Llenguatges de Programació

Sessió 6: més mònades

Jordi Petit, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Contingut

Instanciació de llistes com a mònades

Recordatori de la classe `Monad`:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Les llistes d'`a` (tipus `[a]`) instancien les mònades d'aquesta forma:

```
instance Monad [ ] where
    xs >>= f  =  concat (map f xs)          -- = concatMap f xs
    return x  =  [x]
```

Exemple: `f x = [x, 2*x]` i `xs = [1, 2, 3]`.

```
xs >>= f  =
    =  concat (map f xs)                              -- definició de >>=
    =  concat (map (\x -> [x, 2*x]) [1, 2, 3])        -- definició de f i xs
    =  concat ([[1, 2], [2, 4], [3, 6]])              -- aplicació de map
    =  [1, 2, 2, 4, 3, 6]                             -- aplicació de concat
```

Exemple: salts de cavall

Tenim un cavall que es mou en un tauler d'escacs 8×8.

Les posicions són parells d'enters entre 1 i 8:

```
type Pos = (Int, Int)

dins :: Pos -> Bool
dins (x, y) = dins' x && dins' y    where dins' i = 1 <= i && i <= 8
```

Donada una posició, `moviments` retorna la llista de posicions a on pot anar un cavall:

```
moviments :: Pos -> [Pos]
moviments (f, c) =
  filter dins [ (f + 2, c - 1), (f + 2, c + 1), (f - 2, c - 1), (f - 2, c + 1),
                (f + 1, c - 2), (f + 1, c + 2), (f - 1, c - 2), (f - 1, c + 2) ]
```

La funció `potAnar3`, donada una posició inicial `p` dins del tauler i una posició final `q`, diu si un cavall pot anar de `p` a `q` en (exactament) tres salts:

```
potAnar3 :: Pos -> Pos -> Bool
potAnar3 p q = q `elem` destins
    where destins = (moviments p >>= moviments >>= moviments)
```

Llistes per comprensió

La notació de les llistes per comprensió és **sucre sintàctic** sobre la notació **do**.

L'expressió

```
[f x y | x <- xs, y <- ys]
```

és equivalent a

```
do
    x <- xs
    y <- ys
    return (f x y)
```

Llistes per comprensió

Exemple:

```
[(x,y) | x <- "abc", y <- [1, 2, 3]]
```

⟺

```
do
    x <- "abc"
    y <- [1, 2, 3]
    return (x,y)
```

⟺

```
"abc" >>= \x ->
    [1, 2, 3] >>= \y ->
        return (x,y)
```

Filtres

Per poder disposar de filtres en les llistes de comprensió, les mònades de Haskell també tenen una operació `guard`:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
    guard  :: Bool -> m ()
```

Els filtres en les llistes de comprensió són nou sucre sintàctic sobre `guard`:

```
[f x y | x <- xs, p x, y <- ys]
```

és equivalent a

```
do
    x <- xs
    guard (p x)
    y <- ys
    return (f x y)
```

Nota: Aquesta explicació és una simplificació de la realitat.

Filtres

Exemple: Tripletes pitagòriques

```
[(x, y, z) | x <- [1..n], y <- [x..n], z <- [y..n], x*x + y*y == z*z]
```

⟺

```
do
    x <- [1..n]
    y <- [x..n]
    z <- [y..n]
    guard (x*x + y*y == z*z)
    return (x,y,z)
```

⟺

```
[1..n] >>= \x ->
    [x..n] >>= \y ->
        [y..n] >>= \z ->
            guard (x*x + y*y == z*z) >>
                return (x,y,z)
```

(feu `import Control.Monad` per tenir `guard`)

Filtres

Implementació de `guard`:

```
guard True   =  return ()
guard False  =  []
```

o

```
guard b   =  if b then [()] else []
```

Exemple: Comprovar

```
[x | x <- [1,2,3], odd x]  👉  [1, 3]
```

Filtres

```
[x | x <- [1,2,3], odd x]
```

👉
```
do
    x <- [1,2,3]
    guard (odd x)
    return x
```

👉
```
[1,2,3] >>= (\x -> guard (odd x) >> return x)
```

👉
```
[1,2,3] >>= (\x -> guard (odd x) >>= \_ -> return x)
```

👉
```
concat (map (\x -> guard (odd x) >>= \_ -> return x) [1,2,3])
```

👉
```
concat (map (\x -> guard (odd x) >>= \_ -> [x]) [1,2,3])
```

👉
```
concat [
    (\x -> guard (odd x) >>= \_ -> [x])  1,
    (\x -> guard (odd x) >>= \_ -> [x])  2,
    (\x -> guard (odd x) >>= \_ -> [x])  3
]
```

👉
```
concat [
    guard (odd 1) >>= \_ -> [1],
    guard (odd 2) >>= \_ -> [2],
    guard (odd 3) >>= \_ -> [3]
]
```

Filtres

👉
```
concat [
    guard True  >>= \_ -> [1],
    guard False >>= \_ -> [2],
    guard True  >>= \_ -> [3]
]
```

👉
```
concat [
    [()] >>= \_ -> [1],
    [  ] >>= \_ -> [2],
    [()] >>= \_ -> [3]
]
```

👉
```
concat [
    concat (map (\_ -> [1]) [()]),
    concat (map (\_ -> [2]) [  ]),
    concat (map (\_ -> [3]) [()])
]
```

👉
```
concat [ concat [[1]],  concat [   ],  concat [[3]] ]
```

👉
```
concat [ [1], [ ], [3] ]
```

👉
```
[1, 3]
```

😅

`Lets` en llistes de comprensió

Per desensucrar els `lets` dins de llistes per comprensió, només cal posar-los dins la notació `do`, que ja té `lets`.

Exemple:

