Programowanie Funkcyjne 2020

Lista zadań nr 9 dla grup mabi, mbu, ppo i efes

Na zajęcia 15 i 16 grudnia 2020

Na tej liście zadań będziemy intensywnie używać algebraicznych typów danych do konstruowania poprawnych programów. Niestety Haskell w domyślnej konfiguracji nie ostrzega użytkownika o niewyczerpującym dopasowaniu wzorca. Takie ostrzeżenia będą jednak pomocne na tej liście zadań, więc należy je włączyć uruchamiając kompilator bądź interpreter z flagą `-Wincomplete-patterns`.

Zadanie 1 (4 pkt). Kwantyfikowane formuły Boole'owskie (*Quantified Boolean Formulas*, QBF) to formuły logiczne, które oprócz zmiennych zdaniowych i klasycznych spójników logicznych zawierają kwantyfikatory, które wiążą zmienne zdaniowe. Dla uproszczenia przyjmijmy następującą gramatykę QBF, wyrażoną jako algebraiczny typ danych.

```
module QBF where

type Var = String
data Formula
  = Var Var             -- zmienne zdaniowe
  | Bot                 -- spójnik fałszu (⊥)
  | Not Formula         -- negacja (¬φ)
  | And Formula Formula -- koniunkcja (φ ∧ ψ)
  | All Var     Formula -- kwantyfikacja uniwersalna (∀p.φ)
```

Na przykład $(\forall p.p)$ reprezentowane jest jako `All "p" (Var "p")` i jest formułą, która jest fałszywa, natomiast formuła $(\forall p.\neg\forall q.\neg(\neg(p \wedge q) \wedge \neg(\neg p \wedge \neg q)))$ jest prawdziwa.

Napisz funkcję `isTrue :: Formula -> Bool` sprawdzającą czy podana formuła jest prawdziwa. Możesz założyć, że formuła wejściowa jest zamknięta, tj. nie ma zmiennych wolnych. Jednak gdy zejdziemy pod kwantyfikator, to mogą pojawić się zmienne wolne. W tym celu przyda się pomocnicza funkcja, która ewaluuje formułę w podanym wartościowaniu/środowisku:

```
type Env = Var -> Bool
eval :: Env -> Formula -> Bool
```

Jako, że pracujemy w logice klasycznej, to każda zmienna zdaniowa może być prawdziwa albo fałszywa. Zatem obsłużenie kwantyfikatora jest proste: wystarczy sprawdzić wszystkie dwa przypadki.

Zadanie 2 (4 pkt). Implementacja QBFa z poprzedniego zadania nie jest w pełni zadowalająca, bo nie wymuszamy na użytkowniku, by zawsze podawał zamknięte formuły. Jedyne co możemy zrobić, to zgłosić błąd, gdy napotkamy na zmienną wolną, ale przez leniwość Haskella, może się to nigdy nie wydarzyć. Problem ten możemy rozwiązać przy pomocy *nieregularnych typów danych*, a dokładniej używając techniki reprezentowania wiązania zmiennych za pomocą zagnieżdżonych typów danych (*nested datatypes*). Idea polega na tym, że typ formuł jest sparametryzowany typem używanym do reprezentacji zmiennych wolnych.

```
module NestedQBF where
import Data.Void

data Formula v
  = Var v
  | Bot
  | Not (Formula v)
```

```
  | And (Formula v) (Formula v)
  | All (Formula (Inc v))
```

W bibliotece standardowej znajduje się typ `Void`, który nie ma żadnych elementów. W takim razie typ `Formula Void` opisuje zamknięte formuły — gdy napotkamy zmienną reprezentowaną przez typ `Void`, możemy użyć funkcji `absurd :: Void -> a` by przekonać kompilator, że taka sytuacja jest niemożliwa. Jednak co zrobić z podformułami, które znajdują się pod kwantyfikatorem, skoro mogą one mieć zmienne wolne? Do tego służy następujący typ

```
data Inc v = Z | S v
```

który każdy element typu `v` owija w konstruktor `S` oraz dodaje jeden nowy element `Z`, który reprezentuje tą zmienną która jest związana przez kwantyfikator. Na przykład formułę $\forall p.p \wedge (\forall q.p \wedge q)$ wyrazimy jako:

```
All (Var Z `And` All (Var (S Z) `And` Var Z))
```

Zauważmy, że argument konstruktora `Var` mówi nam ile kwantyfikatorów `All` musimy „przeskoczyć", aby dojść do tego który wiąże naszą zmienną — tyle co `S`-ów w argumencie!

Napisz teraz funkcję `isTrue :: Formula Void -> Bool`, która sprawdza czy podana formuła jest prawdziwa. Również przyda się funkcja `eval` ze środowiskiem. Jaki typ powinna mieć funkcja `eval`?

