Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Altre Astrazioni Algebrico/Categor.
Input/Output in Haskell

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **15**, 26 aprile 2021

Lezione 15a

Ancora Astrazioni: Monoidi, Foldable & Traversable

Classi e pattern di computazione

Oggi vediamo 3 pattern per generalizzare computazioni su strutture dati:

- **`Monoid`**: generalizza l'idea di avere un'operazione associativa e un'identità (ad esempio `[]` e `++` nelle liste)
- **`Foldable`**: generalizza l'idea di foldare operazioni monoidali dentro le liste
- **`Traversable`**: generalizzano ulteriormente l'idea di `map` integrando, ad esempio, la gestione di possibili fallimenti.

classe Monoid

Ricordiamo al solito che un monoide ha un'operazione associativa `mappend` (scritta anche `<>`) con un'identità (`mempty`).

Queste operazioni devono obbedire alle seguenti leggi:

```
mempty <> x = x
x <> mempty = x
x <>(y <> z)= (x <> y) <> z
```

```
 -- classe Monoid
class Monoid a where
   mempty  :: a
   mappend :: a -> a -> a

   mconcat :: [a] -> a
   mconcat = foldr mappend mempty

 -- ovviamente
instance Monoid [a] where
   -- mempty  :: [a]
   mempty = []
   -- mappend :: [a] -> [a] -> [a]
   mappend = (++)
```

Maybe come Monoide

Il monoide Maybe fa distribuire `Maybe` sulle liste, conservando i valori "buoni" e scartando i `Nothing`. Notare che ciò è diametralmente opposto al comportamento di `mapM`!

Ed è legato all'idea di lista come molteplici risultati di una computazione non-deterministica (come l'applicativo `[ ]`)

```
-- osservate che occorre assumere Monoid a
instance Monoid a => Monoid (Maybe a) where
   -- mempty  ::  Maybe a
   mempty = Nothing

   -- mappend :: Maybe a -> Maybe a -> Maybe a
   Nothing `mappend` my = my
   mx `mappend` Nothing = mx
   (Just x) `mappend` (Just y) =
                              Just (x `mappend` y)
```

Wrapper Types: Monoidi Int & Bool

Diversi tipi possono essere **monoidi rispetto a più operazioni**: ad esempio gli **interi** (e tutti gli insiemi numerici) sono un monoide sia rispetto alla **somma** (con identità **0**) che rispetto al **prodotto** (con identità **1**).

Dato che non si possono definire due classi sullo stesso tipo, occorre ricorrere a un **wrapper type**.

Qui vediamo l'esempio di `Bool`.

```
newtype All = All Bool deriving …
getAll (All b) = b
instance Monoid All where
   mempty  = All True
   (All b) `mappend` (All c) = All (b && c)

newtype Any = Any Bool deriving …
getAny (Any b) = b
instance Monoid Any where
   mempty  = Any False
   (Any b) `mappend` (Any c) = Any (b || c)
```

Foldable

La principale applicazione dei monoidi è combinare dei valori in una struttura dati **ottenendo un singolo valore**. Nel caso delle liste, possiamo definire la funzione `fold`.

Di conseguenza, una lista di valori che appartengono a un monoide possono essere combinati in modo standard usando `mempty` e `mappend` senza dover passare una funzione come `foldr`.

Stessa cosa si può fare con gli alberi binari.

```
fold     :: Monoid a => [a] -> a
fold   []    = mempty
fold (x:xs) = x `mappend` fold xs
-- fold [x, y, z] = x<>(y <> (z <> mempty)

fold     :: Monoid a => Tree a -> a
fold   (Leaf x)    = x
fold (Node l r) = fold l `mappend` fold r
```

Foldable

In generale, la classe **`Foldable`** offre una serie di meccanismi per calcolare funzioni basandosi su questo principio.

L'interfaccia della classe `Foldable` è la seguente.

