Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Funtori e Applicativi
Una piccola applicazione

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 7, 22 marzo 2021

Lezione 7a

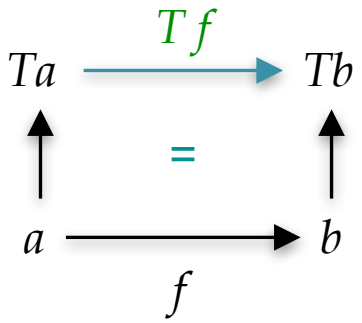
Funtori

More Maps

Abbiamo visto come la `map` e `foldr` generalizzino un gran numero di computazioni sulle liste.

Ora vedremo come il concetto di map si generalizzi a tutti i costruttori di tipo in modo naturale.

Sei io ho un costruttore di tipo T che da oggetti di tipo a permette di costruire oggetti di tipo $T\ a$ allora da ogni funzione $f :: a \rightarrow b$ posso ottenere una funzione $T\ f :: T\ a \rightarrow T\ b$. Questo corrisponde al concetto categoriale di funtore.



Per esempio, nelle liste T è dato da `:` e `[]`

Classe Functor

La classe `Functor` richiede che sia definita una funzione `fmap` che soddisfa al diagramma visto prima.

Osservare che il tipo di `fmap` non è solo parametrico nei tipi `a` e `b`, ma è parametrico rispetto a un costruttore di tipo `t`.

Questo viene inferito perché `t` si applica ad altri tipi.

```
 -- definizione della classe Functor
class Functor t where
   fmap :: (a -> b) -> t a -> t b
```

Esempi

Ovviamente le liste sono istanze della classe `Functor`

e ovviamente `fmap` è proprio la map già nota.

Possiamo definire `fmap` in modo naturale su altri tipi, come ad esempio il costruttore di tipo `Maybe`.

O sugli alberi binari

```
 -- dichiarazione che le liste sono istanze
 -- della classe Functor
istance Functor [] where
   fmap = map
 -- oppure Maybe
istance Functor Maybe where
   fmap f Nothing  = Nothing
   fmap f (Just x) = Just (f x)
 -- oppure BinTree
istance Functor BinTree where
   fmap f EmptyTree  = EmptyTree
   fmap f (Node r Left Right) =
      Node (f r) (fmap f Left)(fmap f Right)
```

```
inc : Functor t => t Int -> t Int
inc = fmap (+1)

inc (Just 2)
(Just 3)
```

Functor Laws

Affinchè il diagramma commuti, deve essere vero che:

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Queste leggi dovrebbero essere soddisfatte da qualunque cosa sia definita come funtore, ma ovviamente **non possono essere verificate dal type-checker**, ma sono responsabilità del programmatore.

Ad esempio, la seguente tipa, ma non soddisfa alle leggi funtoriali.

```
istance Functor [] where
   fmap f []     = []
   fmap f (x:xs) = fmap f xs ++ [f x]
 -- infatti
> fmap (\x -> x) [1,2]
[2,1]
 -- e anche:
> fmap (not . even) [1,2]
[False, True]
> fmap not . fmap even [1,2]
[True, False]
```

Detour: Currying is Beatiful

Non so come abbiate definito `map` usando `foldr`. Ecco la mia soluzione.

Mi ha tuttavia colpito questa versione nel libro di Richard Bird.

`f :: a->b` "nutre" il primo parametro di `(:) :: b->[b]->[b]`, e quindi `(:) . f :: a->[b]->[b]`

che è proprio un'istanza del tipo della funzione da foldare, nel caso particolare in cui il risultato è una lista. Miracoli del currying!

```
 -- la mia soluzione...
myMap  f = foldr (\x y -> f x : y) []

 -- ... e quella nel libro di Richard Bird:
birdMap f = foldr ((:) . f) []
```

Lezione 7b

Applicativi

More More More ...Maps

Abbiamo visto come generalizzare a tutti i tipi la nozione di map, attraverso la classe `Functor`.

È naturale anche considerare il caso di generalizzare a map di varie arità (ad esempo `zipWith` è una forma di `map` binaria).

