Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Monadi & Input/Output

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **17**, 26 aprile 2022

Lezione 17a

Altri esempi

Rivediamo il valutatore con Log

Rivediamo l'esempio del valutatore che produce il log delle operazioni eseguite, vedendo le definizioni complete.

Cominciamo con le definizioni del tipo **`Out`** come Funtore, Applicativo e Monade.

```
newtype Out a = O (a, String)
   deriving Show

instance Functor Out where
   -- fmap: (a->b) -> Out a -> Out b
 fmap f (O (x, s)) = O (f x, s)

instance Applicative Out where
   -- pure: a -> Out a
 pure x = O (x, "")
   -- <*>:Out (a->b) -> Out a -> Out b
 (O(f, x)) <*> (O (a, y)) = O (f a, x++y)

instance Monad Out where
   -- (>>=) :: Out a -> (a -> Out b) -> Out b
  (O(a, x)) >>= f = let (O (b, y)) = f a
                    in O (b, x++y)
```

Codice del valutatore

Vediamo che ancora una volta, il codice del valutatore deve essere solo minimamente modificato.

Domanda: sarebbe più naturale usare un *"log trasformer"* come nel caso precedente con lo **state transformer**?

Probabile Esercizio nel prossimo homework ☺

```
line t a = "eval(" ++ show t ++ ") <= "
                        ++show a ++ ['\n']

logs (O(v, s)) t = O(v, "eval" ++ show t ++ " <= "
                        ++ show v ++ ['\n'] ++ s)

evalOut (Constant a) = O (a, line (Constant a) a)

evalOut (Div t u) =
    evalOut t >>= \a ->
    evalOut u >>= \b ->
    logs (pure (a `div` b)) (Div t u)
```

Relabeling Trees

Vediamo un altro **problema**: preso un albero, etichettare tutti i nodi con un intero diverso.

In un linguaggio imperativo si potrebbe fare uso di una variabile **globale** o **statica**.

La soluzione ricorsiva standard, consiste nel passare l'ultimo intero usato alle chiamate ricorsive: questo implica usare un parametro e in un linguaggio come Haskell anche un valore di ritorno (le operazioni **non sono sequenzializzate come in un linguaggio imperativo**).

```
 -- Tipo degli alberi (etichette solo sulle foglie)
data BinTree a = F a | R (Tree a)(Tree a)

rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (F _) n   = (F n, n+1)
rlabel (R l r) n = (R l' r', n'') where
      (l', n' ) = rlabel l n
      (r', n'') = rlabel r n'
```

Relabeling Trees con Applicativi

Osserviamo subito che il tipo di **rlabel** è

```
Tree a -> Int -> (Tree Int, Int)
                = Tree a -> ST(Tree Int)
```

Osservate che **fresh** ha tipo **ST Int**

e **pure Leaf** ha tipo **ST(Int -> Tree Int)**.

In Haskell è definito anche l'operatore **<$>**, definito da:

```
g <$> x = fmap g x = pure g <*> x
```

Vedremo che questo è il modo standard di definire un funtore.

```
 -- funzione che trasforma lo stato
fresh :: ST Int
fresh = S (\n -> (n, n+1))

alabel :: Tree a -> ST(Tree Int)
alabel (Leaf _)   =  pure Leaf <*> fresh -- Leaf <$> fresh
alabel (Node l r) =  pure Node <*> alabel l <*> alabel r
   -- Node <$> alabel l <*> alabel r
```

Relabeling Trees con Monadi

Probabilmente più semplice la versione con monadi. Che **sembra** (sottolineo **sembra**) un normale programma imperativo ricorsivo, che fa uso di una variabile globale **`fresh`**.

```
 -- funzione che trasforma lo stato
fresh :: ST Int
fresh = S (\n -> (n, n+1))

mlabel :: Tree a -> ST(Tree Int)
mlabel (Leaf _)  = do  n <- fresh
                       return (Leaf n)

mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return (Node l' r')
```

Costruire un albero da una lista

Costruiamo un albero bilanciato da una lista.

Idea: dividere la lista in 2 e costruire con una metà il sottoalbero destro e metà il sottoalbero sinistro.

La soluzione ricorsiva puramente funzionale, si trasmette gli elementi della lista tra le varie chiamate.

Ancora una volta, c'è una simulazione di uno stato.

```
 -- Notare la tecnica di calcolare la lunghezza di
 -- xs una volta sola…
build xs = fst (build' (length xs) xs)

 -- xs una volta sola …
build’:: Int -> [a] -> (Tree a, [a])
build' 1 xs = (Leaf (head xs), tail xs)
build' n xs = (Node u v, xs'') where
          (u, xs') = build' m xs
          (v, xs'')= build' (n-m) xs’
          m = n `div` 2
```

Soluzione con Monadi

Bisogna innanzitutto che vengano ridefinita State transformer e ridefinire Funtori, Monadi, Applicativi.

