Programowanie Funkcyjne 2020

Lista zadań nr 11 dla grup mabi, mbu, ppo i efes

Na zajęcia 19 i 20 stycznia 2021

Zadanie 1 (2 pkt). Obliczenia używające liczb pseudolosowych możemy opisać za pomocą dowolnej monady wzbogaconej o operację `random`.

```
class Monad m => Random m where
  random :: m Int
```

Napisz funkcję `shuffle :: Random m => [a] -> m [a]`, która wybiera losową permutację podanej listy.

Zadanie 2 (2 pkt). Aby uruchomić funkcję z poprzedniego zadania, potrzebujemy instancji klasy `Random`. Można ją zaimplementować, jako obliczenie z jednoelementowym stanem będącym liczbą całkowitą, reprezentującym aktualną wartość zarodka losowego.

```
newtype RS a = RS {unRS :: Int -> (Int, a)}
```

Obliczenie typu `RS a` jest funkcją, która wartość zarodka losowego (przed wykonaniem obliczenia) przekształca w nową wartość zarodka (po wykonaniu obliczenia) oraz wynik. Zainstaluj typ `RS` w klasach `Monad` i `Random`, a następnie użyj go aby przetestować funkcję z poprzedniego zadania. Będziesz potrzebować funkcji wykonującej obliczenie losowe z podanym początkowym zarodkiem, `withSeed :: RS a -> Int -> a`.

Przykładowa funkcja generująca wartości pseudolosowe jest dana przez kolejne wartości ciągu a_i, gdzie a_0 jest początkowym zarodkiem losowym, a kolejne wyrazy zdefiniowane są następująco:

$$\begin{aligned} b_i &= 16807 \cdot (a_i \bmod 127773) - 2836 \cdot (a_i \div 127773) \\ a_{i+1} &= \begin{cases} b_i, & \text{gdy } b_i > 0; \\ b_i + 2147483647, & \text{w p.p.} \end{cases} \end{aligned}$$

Zadanie 3 (4 pkt). Dowolną grę dla dwóch graczy z jawnym stanem można traktować jako obliczenie gdzie efektami ubocznymi są pytania o decyzje graczy. Zatem grę można opisać dowolną monadą, która ma zaimplementowane operacje `moveA` oraz `moveB`, które odpytują odpowiednio gracza A oraz B o kolejny ruch.

```
{-# LANGUAGE FlexibleContexts, FlexibleInstances, FunctionalDependencies #-}

class Monad m => TwoPlayerGame m s a b | m -> s a b where
  moveA :: s -> m a
  moveB :: s -> m b
```

W tej definicji `s` oznacza stan planszy, `a` jest typem wszystkich możliwych ruchów gracza A, natomiast `b` jest typem wszystkich możliwych ruchów gracza B. Definicja ta lekko wykracza poza standard Haskella, więc należy włączyć kilka ogólnie przyjętych rozszerzeń GHC — w szczególności więz `m -> s a b` oznacza że konstruktor typu `m` powinien determinować typy stanu i ruchów (co w tym zadaniu nie powinno sprawić praktycznych problemów).

Wynik gry można opisać następującym typem:

```
data Score = AWins | Draw | BWins
```

Zaimplementuj wybraną przez siebie grę (np. kółko i krzyżyk, reversi czy gomoku[1]) jako obliczenie klasy `TwoPlayerGame`. Jeśli typy (które zdefiniujesz) `Board`, `AMove` oraz `BMove` opisują odpowiednio stan

[1] Stan gry, ruchy graczy i zasady poprawności ruchów powinny być relatywnie proste — szachy czy warcaby wprowadzają komplikacje ortogonalne do istoty zadania.

planszy, przestrzeń ruchów gracza A oraz przestrzeń ruchów gracza B to powinieneś zdefiniować wartość o następującej sygnaturze.

```
game :: TwoPlayerGame m Board AMove BMove => m Score
```

Podanie niedozwolonego ruchu może kończyć się natychmiastową porażką.

