Kurs języka Haskell

Notatki zamiast wykładu i lista zadań na pracownię nr 11

Do zgłoszenia w SKOS-ie do 15 czerwca 2020

Wykorzystanie kwantyfikacji uniwersalnej i egzystencjalnej (cz. 2)

Zadanie 1 (1 pkt). W zeszłym tygodniu omawialiśmy strukturę danych `CoYoneda`, dzisiaj zapoznamy się z alternatywnym sposobem przetwarzania konstruktora typu w funktor, czyli strukturą danych `Yoneda`. Dana jest ona przez następującą definicję:

```
data Yoneda f a = Yoneda (forall x. (a -> x) -> f x)
```

Przeczytaj polecaną na SKOS-ie literaturę i uzupełnij:

```
instance Functor (Yoneda f) where
  fmap = ...

toYoneda :: (Functor f) => f a -> Yoneda f a
toYoneda = ...

fromYoneda :: Yoneda f a -> f a
fromYoneda = ...
```

Zadanie 2 (1 pkt). Powróćmy do naszego zeszłotygodniowego przykładu ze strukturą list różnicowych:

```
newtype DList a = DList { fromDList :: [a] -> [a] }

instance Semigroup (DList a) where
  DList f <> DList g = DList $ f . g
```

Listy te mają być prostą alternatywą dla zwykłych list, umożliwiającą konkatenację bez ciągłego kopiowania lewej strony `++`. Ale mają bardzo przykrą wadę: nie chcą być funktorem. Zmieniliśmy więc je w funktor używając coYonedy, ale mamy mały problem, gdy chcemy zdefiniować konkatenację:

```
instance Semigroup (CoYoneda DList a) where
  h <> j = ???
```

Nie wydaje się, by istniał inny sposób niż konwersja do zwykłęj listy, konkatenacja i konwersja z powrotem do coYonedy (a może istnieje – zastanów się nad tym). Ten sposób jednak sprawia, że tracimy wsystkie korzyści płynące z użycia list różnicowych. Spróbujmy więc użyć Yonedy. Uzupełnij:

```
repDList :: DList a -> Yoneda DList a
repDList = ...

repList :: [a] -> Yoneda DList a
repList = ...

instance Semigroup (Yoneda DList a) where
  h <> j = ...
```

Zwróć uwagę, że skoro `DList` nie jest funktorem, konwersja do `Yoneda DList` nie jest bezbolesna. Czy przez tę konwersję na pewno nie tracimy pożądanych własności list różnicowych? Uzasadnij w komentarzu.

W tym zadaniu przyda się rozszerzenie `FlexibleInstances`.

Zadanie 3 (2 pkt). Zapewne nie uległo Twojej uwadze, że w rozwiązaniu poprzedniego zadania definicja operatora `<>` dla typu `Yoneda DList a` tak naprawdę nie zależy od tego, że to są `DList`-y i aż prosi się to uogólnić na coś w stylu:

```
instance (Semigroup f) => Semigroup (Yoneda f a) where ...
```

Ale tak nie można, bo `f` nie jest kindu `*`, więc napis `Semigroup f` jest źle typowany (kindowany) i odrzucany przez kompilator. Możemy też spróbować tak:

```
instance (Semigroup (f a)) => Semigroup (Yoneda f a) where ...
```

Ta wersja też nie zostanie przez kompilator przyjęta, bo nie wystarczy nam wiedzieć, że `f a` jest półgrupą. Żeby rozwiązanie przeszło, musimy wiedzieć, że `f x` jest półgrupą dla każdego typu `x`. Ale nie możemy niestety napisać tak:

```
instance (Semigroup (f x)) => Semigroup (Yoneda f a) where ...
```

System inferencji klas traktuje zmienne wolne jako skwantyfikowane uniwersalnie, więc skoro zmienna wolna `x` występuje tylko po lewej stronie `=>`, podczas inferencji jest skolemizowana do jakiegoś `x0`, które nie chce zunifikować się z naszym `x`, którego tak naprawdę potrzebujemy.

Do rozwiązywania dokładnie tego problemu powstało rozszerzenie `QuantifiedConstraints`. Poczytaj o nim i zdefiniuj odpowiednią instancję.

Uwaga! `QuantifiedConstraints` to dość świeża sprawa, dostępna w GHC od wersji ok. 8.6, która nie jest domyślnie instalowana na niektórych dystrybucjach niektórych systemów operacyjnych.[1] Dlatego drugi punkt w tym zadaniu przyznawany jest za stworzenie projektu z użyciem narzędzia Stack, który zadba o to, by skompilować plik przy użyciu odpowiedniej wersji GHC.

