Paradigme de Programare

Ș.l. dr. ing. Andrei Olaru

Departamentul Calculatoare

2014 – 2015, semestrul 2

Cursul 8: Anexă opțională: Monade în Haskell

Cursul 8: Anexă opțională: Monade în Haskell

Monade

Motivatie
Pentru existența *monadelor*

- Cum integrăm operațiile de I/O în stilul funcțional?

Motivatie
Pentru existența *monadelor*

- Cum integrăm operațiile de I/O în stilul funcțional?
- Varianta: funcția `inputInt`, ce întoarce un întreg citit de la consolă

```
1  inputInt     ::   Int
2  inputDiff    =    inputInt - inputInt
```

Motivatie
Pentru existența *monadelor*

- Cum integrăm operațiile de I/O în stilul funcțional?
- Varianta: funcția `inputInt`, ce întoarce un întreg citit de la consolă

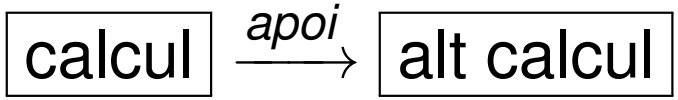
```
1  inputInt     ::   Int
2  inputDiff    =    inputInt - inputInt
```

- Dependența rezultatului de ordinea evaluării parametrilor;

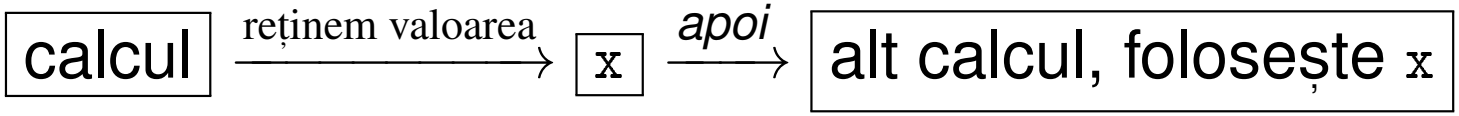
⇒ Ambiguitatea expresiei (`inputDiff` poate diferi de `0`).

- `inputInt` si, drept consecinta, `inputDiff`, opace referential (nu au același efect oriunde ar fi plasate în program) !

Motivație
Ce anume ne este necesar



sau



Acțiuni
Calcule într-o anumita secvență

- Operațiile de I/O $\rightarrow$ actiuni, ce produc efecte laterale: citirea și scrierea la consolă, în fișiere etc.
- Importanța secvențierii acțiunilor;
- Elemente inerent imperative în stilul funcțional!
- Obiective:
  - Controlul construcției programelor ce execută I/O;
  - Limitarea impactului operațiilor de I/O asupra funcțiilor.

Acțiuni
Tipul `IO a`

- Acțiuni $\rightarrow$ valori oarecare, de tipul `IO a`;
- Obiect de tipul `IO a` $\rightarrow$ program, atomic sau compus, ce realizează operații de I/O și întoarce o valoare de tipul `a`;
- Conservare a modelului funcțional al limbajului în sine;
- Separarea construcției obiectelor acțiune (a descrierii acțiunilor ce se doresc efectuate) de execuția propriu-zisă a acțiunilor.

Acțiuni
Acțiuni predefinite (1)

- Citirea unui caracter de la consolă:

```
1 getChar      ::  IO Char
```

- Citirea unei linii de la consolă:

```
1 getLine      ::  IO String
```

- Scrierea unui caracter la consolă:

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

- Scrierea unui sir la consolă:

```
1 putStr       ::  String -> IO ()
2
3 putStrLn     ::  String -> IO ()
4 putStrLn     =   putStr . (++ "\n")
```

Acțiuni
Acțiuni predefinite (2)

- Tipul `()` – *unit type*, similar cu `void` – a cărui unică valoare este reprezentată tot de `()`
- Scrierea unei valori oarecare la consolă:

```
1  print          ::   Show a => a -> IO ()
2  print          =    putStrLn . show
```

- Întoarcerea unei valori:

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

- Toate functiile de mai sus nu declanseaza propriu-zis actiuni, ci doar construiesc un obiect actiune (de tipul `IO a`).

Secvențierea acțiunilor folosind notația `do`
Exemplu

Ex Exemplu

Functia `testGetChar` citeste un caracter de la consola, îl afiseaza si întoarce `True` daca acesta a fost `'y'`.

```
1  testGetChar ::  IO Bool
2  testGetChar =   do
3                      c <- getChar
4                      putChar c
5                      return (c == 'y')
6
7  test        ::  IO ()
8  test        =   do
9                      b <- testGetChar
10                     print b
```

Secvențierea acțiunilor folosind notația `do`
Observații

- Aspectul unui program imperativ $\rightarrow$ doar o deghizare!
- Utilizabilitatea rezultatelor acțiunilor exclusiv în secvența `do` $\rightarrow$ izolarea părților imperative;
- Interzicerea acțiunilor într-o expresie ce nu a fost marcata corespunzator (`IO a`);
- Operatorul `<-` pentru reținerea valorii transmise între operațiunile secvențiate.

Secvențierea acțiunilor folosind notația `do`
Un alt exemplu

Ex Exemplu

Implementarea functiei predefinite `getLine`:

```
1  getL          ::   IO String
2  getL          =    do
3                         c <- getChar
4                         if c == '\n'
5                             then return ""
6                             else do
7                                     line <- getL
8                                     return (c:line)
9  testGetL      ::   IO ()
10 testGetL      =    do
11                        line <- getL
12                        putStr line
```

Secvențierea acțiunilor
Operatorii de secvențiere

- Operatori de secvențiere:

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

Ex Exemplu

```
1  testGetChar      ::  IO Bool
2  testGetChar      =   do
3                           c <- getChar
4                           putChar c
5                           return (c == 'y')
6  testGetChar2     ::  IO Bool
7  testGetChar2     =   getChar
8                       >>= \c -> (putChar c
9                                  >> return (c == 'y'))
```

Aplicație pentru monade

Problemă serializare
Cerința

Ex Exemplu

Să se scrie funcția `serialize`, ce construiește o acțiune compusă dintr-o listă de actiuni mai simple:

```
1  serialize :: [IO a] -> IO ()
```

Pe baza ei, sa se implementeze functia `putStr`.

```
1  actions          ::   [IO ()]
2  actions          =    [   putChar 'a',
3                           do
4                               putChar 'a'
5                               putChar 'b',
6                           do
7                               c <- getChar
8                               putChar c]
```

Problemă serializare
Implementare

```
1  serialize        ::   [IO a] -> IO ()
2  serialize []     =    return ()
3  serialize (h:t)  =    do
4                            h
5                            serialize t
6  serialize2       ::   [IO a] -> IO ()
7  serialize2       =    foldr (>>) (return ())
8
9  putS             =    serialize2 . map putChar
```

! Aplicația `(map putChar lst)` nu produce niciun efect imediat, ci doar construiește lista obiectelor acțiune.