Zadanie 4 (3 pkt). Aby przetestować grę z poprzedniego zadania, potrzebujemy instancji. Jeśli umiemy wyświetlać stan planszy oraz wczytywać ruchy graczy, to monada `IO` będzie świetnie się do tego nadawać. Trzeba ją jedynie opakować w typ, który ma więcej parametrów, by spełnić wymagane zależności funkcyjne:

```
newtype IOGame s a b x = IOGame { runIOGame :: IO x }
```

Dostarcz następującej instancji

```
instance (Show s, Read a, Read b) => TwoPlayerGame (IOGame s a b) s a b where
```

oraz — jeśli potrzebujesz — odpowiednich instancji klas `Show` i `Read`, tak, aby można było zagrać w grę z poprzedniego zadania. Operacje `moveA` oraz `moveB` powinny najpierw wyświetlać stan planszy, a następnie prosić o podanie ruchu odpowiedniego gracza.

Zadanie 5 (2 pkt). Dotychczas patrzyliśmy na monady jako sposób opisu obliczeń z efektami. Alternatywnie, na monady można patrzeć jak na drzewa, które trzymają dane w liściach. Takie drzewa są wszechobecne zarówno w informatyce jak i logice. Na przykład poniższy typ, reprezentujący termy logiki pierwszego rzędu może być postrzegany jako pewien typ drzew, gdzie danymi są zmienne.

```
type Symbol = String
data Term v
  = Var v
  | Sym Symbol [Term v]
```

Odkryj w nim strukturę monady i zainstaluj go w klasie `Monad`. Następnie zastanów się jakim operacjom na drzewach odpowiadają monadyczne operatory `>>=` oraz `return`?

Zadanie 6 (2 pkt). Okazuje się, że związek monad z termami jest dość głęboki i strukturę monady można odnaleźć również w termach z wiązaniem zmiennych reprezentowanym za pomocą *zagnieżdżonych typów danych* (patrz lista nr 9). Przypomnij sobie poniższy typ formuł QBF

```
data Inc v = Z | S v

data Formula v
  = Var v
  | Bot
  | Not (Formula v)
  | And (Formula v) (Formula v)
  | All (Formula (Inc v)),
```

a następnie zainstaluj go w klasie `Monad`.
Wskazówka: Najpierw zainstaluj typy `Inc` oraz `Formula` w klasie `Functor` (patrz lista 9). Funkcja `fmap` okaże się pomocna przy implementowaniu funkcji `>>=`.

Zadanie 7 (1 pkt). Powróćmy jeszcze raz do gier z zadania 3. Wzorując się na typie `StreamTrans` z poprzedniej listy zaproponuj typ `GameTree s a b x` reprezentujący drzewo gry (pytania o ruch są analogiczne do jednej z konstrukcji transformatorów strumieni — zastanów się której). Następnie zainstaluj ten typ w klasach `Monad` oraz `TwoPlayerGame`

Zadanie 8 (4 pkt). Mając drzewo gry z poprzedniego zadania można napisać program grający w grę. Napisz funkcję

```
play :: (Show s, Read a) =>
  Int -> (s -> [a]) -> (s -> [b]) -> GameTree s a b Score -> IO ()
```

która pozwoli zagrać w podaną grę z komputerem. Wywołanie `play depth aMoves bMoves game` prosi tylko gracza A o podanie ruchu, natomiast wybiera ruchy gracza B starając się unikać tych, które w `depth` krokach niechybnie prowadzą do porażki, przy założeniu, że gracz A gra optymalnie, a dozwolone ruchy są opisane funkcjami `aMoves` oraz `bMoves`.

Uwaga: Nie będzie to specjalnie efektywny program — przestrzeń stanów większości gier bardziej skomplikowanych niż kółko i krzyżyk jest olbrzymia, więc głębokość nie będzie mogła być duża. Jeśli chcesz poprawić efektywność wyboru ruchu podstawowym kierunkiem jest definicja funkcji oceniającej pozycję na podstawie stanu planszy, a następnie przeszukiwanie priorytetyzujące lepsze (wyżej oceniane) pozycje i agresywnie odcinające gałęzie drzewa o niskiej ocenie.