Ecco quale potrebbe essere questa gerarchia di funzioni:

```
fmap0 :: a -> t a
fmap1 :: (a -> b) -> ta -> tb
fmap2 :: (a -> b -> c) -> ta -> tb -> tc
fmap3 :: (a -> b -> c ->d) ->
                    ta -> tb -> tc -> td
```

Ovviamente, non vogliamo scrivere una classe `Functor0`, `Functor1`, `Functor2`, `Functor3`, …

Esempio: Maybe

L'idea è quella di considerare una forma astratta di applicazione che generalizza l'usuale applicazione di funzioni.

Vorremmo ottenere effetti come questo sul tipo `Maybe`, che permettono di propagare eccezioni senza dover 'scartare' e 'reincartare' i risultati dal tipo `Maybe`.

E poi generalizzare a un numero arbitrario di argomenti senza scrivere le classi `Functor0`, `Functor1`, `Functor2`, `Functor3`…

Vedremo nel seguito come ottenere questo effetto usando solo 2 operatori:

`pure``:: a->t a` (che corrisponde a `fmap0`) e

`<*>``:: t(a->b) -> t a -> t b`

(che permette di ottenere $\texttt{fmap}_{n+1}$ da $\texttt{fmap}_{n}$)

```
 -- propagare risultati corretti con eccezioni
> fmap2 (+) (Just 2) (Just 3)
Just 5
```

...e quindi

```
fmap0 = pure
fmap1 g x = pure g <*> x
fmap2 g x y = pure g <*> x <*> y
```

Questo si ottiene con una estensione della classe Functor come sotto riportato.

Vediamo anche l'esempio di `Maybe`.

```
 -- dichiarazione che di Applicative
 -- come classe derivata da Functor
class Functor t => Applicative t where
   pure :: a -> t a
   (<*>):: t (a -> b) -> t a -> t b

 -- Definizione di pure e <*> in Maybe
istance Applicative Maybe where
   pure = Just
 -- pure x = (Just x)
   Nothing <*> _   = Nothing
   (Just g)<*> mx = fmap g mx
```

Un semplice esempio

Vediamo anche l'esempio di `Maybe`.
Osservate che <*> associa a **sinistra**.

```
 -- vediamo una semplice riduzione:
pure (+) <*> (Just 2) <*> (Just 3)
    → {pure}
(Just (+)) <*> (Just 2) <*> (Just 3)
    → { <*> }
fmap (+) (Just 2) <*> (Just 3)
    → { fmap }
(Just  (+2)) <*> (Just 3)
    → { <*> }
fmap (+2) (Just 3)
    → { fmap }
(Just  (+5))
```

Vediamo sulle liste

La definizione canonica sulle liste in `Prelude` **non porta tuttavia** `pure f <*> xs <*> ys` a essere la `zipWith f xs ys`!

L'idea è (come in Maybe) di modellare una sorta di **nondeterminismo**: per cui una lista è una sequenza di risultati possibili, e applicando una lista di funzioni ad esse, produce tutte le possibili applicazioni delle funzioni a tutti i risultati.

```
-- Definizione di pure e <*> in []
istance Applicative [] where
   -- pure : a -> [a]
   pure x = [x]

   -- <*> : [a -> b] -> [a] -> [b]
   gs <*> xs = [g x | g <-gs, x<-xs]

> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
 -- e non [3,8]!!
```

zipList

Esiste una definizione alternativa dei funtori applicativi sulle liste.

Innanzitutto vediamo la keyword `newtype` che serve a 'wrappare' (incartare) un tipo dentro un altro tipo **con un solo costruttore unario** (qui **l'equivalenza** diventa **per nome**, per cui `ZipList` non è equivalente a (o sinonimo di) `[]`).

Ecco un vecchio amico (`applyList`!)

```
-- altro tipo list:
newType ZipList a = Z [a]

instance Applicative ZipList where
   -- pure : a -> [a]
   pure x = Z (repeat x) -- repeat x = x : repeat x

   -- <*> : [a -> b] -> [a] -> [b]
   Z fs <*> Z xs = Z (zipWith (\f x -> f x) fs xs)

   -- è equivalente ad applyList, ma ovviamente:
   -- pure f <*> Z xs <*> Z ys = Z zipWith f xs ys
```

Applicative Laws

Affinchè il diagramma commuti, deve essere vero che:

