Programare declarativă[1]

Intrare/Ieșire

Ioana Leuștean
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
traian.serbanuta@unibuc.ro

5 ianuarie 2021

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Amintiri dinainte de vacanță

Clasa de tipuri Functor

Definiție

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

Dată fiind o funcție `f :: a -> b` și o acțiune cu efecte laterale `ca :: m a`, `fmap f ca` produce o acțiune `cb :: m b` care

- va avea același efect ca acțiunea `ca`, dar
- al cărei rezultat va fi transformat prin funcția `f`.

Clasa de tipuri Applicative

Definiție

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

- pure transformă o valoare într-o computație care are acea valoare ca rezultat și nu are efecte laterale
- pure f <*> ca1 <*> ... <*> can produce agregarea computațiilor
  - propagând efectele laterale ale computațiilor argument; și
  - combinând rezultatele lor folosind funcția *f*

Operatori

- f <$> x == fmap f x == pure f <*> x
- ax *> ay
  - Păstrează al doilea rezultat și îl ignoră pe primul
  - Propagă efectele laterale

Comenzi I/O în Haskell

Afișează un caracter!

```
putChar :: Char -> IO ()
```

Exemplu

```
putChar '!'
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de exclamare.

Combină două comenzi!

```
(>>) :: IO () -> IO () -> IO ()
putChar :: Char -> IO ()
```

Exemplu

```
putChar '?' >> putChar '!'
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare .

Observație

Deoarece **IO** este instanță a lui Applicative, (>>) este același lucru cu (*>), i.e., ignoră rezultatul acțiunii, dar propagă efectele laterale.

Stai locului! (nu face nimic!)

Putem defini o comandă care, dacă va fi executată, nu va face nimic:

```
done :: IO ()
```

Observație

Deoarece **IO** este instanță a lui Applicative, done poate fi definită ca

```
done = pure ()
```

Afișează un șir de caractere

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs

putStrLn :: String -> IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

Exemplu

```
putStr "?!" == putChar '?' >> (putChar '!' >> done)
```

rerpezintă o comandă care, dacă va fi executată, va afișa un semn de întrebare urmat de un semn de exclamare.

putStr folosind funcționale

```
putStr      :: String -> IO ()
putStr      = foldr (>>) done . map putChar
```

Observație: (**IO** (), (>>), done) e monoid

```
m >> done           =  m
done >> m           =  m
(m >> n) >> o       =  m >> (n >> o)
```

Și totuși, când sunt executate comenzile I/O?

main

Fișierul PutStr.hs

```
main :: IO ()
main = putStr "?!\n"
```

Rularea programului are ca efect executarea comenzii specificate de main:

```
$ runghc PutStr.hs
?!
$
```

Validitatea raționamentelor

Raționamentele substitutive își pierd valabilitatea

În limbaje cu efecte laterale

Program 1

```
int main() { cout << "HA!"; cout << "HA!"; }
```

Program 2

```
void dup(auto& x) { x ; x; }
int main() { dup(cout << "HA!"); }
```

Program 3

```
void dup(auto x) { x() ; x(); }
int main() { dup( []() { cout << "HA!"; } ); }
```

Raționamentele substitutive sunt valabile

În Haskell

Expresii

```
(1+2) * (1+2)
```

este echivalentă cu expresia

```
let x = 1+2 in x * x
```

și se evaluează amândouă la 9

Comenzi

```
putStr "HA!" >> putStr "HA!"
```

este echivalentă cu

```
let m = putStr "HA!" in m >> m
```

si amândouă afișează "HA!HA!".

Comenzi cu valori

Comenzi cu valori

- **IO** () corespunde comenzilor care nu produc rezultate
  - () este tipul unitate care conține doar valoarea ()

- În general, **IO** a corespunde comenzilor care produc rezultate de tip a.
  - **IO Char** corespunde comenzilor care produc rezultate de tip **Char**

Citește un caracter!

getChar :: IO Char

- Dacă „șirul de intrare” conține "abc"
- atunci **getChar** produce:
  - 'a'
  - șirul rămas de intrare "bc"

Produu o valoare fără să faci nimic!

