Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Haskell: Tipi & Funzioni

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **3**, 8 marzo 2021

Lezione 3a:

Tipi base, costruttori di tipo e definizioni di funzioni

Tipi predefiniti I: Bool

Esistono una serie di tipi predefiniti (o **tipi base**): `Bool`, `Int`, `Integer` (interi a precisione illimitata) `Char`, `String`, `Float`, `Double` con significato (spero) chiaro.

Vediamo alcuni esempi, cominciando dal tipo Bool.

```
 -- il tipo Bool ha le usuali costanti
>:t False
False :: Bool
>:t True
True :: Bool
 -- e le usuali funzioni:
>not True
False
>True && False
False
```

Operator Section

Ogni operazione binaria, come ad esempio `&&` (and logico) e `||` (or logico) può essere convertito in una funzione usando le parentesi (**sections**).

Ma anche ogni funzione binaria può essere convertita in un operazione infissa.

Vedremo in seguito la comodità di questa "doppia visione" di operatori e funzioni.

```
>(&&) True False
False
 -- infatti:
>:t (&&)
(&&) :: Bool -> Bool -> Bool
> myAnd x y =  x && y
> myAnd True False
False
>True `myAnd` False
False
```

Definizione di Funzioni

Abbiamo già visto come definire funzioni con **equazioni ricorsive** e **lambda-notazione**, affidandoci al vostro intuito di informatici. Vediamo ora altri dettagli.

Osservo che il **ramo else** nei linguaggi funzionali **non è opzionale**! infatti è un'espressione e deve sempre restituire un valore!

Qualora i rami siano più complessi (ma qui lo vediamo ancora con il `not`) si possono usare le **espressioni guardate** (analoghi ai **case**)

```
-- Possiamo usare espressioni condizionali…
myNot x = if x then False else True

-- oppure espressioni guardate,
-- tipicamente quando i rami sono almeno 3
-- e le condizioni un po' più complesse…
myNot x
  | x          = False
  | otherwise = True
```

Pattern Matching (1)

Il Vero Programmatore Haskell (VPH) però userà di preferenza il **pattern matching**: la funzione viene definita per ogni forma (**sintattica**) del parametro.

Nel pattern-matching l'ordine è importante (viene eseguita la **prima clausola** che fa **matching** col **parametro**).

Quando non utile nella definizione della funzione una parte del pattern si può usare _ (underscore, **parametro anonimo**)

```
-- Pattern matching sui booleani:
myNot' False = True
myNot' True  = False
-- La seconda clausola fa SEMPRE matching…
-- ma vengono analizzate in ordine…
myNot' False = True
myNot'   x   = False
-- Non è utile il nome del secondo parametro
myNot' False = True
myNot'   _   = False
```

Pattern Matching (2)

Vediamo qualche esempio più complesso con funzioni binarie.

Attenzione: il pattern matching non permette di fare `unificazioni', cioè di richiedere che parti del pattern siano uguali tra loro.

```
-- Pattern matching con funzioni binarie
myAnd' True True = True
myAnd'    _    _  = False
-- similmente con l'or logico
myOr' False False = False
myOr'    _    _    = True
-- Il Pattern matching non permette unificazioni!
xor x x = False
xor _ _ = True
    Conflicting definitions for 'x'
-- Meglio un'equazione guardata
xor x y
| x==y = False
| x!=y = True
```

Tipi predefiniti: Int e Integer

Vediamo la definizione di **funzioni ricorsive** sugli interi e l'uso del pattern matching con gli interi (sarebbe meglio dire **naturali**).

Sperimentazione: verificare che potete calcolare numeri di Fibonacci comuque grandi.

```
-- è necessario mettere prima la clausola per 0!
fatt 0 = 1
fatt n = n * fatt (n-1)

-- oppure fibonacci: la famigerata funzione
inefficiente!
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- qui non abbiamo l'iterazione? Come renderla
efficiente?
```

Costruttori di Tipo: freccia funzionale

Abbiamo già visto in Haskell il costruttore di tipo più caratteristico e cioè `->` che permette di costruire i **tipi funzione**.

I tipi funzione originano dai lambda o dalle definizioni di funzioni.

Riflettete sul perché Haskell trova il tipo 'giusto' di `myAnd`.

```
>:t (&&)
&& :: Bool->Bool->Bool

myAnd x y = if (not x) then False
                       else y
>:t myAnd
myAnd :: Bool->Bool->Bool
```

Costruttori di tipo: Tuple

Avrete notato che non scriviamo l'applicazione di funzione come è tipico nei linguaggi di programmazione `f(v1, …, vn)` ma bensì come `(f v1 … vn)`.