Domanda: ma si può generalizzare? **Esercizio**.

```
 -- Bisogna reistanziare State con [Int]
 -- ma generalizzare?
newtype ST a = S (State -> (a, State))
type State = [Int]

 -- Ricordare che usiamo uno state transformer
buildM xs = fst (app (buildM' (length xs)) xs)

buildM' 1 = S (\s->(Leaf (head s), tail s))

buildM' n  = do u <- buildM' m
                v <- buildM' (n - m)
                return (Node u v)
             where m = n `div` 2
```

Monade State Transformer

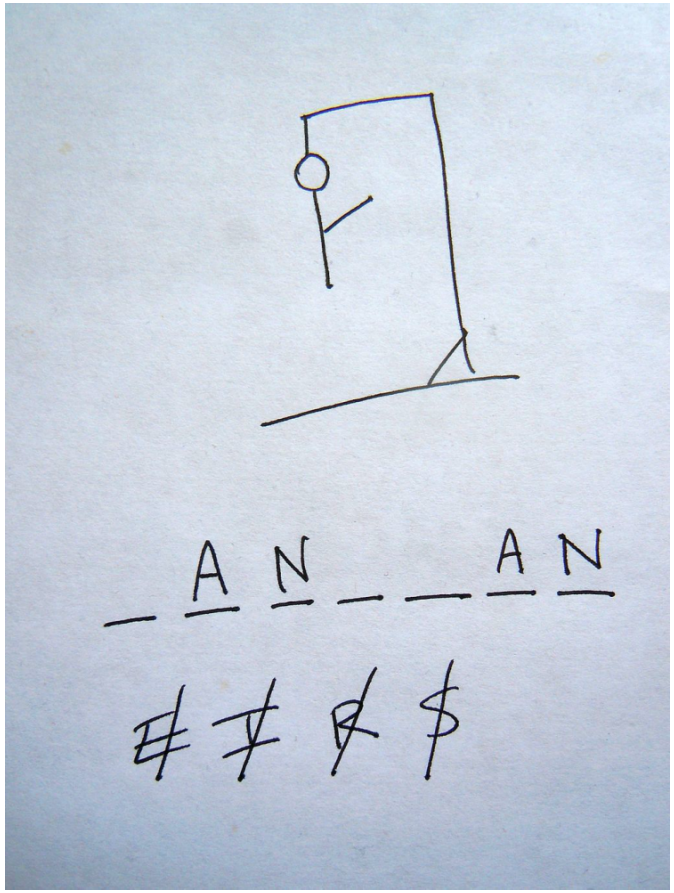
Ci sono una serie di funzioni definite sulla monade State Transformer (che qui vediamo nella sua forma generale con variabile di tipo `s`).

Senza entrare nei dettagli, dai tipi potete evincere che funzioni come **`runState`** e **`get`** permettono di **ispezionare lo stato**, mentre **`put`** e **`state`** permettono **di creare/modificare uno stato**, in accordo col fatto che lo **stato è trattato in modo prettamente funzionale**.

Vedremo che queste operazioni sono vietate nelle monadi **`IO`** (input/output) e **`StateThread`** (stato veramente mutabile).

```
type State s a = s -> (a, s)

put      :: s -> State s ()
get      :: State s s
state    :: (s -> (a, s)) -> (s -> (a, s))
runState :: State s a -> (s -> (s, a))
evalState:: State s a -> s -> a
```



Lezione 17b

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'azione 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)

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

Primitive derivate: putStrLn

Ci sono molte primitive per scrivere dati a video o su file, e leggere dati da file o da tastiera.

Qui vediamo qualche semplice esempio di funzioni predefinite.

Quando non è interessante il valore tornato (come da **`putChar`**) è possibile sequenzializzarle con l'operatore **`>>`**, invece che con il bind (**`>>=`**).

Analogamente **`return ()`** è equivalente a **`done`**.

Osservate che siamo comunque pienamente nel mondo della programmazione funzionale con **`foldr`**, **`map`** & friends.

```
 -- scrivere una stringa a video
putStrLn :: String -> IO ()
putStrLn xs = foldr (>>) done (map putChar xs)
          >> putChar '\n'
```

Primitive derivate: getLine

Sequenziare operazioni di **getChar** è più `complicato', perché in tal caso ci interessa il valore tornato (cioè il carattere letto).

La versione 2 è quella a cui siamo abituati.

```
 -- versione 1
getLine :: IO String
getLine = getChar >>= f where
             f x = if x==‘\n’ then return []
                   else getLine >>= g where
                      g xs = return (x:xs)

 -- versione 2: lambda anonimi
getLine = getChar >>= \x -> f where
             if x==‘\n’ then return []
             else getLine >>= \xs ->
                   return (x:xs)
```

Sequenze

Usando il costrutto **do** **è possibile mettere in sequenza azioni** in un'unica azione complessa, come visto in tutte le monadi.

