Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 8

Do zgłoszenia w SKOS-ie do 4 maja 2020

DataKinds

Typy indeksowane typami i GADT-y omawialiśmy już na ostatnim wykładzie. Klasycznym przykładem tego są *wektory* czyli listy indeksowane długością:

```
data Zero
data Succ n

data Vec :: * -> * -> * where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)
```

Rozszerzenie `DataKinds` pozwala nam nie musieć udawać algebraicznego typu danych na poziomie typów przy pomocy innych typów i automatycznie *promować* typy do *kindów*. Możemy więc zdefiniować:

```
data Nat = Zero | Succ Nat
```

Dzięki rozszerzeniu `DataKinds` nasz świat kindów, do tej pory złożony z `*` i rzeczy konstruowanych przy pomocy `->`, rozbudował się o kind `Nat`, który zawiera dwa „konstruktory typów”: `Zero` i `Succ`. To nie są prawdziwe typy (czyli rzeczy kindu `*`) i nie można np. poprosić o

```
undefined :: Maybe Zero
```

bo `Maybe` ma kind `* -> *`, a nie `Nat -> *`. Daje nam to „typowanie” na poziomie typów. Wcześniej można było skonstruować typ `Vec Int String`, problem był dopiero, gdy próbowaliśmy skonstruować wartość tego typu inną niż `undefined`. Teraz definiujemy:

```
data Vec :: Nat -> * -> * where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)
```

i sama próba skonstruowania typu `Vec Int Int` skończy się błędem kindu (bo `Int :: *` a czekamy na argument kindu `Nat`).

Rodziny typów

Zamknięta rodzina typów to funkcja na poziomie typów. Można w niej robić pattern matching na typach i rekursję. Np. dodawanie można zdefiniować jak poniżej, niezależnie od tego, czy użyliśmy `DataKinds` czy ręcznie zdefiniowaliśmy typy `Zero :: *` i `Succ :: * -> *`:

```
type family Add x y where
  Add Zero     k = k
  Add (Succ n) k = Succ (Add n k)
```

A z rozszerzeniem `TypeOperators` możemy napisać tak:

```
type family x + y where
  Zero   + k = k
  Succ n + k = Succ (n + k)
```

Pozwala nam to zdefiniować konkatenację dwóch wektorów:

```
vappend :: Vec a n -> Vec a k -> Vec a (n + k)
vappend VNil        w = w
vappend (VCons a v) w = VCons a (vappend v w)
```

Proszę zwrócić uwagę, jak definicja funkcji `vappend` współgra z rekurencyjną strukturą dodawania i kompilator akceptuje taką definicję. Nie ma tak lekko, jeśli spróbujemy zrobić odwracanie wektora, nawet w prostszej, naiwnej wersji:

```
vreverse :: Vec a n -> Vec a n
vreverse VNil = VNil
vreverse (VCons a v) = vreverse v `vappend` VCons a VNil
```

Niestety, taka próba definicji skończy się błędem kompilacji:

```
Could not deduce: (n1 + 'Succ 'Zero) ~ 'Succ n1
```

Niestety, kompilator nie chce za nas robić matematyki i nie wie, że

$$\forall n \in \mathbb{N}.\ n + 1 = 1 + n$$

(dodajmy, że `t ~ r` to GHC-owe oznaczenie równości na typach, do czego za chwilę wrócimy). Można obejść problem, każąc kompilatorowi liczyć elementy w innej kolejności. Najpierw definiujemy dołączanie elementu na koniec:

```
snoc :: Vec a n -> a -> Vec a (Succ n)
snoc VNil        a = VCons a VNil
snoc (VCons b v) a = VCons b (snoc v a)
```

Przykładowo:[1]

```
> VCons 1 (VCons 2 (VCons 3 VNil)) `snoc` 4
VCons 1 (VCons 2 (VCons 3 (VCons 4 VNil)))
```

Teraz już można:

```
vreverse :: Vec a n -> Vec a n
vreverse VNil = VNil
vreverse (VCons a v) = vreverse v `snoc` a
```

Naprawdę bardzo dobrym ćwiczeniem jest postarać się zrozumieć, czemu kompilator odrzuca poprzednią definicję, a ta jest Ok.

