Programowanie Funkcyjne 2020

Lista zadań nr 12 dla grup mabi, mbu, ppo i efes

Na zajęcia 26 i 27 stycznia 2021

Zadanie 1 (3 pkt, dodatkowe). Rozwiąż zadania 3 i 4 z poprzedniej listy zadań.

Zadanie 2 (5 pkt). Rozwiąż zadania 7 i 8 z poprzedniej listy zadań.

Zadanie 3 (3 pkt). Na ćwiczeniach do listy zadań nr 10 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]
```

Do przetestowania swojego rozwiązania możesz wykorzystać parser zamieszczony w SKOSie. Nie przejmuj się, jeśli Twoje rozwiązanie nie działa dla programów, które się nie zatrzymują.

Zadanie 4 (2 pkt). Dostarcz instancję klasy `TapeMonad`, aby można było uruchomić interpreter z poprzedniego zadania. Oczywiście będzie to szczególny przypadek monady stanowej. Możesz zdefiniować typ analogiczny do typu `RS` z poprzedniej listy, 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 5 (3 pkt). Implementacja interpretera z zadania 3 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 6 (2 pkt). Zdefiniuj instancję klasy `BFMonad` i użyj jej do zdefiniowania funkcji `runBF`, podobnie do tego, jak to robiliśmy w zadaniu 4.

Zadanie 7 (3 pkt). Wyrażenia regularne można opisać następującym typem danych.

```
data RegExp a
  = Eps
  | Lit  (a -> Bool)
  | Or   (RegExp a) (RegExp a)
  | Cat  (RegExp a) (RegExp a)
  | Star (RegExp a)
```

W tej definicji `a` oznacza alfabet nad którym pracujemy (zwykle będzie to typ `Char`). Znaczenia konstruktorów są następujące.

`Eps` dopasowuje się tylko do słowa pustego.

`Lit p` dopasowuje się tylko do jednoliterowych słów, których jedyna litera spełnia predykat `p`.

`Or r1 r2` dopasowuje się tylko do słów, które są opisane przynajmniej jednym z wyrażeń `r1` oraz `r2`.

`Cat r1 r2` dopasowuje się do słów które można utworzyć poprzez konkatenację słowa opisanego przez `r1` ze słowem opisanym przez `r2`.

`Star r` dopasowuje się do słów, które można rozbić na ciąg słów (być może pusty), z których każde pasuje do `r`.

Na przykład słowa, w których bezpośrednio po każdej literce `'b'` występuje `'a'` można opisać następującym wyrażeniem regularnym.

```
Star (Star (Lit (/= 'b')) `Or` (Lit (== 'b') `Cat` Lit (== 'a')))
```

Napisz funkcję, która dla dowolnej monady z plusem próbuje dopasować wyrażenie regularne do prefiksu danego słowa.

```
match :: MonadPlus m => RegExp a -> [a] -> m (Maybe [a])
```

Aby uniknąć zapętlenia dla gwiazdki na pustym słowie, obliczenie powinno zwracać `Nothing`, gdy dopasowano się do pustego słowa oraz `Just xs`, gdy dopasowano się do niepustego słowa, a pozostały sufiks to `xs`. Dla przypomnienia, monady z plusem służą do wyrażania obliczeń z nawracaniem i wzbogacają zwykłe monady o dwie operacje: `mzero` która jest obliczeniem, które nie zwraca żadnego wyniku (kończy się porażką), oraz `mplus` która równolegle składa dwa obliczenia — możliwym wynikiem całości jest możliwy wyniki każdego ze składników.

Zadanie 8 (2 pkt). Przy pomocy funkcji `match` z poprzedniego zadania oraz korzystając z faktu, że listy tworzą monadę z plusem, napisz funkcję o typie `RegExp a -> [a] -> Bool`, która sprawdza, czy dane wyrażenie regularne dopasowuje się do całości podanego słowa. Następnie zamień monadę listową na monadę `Maybe` (która też jest monadą z plusem). Czy po tej zamianie program działa zgodnie z oczekiwaniem?