Questo modo è decisamente **più popolare ed efficace** rispetto a **foldr >>**.

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: **non c'è redenzione** e il **contrario non si può fare**.

```
 -- sequenza di azioni:
do  v1 <- a1
    v2 <- a2
       ...
    vn <- an
    return (f v1 v2 ... vn)
```

Osservazione su input/output

La funzione **`return`** è a senso unico. Non è infatti 'sicuro' avere una funzione **`runIO :: IO a -> a`** che ispeziona un valore dall'IO e lo **trasporta nel mondo funzionale puro**.

Supponiamo ci sia: potremmo scrivere la funzione nel riquadro.

In un linguaggio funzionale, l'ordine di valutazione non è rilevante (**Teorema di Church-Rosser** o di **confluenza**) ma in questo caso lo è, perché l'Input/Output è effectful, e l'ordine con cui si legge un valore dall'**input è rilevante**.

Ciò implica anche che la **valutazione** nella **monade IO è strict**.

Ad esempio: **`do {undefined; return 3}`** valuta a **`undefined`** e non a 3, anche se `return 3` **non usa l'eventuale valore** calcolato dall'azione precedente.

```
minus :: Int
minus = x - y where
      x = runIO readInt
      y = runIO readInt
```

Review su do notation

Abbiamo usato in modo **intuitivo** la notazione "simil imperativa" per scrivere la sequenzializzazione di azioni fornita dall'operatore `>>=` detta **do notation**.

Vediamo la **traduzione standard** della do-notation in espressioni che coinvolgono bind e return.

Valgono le seguenti uguaglianze, dove **`p`** è un'azione, cioè un'espressione del tipo **`Monad m => m a`**: mentre **`stmts`** è una **sequenza non vuota di comandi**, che sono o un'**azione** oppure un **comando** nella forma **`x <- p`** (che invece **non è un'azione**)

```
do {p} = p
do {p; stmts} = p >> do {stmts}
do {x <- p; stmts} = p >>= \x -> do {stmts}
```

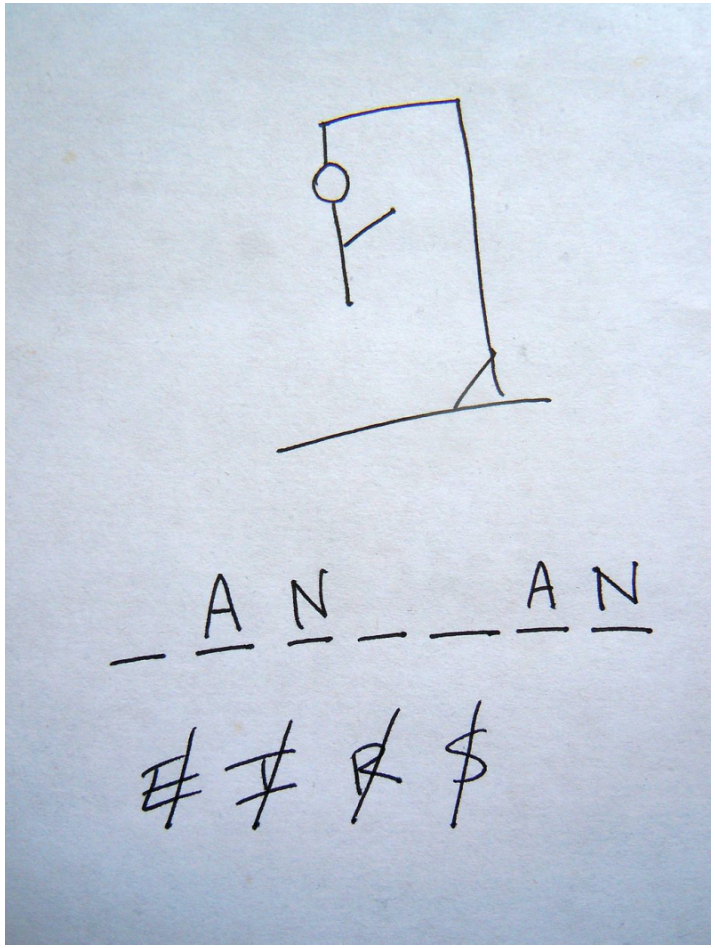
Ad esempio `do { x <- getChar }` non è valida.

Primitive derivate con do notation

```
 -- 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`
```



Lezione 17b

Hangman

(gioco dell'impiccato)

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 17

That's all Folks…

Grazie per l'attenzione…

…Domande?