Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Monade State Thread e variazioni

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **18**, 29 aprile 2022

Lezione 18a

Monade State Thread

Strutture dati mutabili e immutabili

Abbiamo visto che la maggioranza di algoritmi su liste, anche se resi inefficienti dalla semantica "immutabile" delle liste, **hanno generalmente la stessa complessità asintotica** (in tempo) dei corrispondenti programmi iterativi con array.

Ricordiamo che le **funzioni ricorsive** hanno **sempre** (anche nel mondo imperativo) un **overhead di spazio** dovuto agli activation record delle chiamate ricorsive.

Alcune cose sono però **intrinsecamente impossibili** in un mondo funzionale puro. Una di queste sono ad esempio le **tabelle hash** e **dizionari**, che sono **intimamente legati** agli **array mutabili** e all'**accesso diretto** in memoria.

Vediamo nel seguito che Haskell può **interfacciarsi** con un mondo di array e variabili mutabili in modo analogo a come viene trattato l'input/output e con **analoghe limitazioni**.

State Thread Monad

Questa monade è definita in **`Control.Monad.ST`** e la sua struttura è del tutto analoga a quella della monade **`State Transformer`**, vista nelle lezioni scorse.

Tuttavia, gli elementi della variabile di tipo **`s`** **non sono accessibili**, esattamente come la variabile **`World`** in **`IO a`** se non attraverso le operazioni definite.

`s` è una specie di label che identifica uno specifico thread e non si possono trasmettere valori da un thread a un altro.

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

Variabili Mutabili

Il principale valore mutabile, sono le **variabili** dei **programmi imperativi** che vengono introdotte sottoforma di **reference**.

Le variabili sono entità del tipo `STRef s a` e appartengono al thread **s** nominato nel loro tipo.

Come valore mutabile, una variabile di tipo STREF s a può:

- essere **creata** e **inizializzata** (ottenendo una **reference**)
- **letto** il valore (ottenendo il **valore** letto come risultato)
- **modificata** (con un **valore di tipo a**, **non si ottengono** risultati)

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Esempio: Fibonacci

Abbiamo visto con tupling si può ottenere un programma lineare per Fibonacci, ma che naturalmente, essendo ricorsivo è $\theta(n)$ in termini di spazio.

Vediamo un programma imperativo in spazio $\theta(1)$.

Osservate come viene simulato un ciclo, con **`repeatFor`**.

```
repeatFor :: Monad m => Int -> m a -> m ()
repeatFor n = foldr (>>) done . replicate n
replicate :: Int -> a -> [a]
> replicate 3 1
[1, 1, 1]
```

```
fibST :: Int -> ST s Integer
fibST n =
    do  a <- newSTRef 0
        b <- newStref 1
        repeatFor n
            (do x <- readSTRef a
                y <- readSTRef b
                writeSTRef a y
                writeSTRef b $! (x+y) )
        readSTRef a
```

Recuperare il valore di Fibonacci

Abbiamo alla fine un risultato di tipo `s -> (a, s)` ma **come ottenere l'intero desiderato**? Serve una funzione **`runST`** analoga alla **`runState`** vista per gli state transformer. Esiste?

Sì, ma ha uno strano tipo, che **non appartiene** ai consueti **tipi di Haskell** (à la Hindley-Milner): il $\forall s$ **non è premesso** (e quindi si omette sempre in Haskell) ma è **dentro il tipo a sinistra di** $\rightarrow$.

Un tipo così è detto **rank 2** (mentre i tipi di Haskell sono rank 1).

L'obiettivo del tipo è rendere inacessibile lo stato e quindi non ben tipate espressioni come:

```
let v = runST (newSTRef True) in runST (readSTRef v)
```

e il type-checker non può unificare **`STRef s a`** con un tipo che dipende da **`s`**, e quindi non è possibile leggere un valore da un thread e trasportarlo in un altro.

```
runST :: (forall s. ST s a) -> a
fib n = runST (fibST n)
```

Esempio: quickSort (interfaccia)

Vediamo brevemente, a chiusura di questa piccola esplorazione, gli array mutabili e come ci si interfaccia in Haskell.

Un array in Haskell ha tipo: `STArray s i e`, dove **`s`** è il thread a cui appartiene, **`i`** è il tipo degli indici, **`e`** il tipo dei valori contenuti.