Ovviamente, ancora una volta, sono le liste l'esempio più ovvio della classe `Foldable`.

```
 -- Interfaccia foldable
class Foldable t where
   fold    :: Monoid a => t a -> a
   foldMap :: Monoid b => (a -> b) -> t a -> b

 -- queste non necessitano a o b essere un monoide
 -- perché la funzione di composizione è fornita
   foldr   :: (a -> b -> b) -> b -> t a -> b
   foldl   :: (a -> b -> a) -> a -> t b -> a
```

Trees as Foldable

Visto che le definizioni delle liste dovrebbero essere immediate, noi vediamo come esempio di `Foldable` gli alberi binari come definiti la lezione scorsa.

```
 -- Trees come Foldable
instance Foldable Tree where
 --fold :: Monoid a -> Tree a -> a
   fold (Leaf x) = x
   fold (Node l r) = fold l `mappend` fold r
 --foldMap :: Monoid b -> Tree a -> (a -> b) -> b
   foldMap f (Leaf x) = f x
   foldMap f (Node l r) =
     foldMap f l `mappend` foldMap f r
 --foldr :: (a -> b -> b) -> b -> Tree a -> b
   foldr f v (Leaf x) = f x v
   foldr f v (Node l r) = foldr f (foldr f v r) l
 --foldl :: (a -> b -> a) -> a -> Tree b -> a
   foldl f v (Leaf x) = f v x
   foldl f v (Node l r) = foldl f (foldl f v l) r
```

Default definitions in Foldable

Come si può immaginare, molte di queste funzioni sono interdefinibili tra loro.

La funzione `toList` gioca un ruolo cruciale nelle definizioni di default fornite insieme alla classe `Foldable`.

```
-- definizioni reciproche…
fold       = foldMap id
foldMap f = foldr (mappend . f) mempty
-- inoltre tutti i Foldable sono riducibili a
-- una lista con:
toList     = foldMap (\x->[x])
-- altre funzioni in Foldable: default definitions
null    = null . toList
length  = length . toList
elem x  = elem x . toList
maximum = maximum . toList
minimum = minimum . toList
sum     = sum . toList
product = product . toList
```

Discussione del design di Foldable

- **Perché così tante funzioni nell'interfaccia?** Perché è possibile dare una definizione comune di default, dando la possibilità, se necessario di fare overriding in una specifica classe.

- **Cosa è necessario definire manualmente?** È sufficiente fornire l'implementazione di **solo 1 tra `foldr` e `foldMap`** e tutte le altre saranno derivate dalle definizioni di default. Usualmente, la più semplice da definire è foldMap.

- **Efficienza?** In generale GHC implementa versioni più efficiente di quelle viste in Haskell, ma che soddisfano alle stesse equazioni.

Generic Functions

Come sempre, uno degli effetti positivi di definire questo tipo di astrazioni è la possibilità di definire funzioni generiche, che dipendono dai **nomi** e dalle **proprietà** delle classi.

Vediamo qualche semplice esempio.

```
average :: Foldable t, Num a => t a -> a
average ns = sum ns / length ns

and :: Foldable t => t Bool -> Bool
and = getAll . foldMap All

all :: Foldable t => (a -> bool) -> t a -> Bool
all p = getAll . foldMap (All . p)

 -- definizioni simili per or e any
> any even (Node(Leaf 1)(Leaf 2)
True

concat :: Foldable t => t [a] -> a
concat = fold
> concat (Node(Leaf [1])(Leaf [2,3])
[1,2,3]
```

Traversables

Concludiamo con una generalizzazione di map che considera la possibilità di fallimenti.

Ancora una volta, ciò **non è strettamente specifico delle liste**.

```
traverse :: (a->Maybe b) -> [a] -> Maybe [b]
traverse g []     = pure []
traverse g (x:xs) = pure (:)<*> g x <*>
        traverse g xs

 -- Esempio
pred :: Int -> Maybe Int
pred n = if n > 0 then Just (n-1) else Nothing

>traverse pred [1,2,3]
Just [0,1,2]

>traverse pred [2,1,0]
Nothing
```

Traversable

Ci limitiamo a vedere la definizione delle classe **Traversable** e la definizione degli alberi binari come sua istanza.

```
class (Functor t, Foldable t) => Traversable where
  traverse :: Applicative f =>
                        (a -> f b) -> t a -> f (t b)
instance Traversable Tree where
  -- traverse :: Applicative f =>
                  (a -> f b) -> Tree a -> f (Tree b)
   traverse g (Leaf x) = pure Leaf <*> g x
   traverse g (Node l r) =
        pure Node <*> traverse g l <*> traverse g r
```

Lezione 15b

Input/Output in Haskell

Il problema dell'input/output

Abbiamo usato Haskell semplicemente per valutare funzioni. Scriviamo definizioni di funzioni e valutiamo il valore delle funzioni sugli argomenti.

