Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Shall I be pure or impure?

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **16**, 22 aprile 2022

Lezione 16a

Shall I be pure or impure?

Pure/Impure functional programming

Purtroppo, alcuni importanti aspetti della programmazione **non trovano** una efficace rappresentazione nel mondo funzionale:

- strutture dati **mutabili**
- **input/output**

mentre altri aspetti **potrebbero beneficiare** (in efficienza e/o comodità) dall'avere una programmazione con side-effects:

- **eccezioni**
- **tracciamento** e **sequenziamento** delle computazioni

Alcuni linguaggi funzionali (ad esempio ML o Scheme) esplicitamente introducono costrutti non funzionali, come **reference** (= puntatori), **eccezioni**, etc.

Haskell cerca un'integrazione basata su precisi concetti matematici. Abbiamo visto una prima forma negli **applicativi**, ora vedremo le **monadi**.

L'Essenza di FP

Le funzioni sono moduli che si possono **facilmente comporre** e la loro semantica:

- **non dipendono** dall'**ordine di valutazione** (**laziness**)
- **ragionamento algebrico**: si possono sostituire **uguali per uguali**

Lazy evaluation/astrazione/curryficazione permettono alle funzioni di comporre molto bene: d'altro canto a volte è necessario introdurre **complicazioni** per far **fluire i dati** da **dove** vengono **prodotti** a dove vengono **utilizzati**:

- tupling
- uso di parametri

che a volte appesantiscono i programmi, di solito per **guadagnare efficienza**.

Esempio: un piccolo valutatore

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Faremo un piccolo esempio e vedremo tre variazioni (essenzialmente **banali** nel mondo **imperativo**) che richiedono una profonda ristrutturazione del codice.

Vedremo poi come in Haskell si evitino queste difficoltà.

Cominciamo con la versione base: un piccolo valutatore di espressioni che contengono solo la divisione.

Vedremo cosa occorre fare per:

- aggiungere **eccezioni**
- stampare un **log**
- **contare** le operazioni

```
 -- prima approssimazione:
data Term = Const int | Div Term Term

eval :: Term -> Int
eval (Const a) = a
eval (Div t u) = eval t `div` eval u
```

Variazione 1: Trattamento Eccezioni

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Siccome la divisione **non è totale**, la valutazione di un elemento del tipo `Term` può condurre a un errore.

Prendiamo le due espressioni:

```
answer = (Div (Div (Const 1972)(Const 2))(Const 23))
error = (Div (Const 1) (Const 0))
```

la prima dovrebbe essere valutata a 42, mentre la seconda dovrebbe dare un errore di **divisione per zero**.

Vediamo come modificare il valutatore per trattare le eccezioni.

```
 -- prima approssimazione:propagare Nothing funz.
eval :: Term -> Maybe Int

eval (Const a) = Just a
eval (Div t u) = case eval t of
   Nothing -> Nothing
   Just a -> case eval u of
      Nothing -> Nothing
      Just b -> if b==0 then Nothing
                        else Just (a `div` b)
```

Occorre ogni volta trattare il caso di valutazione fallita

Variazione 2: Log in Output

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Immaginiamo di voler stampare tutte le operazioni eseguite producendo un log di quanto accaduto.

Anche qui usiamo la tecnica di accoppiare i log con i valori di ritorno della funzione **`eval`** e dobbiamo quindi modificare il prototipo di **`eval`**.

Usiamo una funzione **`line`** per produrre l'output.

```
 -- prima approssimazione:
type M a = (Output, a) -- a è il risultato
type Output = String -- Output è l'effetto computaz.

eval :: Term -> M Int

eval (Const a) = (line (Const a) a, a)
eval (Div t u) = let (x, a) = eval t
  in let (y, b) = eval u
    in (x++y++line (Div t u)(a `div` b), a `div` b)

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

Variazione 3: Stato Mutabile

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Immaginiamo di voler contare il numero di divisioni eseguite: funzionalmente questo è possibile **aggiungendo informazione** nei **valori di ritorno** e trasmettere il valore **riaggiornandolo opportunamente a ogni chiamata ricorsiva**.

È possibile risolvere questo problema senza propagare il numero di operazioni sui parametri ma solo nei risultati?

È stato anche necessario modificare anche i prototipo di `eval`.

