Programowanie Funkcyjne 2018

Lista zadań nr 12: Monady

Na zajęcia 17 stycznia 2019
Lista została przygotowana przez Filipa Sieczkowskiego

Zadanie 1 (2 pkt). Powiemy że `m` jest typem *transformatorów strumieni*, jeśli jest zainstalowany w następującej klasie typów:[1]

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

class Monad m => StreamTrans m i o | m -> i o where
   readS :: m (Maybe i)
   emitS :: o -> m ()
```

Transformator strumieni `StreamTrans m i o` konsumuje elementy strumienia wejściowego (typu `i`) i wytwarza elementy strumienia wyjściowego (typu `o`). W tym celu może przeczytać element strumienia wejściowego (operacja `readS`, która zwraca `Nothing` gdy przeczytaliśmy już wszystkie elementy strumienia wejściowego) lub wyemitować element strumienia wyjściowego (operacja `emitS`) — a ponieważ kolejność obliczeń ma znaczenie, od konstruktora typu `m` wymagamy aby był monadą.

Przykładem prostego transformatora strumieni jest obliczenie zamieniające w napisie wszystkie wystąpienia dużych liter na odpowiadające im małe litery. Zaimplementuj obliczenie `toLower :: StreamTrans m Char Char => m Integer`, które zamienia wielkie litery w strumieniu wejściowym na odpowiadające im małe litery, pozostawia resztę znaków bez zmian, a którego wynikiem jest liczba wykonanych zamian.

Zadanie 2 (3 pkt). Obliczenia z poprzedniego zadania nie umiemy jeszcze przetestować, gdyż nie mamy żadnej instancji klasy `StreamTrans`. Możemy jednak sprawić żeby monada `IO` służyła do transformowania ciągów znaków wczytywanych ze standardowego wejścia na ciągi znaków wyświetlane na ekranie. Uzupełnij poniższą definicję i przetestuj rozwiązanie poprzedniego zadania:

```
instance StreamTrans IO Char Char where
  ...
```

Transformatorów strumieni możemy też używać żeby przetwarzać listy wartości dowolnego typu wejściowego na listy wartości odpowiedniego typu wyjściowego. W tym celu, zdefiniujmy typ obliczeń wczytujących dane z listy elementów typu `i`, tworzących listę wypisanych elementów typu `o`, a których wynik ma typ `a`:

```
newtype ListTrans i o a = LT { unLT :: [i] -> ([i], [o], a) }
```

Zainstaluj typ `ListTrans i o` w klasach `Monad` i `StreamTrans`, uzupełniając poniższe definicje.

```
instance Monad (ListTrans i o) where
  ...
instance StreamTrans (ListTrans i o) i o where
  ...
```

Zauważ jednak, że wciąż nie umiemy *wywołać* obliczenia z poprzedniego zadania przy użyciu tych definicji. W tym celu potrzebować będziemy funkcji `transform :: ListTrans i o a -> [i] -> ([o], a)`. Zdefiniuj ją, i użyj do przetestowania obliczenia z poprzedniego zadania.

[1] Używamy rozszerzenia kompilatora GHC pozwalającego na ustalanie zależności pomiędzy parametrami klasy typów.

Zadanie 3 (4 pkt). Jednym z najbardziej klasycznych zastosowań transformatorów strumieni są leksery, czyli programy zamieniające ciągi znaków w ciągi (bardziej abstrakcyjnych) tokenów, w pierwszym etapie parsowania. Rozważmy składnię konkretną języka wyrażeń arytmetycznych daną następującą gramatyką EBNF (przyjmujemy że klasy znaków `letter`, `digit`, `alphaNum`, `blank` to odpowiednio litery, cyfry, znaki alfanumeryczne i białe znaki):

```
number ::= digit+
var    ::= letter alphaNum*
binop  ::= '+' | '-' | '*' | '/'
ws     ::= blank*
expr   ::= var | number | expr ws binop ws expr | '(' ws expr ws ')'
```

Powyższa gramatyka nie jest jednoznaczna (operatory nie mają ustalonej siły wiązania ani łączności), ale ponieważ nie parsujemy programu a tylko przetwarzamy go na bardziej abstrakcyjną postać — nie ma to znaczenia.

Zdefiniuj typ `Token` definiujący tokeny powyższego języka i obliczenie `lexer :: StreamTrans m Char Char => m ()`, a następnie przetestuj jego działanie. Białe znaki nie są istotne semantycznie i powinny być przez lekser usuwane.

Zadanie 4 (3 pkt). Podobnie do klasy reprezentującej transformatory strumieni możemy reprezentować obliczenia używające liczb pseudolosowych:

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

Prosty typ obliczeń z losowością możemy zdefiniować jako obliczenia z jednoelementowym stanem będącym liczbą całkowitą, reprezentującym aktualną wartość zarodka losowego:

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

Zainstaluj `RS` w klasach `Monad` i `Random`, a następnie użyj go aby zrandomizować działanie Twojego programu grającego w Nim (z poprzedniej listy).

Wskazówka: Będziesz potrzebować funkcji wykonującej obliczenie losowe z podanym początkowym ziarnem, `withSeed :: RS a -> Int -> a`. Przykładowa funkcja generująca wartości pseudolosowe jest dana przez kolejne wartości ciągu a_i, gdzie a_0 jest początkowym ziarnem 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 5 (6 pkt). Kolejnym przydatnym rodzajem obliczeń są obliczenia niedeterministyczne. Możemy je zamodelować następująco:

```
class Monad m => Nondet m where
  amb  :: m a -> m a -> m a
  fail :: m a
```

Przyjmujemy tu, że operator `amb` jest operatorem niedeterministycznego wyboru, zwracającym prawdę lub fałsz, zaś operator `fail` jest operatorem porażki (w pewnym sensie odpowiadającym zgłoszeniu wyjątku). Zdefiniuj typ danych `RegExpr` reprezentujący wyrażenia regularne, a następnie użyj obliczeń niedeterministycznych aby zdefiniować obliczenie `match :: Nondet m => RegExpr -> String -> m ()` sprawdzające czy dany napis jest rozpoznawany przez wyrażenie regularne.

Wskazówka: Najłatwiej zdefiniować takie obliczenie przetwarzając prefiks sprawdzanego napisu i (w przypadku sukcesu) zwracając pozostałą jego część. Wtedy aby zdefiniować `match` wystarczy sprawdzić czy umiemy dopasować się do takiego prefiksu żeby pozostały fragment był listą pustą (w przeciwnym wypadku wystarczy użyć operacji `fail`).

Zadanie 6 (2 pkt). Aby uruchomić obliczenia niedeterministyczne możemy użyć (między innymi) typów `[]` i `Maybe`. Zdefiniuj odpowiednie instancje i funkcje uruchamiające dla tych typów. Jak użycie jednej lub drugiej z tych implementacji wpływa na Twoje rozwiązanie poprzedniego zadania?