Równość na typach

W Haskellu mamy dostępny *constraint* mówiący, że dwa typy są równe. Oznaczamy go `~`. Możemy go użyć do zdefiniowania funkcji, która odwraca wektory tylko parzystej długości. Próba odwrócenia wektora długości nieparzystej kończy się błędem typów. Proszę zwrócić uwagę, że korzystamy z promocji typu `Bool`:

```
type family Even n where
  Even Zero            = True
  Even (Succ Zero)     = False
  Even (Succ (Succ n)) = Even n

vreverseEven :: (Even n ~ True) => Vec a n -> Vec a n
vreverseEven = vreverse
```

[1] Automatyczne wyprowadzenie instancji klasy `Show` dla wektorów (i innych GADT-ów) jest trochę upierdliwe, bo nie wystarczy napisać `deriving (Show)` pod definicją typu. Trzeba użyć rozszerzenia `StandaloneDeriving` i napisać `deriving instance Show a => Show (Vec a n)` jako osobny wiersz programu. Na szczęście kompilator sam podpowiada, które rozszerzenia włączyć, inaczej programista spędzałby całe godziny na przeszukiwaniu dokumentacji.

Np.

```
> vreverseEven (VCons 1 (VCons 2 VNil))
VCons 2 (VCons 1 VNil)
```

Ale za to zapytanie

```
> vreverseEven (VCons 1 (VCons 2 (VCons 3 VNil)))
```

kończy się błędem typu

```
Couldn't match type ''False' with ''True'
```

Własne komunikaty o błędach typów

Jednym z ciekawych zastosowań rodzin typów jest możliwość generowania własnych komunikatów o błędach typów, które często lepiej opisują sytuację niż generyczne „nie mogę zunifikować `t1` z `t2`". Do tego przydaje się zaimportować

```
import GHC.TypeLits (TypeError, ErrorMessage(Text))
```

(czasem też trzeba włączyć rozszerzenie `UndecidableInstances`). Np.

```
type family Even n where
  Even Zero           = True
  Even (Succ Zero)     = TypeError (Text "Funkcja vererseEven dziala tylko dla \
                                  \wektorow parzysej dlugosci")
  Even (Succ (Succ n)) = Even n
```

Wówczas próba wywołania

```
> vreverseEven (VCons 1 (VCons 2 (VCons 3 VNil)))
```

kończy się błędem

```
• Funkcja vreverseEven dziala tylko dla wektorow parzysej dlugosci
• In the expression:
    vreverseEven (VCons 1 (VCons 2 (VCons 3 VNil)))
```

Przykład: drzewa lewicowe

Drzewo lewicowe to kolejny rodzaj drzewa binarnego użytecznego do reprezentowania kopców. Przez *rangę* rozumiemy długość prawego kręgosłupa drzewa. Niezmiennik to: W każdym wierzchołku ranga prawego poddrzewa jest niewiększa niż ranga lewego poddrzewa. Niezmiennik ten można wyrazić używając rodzin typów tak:

```
type family OkLeft l r where
  OkLeft n        n = True
  OkLeft (Succ n) m = OkLeft n m
  OkLeft _        _ = TypeError (Text "Leftist tree invariant not satisfied!")

data TreeL :: Nat -> * where
  LeafL :: TreeL Zero
  NodeL :: (OkLeft lrank rrank ~ True)
        => TreeL lrank -> TreeL rrank -> TreeL (Succ rrank)
```

Ponieważ użytkownika modułu mało obchodzi sam indeks, możemy go zamknąć egzystencjalnie (używając rozszerzenia `ExistentialQuantification`) tak:

```
data LeftistTree = forall n. LeftistTree (TreeL n)
```

Singletony

Singletony to typy, które mają tylko jedną wartość (poza `undefined`). Po co komu takie głupie typy? Rozważmy funkcję `replicate :: Int -> a -> [a]`, która bierze element i tworzy listę zawierającą go n razy, np. `replicate 5 'a'` daje nam `"aaaaa"`. Jak otypować jej indeksowaną wersję? W Agdzie mamy typy zależne i moglibyśmy napisać tak:

```
replicate : (n : Int) -> a -> Vec a n
```

