Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Introduzione a Haskell
Strategia di valutazione

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **2**, 4 marzo 2021

Lezione 2a:

Informazioni Pratiche

Esame

Esame:

Esercizi settimanali.

Vengono dati nella lezione di giovedì.

Da consegnare entro la domenica successiva (10g).

Colloquio Finale (si spera, formalità).

Eventualmente:

discussioni esercizi nella **terza ora** del venerdì.

Ho aperto un classroom.

Lezione 2c:

In principio era il λ-calcolo II

In principio era il λ-calcolo

[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)

[Church/Kleene, 1936]

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

$$\mathrm{FV}(x) = \{x\}$$

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

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

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**).

Associazioni & Abusi

[Church/Kleene, 1936]

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:

$x[^N/_x] = N \qquad y[^N/_x] = y$

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

$(\lambda x.M)[^N/_x] = \lambda x.M$ [Il λx "**oscura**" x]

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

Numeri, Iteratori e Ricorsori

[Church/Kleene, 1936]

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 $\equiv \lambda xsz.s(xsz)$ 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? A cosa riduce $\underline{n}$ succ $\underline{m}$?

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**)

Esempi di λ-termini e computazioni

[Alonso Church, 1931]

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

Cancellatori:

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

O (o **F** per FALSE)$\equiv \lambda xy.y$

if x ***then*** M ***else*** $N \equiv \lambda x.x\ M\ N$

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

Compositori:

S $\equiv \lambda xyz.xz(yz)$ $\qquad\qquad$ **S K K** $\equiv$ **I**, infatti:

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

Duplicatori:

$$\boldsymbol{\omega} \equiv \lambda x.xx \ \text{-}\ \boldsymbol{\omega}_3 \equiv \lambda x.xxx \ \text{-}\ \boldsymbol{\Omega} \equiv \boldsymbol{\omega}\ \boldsymbol{\omega} \ \text{-}\ \boldsymbol{\Omega}_3 \equiv \boldsymbol{\omega}_3\ \boldsymbol{\omega}_3$$

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

Il Punto fisso del λ-calcolo

[Church/Kleene, 1936]

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 $\mathbf{Y}=\theta\theta$.
Abbiamo che:

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

Abbiamo chiamato $\mathbf{Y}$ il **combinatore di punto fisso di Turing**, nome che viene usualmente dato al **combinatore paradossale di Curry**:

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

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

Lezione 1d:

Introduzione a Haskell
λ-calcolo in Haskell

La funzione Identità e il suo tipo

```
>i x = x
   -- definisce la funzione identità
   -- posso chiedere il tipo della funzione:
>:type i
i :: t -> t
   -- posso applicare la funzione a un argomento
   -- l'interprete valuta il risultato:
>i 42
42
   -- ma anche:
>i i
<interactive>:5:1:
    No instance for (Show (t0 -> t0)) arising from a use of 'print'
    In a stmt of an interactive GHCi command: print it
   -- semplicemente non sa stampare una funzione!
   -- però:
> let j = i i
> :type j
j :: t -> t
> j 42 – anche > i i 42
42
```

Questo tipo va letto quantificato universalmente: $\forall \tau.\tau \to \tau$

42 è un intero ed è quindi un'istanza del tipo del dominio di **i**

Anche $\tau \to \tau$ *è un'istanza del dominio e il risultato sarà di tipo* $\tau \to \tau$

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 **type-safe**: se un programma è tipato correttamente nessun errore di tipo può verificarsi durante l'esecuzione.

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

Altre piccole funzioni famose

```
>k x y = x
   -- definisce il proiettore che cancella il
   -- secondo argomento
   -- posso chiedere il tipo della funzione
>:type k
k :: t1 -> t2 -> t1
   -- posso usare in Haskell la lambda notazione
>k' = \x y -> x
   -- che dovrebbe far capire la relazione tra
   -- lambda astrazioni e parametri di una funzione.
   -- Anche:
>i' = \x -> x
   -- e il combinatore S:
>s' = \x y z -> x z (y z)
>:type s'
s' :: (t2 -> t1 -> t) -> (t2 -> t1) -> t2 -> t
 -- e ovviamente:
>s' k' k' 42
42
```

Anche questo tipo va letto quantificato universalmente: $\forall \sigma\, \tau.\tau \rightarrow \sigma \rightarrow \tau$

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!
```

La **semantica** di una **equazione ricorsiva** è il **punto fisso** della **trasformazione** indotta dalla definizione. La trasformazione indotta da **h** è chiaramente **l'identità**

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

`omega'` è equivalente a $\mathbf{Y}\ \mathbf{I} \equiv \boldsymbol{\Omega}$

Strategia di valutazione - I

```
 -- definiamo una semplice funzione...
>double x = x + x
```

Come viene valutato, ad esempio **`double 3`**? Facile…

