e delle **due occorrenze** di **K**. Lasciando immutate quelle di **S**, abbiamo:

$\mathbf{S}::(t_2 \to t_1 \to t) \to (t_2 \to t_1) \to (t_2 \to t)$ $\mathbf{K}_1:: a' \to b' \to a'$ $\mathbf{K}_2:: a'' \to b'' \to a''$

Quindi devo unificare $t_2 \to t_1 \to t$ con $a' \to (b' \to a')$ e $t_2 \to t_1$ con $a'' \to (b'' \to a'')$ in modo coerente. Calcoliamo la sostituzione θ.

Dal primo vincolo abbiamo che $\theta(t_2) = \theta(t) = \theta(a')$, diciamo u_1 e $\theta(t_1) = \theta(b')$, diciamo u_2.

Dal secondo vincolo abbiamo che $\theta(t_2) = \mathbf{\theta(a'') = u_1}$ (e va bene senza ulteriori sostituzioni) e $\theta(t_1) = \theta(b'' \to a'')$.

Avendo già $\theta(t_1) = u_2$ basta "comporre" con una sostituzione θ' tale che $\theta'(u_2) = u_3 \to \mathbf{u_1}$ e $\theta'(b'') = u_3$ e $\theta'(a'') = \mathbf{u_1}$. Il risultato è $\theta' \circ \theta(t_2 \to t) = \theta' \circ \theta(t_2) \to \theta' \circ \theta(t) = u_1 \to u_1$ che è il tipo di **I** ☺.

Lambda termini NON tipabili

```
>omega = \x -> x x
    Occurs check: cannot construct the infinite type:
                  t1 ~ t1 -> t
    Relevant bindings include
      x :: t1 -> t (bound at lezione1.hs:62:10)
      omega :: (t1 -> t) -> t (bound at lezione1.hs:62:1)
    In the first argument of ‘x’, namely ‘x’
    In the expression: x x
```

Il problema è che occorrebbe tipare la variabile ***x*** **con due diversi tipi**, il primo con "una freccia in più" del secondo

Esistono teorie dei tipi più sofisticate in cui `omega` è correttamente tipabile, ma **non sono decidibili** (quindi un compilatore non sarebbe in grado di calcolare i tipi)

Tutti i lambda-termini **tipabili** in Haskell **terminano**

Non sono ovviamente **tipabili** di conseguenza neanche gli **operatori di punto fisso**. Tuttavia…

Defin. ricorsive e non-terminazione

```
>omega' x = omega' x
>:type omega'
omega' :: t -> t1
 -- quale strana funzione può essere h che prendendo
 -- un parametro di un qualsiasi tipo t restituisce
 -- un risultato di un qualsiasi altro tipo t1?
 -- Ovviamente:
> omega' 42
^CInterrupted.
 -- omega' non termina.
 -- A ben vedere è il punto fisso dell'identità!
```

Ogni funzione soddisfa questa equazione. In particolare, la funzione **ovunque indefinita**, che è il **minimo punto fisso**

La **semantica** di una **equazione ricorsiva** è il **punto fisso** della **trasformazione** indotta dalla definizione. La trasformazione indotta dalla definizione di **`omega`** è un'η-equivalenza dell'**identità**, per la precisione $\lambda xy.xy$

home-made undefined

```
myUndefined = myUndefined
>:type myUndefined
myUndefined :: a
```

Ancora una volta, **ogni termine di Haskell soddisfa questa equazione** (e, mi sbilancio, non solo di Haskell!): non abbiamo definito nessun termine specifico.

Quindi stavolta definiamo l'**oggetto indefinito** puro.

Notate che il suo **tipo** è il **più generale di tutti** (anche di omega' il cui **tipo prescrive** che sia una **funzione**).

Infatti `myUndefined` appartiene a **tutti i tipi**.

Stavolta il lambda termine di cui calcolare il punto fisso è proprio l'identità…

… e come potete facilmente verificare, **Y I ≡ Ω**

Numerali di Church

Vediamo i numerali di Church, e soprattutto i loro tipaggi.

Vedete che i tipi diventano via via **più specifici** (cioè hanno **più vincoli**)…

fino al tipo di due, che è il **tipo generale dei numerali di Church** $\forall a.(a \to a) \to a \to a$ (e che uno si aspetterebbe anche per zero e uno, che invece hanno tipi più generali).

Se provo a valutare due, ottengo errore: niente paura, due **è una funzione** e Haskell **non sa come stamparla**, però…

```
zero = \s z -> z
zero :: t -> t1 -> t1
uno = \s z -> s z
uno :: (t1 -> t) -> t1 -> t
due = \s z -> s (s z)
due :: (t -> t) -> (t -> t)
```

```
>due
<interactive>:13:1:
    No instance for (Show ((t0 -> t0) -> t0 -> t0))
      arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
```

```
> due (+1) 0
2
```

Numerali di Church come iteratori

Chiudiamo questo piccolo viaggio nel mondo del λ-calcolo con un altro esempio.

Definiamo una funzione f sulle coppie numeriche.

A quanto valuta $\underline{n}\, f\, (1, 0)$?

Definiamo il numerale $\underline{5}$ e proviamo a vedere…

(ricordate sempre di applicare alla fine le funzioni (+1) e 0 per poter visualizzare i risultati come intere!)

```
f (m, n) = (m+n, m)
f :: (Number a)=>(a, a) -> (a, a)
```

```
cinque = \s z->s(s(s(s(s z))))
> cinque f (1, 0) (+1) 0
(8, 5)
```

Esercizi della Lezione 2

- **Esercizio**: Scrivere due funzioni Haskell:

```
toCN :: Integer-> (a->a)->(a->a)
fromCN :: (a->a)->(a->a)->Integer
```

per codificare/decodificare numerali di Church e interi.

- **Esercizio**: Anche se `\x->x x` non è tipabile in Haskell, alcuni termini hanno i tipi giusti per auto-applicarsi, come **I**.

Verificare, prima manualmente, e poi con GHCi che i numerali si possono applicare con sé stessi. Ma quanto vale l'auto-applicazione n n di due numerali di Church?

Per scoprirlo, potete aiutarvi con le funzioni `toCN` e `fromCN`.

Intuita la funzione… dimostrate che è proprio quella la funzione!

Verificare poi che la versione binaria `\x y->x y` è tipabile e applicata a due numerali di Church calcola la stessa funzione (in versione binaria).

Lezione 2

That's all Folks…

Grazie per l'attenzione…

…Domande?