Zadanie 4 (2 pkt). Kolejną przydatną abstrakcją jest coś, co nosi nazwę *monada kogęstościowa* (nazwa pochodzi oczywiście od jej teorio-kategoriowych własności, których tu nie będziemy omawiać). Motywacją dla jej użycia w Haskellu jest fakt, że chociaż operator bind (`>>=`) jest łączny z semantycznego punktu widzenia, nie jest łączny z punktu widzenia wydajnościowego: bardzo często jest tak, że łączenie w lewo (czyli `(m >>= f) >>= g`) jest mniej wydajne niż łączenie w prawo (czyli `m >>= (\x -> f x >>= g)`). Najbardziej oczywistym przykładem jest monada wolna, omawiana na zeszłej liście, o której można myśleć, że to typ termów, gdzie bind to podstawienie. Podstawienie musi odbudować całą strukturę (podobnie jak w omawianym na zeszłej liście przykładzie z `fmap`). Więc łańcuszek bindów łączony w lewo za każdym razem musi odbudować strukturę, co produkuje dużo nieużytków i powoduje kwadratową złożoność. Ten sam łańcuszek łączony w prawo działa liniowo i nie produkuje żadnych nieużytków. Niestety, czasem dość trudno przebudować program tak, by wszystkie bindy były łączone w lewo. Z pomocą przychodzi monada kogęstościowa (poniższa implementacja jest też na SKOS-ie):

```
newtype Cod f a = Cod { runCod :: forall x. (a -> f x) -> f x }

instance Functor (Cod f) where
  fmap f (Cod m) = Cod $ \k -> m (\x -> k (f x))

instance Monad (Cod f) where
  return x = Cod $ \k -> k x
  m >>= k  = Cod $ \c -> runCod m (\a -> runCod (k a) c)
```

[1] Np. na Ubuntu autora zadania zainstalowana jest wersja 8.0.2. z 11 stycznia 2017. Autor nie próbował nigdy dochodzić, czemu aż tak stara, bo i tak używa Stacka.

Jeśli `m` jest monadą, to `m` $\cong$ `Cod m`. Zdefiniuj ten izomorfizm w dwie strony:

```
fromCod :: (Monad m) => Cod m a -> m a
fromCod = ...

toCod :: (Monad m) => m a -> Cod m a
toCod = ...
```

(Wskazówka: podążaj za typami, a zrobisz dobrze.)

Monady kogęstościowej używamy w kolejnej próbie zrobienia list z sensowną złożonością konkatenacji. Zainstaluj `Cod m` w klasie `MonadPlus` (o ile jest w niej `m`):

```
instance MonadPlus m => MonadPlus (Cod m) where
  mzero = ...
  mplus = ...
```

Zbadaj, jaka jest złożoność konkatenacji przy użyciu `mplus` dla `Cod []`, użytej np. tak:

```
fromCod $ (((toCod "a" <|> toCod "b") <|> toCod "c") <|> toCod "d") <|> toCod "e"
```

Możesz to zbadać empirycznie (badając np. czas działania programów albo zużycie pamięci używając profilera) albo teoretycznie, rozpisując jak przebiega ewaluacja (dla uproszczenia można założyć gorliwą semantykę). Opisz swoje eksperymenty/wnioskowanie i wnioski w komentarzu.

Teoria kategorii

Zadanie 5 (1 pkt). Według matematycznej definicji kategoria składa się z kolekcji *obiektów* i typowanych *morfizmów* między nimi. Niezbyt istotne jest to czym są obiekty i morfizmy, byle dobrze typowane morfizmy się ze sobą *składały* (czymkolwiek to złożenie by nie było) $\longleftarrow$ właśnie dlatego nikt nigdy nie rozumie teorii kategorii, bo jest bardzo ogólna.

W Haskellu najczęściej przyjmuje się, że obiekty to „wartości” jakiegoś kindu `k`, a morfizmy to wartości jakiegoś typu `t :: k -> k -> *`. Zapisujemy to jako klasę typów:

```
class Category (t :: k -> k -> *) where
  ident :: a `t` a
  comp  :: b `t` c -> a `t` b -> a `t` c
```

Funkcja `comp` to kompozycja morfizmów, a `ident` to morfizm identycznościowy. Wymagamy, by instancje spełniały (niezbyt zaskakujące) równości:

```
ident `comp` f = f
f `comp` ident = f
f `comp` (g `comp` h) = (f `comp` g) `comp` h
```

Ta definicja wymaga rozszerzenia `PolyKinds`, które pozwala konstruktorom typów przyjmować argumenty polimorficzne w kindzie. Przykładowo, jeśli `k` = `*`, to morfizmami mogą być funkcje:

```
instance Category (->) where
  ident = id
  comp  = (.)
```

Innym przykładem jest `k` = `Nat` oraz `t` = `Matrix` (zad. 2 z listy 8), gdzie `ident` to macierz identycznościowa o odpowiednim rozmiarze, a `comp` to mnożenie macierzy.