```
[(x, y, z) |
    x <- [1..n],
    y <- [x..n],
    let z = isqrt (x*x + y*y),
    z <= n,
    x*x + y*y == z*z
]
```

Equival a

```
do
    x <- [1..n]
    y <- [x..n]
    let z = isqrt (x*x + y*y)
    guard (z <= n)
    guard (x*x + y*y == z*z)
    return (x,y,z)
```

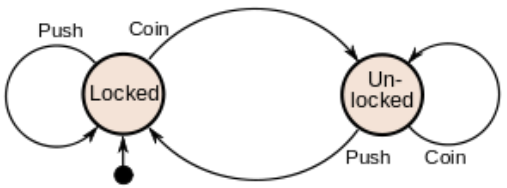
Sumari

- Les llistes són mònades (`>>=` és `concatMap`).
- Les llistes per comprensió són sucre sintàctic.
- Cal ampliar les mònades amb `guard` per implementar filtres.

Contingut

- Mònade llista
- Mònade estat
- Combinació de mònades

Exemple: torn d'entrada en C++




```
FSM torn;
torn.getEstat()     👉  Locked
torn.coinTrigger()  👉  Thank
torn.getEstat()     👉  Unlocked
torn.pushTrigger()  👉  Open
torn.getEstat()     👉  Locked
```

```
class FSM {
    private:
        string estat;
    public:
        FSM() {
            estat = "Locked";
        }
        string getEstat() {
            return estat;
        }
        string coinTrigger() {
            estat = "Unlocked";
            return "Thank";
        }
        string pushTrigger() {
            if (estat == "Locked")
                return "Error";
            estat = "Locked";
            return "Open";
        }
};
```

Fonts: wikipedia i wikibooks

Exemple: torn d'entrada en Haskell

Codi

```haskell
data TurnstileState = Locked | Unlocked
  deriving (Eq, Show)

data TurnstileOutput = Thank | Open | Error
  deriving (Eq, Show)

coinTrigger :: TurnstileState -> (TurnstileOutput, TurnstileState)
coinTrigger _ = (Thank, Unlocked)

pushTrigger :: TurnstileState -> (TurnstileOutput, TurnstileState)
pushTrigger Locked   = (Error , Locked)
pushTrigger Unlocked = (Open, Locked)
```

Ús

```haskell
pushTrigger Locked    👉  (Error,Locked)
coinTrigger Locked    👉  (Thank,Unlocked)
pushTrigger Unlocked  👉  (Open,Locked)
coinTrigger Unlocked  👉  (Thank,Unlocked)
```

* font: wikibooks

Exemple: torn d'entrada

En seqüència

```
triggers :: TurnstileState -> ([TurnstileOutput], TurnstileState)
triggers s0 =
  let (a1, s1) = coinTrigger s0
      (a2, s2) = pushTrigger s1
      (a3, s3) = pushTrigger s2
  in ([a1, a2, a3], s3)

triggers Locked  👉  ([Thank,Open,Error],Locked)
```

Aniria bé tenir l'operador `>>=` per encadenar els *triggers*.

* font: wikibooks

Mònada State

La mònada *State* encapsula una estructura de dades en forma d'estat.

El tipus *State*

- representa funcions d'estat a tuples (valor, estat):

```
s -> (a, s)
```

- té funcions per aplicar funcions d'estat:

```
runState :: State s a -> s -> (a, s)
runState coinTrigger Locked        -- aplica una funció d'estat
```

- és instància de `monad`
- té l'operador `>>=` per enllaçar funcions d'estat

Exemple: torn d'entrada I

Data

```haskell
import Control.Monad
import Control.Monad.State

data TurnstileState = Locked | Unlocked
  deriving (Eq, Show)

data TurnstileOutput = Thank | Open | Error
  deriving (Eq, Show)
```

Funció `coin`

```haskell
coinTrigger :: State TurnstileState TurnstileOutput
coinTrigger = do
    put Unlocked   -- nou estat
    return Thank   -- valor de tornada
```

Ús amb `runState`

```haskell
runState coinTrigger Locked    👉  (Thank,Unlocked)
runState coinTrigger Unlocked  👉  (Thank,Unlocked)
```

Exemple: torn d'entrada II

Funció `push`