```
 -- prima approssimazione: pairing dei risultati
newtype M a = (a, State)
type State = Int

eval :: Term -> M Int

eval (Const a) x = (a, x)
eval (Div t u) x =
   let (a, y) = eval t x
      in let (b, z) = eval u y
         in (a `div` b, z+1)
```

Osservazioni

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Cos'hanno in comune i tre esempi precedenti?

Occorre estendere **`eval`** cambiandone la **segnatura** da **`Term -> Int`** a **`Term -> M Int`**: questo tuttavia obbliga nelle chiamate ricorsive a "**spacchettare**" i risultati dal tipo **`M Int`** e poi re-impacchettarli. Si tratta di un lavoro **noioso e ripetitivo** che offusca la semplicità del valutatore ricorsivo.

In generale, avendo una funzione **`f: a -> M b`** e un valore di tipo **`M a`**, voglio costruire un nuovo valore di tipo **`M b`**, "spacchettando" il valore dentro **`M a`** e passandolo come parametro a **`f`**.

Serve inoltre una funzione per **impacchettare** un valore di tipo **`a`** dentro il tipo **`M a`** (questo ricorda **`pure`** negli applicativi!).

Lezione 13c

Monadi, do notation e Monad laws

Generalizzando: Monadi

Una funzione `f : a -> b` viene rimpiazzata da una funzione di tipo `f' : a -> M b`, dove `M` cattura effetti computazionali di `f`.

L'operazione `m >>= f` **sequenzializza la computazione**: prima viene valutato `m : M a` e poi il risultato viene passato (spacchettato da `M`) come parametro a `f': a -> M b`.

Esiste anche `m >> f` utile se il valore `m` non è significativo per `f`.

La funzione **`return`** è inclusa per ragioni storiche: **`pure`** sarebbe sufficiente.

```
 -- dichiarazione della classe Monad
class Applicative m => Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   (>>)   :: m a -> m b -> m b
   return :: a -> m a

   return = pure
```

Reminder su Applicativi

A prima vista potrebbe sembrare che l'operatore **<*>** sia quello che serve, per cui una volta definita (per esempio) **safeDiv**:

```
eval (Term t u) = pure safeDiv <*> eval t <*> eval u
```

Tuttavia osservate che i **tipi non tornano**: **safeDiv** ha tipo `Int -> Int -> Maybe Int` mentre **pure div** avrebbe tipo `Maybe (Int -> Int -> Int)` e **pure safeDiv** avrebbe tipo `Maybe (Int -> Int -> Maybe Int)`.

```
 -- dichiarazione di Applicative
class Functor t => Applicative t where
   pure  :: a -> t a
   (<*>) :: t (a -> b) -> t a -> t b

 -- Definizione safeDiv
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)
```

Monad Laws

Vediamo quali sono le **equazioni** che dovrebbero essere soddisfatte da `return` e `>>=`.

1. `return` è un'**identità destra**:

```
p >>= return = p
```

che può essere scritta anche in do-notation:

```
do {x<-p; return x} = do {p}
```

2. `return` è anche una **specie di identità sinistra**:

```
return e >>= f = f e
```

che può essere scritta anche in do-notation:

```
do {x<-return e; f x} = do {f e}
```

3. `>>=` è un **compositore "associativo"**:

```
((p >>= f)>>=g) = p >>= (\x-> (f x >>= g))
```

che può essere scritta anche in do-notation:

```
do {y <- do {x <- p; f x}; g y}
  = do {x <- p; do {y <- f x; g y}}
  = do {x <- p; y <- f x; g y}
```

Monade Identità

Il valutatore base è ottenuta usando la monade identità.

La **monade Identità** è simile **all'elemento neutro in un'algebra** e viene ad esempio applicata ogni qualvolta (come in questo caso) si dà una definizione generale, parametrizzata rispetto a una monade, ma non è necessario usarla.

```
instance Identity Monad where
  --(>>=) :: Identity a -> (a -> Identity b)
          -> Identity b
   pure a  = a
   a >>= f = f a
```

Liste come Monade

Anche le liste sono una monade.
>>= è analogo a **List comprehension**.

