Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

More on Monads and more

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **14**, 22 aprile 2021

Lezione 14a

Monade State Transformer

Il Tipo State Transformer

La monade `ST` serve a introdurre uno stato mutabile: non è possibile definire un tipo sinonimo come monade: occorre introdurre un **costruttore fittizio**.

Per semplicità, lo stato mutabile è costituito da un solo intero.

A causa del costruttore "fittizio" `S`, conviene definire un'operazione di applicazione che semplicemente applica uno state transformer rimuovendo il costruttore `S`.

```
 -- per definire una monade non basta un sinonimo
newtype ST a = S (State -> (a, State))
type State = Int

app (S st) x = st x
-- anche app (S st) = st
-- funzione che incrementa uno stato di 1
tick :: ST Int -> ST Int
tick (S st) =
   S (\s -> let (a, s') = st s
                in (a, s'+1))
```

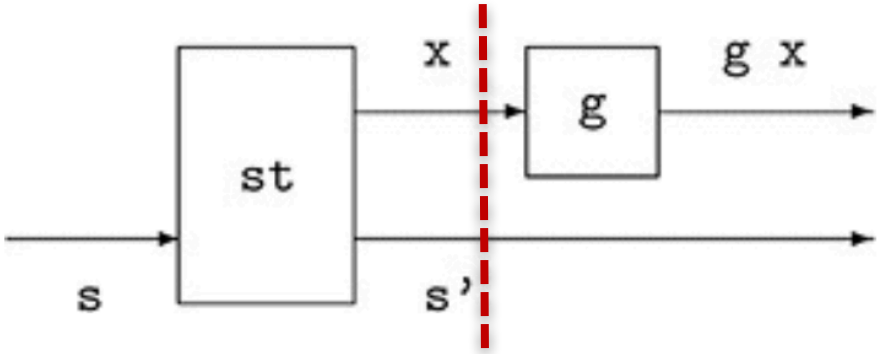
ST come Funtore Applicativo

Cominciamo col vedere il tipo `ST` degli state transformer come un Funtore…

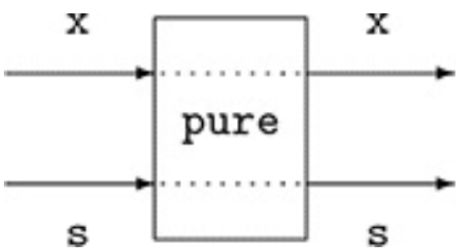
… e poi come un applicativo.

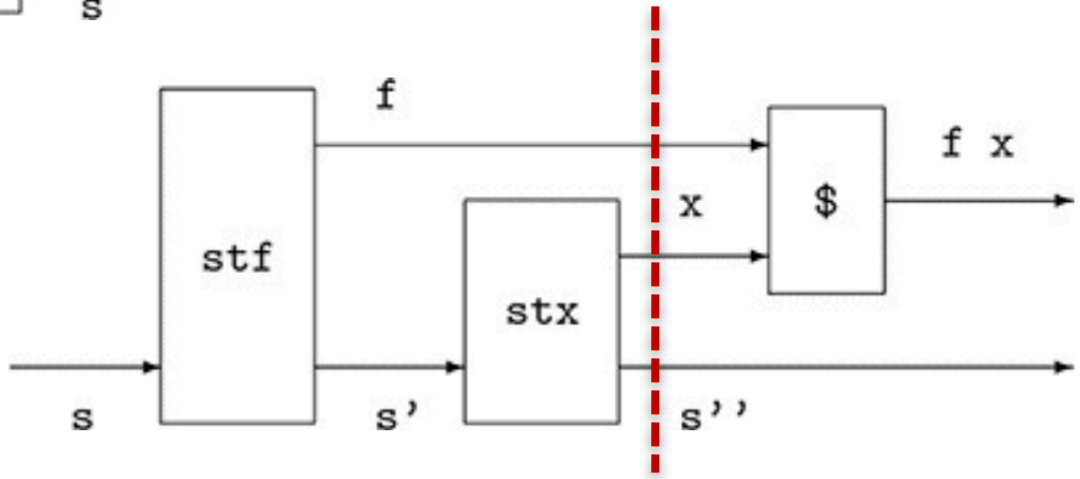
```
instance Functor ST where
   -- fmap: (a -> b) -> ST a -> ST b
 fmap f st =
    S (\s -> let (x, s') = app st s
                 in (f x, s')
       )
instance Applicative ST where
    -- pure :: a -> ST a
  pure x = S (\s -> (x, s))

    -- <*> :: ST (a -> b) -> ST a -> ST b
  stf <*> stx = S (\s ->
      let (f, s') = app stf s in
         let (x, s'') = app stx s' in
             (f x, s''))
```