Ale w Haskellu typy nie mogą zależeć od wartości, jedynie od innych typów ☹. Ale jeśli typ ma tylko jedną wartość, to zależenie od tego typu to to samo, co zależenie od tej wartości! ☺

Przykładowo, możemy zdefiniować singletona dla każdego typu w rodzinie `Nat`:

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSucc :: SNat n -> SNat (Succ n)
```

I już umiemy zdefiniować `replicate`:

```
vreplicate :: SNat n -> a -> Vec a n
vreplicate SZero     _ = VNil
vreplicate (SSucc n) a = VCons a (vreplicate n a)
```

Co więcej, singletony pozwalają nam dowodzić (!) rzeczy o typach. Zdefiniujmy własną równość na typach:

```
data Equal t s where
  Refl :: Equal a a
```

Możemy dowieść twierdzenie, że $\forall n \in \mathbb{N}.\ n + 1 = 1 + n$ przez indukcję (czyli rekursję):

```
type One = Succ Zero

xPlusOne :: SNat x -> Equal (x + One) (Succ x)
xPlusOne SZero = Refl
xPlusOne (SSucc n) = case xPlusOne n of Refl -> Refl
```

Teraz za każdym razem, gdy robimy dopasowanie wzorca na `xPlusOne x`, do kontekstu trafia informacja, że `(x + One) ~ Succ x`, a to type-checkerowi wystarczy, żeby otypować naiwną wersję funkcji `vreverse`:

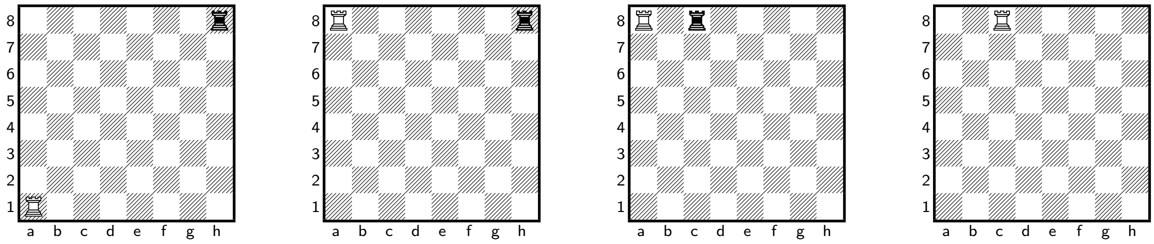
```
vlength :: Vec a n -> SNat n
vlength VNil        = SZero
vlength (VCons _ v) = SSucc (vlength v)

vreverse' :: Vec a n -> Vec a n
vreverse' VNil        = VNil
vreverse' (VCons x v) = case xPlusOne (vlength v) of
                          Refl -> vreverse' v `vappend` VCons x VNil
```

Zadania

Zadanie 1 (1 pkt). Zdefiniuj pełne drzewo binarne. Wymuś niezmiennik pełności przez dodanie indeksu, czyli dodatkowego argumentu w typie drzewa kindu `Nat`, który mówi, jaka jest wysokość drzewa.

Zadanie 2 (1 pkt). Zdefiniuj typ macierzy o wymiarach $n \times m$, gdzie n i m to indeksy kindu `Nat`. Zdefiniuj dobrze typowane (względem wymiaru) mnożenie i dodawanie macierzy.

Zadanie 3 (3 pkt). Zdefiniuj typ danych reprezentujący przebieg partii szachowej, który pozwala zapisać tylko partie zgodne z zasadami gry. Dla uproszczenia przyjmij, że na szachownicy są tylko dwie figury: biała wieża znajdująca się na polu A1 i czarna na polu H8. Jeśli jedna wieża zbije drugą, patia się kończy. Dla przykładu, rozważmy poniższą rozgrywkę:

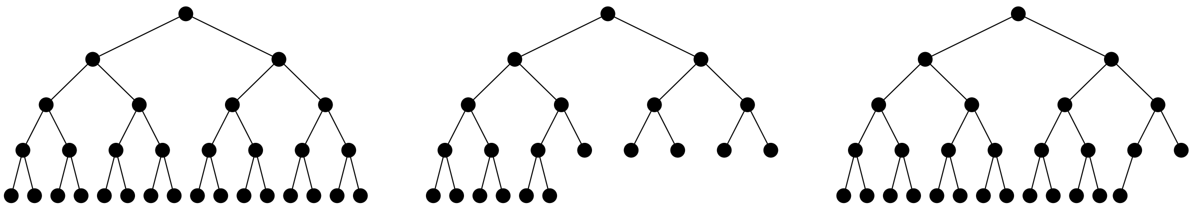


Bez niezmienników zapisalibyśmy ją np. jako

