Programowanie Funkcyjne 2023

Lista zadań nr 9

Na zajęcia 20 i 22 grudnia 2023

Zadanie 1 (4p). Dowolną grę dla dwóch graczy 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[1].

Natomiast wynik gry można opisać następującym typem.

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

Zaimplementuj wybraną przez siebie grę (np. kółko i krzyżyk) jako obliczenie klasy `TwoPlayerGame`. Jeśli typy (które zdefiniujesz) `Board`, `AMove` oraz `BMove` opisują odpowiednio stan 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 powinno kończyć się natychmiastową porażką.

Zadanie 2 (2p). 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
```

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 3 (2p). Wzorując się na typie `StreamTrans` z poprzedniej listy zaproponuj typ `GameTree s a b x` reprezentujący drzewo gry, jako wolną monadę (staraj się nie szukać w internecie o wolnych monadach, bo natkniesz się na ogólniejszą konstrukcję monady z dowolnego funktora). Następnie zainstaluj ten typ w klasach `Monad` oraz `TwoPlayerGame`

Zadanie 4 (1p). 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 ()
```

[1] Bardziej eleganckie rozwiązanie korzystało by z indeksowanych rodzin typów zamiast wieloparametrowych klas z zależnościami funkcyjnymi, ale indeksowanych rodzin typów jeszcze nie widzieliśmy.

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`.

Zadanie 5 (3p). Na ćwiczeniach do listy zadań nr 8 widzieliśmy interpreter języka *Brainfuck*, który jawnie przekazywał stan taśmy, natomiast operacje wejścia-wyjścia były wbudowane w model obliczeń (rozważaliśmy wtedy transformatory strumieni). W tym zadaniu zrobimy odwrotnie: operacje na taśmie będą wbudowane, a stan wejścia-wyjścia będziemy jawnie przekazywać. W tym celu wzbogacimy klasę `Monad` o operacje, które odpowiednio czytają i zapisują wartość na taśmę, oraz przesuwają taśmę w lewo i w prawo.

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

class Monad m => TapeMoand m a | m -> a where
  tapeGet   :: m a
  tapePut   :: a -> m ()
  moveLeft  :: m ()
  moveRight :: m ()
```

Napisz funkcję, która interpretuje składnię abstrakcyjną języka *Brainfuck* jako obliczenie klasy `TapeMonad`. Twój interpreter powinien jako dodatkowy argument przyjąć listę znaków czekających na standardowym wejściu, natomiast zwracane obliczenie powinno produkować listę znaków wypisanych na standardowe wyjście. Powinieneś otrzymać funkcję o następującej sygnaturze.

```
evalBF :: TapeMonad m Integer => [BF] -> [Char] -> m [Char]
```

Nie przejmuj się, jeśli Twoje rozwiązanie nie działa dla programów, które się nie zatrzymują.

Zadanie 6 (3p). Dostarcz instancję klasy `TapeMonad`, aby można było uruchomić interpreter z poprzedniego zadania. Oczywiście będzie to szczególny przypadek monady stanu. Możesz odpowiedni typ zdefiniować samemu albo użyć bibliotecznego typu `State`. Następnie napisz funkcję

```
runBF :: [BF] -> [Char] -> [Char]
```

która uruchamia podany program na pustej taśmie. Funkcja ta powinna być zdefiniowana na bazie funkcji `evalBF` z poprzedniego zadania.

Zadanie 7 (3p). Implementacja interpretera z zadania 5 nie jest najwygodniejsza, ponieważ trzeba jawnie przekazywać stan strumienia wejściowego i wyjściowego. W istocie, interpretacja języka *Brainfuck* jest obliczeniem, które korzysta z trzech niezależnych od siebie efektów ubocznych: operacji na taśmie, czytania strumienia wejściowego i pisania do strumienia wyjściowego. Niestety nie znamy jeszcze mechanizmów, które pozwalają podejść do zagadnienia w sposób modularny (takich jak transformatory monad), a zdefiniowanie trzech niezależnych monad nie wystarczy (dlaczego?). Na razie zadowolimy się niezbyt modularnym, rozwiązaniem: zdefiniuj klasę typów `BFMonad`, rozszerzającą monady o operacje związane ze wszystkimi trzema wspomnianymi efektami. Następnie napisz interpreter o następującej sygnaturze.

```
evalBF :: BFMonad m => [BF] -> m ()
```

Zadanie 8 (2p). Zdefiniuj instancję klasy `BFMonad` i użyj jej do zdefiniowania funkcji `runBF`, podobnie do tego, jak to robiliśmy w zadaniu 6.