In realtà, i programmi interattivi sono un problema nel mondo funzionale, a causa del fatto che **input** e **output** sono, a rigore dei **side-effects**: il **valore di una funzione pura dipende solo dai suoi argomenti**.

Idea: immaginare esista un argomento implicito nelle funzioni interattive che rappresenta lo "stato del mondo".

Si usa quindi una Monade e le espressioni di tipo `IO a` sono dette **actions**.

```
 -- prima approssimazione:
type IO = World -> World

 -- ma un programma può anche tornare valori
type IO a = World -> (a, World)
```

Azioni Base

Il tipo `IO Char` è il tipo delle **azioni che tornano un carattere**, mentre `IO ()` è il tipo che torna la tupla vuota come un risultato (= nessun risultato, assomiglia a `void`), cioè il tipo delle azioni che sono solo side-effects.

Osserviamo che il tipo `IO a` è predefinito in Haskell ed è visto come un tipo dalla definizione nascosta.

Ecco le primitive base per:

- leggere un carattere da tastiera,
- scrivere un carattere da tastiera,
- fornire un valore alle azioni (è un ponte tra le funzioni pure e le azioni: è a senso unico, in quanto una volta impuri **non c'è redenzione!**: a differenza di ST, **`World`** non è accessibile.

```
 -- basic actions:
getChar :: IO Char
putChar :: Char -> IO ()
return :: a -> IO a
```

Sequenze

Usando il costrutto `do` **è possibile mettere in sequenza azioni** in un'unica azione complessa.

L'istruzione return è il modo in cui le azioni possono tornare un risultato ed è un ponte tra il mondo impuro e quello puro: ovviamente **non c'è redenzione** e il contrario non si può fare.

```
 -- sequenza di azioni:
do  v1 <- a1
    v2 <- a2      do {undefined; return 3}=undefined
    ...
    vn <- an
    return (f v1 v2 ... vn)
```

Primitive derivate

```
 -- leggere una stringa
getLine :: IO String
getLine =
   do x <- getChar
      if x == '\n' then
         return []
      else
         do xs <- getLine
         return (x:xs)

 -- scrivere una stringa
putStr :: String -> IO ()
putStr   []    = return ()
putStr (x:xs) = do putChar x
                   putStr xs

 -- scrivere una stringa e andare a capo
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar `\n`
```

Il gioco dell'impiccato (1)

[da *Programming in Haskell*, G. Hutton, 2016]

Come esempio di programma interattivo, vediamo una semplice variazione del gioco dell'impiccato.

```
 -- inizio del gioco:
hangman :: IO ()
hangman = do putStrLen "Think a word: "
             word <- secretGetLine
             putStrLn "Try to guess it: "
             play word

 -- secretGetLine evita di riprodurre la stringa
secretGetLine =
   do x <- secretGetChar
   if x == '\n' then
      do putChar x
         return []
   else
      do putChar '-'
         xs <- secretGetLine
        return (x:xs)
```

Il gioco dell'impiccato (2)

[da *Programming in Haskell*, G.Hutton, 2016]

Occorre interagire col sistema per impedire la riproduzione dei caratteri a video ☺

```
 -- bisogna estendere getChar:
secretGetChar :: IO ()
secretGetChar =
          do hSetEcho stdin False
             x <- getChar
             hSetEcho stdin True
             return x

 -- ci resta da programmare il ciclo di gioco
play word =
   do putStr "?"
      guess = getLine
      if guess == word then
         putStrLen "You got it!!"
      else
         do putStrLen (match word guess)
            play word
```

Il gioco dell'impiccato (3)

[da *Programming in Haskell*, G.Hutton, 2016]

La funzione match mostra i caratteri comuni tra il tentativo e la parola segreta.

```
 -- funzione che genera la lista risultato:
match xs ys = [if elem x ys then x else `-`
                  | x <- xs]
```

Output del programma:

```
*Main> hangman
Think of a word:
------
Try to guess it:
? s
s-----
? test
se--et
? cross
s-cr--
? secret
You got it!
```

Lezione 15

That's all Folks…

Grazie per l'attenzione…

…Domande?