```
[('a',8), ('c',8), ('c',8)]
```

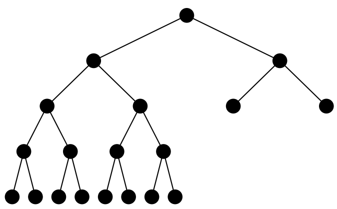
Jako przykład pokaż jak Twoja struktura danych reprezentuje powyższą partię.

Zadanie 4 (3 pkt). *Drzewo z uskokiem* to drzewo, które jest prawie binarne i prawie pełne. Znaczy to tyle, że liście znajdują się w nim na wysokości h lub $h-1$ z tym, że wszystkie na wysokości $h-1$ są na prawo od tych na wysokości h. Przykładowo:

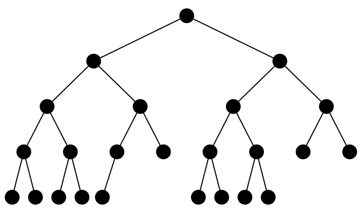


Jak widać na ilustracji, w niektórych takich drzewach znajduje się węzeł (ale zawsze co najwyżej jeden), który ma jedno poddrzewo. Ta struktura zanana jest z AiSD jako abstrakcyjna postać implementacji kopca przy użyciu tablicy.

Zaimplementuj strukturę danych, która reprezentuje takie drzewo. Niech ma ono trzy konstruktory: dla liścia, węzła unarnego i węzła binarnego. Niech drzewo będzie etykietowane, z etykietami i w liściach, i węzłach. Użyj typów fantomowych i rodzin typów, żeby wymusić niezmiennik kształtu. Użyj konstrukcji `TypeError`, żeby poinformować użytkownika o tym, kiedy próbuje stworzyć nieprawidłowe drzewo. Przykładowe możliwe komunikaty to:



`Error: brak zbalansowania`

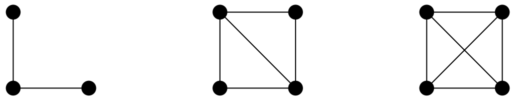


`Error: za dużo uskoków w drzewie`

Do konstruktorów węzłów dodaj także wartość mówiącą, w którym poddrzewie znajduje się uskok i zaimplementuj operację kopca `findMin`, `removeMin` i `insert`.

Zadanie 5 (4+4 pkt). Zdefiniuj typ danych `Graph` służący do reprezentowania grafów skończonych. (Wskazówka: np. lista wierzchołków i lista krawędzi.) Zdefiniuj typ `Graph3Col`, który reprezentuje grafy 3-kolorowalne poprzez odpowiednio indeksowane typy danych. Zdefiniuj funkcję `colorGraph :: Graph`

`-> Maybe Graph3Col` taką, że jeśli graf jest 3-kolorowalny, to wynikiem jest jego reprezentacja w typie `Graph3Col`, a jeśli nie jest, wynikiem jest `Nothing`. Zwróćmy uwagę, że sam typ funkcji gwarantuje jej (częściową) poprawność! Przetestuj na poniższych grafach:



Uwaga: 4 punkty są za reprezentację grafów (razem z reprezentacją 3-kolorowalnych przykładów z powyższej ilustracji), a 4 punkty za funkcję `colorGraph`. Autor zadania zrobił tę pierwszą część, a drugiej nie próbował, więc trudno mu oszacować, czy się w ogóle da i jaki jest wymagany nakład pracy. Dodajmy, że Autorowi przydały się także singletony i rozszerzenia `ExistentialQuantification` (żeby zapakować indeksowany graf w typie `Graph3Col`) i `PolyKinds` (żeby ta kwantyfikacja była nie po kindzie `*` tylko po kindach wypromowanych z typów reprezentujących dowody prawidłowości kolorowania).