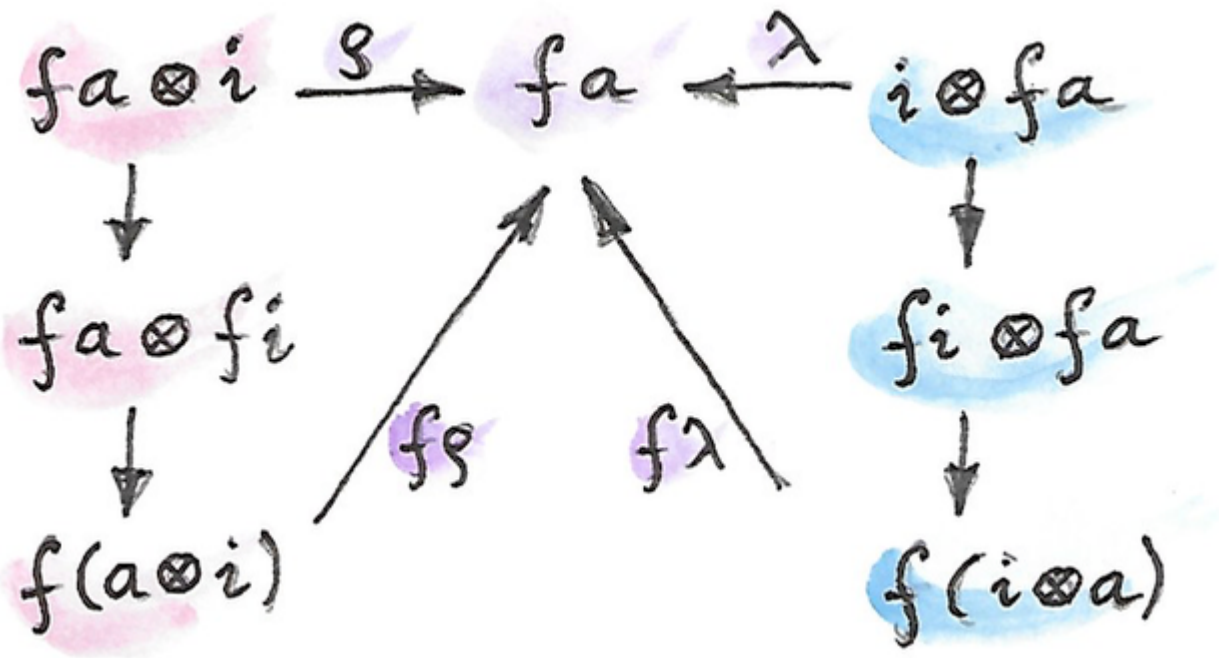
```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Queste leggi dovrebbero essere soddisfatte da qualunque cosa sia definita come funtore, ma ovviamente **non possono essere verificate dal type-checker**, ma sono responsabilità del programmatore.

Ad esempio, la seguente tipa, ma non soddisfa alle leggi funtoriali.

```
istance Functor [] where
   fmap f []     = []
   fmap f (x:xs) = fmap f xs ++ [f x]
 -- infatti
> fmap (\x -> x) [1,2]
[2,1]
 -- e anche:
> fmap (not . even) [1,2]
[False, True]
> fmap not . fmap even [1,2]
[True, False]
```

Applicative Laws



Applicative Laws

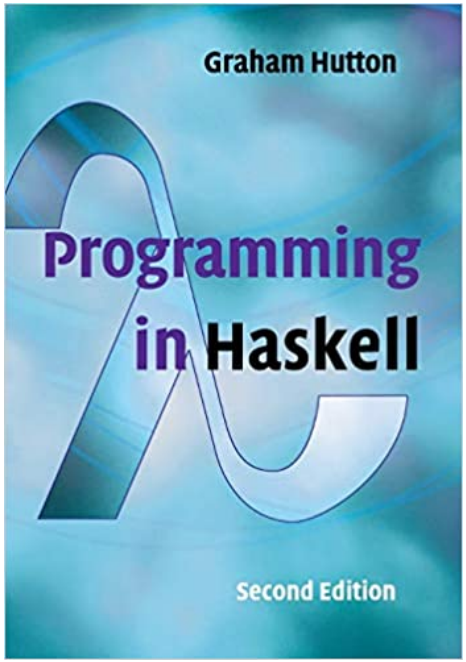
Ecco le leggi (scritte in Haskell) che devono essere soddisfatte dagli applicativi:

1. `<*>` preserva l'identità
2. `<*>` preserva l'applicazione di funzioni
3. Non è importante l'ordine con cui si fa l'embedding
4. Vale l'associatività rispetto a `(.)`