```
 -- Liste come monade
instance [] Monad where
   xs >>= f  = concat (map f xs)
   -- xs >>= f = [y | x <- xs, y <- f x]

   return x = [x]

 -- ad esempio…
> [1,2,3] >>= (\x->([4,5] >>= (\y->[(x,y)])))
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

 -- oppure …
do {x <- [1,2,3]; y <- [4,5]; return (x,y)}
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
 -- infatti:
pairs xs ys = [(x,y) | x<-xs, y<-ys]
```

Lezione 16c

Monadi Maybe, Log & State Transformer (Esempi Completi)

Valutatore Generalizzato

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Scriviamo il codice del valutatore generalizzato.

In un certo senso, `m >>= f`, può essere visto come un'espressione della forma `let a = m in f`, con `f` che dipende da `a`: la computazione di `m` precede quella di `f`.

Ogni variante del valutatore sarà ottenuta semplicemente (o quasi) **cambiando la definizione della monade** `m` (e quindi di `>>=`).

```
eval :: Monad m => Term -> m Int

eval (Const a) = pure a
eval (Div t u) = eval t >>= \a ->
                 eval u >>= \b ->
                 pure (a `div` b)
```

Monade Eccezioni

[Philip Wadler, *Monads for Functional Programming*, AFP, 1996]

Nella monade **Maybe**, `Nothing` viene propagato da **`>>=`**. In caso di un valore `(Just x)`, semplicemente si accede all'elemento contenuto e lo si passa come parametro alla funzione che segue.

Per ottenere il valutatore, occorre "**generare i casi base**" delle eccezioni, cioè quando si cerca di fare una divisione per zero.

Occorre di conseguenza sostituire `pure div` con **`safediv`**.

```
 -- Maybe come monade
instance Maybe Monad where
-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
   Nothing  >>= _ = Nothing
   (Just x) >>= f = f x

eval (Const a) = Just a
eval (Div t u) = eval t >>= \a ->
                 eval u >>= \b ->
                 safeDiv a b
```

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 classe 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))
```

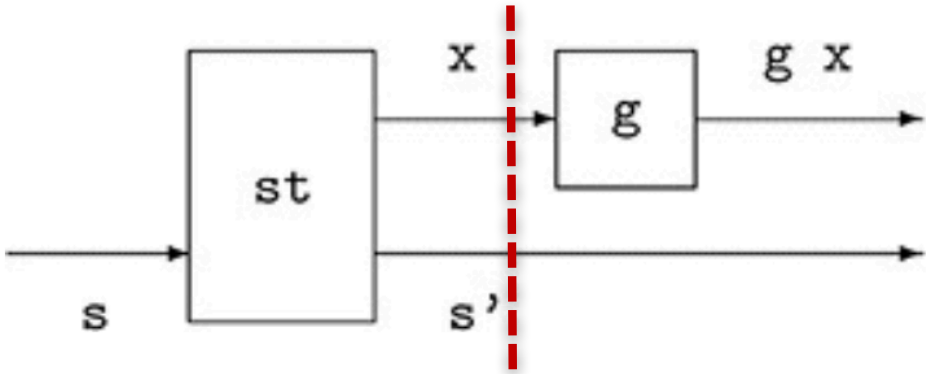
ST come Funtore Applicativo

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

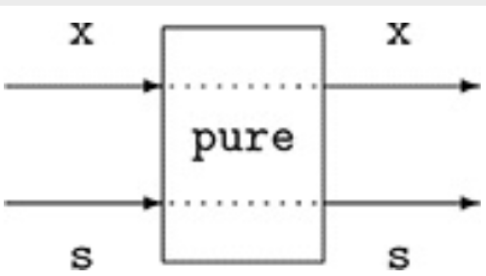
… e poi come un applicativo.