Fără efecte laterale

```
return :: a -> IO a
```

Asemănatoar cu done, nu face nimic, dar produce o valoare.

Exemplu

```
return ""
```

- Dacă „șirul de intrare” conține "abc"
- atunci **return** "" produce:
  - valoarea ""
  - șirul (neschimbat) de intrare "abc"

Observație

Deoarece **IO** este instanță a clasei Applicative, **return** este identic cu pure

Combinarea comenzilor cu valori

Operatorul de legare / bind

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Exemplu

```
getChar >>= \x -> putChar (toUpper x)
```

- Dacă „șirul de intrare” conține "abc"
- atunci comanda de mai sus, atunci când se execută, produce:
  - ieșirea "A"
  - șirul rămas de intrare "bc"

Observație

- Operatorul (>>=) nu poate fi obținut în Applicative
- În schimb, <*> poate fi obținut din (>>=)

```
af <*> ax = af >>= (\f -> ax >>= (\x -> return (f x)))
```

Operatorul de legare / bind

Mai multe detalii

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- Dacă fiind o comandă care produce o valoare de tip a
  m :: **IO** a
- Data fiind o funcție care pentru o valoare de tip a se evaluează la o comandă de tip b
  k :: a -> **IO** b
- Atunci
  m >>= k :: **IO** b
  este comanda care, dacă se va executa:
  - Mai întâi efectuează m, obținând valoarea x de tip a
  - Apoi efectuează comanda k x obținând o valoare y de tip b
  - Produce y ca rezultat al comenzii

Citește o linie!

```
getLine :: IO String
getLine = getChar >>= \x ->
            if x == '\n' then
              return []
            else
              getLine >>= \xs ->
              return (x:xs)
```

Exemplu

Dat fiind șirul de intrare "abc\ndef", getLine produce șirul "abc" și șirul rămas de intrare e "def"

Comenzile sunt cazuri speciale de comenzi cu valori

done e caz special de return

```
done        :: IO ()
done        = return ()
```

>> e caz special de >>=

```
(>>)        :: IO () -> IO () -> IO ()
m >> n      = m >>= \ () -> n
```

Operatorul de legare e similar cu **let**

Operatorul **let**

```
let x = m in n
```

let ca aplicație de funcții

```
(\ x -> n) m
```

Operatorul de legare

```
m >>= \ x -> n
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

De la intrare la ieșire

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo

main :: IO ()
main = echo
```

Test

```
$ runghc Echo.hs
One line
ONE LINE
And, another line!
AND, ANOTHER LINE!
```

Notația do

Citirea unei linii în notație „do”

```
getLine :: IO String
getLine = getChar >>= \x ->
            if x == '\n' then
              return []
            else
              getLine >>= \xs ->
              return (x:xs)
```

Echivalent cu:

```
getLine :: IO String
getLine = do {
            x <- getChar;
            if x == '\n' then
              return []
            else do {
              xs <- getLine;
              return (x:xs)
            }
          }
```

Echo în notația „do”

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo
```

Echivalent cu

```
echo :: IO ()
echo = do {
         line <- getLine;
         if line == "" then
           return ()
         else do {
           putStrLn (map toUpper line);
           echo
         }
       }
```

Notația „do” în general

- Fiecare linie x <– e; ... devine e >>= \x –> ...
- Fiecare linie e; ... devine e >> ...

De exemplu

```
do { x1 <– e1;
     x2 <– e2;
     e3;
     x4 <– e4;
     e5;
     e6 }
```

e echivalent cu

```
e1  >>= \x1 –>
e2  >>= \x2 –>
e3  >>
e4  >>= \x4 –>
e5  >>
e6
```

Monade

Clasa de tipuri **Monad**

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

    (>>) :: m a -> m b -> m b
    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul comenzilor care produc rezultate de tip a (și au efecte laterale)
- a -> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a comenzilor

În Haskell, monada este o clasă de tipuri!