```haskell
pushTrigger :: State TurnstileState TurnstileOutput
pushTrigger = do
    s <- get           -- obté l'estat
    if s == Locked
    then return Error  -- valor de tornada
    else do
        put Locked     -- nou estat
        return Open    -- valor de tornada
```

Ús amb `runState`

```haskell
runState pushTrigger Locked   👉  (Error,Locked)
runState pushTrigger Unlocked 👉  (Open,Locked)
```

Enllaç i seqüència

Enllaç de *triggers*

```
runState (coinTrigger >>= \x -> pushTrigger >>= \y -> return [x,y]) Locked

👉   ([Thank,Open],Locked)
```

```
runState (sequence [coinTrigger, pushTrigger]) Locked

👉   ([Thank,Open],Locked)
```

Exemple més llarg

```
runState (sequence
  [coinTrigger, pushTrigger, pushTrigger, coinTrigger, pushTrigger])
  Locked

👉   ([Thank,Open,Error,Thank,Open],Locked)
```

Funcions de la mònada State

Funcions de treball

- `get`: obté l'estat
  `get :: MonadState s m => m s`
- `put`: modifica l'estat
  `put :: MonadState s m => s -> m ()`
- `return`: retorna el valor
  `return :: Monad m => a -> m a`

Funcions de crida

- `runState`: crida a una funció i torna el valor i el nou estat
  `runState coinTrigger Locked 👉 (Thank,Unlocked)`
- `evalState`: crida a una funció i torna el valor
  `evalState coinTrigger Locked 👉 Thank`
- `execState`: crida a una funció i torna el nou estat
  `execState coinTrigger Locked 👉 Unlocked`

Contingut

- Mònade llista
- Mònade estat
- Combinació de mònades

Petit LP I

Tipus

```
data Expr = Val Int
    | Var String
    | Add Expr Expr
    | Sub Expr Expr
    | Mul Expr Expr
    | Div Expr Expr
    deriving (Show)

data Accio = Ass String Expr
    | View Expr
    deriving (Show)
```

Exemple

```
View (Val 2)                    👉  Just 2
Ass "a" (Val 3)                 👉  Just 3
View (Add (Var "a") (Val 2))    👉  Just 5
👉
Estat final: [("a",Just 3)]
```

Petit LP II

Requeriments

```
import Control.Monad
import Control.Monad.State
```

Estat: taula de símbols

- tipus:

```
type TS = [(String, Maybe Int)]
```

- nou símbol:

```
addEntry :: Eq a => a -> b -> [(a, b)] -> [(a, b)]
addEntry k v l = (k, v) : filter (\p -> fst p /= k) l
```

- consulta símbol:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Petit LP III

Expressions

```
eval :: Expr -> State TS (Maybe Int)
eval (Val x) = return (Just x)
eval (Var x) = do
    ts <- get                          -- recuperem la taula de símbols
    return (join $ lookup x ts)        -- join $ Just $ Just 2 👉 Just 2
eval (Add x y) = evalf (+) x y
eval (Sub x y) = evalf (-) x y
eval (Mul x y) = evalf (*) x y
eval (Div x y) = do
    ts <- get                          -- recuperem la taula de símbols
    let vlx = evalState (eval x) ts    -- només volem el valor, l'estat no canvia
    let vly = evalState (eval y) ts    -- idem
    return $ evalDiv vlx vly
```

Petit LP IV

Funcions auxiliars d'`eval`

```haskell
evalf :: (Int -> Int -> Int) -> Expr -> Expr -> State TS (Maybe Int)
evalf f x y = do
    ts <- get                           -- recuperem la taula de símbols
    let vlx = evalState (eval x) ts     -- només volem el valor, l'estat no canvia
    let vly = evalState (eval y) ts     -- idem
    return $ liftM2 f vlx vly           -- vlx i vly són Maybe Int

evalDiv :: Maybe Int -> Maybe Int -> Maybe Int
evalDiv x y = do
    x' <- x
    y' <- y
    if y' == 0
    then Nothing
    else return $ div x' y'
```

Petit LP V

Accions

```
exec :: Accio -> State TS (Maybe Int)
exec (Ass s x) = do
    ts <- get
    let (vlx, tsx) = runState (eval x) ts
    put $ addEntry s vlx tsx          -- actualització de la taula de símbols
    return vlx
exec (View x) = do
    ts <- get
    let vl = evalState (eval x) ts
    return vl
```

Ús

```
runState (exec (View (Val 2))) []

👉  (Just 2,[])
```

Petit LP VI

Enllaç

```
let l = [Ass "a" (Val 3), View (Add (Var "a") (Val 2))]
```

```
runState (exec (l!!0) >>= \x -> exec (l!!1) >>= \y -> return [x, y]) []
👉   ([Just 3,Just 5],[("a",Just 3)])       -- (valors resultants, TS final)
```

```
runState (mapM exec l) []
👉   ([Just 3,Just 5],[("a",Just 3)])
```

Un altre exemple

```
let l = [View (Val 2), Ass "a" (Val 3), View (Add (Var "a") (Val 2))]
runState (mapM exec l) []
👉
([Just 2,Just 3,Just 5],[("a",Just 3)])
```