# Programowanie Funkcyjne 2018

Lista zadań nr 13 [autor: TWi]

30 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, 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.

**Zadanie 9 (0 pkt).** *Języki dedykowane* (*Domain-Specific Languages*, *DSL*) są, w odróżnieniu od języków ogólnego przeznaczenia, używane do rozwiązywania problemów w konkretnych, wąskich dziedzinach. Prominentnymi przykładami takich języków są SQL i HTML, które są powszechnie wykorzystywane w oprogramowaniu serwerów WWW. Oprogramowanie serwera jest zwykle napisane w języku ogólnego przeznaczenia (PHP, Ruby, Python, Java itp.), a fragmenty kodu w językach SQL i HTML osadza się w nim w postaci zwykłych napisów. Nie tylko wykonanie, ale również analiza składniowa i kompilacja takich wstawek odbywa się dynamiczne podczas wykonania programu serwera, a nie statycznie podczas jego kompilacji. Jedynym sposobem sprawdzenia poprawności tych wstawek jest testowanie — kompilator nie wspiera nawet sprawdzenia ich poprawności składniowej, kontroli typów itp.

Języki o dużej sile wyrazu, w szczególności języki funkcyjne, pozwalają na osadzanie DSL-i w postaci wyrażeń określonego typu, kompilowanych i sprawdzanych razem z kodem gospodarza. Jednym z najstarszych przykładów tego typu są *kombinatory parsujące*, które pozwalają na zapisanie w języku funkcyjnym gramatyki bezkontekstowej i tym samym osadzenie w kodzie programu dowolnego parsera. Takie rozwiązanie jest znacznie wygodniejsze niż generowanie parsera za pomocą osobnego programu, takiego jak Bison. W Haskellu kombinatory parsujące są dostępne w module `Parsec`.

W tym zadaniu osadzimy w Haskellu bardzo prosty DSL — mały kalkulator. Wyrażenia naszego kalkulatora są opisane następującą składnią konkretną:

$$
\begin{array}{rcl}
\langle\text{expression}\rangle & ::= & (\ \langle\text{expression}\rangle\ ) \\
 & \mid & \langle\text{integer literal}\rangle \\
 & \mid & \texttt{True} \\
 & \mid & \texttt{False} \\
 & \mid & (\ \langle\text{expression}\rangle\ ,\ \langle\text{expression}\rangle\ ) \\
 & \mid & \langle\text{unary operator}\rangle\ \langle\text{expression}\rangle \\
 & \mid & \langle\text{expression}\rangle\ \langle\text{binary operator}\rangle\ \langle\text{expression}\rangle \\
 & \mid & \langle\text{expression}\rangle\ ?\ \langle\text{expression}\rangle\ :\ \langle\text{expression}\rangle \\
\langle\text{unary operator}\rangle & ::= & \texttt{!} \mid \texttt{fst} \mid \texttt{snd} \\
\langle\text{binary operator}\rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{+}
\end{array}
$$

Operatory binarne łączą w lewo. Operator `/` wyznacza parę złożoną z części całkowitej ilorazu i reszty z dzielenia podanych argumentów. Operatory uszeregowane według priorytetów: operatory unarne `!`, `fst`, `snd`, operatory multiplikatywne `*`, `/`, operatory addytywne `+` i `-`, operatory relacyjne `<`, `<=`, `>`, `>=`, `!=`, `==`, operator `&&`, operator `||`, operator ternarny `?:`. Reguły typowania wyrażeń są przedstawione na Rysunku 1.

Moglibyśmy z łatwością napisać interpreter wyrażeń całkowitoliczbowych:

```
interpreter :: String -> Integer
```

i umieszczać w programie Haskellowym wstawki w języku kalkulatora np. tak:

$$\frac{n \text{ — integer literal}}{n : \texttt{Integer}} \qquad \frac{}{\texttt{True} : \texttt{Bool}} \qquad \frac{}{\texttt{False} : \texttt{Bool}}$$

$$\frac{e_1 : \sigma_1 \quad e_2 : \sigma_2}{\texttt{(}e_1\texttt{,}e_2\texttt{)} : \texttt{(}\sigma_1\texttt{,}\sigma_2\texttt{)}} \qquad \frac{p : \texttt{(}\sigma_1\texttt{,}\sigma_2\texttt{)}}{\texttt{fst}\ p : \sigma_1} \qquad \frac{p : \texttt{(}\sigma_1\texttt{,}\sigma_2\texttt{)}}{\texttt{snd}\ p : \sigma_2}$$

$$\frac{e_1 : \texttt{Integer} \quad e_2 : \texttt{Integer}}{e_1 \oplus e_2 : \texttt{Integer}} \qquad \frac{e_1 : \texttt{Integer} \quad e_2 : \texttt{Integer}}{e_1 \odot e_2 : \texttt{Bool}} \qquad \frac{e_1 : \texttt{Integer} \quad e_2 : \texttt{Integer}}{e_1\ \texttt{/}\ e_2 : \texttt{(Integer,Integer)}}$$

$$\frac{b : \texttt{Bool}}{\texttt{!}\ b : \texttt{Bool}} \qquad \frac{b_1 : \texttt{Bool} \quad b_2 : \texttt{Bool}}{b_1 \otimes b_2 : \texttt{Bool}} \qquad \frac{b : \texttt{Bool} \quad e_1 : \sigma \quad e_2 : \sigma}{b\ \texttt{?}\ e_1\ \texttt{:}\ e_2 : \sigma}$$

Symbol $\oplus$ oznacza dowolny z operatorów `+`, `-`, `*`, symbol $\odot$ oznacza dowolny z operatorów `<`, `<=`, `>`, `>=`, `!=`, `==`, a symbol $\otimes$ oznacza dowolny z operatorów `&&` i `||`.