In Haskell queste due scritture sono **entrambe possibili**, ma hanno **due significati diversi**. Le parentesi tonde sono il costruttore del tipo **tupla**.

```
>:t (False, True)
(False, True) :: (Bool, Bool)
 -- le tuple hanno dimensione arbitraria
 -- e possono essere non omogenee
>:t (“tpFi”, 42, True, ‘E’)
(“tpFi”,42,True,‘E’)::(String, Integer, Bool, Char)
 -- sulle coppie ci sono le funzioni predefinite
 -- fst e snd che estraggono primo e secondo elemento
>:t fst
fst :: (a, b) -> a
>:t snd
snd :: (a, b) -> b
```

Comporre funzioni

Le terne come nesting di coppie ci danno l'opportunità di vedere come in Haskell sia naturale comporre funzioni.

```
-- distruttori delle terne…
myFst(x, y) = x

fst3 x = fst x
>t fst3
fst3 :: (a, b) -> a

snd3 x = fst (snd x)
>t snd3
snd3 :: (a, (c, b)) -> c
thrd3 x = snd (snd x)
snd3'(x, (y, z)) = y
snd3"(_, (y, _)) = y
-- ma anche:
snd3  = fst . snd
thrd3 = snd . snd

-- . è la composizione di funzioni
> t (.)
```

Curryficazione (1)

```
> k''(x,y) = x
> k x y = x -- k = \x y -> x
> :type 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
> :type somma
somma:: (Num t) => t->t->t
 -- somma è funzione di due numeri,
 -- non di una coppia di numeri!!!
 -- anche semplicemente (+)
> :type (+)
(+) :: (Num t) => t->t->t
 -- Ovviamente possiamo definire somma' sulle coppie
> somma' (x, y) = x+y
> :type 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 = somma 3
>:type somma3
somma3 :: Integer -> Integer
 -- è una funzione di un solo parametro.
 -- osservate che l'applicazione a 3 ha specificato
 -- il tipo a essere Integer

 -- ma anche:
>somma3 = (+3) = (3+)
```

Curryficazione (3) – Ordine superiore

```
>myCurry f a b = f(a,b)
>:type 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**!

Lezione 3b:

Programmazione su Liste in Haskell

Tipi predefiniti e Costruttori di Tipo

Abbiamo già visto alcuni tipi predefiniti (o **tipi base**) come `Bool` e `Integer`. Ce ne sono altri come `Char`, `String`, `Float`, `Double` con significato spero chiaro.

Osserviamo che alcuni valori (ad esempio costanti intere come 42) possono avere tutti i tipi numerici (Integer, Float, Double).

Abbiamo già visto due costruttori di tipo: le **coppie** (e più in generale le **tuple**) e le **funzioni**.

```
>:t (False, True)
(False, True) :: (Bool, Bool)
 -- le tuple hanno dimensione arbitraria
 -- e possono essere non omogenee
>:t (“LdpMat”, 42, True, ‘E’)
(“LdpMat”,42,True,‘E’)::(String,Integer,Bool,Char)
```

Costruttori di Tipo: Liste

Nella tradizione dei Linguaggi Funzionali, c'è predefinito il costruttore di tipo **lista**: `[ ]`.

Il **cons** (inserzione in testa) è un operatore infisso e si scrive `:`.

La scrittura `[1,2,3]` è zucchero sintattico per `1:2:3:[ ]`.

```
 -- la costante lista vuota ha tipo polimorfo
>:t []
[] :: [t]
 -- il cons si scrive : (infisso)
>:t (:)
(:) :: t -> [t] -> [t]
 -- notazioni abbreviate per le liste
> 1:2:3:[]==[1,2,3]
True
```

Funzioni su Liste

Tutte le liste hanno la forma `x:xs`, `x` **testa** e `xs` **coda**: gli Haskellioti usano un nome che finisce in s (plurale) per denotare liste. Possiamo usare questo fatto per scrivere funzioni per **pattern matching**.

Attenzione! l'applicazione di **funzione associa sempre più di tutto** e quindi `f x:xs` viene inteso `(f x):xs` e non `f (x:xs)` Occorre mettere le parentesi se necessario.

```
 -- testa e coda si scrivono facilmente
testa (x:_) = x
coda (_:xs) = xs
> testa [1,2,3]
1
>:t testa
testa :: [t] -> t
> coda [1,2,3]
[2,3]
>:t coda
coda :: [t] -> [t]
```

Pattern matching non esaustivi

Testa e coda **non sono definite** su lista vuota: significa che ci sono dei **casi che sfuggono al pattern matching**.

In questi casi viene sollevata **un'eccezione**, cioè un errore

… che può eventualmente essere **catturato** e **gestito**.

```
 -- xs fa matching con lista vuota sulla lista [42]
> coda [42]
[]
 -- testa e coda non sono definite su lista vuota
> testa []
*** Exception: lezione2.hs:2:1-16: Non-exhaustive
patterns in function testa
 -- la funzione predefinita head si comporta in
 -- modo leggermente diverso
> head []
*** Exception: Prelude.head: empty list
 -- l'eccezione è intercettata e trasformata nel suo
 -- significato `logico'
 -- In questo caso eentrambe le funzioni si bloccano
```

Annidamento dei costruttori di tipo

Le liste si possono annidare: è facile avere le **liste di liste**.

Scriviamo una funzione che sostituisce ogni elemento di una lista con la lista contenente quell'elemento…

… oppure la lista dei suffissi [`xs@(h:txs)` permette di riferire a destra di = sia la lista (`xs`) che le sue componenti (`h` e `txs`)].