Pittoricamente...



fmap





stf <*> stx

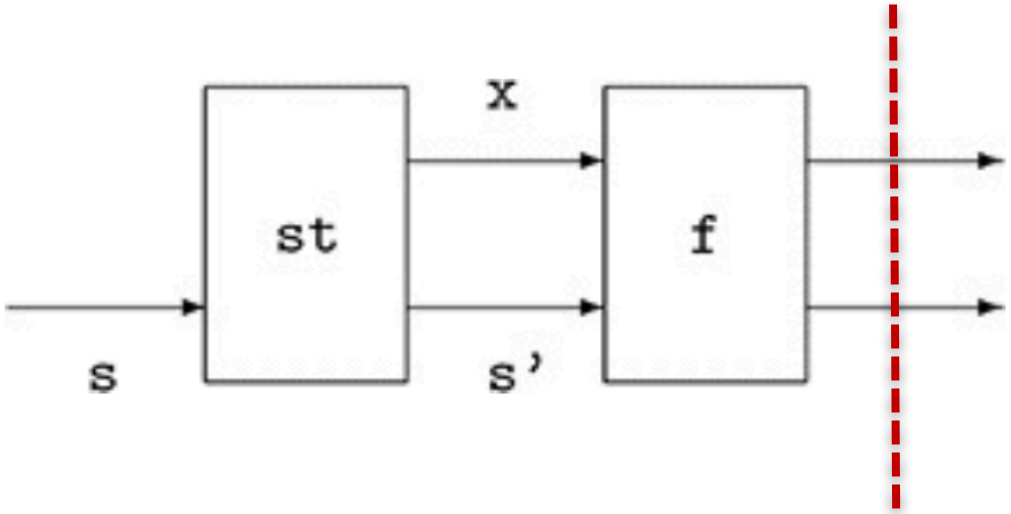
State Transformer come Monade

Infine definiamo la Monade (ricordiamo che Monad è un'istanza derivata da Applicative).

Notate sempre la necessità di astrarre su uno `State` ogni volta che si costruisce un oggetto di `ST`.

```
instance Monad ST where
   -- return: a -> ST a
  return = pure

   -- (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f =
        S (\s ->
             let (x, s') = app st s in
                app (f x) s'
          )
```

Pittoricamente…



Valutatore con lo stato 1

Vediamo ora l'esempio del valutatore che conta il numero di divisioni eseguite, usando uno stato.

Vedremo 3 versioni.

Nella prima versione, usiamo >>= e costruiamo "esplicitamente" il risultato finale… notate tuttavia che il risultato è uno "state transformer" più che uno stato e che viene applicato agli state transformer già calcolati ricorsivamente.

```
eval (Constant a) = S (\s -> (a, s))
eval (Div t u) = eval t >>= \a ->
                    eval u >>= \b ->
                       S (\s -> (a `div` b, s+1))

 -- esempio d'uso
answer = (Div (Div (Const 1972)(Const 2))(Const 23))
>app (eval answer) 5
(42, 7)
```

Valutatore con lo stato 2 e 3

Usiamo la funzione `tick`, in congiunzione con `pure`.