Rysunek 1: Reguły typowania wyrażeń kalkulatora

```
interpreter "2*7 > fst(6/3) && 3 < 5 : snd(15/12)+3 ? 7*8"
```

(widać, że język należałoby znacznie rozbudować, żeby był użyteczny, ale nie o użyteczność, tylko o ideę tu chodzi).

W powyższym przykładzie wyrażenie w języku kalkulatora jest wstawione do programu Haskellowego w postaci napisu. Kompilator zaakceptuje dowolny napis, również zawierający błędy składniowe lub typowe, które zostaną wykryte dopiero przez funkcję `interpreter` podczas biegu programu. Lepszym rozwiązaniem byłoby zastąpienie w programie Haskellowym składni konkretnej wyrażeń kalkulatora składnią abstrakcyjną, np. tak:

```
infixl 6 :*
infixl 5 :+, :-
data Expr = C Integer | (:+) Expr Expr | (:-) Expr Expr | (:*) Expr Expr | ...
```

Zamiast `"2 + 3 * 7" :: String` moglibyśmy napisać

```
C 2 :+ C 3 :* C 7 :: Expr
```

Składnia niewiele straciła na czytelności, a kompilator Haskella sprawdza teraz poprawność składniową wyrażenia. Niestety w powyższej składni abstrakcyjnej wszystkie wyrażenia, niezależnie od swojego typu, są reprezentowane przez wartości typu `Expr`, nie jest więc możliwa kontrola poprawności typowej. Aby odróżnić skończoną liczbę typów moglibyśmy zdefiniować osobne typy danych, np. `BoolExpr`, `IntegerExpr` itd. Niestety w języku kalkulatora typów jest nieskończenie wiele. Możemy sparametryzować typ `Expr` typem wyrażenia. Chcielibyśmy np. żeby `C True :: Expr Bool`, a

```
(:+) :: Expr Integer -> Expr Integer -> Expr Integer
```

Argumenty i wynik konstruktora `(:+)` mają zawężony typ. Definicja typu `Expr a` będzie więc zawierać rekursję niejednorodną, którą rozważaliśmy w zadaniach z poprzedniej listy. Tym razem jednak niejednorodność dotyczy nie tylko argumentów, ale też *wyniku* konstruktora `(:+)`. Używana przez nas do tej pory deklaracja `data` nie pozwala na taką niejednorodność. Haskell został zatem rozszerzony o dodatkową, ogólniejszą deklarację `data`, w której wymienia się nie same typy argumentów, tylko kompletne typy definiowanych konstruktorów:

```
data Expr a where
   C :: a -> Expr a
   P :: (Expr a, Expr b) -> Expr (a,b)
   Not :: Expr Bool -> Expr Bool
   (:+), (:-), (:*) :: Expr Integer -> Expr Integer -> Expr Integer
   (:/) :: Expr Integer -> Expr Integer -> Expr (Integer,Integer)
   (:<), (:>), (:<=), (:>=), (:!=), (:==)
```

```
    :: Expr Integer -> Expr Integer -> Expr Bool
  (:&&), (:||) :: Expr Bool -> Expr Bool -> Expr Bool
  (:?) :: Expr Bool -> Expr a -> Expr a -> Expr a
  Fst :: Expr (a,b) -> Expr a
  Snd :: Expr (a,b) -> Expr b
```

Przypomnijmy, że identyfikatory symboliczne będące nazwami konstruktorów muszą rozpoczynać się znakiem `:`. W zapisie wyrażeń zawierających trójargumentowy operator `:?` możemy się posłużyć operatorem `$` w celu oddzielenia trzeciego argumentu, pisząc np. $b$ `:?` $e_1$ `$` $e_2$. Daje to w Haskellu namiastkę ternarnego operatora miksfiksowego. Należy jeszcze dodać dyrektywy ustalające priorytety operatorów:

```
infixl 6 :*, :/
infixl 5 :+, :-
infixl 4 :<, :>, :<=, :>=, :!=, :==
infixl 3 :&&
infixl 2 :||
infixl 1 :?
```

Wyrażenie z wcześniejszego przykładu zapisane w naszym osadzonym DSL-u wygląda następująco:

```
C 2 :* C 7 :> Fst (C 6 :/ C 3) :&& C 3 :< C 5 :?
  Snd (C 15 :/ C 12) :+ C 3 $ C 7 :* C 8
```

i ma typ `Expr Integer`.

Ponieważ typy wprowadzone deklaracją `data` nazywa się *algebraicznymi typami danych* (bo ich deklaracja, to w istocie definicja algebry wolnej o podanej sygnaturze), więc typy wprowadzone powyższą deklaracją nazwano *uogólnionymi algebraicznymi typami danych* (*Generalized Algebraic Data Types*, *GADT*).

Zaprogramuj ewaluator wyrażeń języka kalkulatora, tj. funkcję

```
eval :: Expr a -> a
```

*Uwaga:* zaprogramowanie funkcji `eval` nie wymaga rozwiązania jakichkolwiek problemów, jest śmiesznie łatwe, oczywiste, wręcz nudne. Prawdziwym celem zadania nie jest rozwój umiejętności programistycznych, tylko dostarczenie Rozwiązującemu ciekawego przeżycia emocjonalnego — podczas zapisywania funkcji `eval` (która jest w istocie izomorfizmem pomiędzy naszym językiem kalkulatora i niewielkim podzbiorem Haskella) cały czas ma się wrażenie, że ta funkcja nie ma prawa się skompilować, gdyż zawiera błędy typowe — prawe strony różnych klauzul są różnych, nieuzgadnialnych typów. GADT, wbrew pozorom, są potężnym rozszerzeniem możliwości języka!