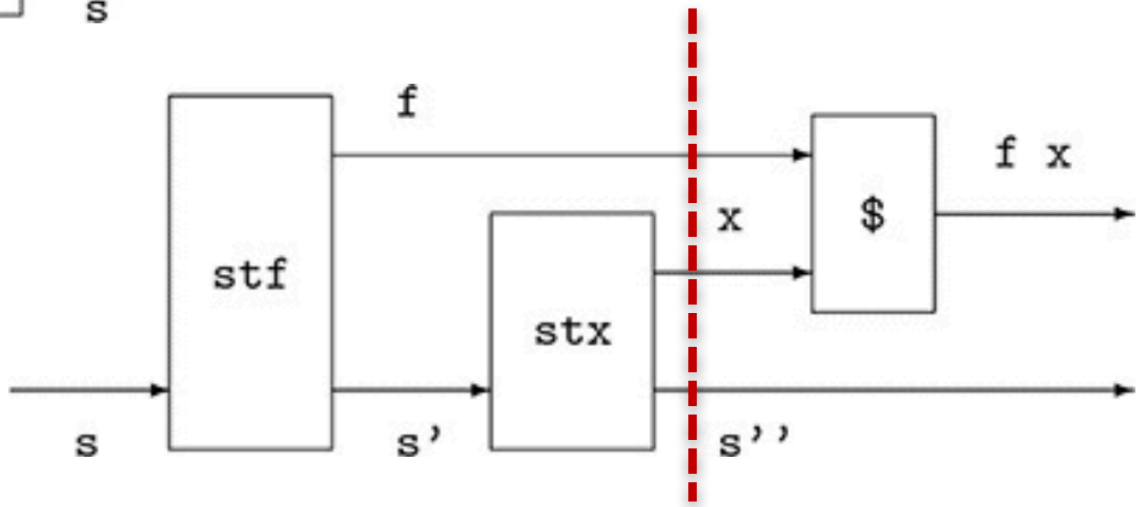
```
instance Functor ST where
   -- fmap: (a -> b) -> ST a -> ST b
 fmap f st =
    S (\s -> let (x, s') = app st s
                 in (f x, s')
       )
instance Applicative ST where
    -- pure :: a -> ST a
  pure x = S (\s -> (x, s))

    -- <*> :: 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

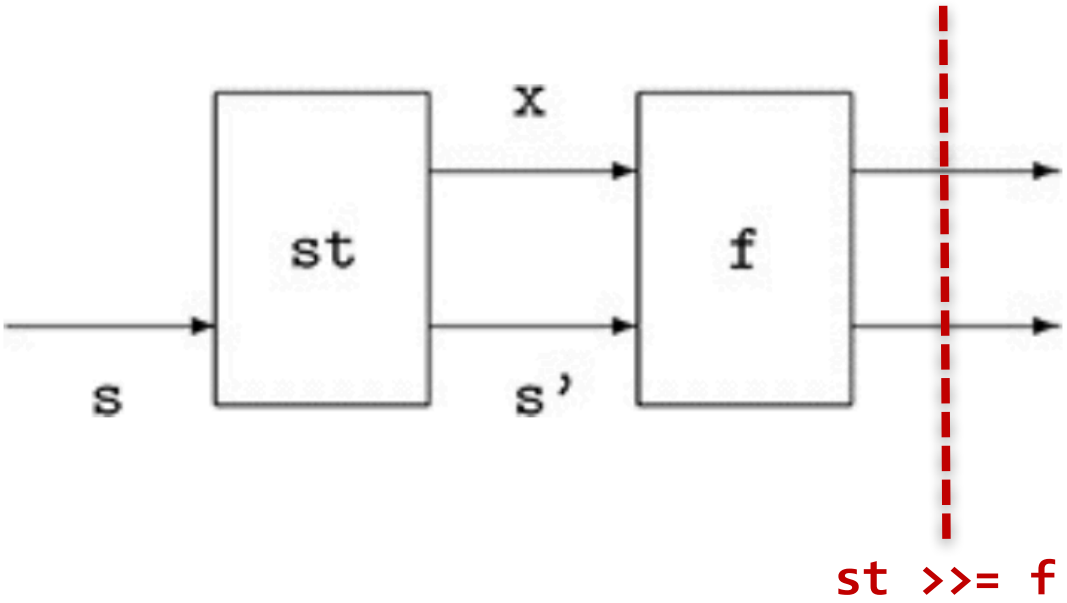
State Transformer come Monade

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 16

That's all Folks…

Grazie per l'attenzione…

…Domande?