```
pure id <*> x = x

pure (g x) = pure g <*> pure x

x <*> pure y = pure (\g -> g y) <*> x

x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

Lezione 7c:
Comporre, Comporre, Comporre &
Una piccola applicazione

tratta da:



Verificare se una lista è ordinata

Possiamo ovviamente scrivere un programma per pattern matching, che ricalca gli analoghi programmi imperativi…

Ma il vero programmatore Haskell probabilmente preferisce comporre funzioni, vedendo le strutture dati nella loro globalità.

```
 -- stile imperativo
ordinata []  = True
ordinata [x] = True
ordinata (x:y:xs) = x <= y && ordinata (y:xs)

 -- stile funzionale
ordinataVPH xs =
   foldr (&&) True (zipWith (<=) xs (tail xs))

  -- c'è l'and predefinito sulle liste
and = foldr (&&) True  -- io lo chiamerei forall
ordinataVPH' xs = and (zipWith (<=) xs (tail xs))

  -- osservate che:
> and []
True
```

Voting System - 1

Problema: Abbiamo una lista che raccoglie le preferenze. Vogliamo determinare quale candidato ha avuto più voti.

Scriviamo intanto una funzione che conta i voti. Privilegiamo la programmazione "composizionale"

Tornerà infine utile rimuovere i duplicati. Vediamo ancora una soluzione composizionale.

```
> votes :: [String]
votes = ["Rossi", "Bianchi", "Verdi", "Bianchi",
"Bianchi", "Rossi"]

>count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
> count "Rossi" votes
2

rmvDups :: Eq a => [a] -> [a]
rmvDups   []   = []
rmvDups (x:xs) = x : rmvDups (filter (/=x) xs)
>rmvDups votes
["Rossi", "Bianchi", "Verdi"]
```

Voting System - 2

Scriviamo una funzione `result` che presenta i risultati ordinati in ordine di preferenza. Con le definizioni date finora, posso:

1. calcolare la lista dei candidati (senza ripetizioni)
2. per ciascun candidato, contare i voti
3. accoppiare in una lista voti/candidati e ordinarla.

Infine, il vincitore sarà l'ultimo di questa lista (e proietto il secondo elemento della coppia)

Esercizio: definire `last`.

```
results :: Ord a => [a] -> [(Int, a)]
results vs = sort [(count v vs, v) | v <- rmvDups vs]

>results votes
[(1, "Verdi"), (2, "Rossi"), (3,"Bianchi)]

-- and the winner is…

winner :: Ord a => [a] -> a
winner = snd . last . results
```

Voting System - 3

Problema: Consideriamo un caso più complesso: la lista di voti è una sequenza di liste di preferenze.

Il vincitore si ottiene **eliminando iterativamente quelli che ricevono meno prime scelte**.

Nel nostro esempio, il primo a essere eliminato è `"Rossi"` e poi viene rimosso… finché non resta un solo candidato.

```
> ballots :: [[String]]
votes = [["Rossi", "Verdi"],
         ["Bianchi"],
         ["Verdi", "Rossi", "Bianchi"],
         ["Bianchi", "Verdi", "Rossi"],
         ["Verdi"]]
votes' = [["Verdi"],
         ["Bianchi"],
         ["Verdi", "Bianchi"],
         ["Bianchi", "Verdi"],
         ["Verdi"]]
votes''=[["Verdi"],[],["Verdi"], ["Verdi"],["Verdi"]]
```

Voting System - 4

Possiamo sfruttare parte del lavoro precedente per scrivere una funzione `rank` che ordina i candidati.

Poi dobbiamo eliminare il candidato eliminato ed eventuali ballot vuoti.

E infine comporre i pezzi.

```
rank :: Ord a => [[a]] -> [a]
rank = map snd . result . map head

elim :: Eq a => [[a]] -> [[a]]
elim x = map (filter (/=x))

rmvEmpty = filter (/=[])

winnerB :: Ord a => [[a]] -> a
winnerB bs = let (c:cs) = rank (rmvEmpty bs)
   in if cs =[] then c
                else winnerB (elim c bs)
```

Lezione 7

That's all Folks…

Grazie per l'attenzione…

…Domande?