Infine usiamo do-notation. Osservate che `return` non è un return nel senso dei linguaggi imperativi ☺

```
 -- usando tick e pure
eval' (Constant a) = S (\s -> (a, s))
eval' (Div t u) = eval' t >>= \a ->
                      eval' u >>= \b ->
                         tick (pure a `div` b)

 -- usando do-notation e layout convention
eval'' (Constant a) = S (\s -> (a, s))
eval'' (Div t u) = do a <- eval'' t
                      b <- eval'' u
                      tick (return a `div` b)
```

layout convention!

Lezione 14b

Altri esempi

Relabeling Trees

Vediamo un altro **problema**: preso un albero, rietichettare 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 Tree a = Leaf a | Node (Tree a)(Tree a)

 -- rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n   = (Leaf n, n+1)
rlabel (Node l r) n = (Node 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)`**.

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

-- rlabel :: Tree a -> ST(Tree Int)

alabel (Leaf _)   =  pure Leaf <*> fresh
alabel (Node l r) =
        pure 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))

-- rlabel :: 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')
```

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)
```

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
```

Lezione 14c

Monad Laws revisited

Kleisli composition

Consideriamo il seguente operatore con il seguente tipo:

```
(>=>)::Monad m =>(a->m b)->(b->m c)->(a->m c)
```

che ha essenzialmente il tipo della composizione di funzioni (.):

```
(.) :: (a -> b) -> (b -> c) -> (a -> c)
```

a parte le occorrenze della monade m.

L'operatore è definito come segue:

```
(f >=> g) x = f x >>= g
```

Notate che l'ordine di `applicazione' è opposto rispetto a `(.)`.

Esiste anche l'operatore `opposto':

```
(<=<)::Monad m =>(b->m c) ->(a->m b)->(a->m c)
```

definito da:

```
(g >=> f) x = f x >>= g
```

Possiamo anche definire `(>>=)` in termini di `(>=>)`:

Kleisli composition: Monad Laws

Usando l'operatore `(>=>)` della Kleisli composition (o quello duale `(<=<)` della Kleisli composition inversa) le leggi che deve soddisfare una monadi si riducono a dire che `return` e `(>=>)` formano un **monoide**, e cioè:

- `(>=>)` è associativa.
- `return` è l'identità destra e sinistra per `(>=>)`

Generic Functions I

Come già visto coi Funtori, le Monadi (ma più in generale tutte le classi) permettono forme di programmazione generica che si basa sul fatto che:

- gli operatori hanno lo stesso nome (overloading)
- soddisfano a leggi ben precise (questo dovrebbe farvi riflettere sull'importanza di verificare le proprietà algebriche richieste).

Vediamo alcuni esempi famosi.

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y <- f x
                   ys <- mapM f xs
                   return (y:ys)

conv:: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing

> mapM conv “1234”
Just [1,2,3,4]
> mapM conv “123a"
Nothing
```

Generic Functions II

Vediamo la versione monadica `filterM` di `filter`, che generalizza `filter` in modo del tutto analogo a `mapM` rispetto a `map` (del resto `filter` è derivabile da `map`).

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [b]
filterM p []     = return []
filterM p (x:xs) = do b <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)

 -- si può facilmente ottenere il powerset
 -- occorre ricordare come sono definiti >>= e <*>
 -- sulle liste! (moltiplicano le computazioni!)
>filterM (\x->[True, False])[1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Generic Functions III

Vediamo infine la generalizzazione di `concat` a una generica monade m: l'idea è sempre quella di sciogliere un'applicazione annidata di un tipo (monade, in questo caso) dentro un'unica applicazione.

Si potrebbe al contrario definire `(>>=)` in termini di `mapM` e `join` e definire le Monad Laws in termini di `join`, `mapM` e `return`!

```
join :: Monad m => m (m a) -> m a
join mmx      = do mx <- mmx
                   x <- mx
                   return x
 -- join corrisponde a concat sulle liste…
>join [[1,2],[3,4],[5,6]]
[1,2,3,4,5,6]
> join (Just (Just 1))
Just 1
> join (Just (Nothing))
Nothing
```

Lezione 14

That's all Folks…

Grazie per l'attenzione…

…Domande?