Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Effectful Programming in Haskell: Monadi

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **13**, 19 aprile 2021

Lezione 13a

Shall I be pure or impure?

Pure/Impure functional programming

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

- **eccezioni**
- **input/output**

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

- strutture dati **mutabili**
- **tracciamento** e **sequenziamento** lle 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 dipende 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**:

- accoppiamenti
- uso di parametri

che a volte appesantiscono i programmi.

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 Haskell permette di evitare 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)
```

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)
type Output = String

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 = State -> (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!).

Generalizzando: Monadi

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

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`** **sequenzalizza 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

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

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

 -- Definizione di pure e <*> in Maybe
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)
```

Lezione 13b

Valutatore con Monadi

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 >>=).

```
 -- prima approssimazione:
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 Identità

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

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
```

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 paramentro 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
```

Monade Output

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

La monade State (per questo problema, poi vedremo un caso più generale) semplicemente propaga e passa i parametri.

```
 -- M = (a, Output)
instance M Monad where
  --(>>=) :: Output a -> (a -> Output b) -> Output b
   m >>= k = let (a, x) = m in
                 let (b, y) = k a in
                         (b, x ++ y)
   pure a  = (a, "")

 -- modifica al valutatore:
eval (Const a) = (a, line (Const a) a)
eval (Div t u) = eval t >>= \a ->
                 eval u >>= \b ->
                 \x->(a `div` b,x++line (Div t u))
```

Lezione 13c

Do notation e Monad laws

Do notation

Haskell permette di usare una notazione "simil imperativa" per scrivere la sequenzializzazione di azioni fornita dall'operatore `>>=` delle monadi, detta **do notation**.

Riscriviamo l'esempio del valutatore con eccezioni.

In generale, valgono le seguenti uguaglianze:

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

dove `stmts` è una sequenza non vuota di staments, che sono o azioni o statements nella forma `x<-p` (che non è un'azione!)

```
eval (Div t u) = do { a <- eval t;
                      b <- eval u;
                      safeDiv a b
                    }
```

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}
```

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 13

That's all Folks…

Grazie per l'attenzione…

…Domande?