Il tipo **`i`** deve essere istanza della classe **`i`** che essenzialmente contiene tipi enumerabili (con operazioni di **`succ`**) tra cui ci sono **`Int`**, **`Char`** etc. Nella definizione sotto:

- **`xa`** è un **array mutabile** con indici nell'intervallo [0, n) in cui sono stati **copiati gli elementi di xs**
- **`getElems`** **ricopia** all'indietro **su una lista**

```
qSort :: Ord a => [a] => [a]
qSort xs = runST $
             do xa <- newListArray (0,n-1) xs
                qsortST xa (0, n)
                getElems xa
             where n = length xs
```

Esempio: quickSort (codice ST)

Questo è l'algoritmo "imperativo" molto simile a quello che siamo abituati a vedere.

`(a, b)` è l'intervallo di indici su cui ordinare, mentre al solito **`partiziona`** restituisce il punto in cui suddividere l'array.

Quando **`a==b`**, l'intervallo da ordinare è vuoto e quindi la computazione è finita.

Rimane da vedere **`partiziona`**.

```
qsortST :: Ord a => STArray s Int a ->
              (Int, Int) -> ST s ()
qsortST xa (a, b)
 | a == b    = return ()
 | otherwise = do   m <- partition xa (a, b)
                    qsortST xa (a, m)
                    qsortST xa (m+1, b)
```

Esempio: quickSort (partiziona)

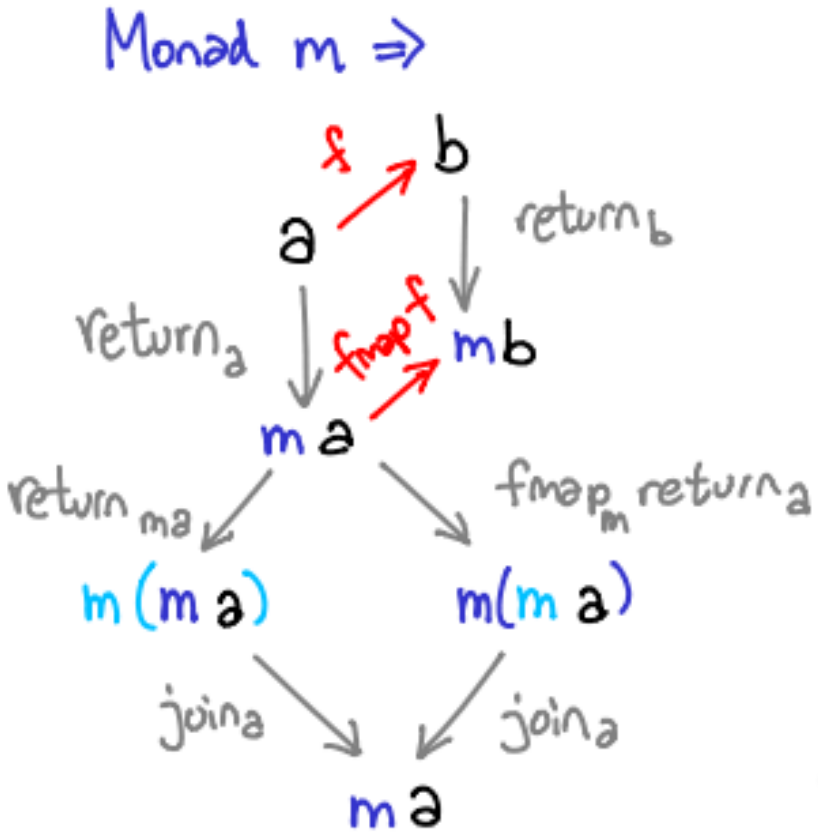
Vediamo alcune primitive come **`readArray xa a`** che è equivalente a prendere il valore in posizione **`a`**.

`loop` è sostanzialmente una funzione ricorsiva che mima un ciclo di tipo **`repeat-until`** (o **`do-while`**).

`swap` invece assomiglia all'assegnamento parallelo.

```
swap :: STArray s Int a ->
        Int -> Int -> ST s ()
swap xa i j = do v <- readArray xa i
                 w <- readArray xa j
                 writeArray xa i w
                 writeArray xa j v
```

```
partition xa (a, b) =
 do x <- readArray xa a
    let loop (j, k) =
        if j==k then do swap xa a (k-1)
                        return (k-1)
                 else do y <- readArray xa j
                         if y < x then loop (j+1, k)
                         else do  swap xa j (k-1)
                                  loop (j, k-1)
    in loop (a+1, b)
```



Lezione 18b

Altre definizioni di Monade

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 parte le occorrenze della monade **`m`**:

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

L'operatore è definito come segue (tipica definizione di un operatore **diretta dai tipi**):

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

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

Si può anche definire l'**operatore `opposto'**:

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

ovviamente rovesciando il ruolo di **`g`** ed **`f`**:

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

La cosa interessante è che possiamo anche **definire `(>>=)` in termini di `(>=>)`**.