Przykładem kategorii jest tzw. kategoria Kleislego `https://en.wikipedia.org/wiki/Heinrich_Kleisli`, w której morfizmami są funkcje nieczyste z efektami pochodzącymi z jakieś monady `m`:

```
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }
```

Zainstaluj ten typ w klasie `Category`:

```
instance (Monad m) => Category (Kleisli m) where
  ident = ...
  comp  = ...
```

Zadanie 6 (2 pkt). Monada to monoid w kategorii endofunktorów. W tym zadaniu zaimplementujemy tę często spotykaną (choć nie do końca precyzyjną) definicję.[2].

Zaczynamy od zdefiniowania *kategorii ściśle monoidalnej*:

```
class (Category t) =>
  MonoidalCategory (t    :: k -> k -> *)  -- kategoria
                   (tens :: k -> k -> k)  -- tensor
                   (unit :: k)            -- element neutralny tensora
   | t -> tens, t -> unit where
  bimap :: a `t` a' -> b `t` b' -> tens a b `t` tens a' b'
```

Instancje powinny spełniać następujące równania na typach:

```
tens unit x = x
tens x unit = x
tens (tens x y) z = tens x (tens y z)
```

A także na wartościach:

```
bimap ident ident = ident
bimap f g `t` bimap f' g' = bimap (f `t` f') (g `t` g')
```

Więc kategoria ściśle monoidalna to taka kategoria, której obiekty tworzą coś w rodzaju monoidu, z operacją daną przez bifunktor `tens` i elementem neutralnym przez „wartość” `unit`. W takiej kategorii możemy zdefiniować *monoid*:

```
class (MonoidalCategory t tens unit) =>
    MonoidInCategory (t :: k -> k -> *) tens unit (m :: k) where
  one  :: unit `t` m
  mult :: tens m m `t` m
```

Instancje powinny spełniać:

```
mult . bimap one ident :: tens unit m `t` m
=
ident :: m `t` m

mult . bimap ident one :: tens m unit `t` m
=
ident :: m `t` m

mult . bimap ident mult :: tens m (tens m m) `t` m
=
mult . bimap mult ident :: tens (tens m m) m `t` m
```

Mamy więc trzy struktury monoidalne: kategoria to trochę monoid (zdefiniowany przez łączne składanie morfizmów), ściśle monoidalną kategorię (dającą monoid na obiektach) i monoid w takiej kategorii. Przykładem kategorii ściśle monoidalenj jest kategoria endofunktorów:

[2] Brak precyzji wynika z faktu, że pojęcie monoidu w kategorii ma sens tylko w kategorii monoidalnej, a kategoria endofunktorów jest zazwyczaj monoidalna na kilka sposobów. Na przykład funktory aplikatywne też są monoidami w kategorii endofunktorów, ale wyposażonej w inną strukturę monoidalną.

```
newtype NatTrans f g = NatTrans { runNatTrans :: forall a. f a -> g a }

instance Category NatTrans where
  ident = NatTrans id
  NatTrans f `comp` NatTrans g = NatTrans $ f . g

newtype Identity a = Identity { runIdentity :: a }
  deriving (Functor)

newtype Comp f g x = Comp { runComp :: f (g x) }
  deriving (Functor)

instance MonoidalCategory NatTrans Comp Identity where
  bimap = ...
```

Uzupełnij definicję `bimap` (wskazówka: podążaj za typami).

Uzupełnij definicję przykładowego monoidu:

```
instance MonoidInCategory NatTrans Comp Identity [] where
  one  = ...
  mult = ...
```

Monoidy (czyli instancje klasy `MonoidInCategory`) w tej właśnie kategorii monoidalnej to monady. Pokaż to uzupełniając następujące definicje:

```
newtype MonadFromMonoid m a = MonadFromMonoid (m a) deriving (Functor)

instance (Functor f, MonoidInCategory NatTrans Comp Identity f)
 => Monad (MonadFromMonoid f) where
  return = ...
  (>>=)  = ...
```

A także:

```
newtype MonoidFromMonad m a = MonoidFromMonad (m a) deriving (Functor)

instance (Monad m) => MonoidInCategory NatTrans Comp Identity (MonoidFromMonad m a) where
  one  = ...
  mult = ...
```