Metody programowania 2017

Lista zadań nr 13

Na zajęcia 6 i 7 czerwca 2017

Zadanie 1 (1 pkt). Problem n-hetmanów polega na ustawieniu n hetmanów na szachownicy o wymiarach $n \times n$ tak, by się wzajemnie nie szachowały. Rozwiąż ten problem w Prologu i Haskellu implementując algorytm przeszukiwania z nawrotami. Do zorganizowania nawrotów w Haskellu użyj notacji do i metod klasy `MonadPlus`, takich jak `guard`.

Zadanie 2 (1 pkt). Oto prosty generator losowy zaprogramowany w C:

```
int seed;

void init(int newSeed) {
   seed = newSeed;
}
int random() {
   int newSeed = 16807 * (seed % 127773)
               - 2836 * (seed / 127773);
   return seed = (newSeed > 0 ? newSeed
                     : (newSeed + 2147483647));
}
```

Zaprogramuj abstrakcyjny typ `Random a` reprezentujący obliczenie z wynikiem typu a i stanem będącym wartością zarodka losowego. Zainstaluj go w klasie `Monad` i zdefiniuj dla niego operacje

```
init :: Int → Random ()
random :: Random Int
```

Zadanie 3 (1 pkt). Napis i strumień wejściowy są bardzo podobnymi strukturami danych — udostępniają sekwencyjny dostęp do znaków. Np. w Javie napisy (`StringReader`) i pliki (`FileReader`) są instancjami tego samego interfejsu `Reader`. Obliczenie, które działa na strumieniu (napisie) zwraca pewną wartość pewnego typu a i zmodyfikowany strumień (modyfikacja strumienia polega na przeczytaniu zeń znaków), jest więc typu

```
String → (a, String).
```

Niech zatem

```
newtype SSC a = SSC (String → (a, String))
```

(SSC to skrót od „string stream computation"). Uczyń typ `SSC` monadą. Zdefiniuj dla niego operacje

```
runSSC :: SSC a → String → a
getc :: SSC Char
ungetc :: Char → SSC ()
isEOS :: SSC Bool
```

tak, by dało się zdefiniować np. takie obliczenie:

```
countLines :: String → Int
countLines = runSSC $ lines 0 where
   lines :: Int → SSC Int
   lines n = do
      b ← isEOS
      if b
         then return n
         else do
            ch ← getc
            lines (if ch == '\n' then n+1 else n)
```

Zadanie 4 (1 pkt). Zauważ, że definicje operatorów `>>=` i `return` w poprzednich dwóch zadaniach są praktycznie identyczne. Zdefiniuj więc monadę realizującą dowolne obliczenie z dowolnym wynikiem w której typem obliczenia jest

```
newtype StateComput s a = SC (s → (a,s))
```

Zadanie 5 (1 pkt). Rozważmy nieskończoną listę wszystkich liczb pierwszych:

```
primes :: [Integer]
```

Bez cukru syntaktycznego definicja tej listy jest mało czytelna:

```
primes = 2 : filter
   (λ n → all (λ p → n `mod` p ≠ 0)
      (takeWhile (λ p → p*p ≤ n) primes))
   (enumFrom 3)
```

Wyrażenia listowe poprawiają czytelność:

```
primes = 2 : [ n | n ← [3..], all
   (λ p → n `mod` p ≠ 0)
   (takeWhile (λ p → p*p ≤ n) primes) ]
```

Skorzystajmy z faktu, że typ `[]` jest monadą i przyjmijmy definicję

```
type Generator = []
```

Dzięki leniwości listę liczb pierwszych możemy traktować jak generator wyliczający na żądanie kolejne liczby pierwsze:

```
primes :: Generator Integer
primes = return 2 `mplus` do
   n ← enumFrom 3
   guard (all (λ p → n `mod` p ≠ 0)
              (takeWhile (λ p → p*p ≤ n) primes)
   return n
```

Zapisz we wszystkich trzech przedstawionych wyżej stylach (bez cukru syntaktycznego, z wyrażeniami listowymi oraz z do-notacją) haskellowe funkcje:

```
tails :: [a] → [[a]]
```

odpowiadające następującemu predykatowi prologowemu:

```
tails(T,T).
tails([_|T],S) :-
   tails(T,S).
```

Zadanie 6 (1 pkt). Niech

```
data Term sig var = Var var
                  | FunSym sig [Term sig var]
```

Tablica 1: Operacje wejścia/wyjścia oparte na dialogu

```
module IO(Response(..), Request(..), Dialog)
   where
      data Request = PutStrLn String | ReadLine
      data Response = OK | OKStr String
      type Dialog = [Response] → [Request]

module Main where
   import IO(Response(..), Request(..), Dialog)
   main :: Dialog
   main resps = ReadLine :
      (case resps of
         OKStr str : resps →
            if str == ""
               then []
               else (PutStrLn . reverse $ str) :
                  (case resps of
                     OK : resps → main resps))
```

Tablica 2: Operacje wejścia/wyjścia oparte na kontynuacjach

```
module IO(Answer, putStrLn, getLine, done) where
   type Answer
   putStrLn :: String → Answer → Answer
   getLine :: (String → Answer) → Answer
   done :: Answer

module Main where
   import IO(Answer, putStrLn, getLine, done)
   main :: Answer
   main = getLine (λ line →
      if line == ""
         then done
         else putStrLn (reverse line) main)
```

będzie gramatyką abstrakcyjną opisującą termy nad sygnaturą `sig :: * ze` zbiorem zmiennych `var :: *`. Dla ustalonej sygnatury `sig` typ `Term sig :: * -> *` ma naturalną strukturę monady. Odkryj ją i zainstaluj typ `Term sig` w klasie `Monad`.

Zadanie 7 (1 pkt). Operacje wejścia/wyjścia w językach czysto deklaratywnych implementowano dawniej przy pomocy dialogów, które pozwalały na wykonywanie takich obliczeń, jak przedstawione w Tablicy 1. Kontynuacje zapewniają łatwiejszą kontrolę skutków ubocznych, zob. przykład w Tablicy 2. Monady okazały się jednak najlepszym rozwiązaniem, zob. przykład w Tablicy 3. Zdefiniuj funkcję

```
dialogToIOMonad :: Dialog -> IO ()
```

pozwalającą na uruchomienie we współczesnej implementacji Haskella programu napisanego przy użyciu dialogów.

Tablica 3: Operacje wejścia/wyjścia oparte na monadach

```
module IO(IO, putStrLn, getLine) where
   type IO a
   instance Monad IO
   putStrLn :: String → IO ()
   getLine :: IO String

module Main where
   import IO(IO, putStrLn, getLine)
   main :: IO ()
   main = do
      line ← getLine
      if line == ""
         then return ()
         else do
            putStrLn (reverse line)
            main
```