```
> :t [[]]
[[]] :: [[t]]

lol (x:xs) = [x]:lol xs
lol [] = []
> :t lol
lol::[t]->[[t]]
> lol [1,2,3]
[[1],[2],[3]]

suffissi xs@(_:txs) = xs:suffissi txs
suffissi     []     = []
> suffissi [1,2,3]
[[1,2,3],[2,3],[3]]
```

Schemi di programmi (1)

Molti programmi hanno una struttura comune: ad esempio iterano una stessa funzione su una lista.

Un funzionale può generalizzare questa forma di ricorsione: è chiamato spesso **reduce**, ma in Haskell si chiama `foldl` (vedremo altre versioni simili, come `foldr`).

```
 -- La funzione length
myLength (_:xs) = 1 + myLength xs
myLength   []   = 0
 -- La funzione sum è simile a length
mySum (x:xs) = x + mySum xs
mySum   []   = 0
 -- La funzione produttoria
myProd (x:xs) = x * mySum xs
myProd   []   = 1

 -- Possiamo generalizzare...aka reduce
myFold f g (x:xs) = f x (myFold f g xs)
myFold f g []     = g
myFold :: (t -> t1 -> t1) -> t1 -> [t] -> t1
```

Schemi di programmi (2)

C'è un gusto tutto funzionalista di scrivere funzioni come composizione di altre funzioni (one-liner) senza fare ricorsione esplicita decomponente liste o altre strutture dati.

Può a volte aiutare anche il compositore, che esiste già predefinito e si chiama (.)

```
-- e quindi...
> myLength' = myFold (\x y->y+1) 0
> mySum' = myFold (+) 0
> myProd' = myFold (*) 1
 -- ma anche...
> myLength'' = myFold (\x -> (+1)) 0

 -- il funzionale composizione di funzioni
c f g x = f (g x)
c :: (t1 -> t) -> (t2 -> t1) -> t2 -> t
 -- oppure infisso predefinito
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Schemi di programmi (3)

Un famoso funzionale è l'**apply-to-all**, noto soprattutto come `map`, che applica una funzione a tutti gli elementi di una lista.

C'è una versione binaria: `zipWith`.

Volendo c'è pure quella *n*-aria. Vedremo come generalizzare.

```
 -- map applica f a tutti gli elementi di una lista:
myMap f (x:xs) = f x : myMap f xs
myMap f [] = []
myMap :: (t -> t1) -> [t] -> [t1]
 -- ad esempio:
> myMap (+1) [41,36,72]
[42,37,73]
> myMap (\x->[x]) [42,37,73]
[[42],[37],[73]] -- myMap (\x->[x]) . (+1) [41,36,72]

 -- spesso è utile una versione 'binaria' di map
myZipWith f (x:xs)(y:ys)=f x y: myZipWith f xs ys
myZipWith f [] _ = []
myZipWith f _ [] = []
myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Schemi di programmi (4)

Tra i casi particolari, molto famoso è `zip`, o **cerniera** che accoppia ordinatamente gli elementi di una lista.

Può essere ottenuto facilmente da `zipWith`.

Possiamo anche immaginare di applicare una lista di funzioni a una lista di argomenti…

```
 -- speso è utile una versione 'binaria' di map
myZip (x:xs)(y:ys)=(x,y) : myZip xs ys
myZip [] _ = []
myZip _ [] = []
myZip :: [a] -> [b] -> [(a,b)]

 -- ma ovviamente
myZip' = myZipWith (\x y ->(x, y))

applyList (f:fs)(x:xs)=f x: applyList fs xs
applyList [] _ = []
applyList _ [] = []
applyList :: [t -> t1] -> [t] -> [t1]
```

Una famosa applicazione…

John Backus (progettista del Fortran) nel **1978** scrisse un celeberrimo articolo che rilanciò la programmazione funzionale ponendo l'accento sulle sue **virtù composizionali**.

Fece l'esempio del **prodotto scalare**.

```
 -- prodotto scalare con myFold e myZipWith:
prodottoScalare xs ys =
   myFold (+) 0 (myZipWith (*) xs ys)

 -- ma anche con map e applyList
prodottoScalare' xs ys =
   myFold (+) 0 (applyList (myMap (*) xs) ys)

 -- ma a ben vedere
 -- giocando con la composizione
prodottoScalare'' xs =
     mySum . ((applyList . (myMap (*))) xs)
```

Lezione 3

That's all Folks…

Grazie per l'attenzione…

…Domande?