```
double 21
  -- applico la definizione di double
21 + 21
  -- applico la definizione di +
42
  -- 42 è un valore, non si riduce
```

Strategia di valutazione - II

```
 -- definiamo una semplice funzione...
>double x = x + x
```

Ma **`double (double 21)`**? Qui ho due scelte…

Posso ridurre il double interno oppure il double esterno…
Vediamo il primo caso…

```
double (double 21)
  -- applico la definizione
double (21 + 21)
  -- applico la definizione di +
double 42
  -- di nuovo double
42 + 42
  -- di nuovo +
84
```

Strategia di valutazione - III

```
 -- definiamo una semplice funzione...
>double x = x + x
```

Stavolta cominciamo dal più esterno…

```
double (double 21)
  -- applico la definizione
(double 21) + (double 21)
  -- anche qui posso scegliere quale…
  -- cominciamo a sinistra…
42 + (double 21)
  -- di nuovo double
42 + 42
  -- di nuovo +
84
```

E otteniamo lo stesso risultato… anche **se con più conti** (è stato valutato **due volte** `double 21`)

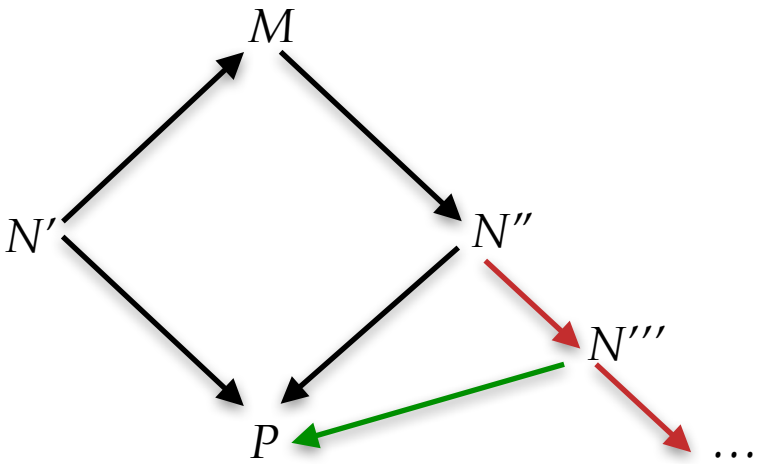
Confluenza (Church-Rosser)

Non è un caso fortunato. In λ–calcolo (e Haskell) vale il seguente:

Teorema [**Church-Rosser**] Se $M \rightarrow^* N'$ e $M \rightarrow^* N''$ allora esiste un termine P tale che $N' \rightarrow^* P$ e $N'' \rightarrow^* P$

Il fatto che esista **non significa che lo troviamo**! (vedi seguito). Tuttavia, se tutte le riduzioni di M terminano, necessariamente P è un valore (in forma normale) e viene raggiunto da tutte le riduzioni.

Corollario [**Unicità delle Forme Normali**] Se *N' ed N'' sono forme normali e* $M \rightarrow^* N'$ e $M \rightarrow^* N''$ allora $N'=N''$.



Strategia di valutazione - IV

```
 -- nonostante la valutazione di (omega' 42) non
 -- termini...
>k i (omega' 42) 42
42
 -- Haskell riduce k quindi:
 -- k i (omega' 42) 42 -> i 42 -> 42
```

Haskell riduce sempre il redex **più esterno e più a sinistra** (**leftmost outermost**)

Questo corrisponde alla valutazione dei parametri di una funzione **call-by-name**.

Nel nostro esempio, la funzione `k` è un **cancellatore** che non usa il secondo argomento e valutarlo è inutile.

Questa scelta non è casuale…

Valutazione degli argomenti

Haskell usa la strategia **call-by-name**: un argomento di una funzione viene valutato **solo se necessario**.

Se ricordate, in C si **valutano sempre gli argomenti nel momento della chiamata della funzione** (**call-by-value**): qualcuno forse ricorda l'impossibilità di scrivere delle funzioni `myAnd` e `myOr` con la stessa semantica di `&&` e `||`.

A differenza di `&&` e `||` le funzioni `myAnd` e `myOr` a causa della call-by-value **valutano sempre entrambi i parametri** anche quando non necessario.

In λ–calcolo (e Haskell) vale il seguente:

> **Teorema**. Se una computazione può terminare, termina la riduzione che valuta prima le espressioni esterne.

Nel nostro esempio:

$$\mathbf{K}\ \mathbf{I}\ \mathbf{\Omega}\ 42 \rightarrow \mathbf{I}\ 42 \rightarrow 42$$

Esercizio della Settimana

Scegliete il vostro linguaggio di programmazione preferito (o, ancora meglio, più d'uno) e scrivete dei programmini di test che dimostrano **quale sia la strategia di valutazione degli argomenti di una funzione**.

Lezione 2

That's all Folks…

…Domande?