Zadanie 3 (4 pkt). Napisz funkcję konwertującą formuły z zadania 1 do formuł z zadania 2.

```
import qualified QBF       as QBF
import qualified NestedQBF as NestedQBF

scopeCheck ::
  (QBF.Var -> Maybe v) -> QBF.Formula -> Maybe (NestedQBF.Formula v)
```

Postaraj się nie używać jawnie konstruktorów typu `Maybe` (również w dopasowaniu wzorca), tylko skorzystać z następujących funkcji bibliotecznych.

```
pure  :: a -> Maybe a
(<$>) :: (a -> b) -> Maybe a -> Maybe b
(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
```

Funkcje te mają ogólniejszy typ niż ten podany w treści zadania. Jeśli zrobisz to zadanie dobrze, to również powinieneś dostać ogólniejszy typ funkcji `scopeCheck`. Jak on będzie wyglądał?

Zadanie 4 (6 pkt). Klasy typów w Haskellu służą do nadawania różnym typom podobnej funkcjonalności: spotkaliśmy się już z klasami `Show a`, dostarczającą funkcję `show :: a -> String` pozwalającą uzyskać reprezentację naszego typu jako napis, czy `Num a`, która dostarcza całą gamę funkcji których oczekujemy od typów liczbowych.

W tym zadaniu zajmiemy się standardową klasą `Functor t`, która dostarcza dla danego *konstruktora typów* `t` funkcję `fmap` będącą uogólnieniem funkcji `map` dla list:

```
class Functor t where
  fmap :: (a -> b) -> t a -> t b
```

Dla danego typu (lub, jak tu, konstruktora typu) możemy zadeklarować implementację naszej klasy używając polecenia `instance`:

```
instance Functor Inc where
  fmap f Z = Z
  fmap f (S x) = S (f x)
```

Jak widać, mając funkcję `f :: a -> b`, możemy zastosować ją do naszego typu indeksów `Inc a` przez rozważenie przypadków: jeśli widzimy konstruktor `Z`, zwyczajnie go zwracamy; jeśli zaś widzimy `S x`, to `x :: a`, więc możemy do niego zaaplikować funkcję `f` i opakować wynik konstruktorem `S`.

- Zdefiniuj instancję klasy `Functor` dla typu `Formula` z Zadania 2.

- Zdefiniuj operację *podstawienia* formuły za zmienną w QBFach reprezentowanych przez typ `Formula` z Zadania 2; może ona wykorzystać funkcję `fmap` z poprzedniego punktu, choć nie jest to konieczne. Jaki będzie jej typ?
- Zdefiniuj funkcję `isTrue :: NestedQBF.Formula Void -> Bool`, która zamiast tworzyć środowiska korzysta z podstawień zdefiniowanych powyżej.

Zadanie 5 (6 pkt). Inną klasą typów, którą możemy rozważyć[1] jest klasa reprezentująca typy skończone. Możemy ją zdefiniować następująco:

```
class Finite a where
  elems :: [a]
  index :: a -> Integer
```

Klasa ta daje nam dwie metody: `elems`, zwracającą listę wszystkich elementów danego typu (bez powtórzeń!) i `index`, zwracającą indeks danego elementu na liście `elems`. Zakładamy, że lista `elems` zawsze będzie skończona.

- Zdefiniuj implementację klasy `Finite` dla typu `Bool`.
- Zdefiniuj implementację klasy `Finite` dla typu `Maybe a`, pod warunkiem że `a` jest typem skończonym. Użyj w tym celu następującej deklaracji instancji:

  ```
  instance Finite a => Finite (Maybe a) where
    ...
  ```

- (trudniejsze) Zdefiniuj implementację klasy `Finite` dla par złożonych z typów skończonych — w tym celu wymagamy żeby obydwa typy w parze były skończone:

  ```
  instance (Finite a, Finite b) => Finite (a, b) where
    ...
  ```

- (trudne!) Zdefiniuj implementację klasy `Finite` dla funkcji z typów skończonych w typy skończone:

  ```
  instance (Finite a, Finite b) => Finite (a -> b) where
    ...
  ```

Ostatnie dwa podpunkty mogą wymagać włączenia rozszerzenia `ScopedTypeVariables`, które daje możliwości podobne do typów lokalnie abstrakcyjnych w OCamlu, przez dodanie na początku pliku komendy

```
{-# LANGUAGE ScopedTypeVariables #-}
```

Zwróć uwagę na podobieństwo tych definicji do funktorów, które implementowaliśmy w OCamlu dla typów uporządkowanych. Jakie zalety/wady poszczególnych podejść widzisz?

[1] W przeciwieństwie do klasy `Functor` nie jest ona klasą biblioteczną — użyj definicji poniżej.