Programowanie Funkcyjne 2018

Lista zadań nr 13

Na zajęcia 24 stycznia 2019

Algebraiczne typy danych, jako typy konkretne, pozwalają na definiowanie danych posiadających ustaloną strukturę, np.:

```
data List a = Cons a (List a) | Nil
```

Łatwo możemy dodawać do nich nowe funkcjonalności, np.

```
length :: List a -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs
```

jednak struktura danych jest ustalona. Wiele operacji (w tym `(++)`, `(!!)` i `length`) nie da się dla tej reprezentacji zaprogramować efektywnie. Jeśli nasz program korzysta z takich list, to jest skazany na ich wady.

Dualnie, klasy typów pozwalają na definiowanie typów abstrakcyjnych o nieustalonej strukturze, ale określonym zbiorze funkcjonalności:

```
import Prelude hiding ((++), head, tail, length, null, (!!))
import qualified Prelude ((++), head, tail, length, null, (!!))

class List l where
   nil :: l a
   cons :: a -> l a -> l a
   head :: l a -> a
   tail :: l a -> l a
   (++) :: l a -> l a -> l a
   (!!) :: l a -> Int -> a
   toList :: [a] -> l a
   fromList :: l a -> [a]
```

Jeśli nasz program korzysta z powyższych list, to ich reprezentację możemy łatwo poprawiać bez konieczności zmiany naszego programu. Z drugiej strony dodawanie nowych funkcjonalności może być utrudnione lub niemożliwe (np. sprawdzenie, czy lista jest pusta), gdyż nie znamy struktury typu, tylko jego abstrakcyjny interfejs.

Aby w powyższym programie uniknąć kolizji między nazwami metod naszej klasy i nazwami funkcji z preludium standardowego ukryliśmy niektóre identyfikatory.

Zadanie 1 (1 pkt). Zainstaluj typ `[]` w klasie `List`. Oddasz mu w ten sposób pożyczone identyfikatory.

Klasa `List` nie oferuje ujawniania długości listy. Jest to nowa funkcjonalność niemożliwa do wyrażenia za pomocą metod tej klasy. Możemy klasę `List` rozszerzyć przez dziedziczenie:

```
class List l => SizedList l where
   length :: l a -> Int
   null :: l a -> Bool
   null l = length l == 0
```

Zadanie 2 (1 pkt). Zainstaluj typ `[]` w klasie `SizedList`. Nie korzystaj z domyślnej implementacji metody `null`, tylko podaj własną, efektywną wersję dla tego typu.

Zadanie 3 (2 pkt). Każdą implementację list można uzupełnić do implementacji efektywnie ujawniającej długość listy za pomocą typu

```
data SL l a = SL { len :: Int, list :: l a }
```

Jeśli `l` należy do klast `List`, to `SL l` w oczywisty sposób należy do klasy `SizedList`. Zdefiniuj tę oczywistość w postaci instancji klas:

```
instance List l => List (SL l) ...
instance List l => SizedList (SL l) ...
```

Zadanie 4 (2 pkt). Jeśli często wykonujemy operację spinania list, to standardowy typ `[]` nie jest efektywny. Listy można przedstawiać w postaci drzew binarnych o etykietowanych liściach, w których wierzchołek wewnętrzny odpowiada spinaniu list:

```
infixr 6 :+
data AppList a = Nil | Sngl a | AppList a :+ AppList a
```

Operacja spinania list jest wówczas konstruktorem typu i działa w czasie stałym. Płacimy za to udogodnienie wydłużeniem czasu działania innych operacji. Zainstaluj typ `AppList` w klasach `Show` (tak, żeby powyższe listy były wypisywane tak samo, jak zwykłe) i `SizedList`. Przemyśl definicję metody `toList` tak, żeby zmaksymalizować efektywność metod `head` i `tail` dla budowanych list.

Zadanie 5 (2 pkt). Jeśli często dodajemy elementy na koniec listy, wówczas efektywną implementacją są listy różnicowe znane z języka Prolog. W Haskellu implementujemy je za pomocą funkcji, które dla podanego ogona zwracają listę złożoną z podanych elementów i elementów podanego ogona:

```
newtype DiffList a = DL ([a] -> [a])
```

Np. Prologową listę różnicową `[1,2,3|X]` reprezentujemy tu w postaci funkcji `DL(\xs->1:2:3:xs)`. Zainstaluj typ `DiffList` w klasach `Show` (tak, żeby powyższe listy były wypisywane tak samo, jak zwykłe) i `SizedList`.

Zadanie 6 (5 pkt). Jeśli za pomocą list symulujemy tablice o dostępie swobodnym (co nie jest dobre, ale często konieczne), to zależy nam na efektywności operacji `(!!)`. Listy, w których operacja `(!!)` działa w czasie logarytmicznym względem liczby elementów listy nazywa się *listami o dostępie swobodnym*. Najprostsza implementacja takich list wykorzystuje reprezentacje numeryczne, które poznaliśmy rozważając kopce, w których kontenerami przechowującymi wartości są pełne drzewa binarne o etykietowanych liściach:

```
data RAL a = Empty | Zero (RAL (a,a)) | One a (RAL (a,a))
```

Lista `[1,2,3,4,5]` jest tu reprezentowana jako wartość

```
One 1 $ Zero $ One ((2,3),(4,5)) $ Empty
```

Koszt metod `head`, `tail` i `(!!)` jest logarytmiczny. Zainstaluj typ `RAL` w klasach `Show` (tak, żeby powyższe listy były wypisywane tak samo, jak zwykłe) i `SizedList`.

Typy należące do klasy `MonadPlus` mogą symulować:

- obliczenia z nawrotami, w których `return` oznacza pojedynczy sukces, a `mzero` porażkę, >>= jest sekwencyjnym złożeniem obliczeń, a `mplus` odpowiada operatorowi `amb` z poprzedniej listy;
- obliczenia z wyjątkami, w których `return` oznacza obliczenie zakończone poprawnie, `mzero` jest zgłoszeniem wyjątku, >>= jest sekwencyjnym złożeniem obliczeń, a `mplus` to operacja obsługi (przechwycenia) wyjątku.

Kanonicznym przykładem typu realizującego pierwsze z wymienionych zadań jest `[]`, a drugie — `Maybe`.

Zadanie 7 (3 pkt). Prostym przykładem obliczenia z nawrotami jest generowanie permutacji przez wstawianie bądź wybieranie. Zaprogramuj funkcje

```
iperm, sperm :: MonadPlus m => [a] -> m [a]
```

Zauważ, jak notacja `do` pozwala elegancko zapisać te algorytmy.

Zadanie 8 (4 pkt). Kanonicznym przykładem obliczenia z nawrotami jest ustawianie hetmanów na szachownicy. Zaprogramuj funkcję

```
queens :: MonadPlus m => Int -> m [Int]
```

Dla podanego argumentu n pojedyncze rozwiązanie powinno być listą długości n, w której jeśli i-ty element ma wartość j, to hetman stoi w polu (i, j). Monada `m` zapewnia zwracanie kolejnych wyników na żądanie.