Kleisli composition: Monad Laws

Ricordiamo i tipi:

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

Nel tipo di `(>=>)` abbiamo 3 tipi, il che fa pensare che sia effettivamente più generale. Infatti, se istanziate **`a`** con **`m b`**, un tipo "particolare" di `(>=>)` è:

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

che suggerisce la definizione di `(>>=)` con `(>=>)`:

```
(p >>= f) = (id >=> f) p
```

oppure, più brevemente:

```
(>>=) = flip (id >=>)
```

La cosa interessante è che definendo le monadi 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**, cioè `(>=>)` è **associativa** con **`return`** è **identità destra e sinistra**.

Rovesciando le cose

È possibile fare il percorso opposto nel senso che è è possibile definire **fmap** in modo naturale da **<*>**:

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

Analogamente, è possibile definire **<*>** in modo naturale da **>>=**:

```
f <*> x = f >>= \g -> x >>= \y -> return g y
```

Questo riprova che **Monad** è un **tipo più specifico** (quindi un **sottotipo**) di **Applicative** e **Applicative** **sottotipo** di **Functor**.

Chiudiamo con una formulazione alternativa delle monadi. L'idea è che le monadi contengano una funzione:

```
join :: Monad m => m (m a) -> m a
```

che è **interdefinibile** con **(>>=)**:

```
join x = x >>= id
p >>= f = join ((fmap f) p)
```

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.

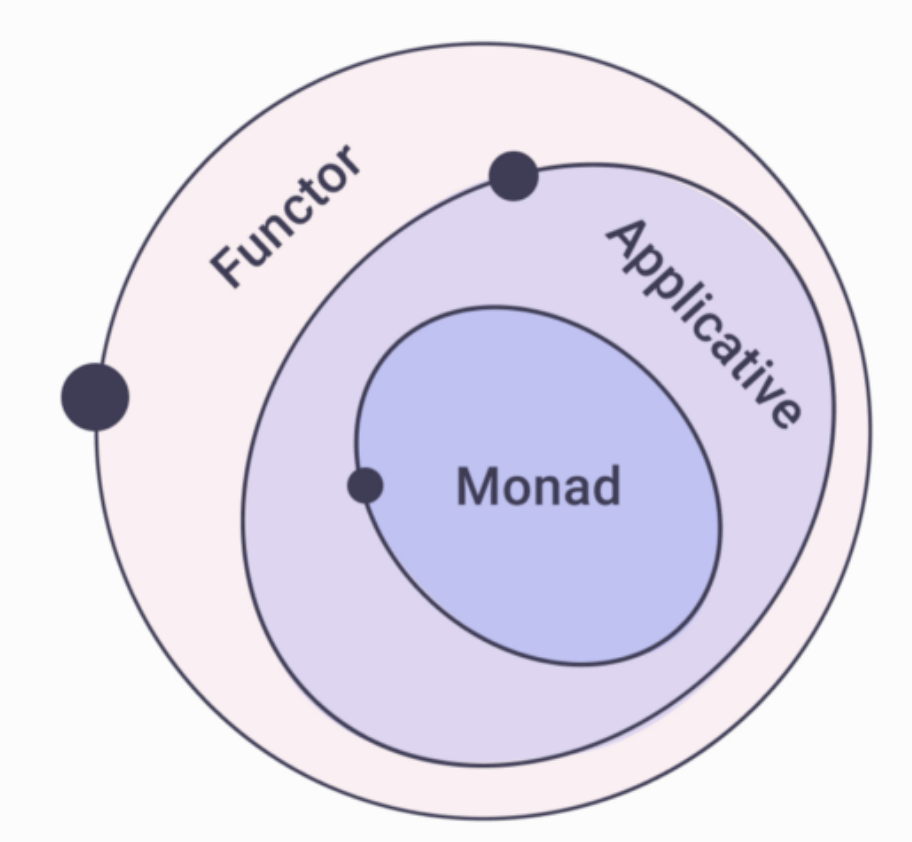
Vediamo tra poco come definire **(>>=)** in termini di **mapM** e **join** e definire le Monad Laws in termini di **join**, **mapM** e **return**.

```
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 18c

Ancora Astrazioni: Monoidi, Foldable & Traversable

Classi e pattern di computazione

Per finire, 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 18

That's all Folks…

Grazie per l'attenzione…

…Domande?