Linguaggi di Programmazione

Ivano Salvo

FP2: Primi Passi in Haskell

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **2**, 6 ottobre 2020

Lezione 2a:

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:
>:t 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
> :t 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)

Regole base di Inferenza di Tipo

Vediamo le tre regole base per derivare il tipo principale

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

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

$$\frac{\alpha \; variabile \; di \; tipo \; nuova}{x: \alpha \vdash x: \alpha} \; (VAR)$$

$$\frac{\Gamma, x: \sigma \vdash M: \tau}{\Gamma \vdash \lambda x. M: \sigma \longrightarrow \tau} \; (ABS)$$

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

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)

Le regole sono **syntax-driven**

Altre piccole funzioni famose

```
>k x y = x
   -- definisce il proiettore che cancella il
   -- secondo argomento
   -- posso chiedere il tipo della funzione:
>:t 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 -> x
   -- e il combinatore S:
>s' = \x y z -> x z (y z)
>:t s'
s' :: (t2->t1 -> t) -> (t2 -> t1) -> t2 -> t
 -- e ovviamente:
>s' k' k' 42
42
```

Anche uesto tipo va letto quantificato universalmente: $\forall \sigma\ \tau.\tau \to \sigma \to \tau$

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
>:t 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.
 -- h 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 $\mathbf{Y\ I \equiv \Omega}$

Strategia di valutazione

```
 -- 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:

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

Curryficazione (1)

```
>k''(x,y) = x
>:t k''
k'' :: (t, t1) -> t
 -- (t, t1) è il tipo coppia o prodotto cartesiano
 -- la funzione k'' è parente di k  e k'
> somma x y = x+y
> :t somma
somma:: (Num t) => t->t->t
 -- somma è funzione di due numeri,
 -- non di una coppia di numeri!!!
 -- anche semplicemente (+)
> :t (+)
(+) :: (Num t) => t->t->t
 -- Ovviamente possiamo definire somma' sulle coppie
> somma' (x, y) = x+y
> :t somma'
somma'':: (Num t) => (t,t)->t
 --Per Haskell somma' è funzione di un solo parametro
 -- coppia!
```

Curryficazione (2)

Tutto si fonda sul fatto che esiste un **isomorfismo** (anche da un punto di vista set-teoretico tra gli insiemi:

$$A \times B \to C \cong A \to (B \to C)$$

Attenzioni alle parentesi!

C'è un **vantaggio** ad avere le versioni curryficate: posso passare solo un argomento! Questo spesso **permette di comporre le funzioni** in modo più efficace.

```
>somma3 = (+3)
>:t somma3
somma3 :: Integer -> Integer
 -- è una funzione di un solo parametro.
 -- osservate che l'applicazione a 3 ha specificato
 -- il tipo a essere Integer
```

Curryficazione (3) – Ordine superiore

```
>myCurry f a b = f(a,b)
>:t myCurry
myCurry :: ((t1, t2) -> t) -> t1 -> t2 -> t
 -- myCurry trasforma una qualsiasi funzione sulle
 -- coppie nella sua versione curryficata.
>myUncurry f (a,b) = f a b
>:t myUncurry
myUncurry :: (t1 -> t2 -> t) -> (t1, t2) -> t
 -- myCurry trasforma una qualsiasi funzione di due
 -- parametri, nella versione sulle coppie
>uncurry (+) (42,31)
73
```

`myCurry` e `myUncurry` sono **funzioni di ordine superiore** e prendono una funzione come parametro (e restituiscono come risultato una funzione).

Questo è naturale nei linguaggi funzionali: **functions are first-class citizens**!

Giochiamo coi numerali di Church

```
>zero = \s z -> z
>:t zero
zero :: t -> t1 -> t1

>uno = \s z -> s z
>:t uno
uno :: (t -> t) -> (t -> t)
 -- osservate che la sostituzione
 -- \theta(t)=t1->t1 mostra che il tipo di zero
 -- è più generale di quello di uno
 -- zero ha meno vincoli di uno!

>uno (+1) 0
1
 -- 😂 (+1) è la funzione somma a cui ho passato 1
>mySucc = \x s z -> s (x s z)
>mySucc uno (+1) 0
2
 -- 🤣
```

Definizioni ricorsive e Pattern Matching

Morale: I numerali sono **funzionali**. Passando le funzioni +1 e 0 otteniamo un numero intero. Ovviamente possiamo scrivere la funzione inversa e lo facciamo scoprendo le definizioni ricorsive date per casi (**pattern matching**).

```
>toChurchNumeral 0 = zero
>toChurchNumeral n = mySucc (toChurchNumeral (n-1))
 -- L'ordine dei casi conta!
 -- Essendo n una variabile, può essere sostituita
 -- a qualsiasi intero, 0 compreso.
 -- Haskell analizza le clausole in ordine testuale
 -- prima vanno messe le MENO GENERALI
>:t toChurchNumeral
toChurchNumeral ::
   (Num a, Eq a) => a -> (t -> t) -> t -> t
 -- ma posso usare l'if-then-else che è un
 -- espressione e quindi DEVE SEMPRE restituire un
 -- valore del giusto tipo ELSE NON è OPZIONALE
>toChurchNumeral n =
     if n==0 then zero
             else mySucc (toChurchNumeral (n-1))
```

Per finire: altri esempi e tecnicismi

```
 -- oltre a pattern-matching i programmatori Haskell
 -- amano le guarded definition:
segno n
 | n == 0     = 0
 | n < 0      = -1
 | otherwise = 1

-- Pattern matching sui booleani:
myNot False = True
myNot True  = False

-- si può usare _ (parametro anonimo)
-- quando il parametro non serve
myNot' False = True
myNot'   _   = False

-- notare che il secondo pattern, matcha sempre
-- quando fallisce il primo. Anche:
myAnd True True = True
myAnd  _    _   = False
```

Lezione 2

That's all Folks…

Grazie per l'attenzione…

…Domande?