Семейства типов & схемы рекурсии

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

11 декабря 2018 г.

Outline

Outline

(Indexed) type families a.k.a. семейства типов

```
{-# LANGUAGE TypeFamilies #-}
```

Type families = data families + type synonym families

- Могут объявляться отдельно
- А можно вместе (*associated*) с классом типов

(Indexed) type families a.k.a. семейства типов

```
{-# LANGUAGE TypeFamilies #-}
```

Type families = data families + type synonym families

- Могут объявляться отдельно
- А можно вместе (*associated*) с классом типов

Type classes – это ad hoc полиморфизм (a.k.a. overloading) для *функций*

Type families – это ad hoc полиморфизм (a.k.a. overloading) для *типов*

Data families

Заявим, что у нас будет a lá список

```
data family XList a
```

и сделаем разную реализацию списка в зависимости от типа элемента.

```
data instance XList Char = XCons Char (XList Char)
                         | XNil

data instance XList () = XListUnit Int
```

- 👍 Семейства типов *расширяемы* (в отличие от GADT)
- 👎 Но, чтобы их поддержать, алгоритм вывода типов надо было выкинуть и переписать

Associated types

Типы, *индексированные* типами.

```
class MapKey k where
  data Map k v
  ...
```

У нас был ad-hoc полимофизм (a.k.a. overloading) для функций, теперь для типов.

Пример. Конечные отображения (1/3)

```
class MapKey k where
  data Map k v

  empty  :: Map k v
  lookup :: k -> Map k v -> v
```

Пример. Конечные отображения (1/3)

```
class MapKey k where
  data Map k v

  empty  :: Map k v
  lookup :: k -> Map k v -> v
```

Сделаем оптимизированное представление, когда ключ – это число

```
instance MapKey Int where
  data Map Int v = MI (Patricia . Dict v)

  empty              = MI Patricia . emptyDict
  lookup k (MI d) = Patricia . lookupDict k d
```

Пример. Конечные отображения (2/3)

Ключ – это одноэлементное множетсво:

```
instance MapKey () where
  data Map () v = MU (Maybe v)

  empty = MU Nothing

  lookup () (MU Nothing)  = error "unknown key"
  lookup () (MU (Just v)) = v
```

Ключ – это пара:

```
instance (MapKey a, MapKey b) => MapKey (a,b) where
  data Map (a,b) v = MP (Map a (Map b v))

  empty                     = MP empty
  lookup (a,b) (MP fm) = lookup b (lookup a fm)
```

Пример. Конечные отображения (3/3)

В зависимости от ключа будем заглядывать в одну или другую “мапу”

```
instance (MapKey a, MapKey b) =>
                 MapKey (Either a b) where

  data Map (Either a b) v = ME (Map a v) (Map b v)

  empty = ME Nothing Nothing

  lookup (Left a)  (ME fm1   _) = lookup a fm1
  lookup (Right b) (ME _   fm2) = lookup b fm2
```

Outline

Схемы рекурсии

При *рекурсивном* преобразовании почти всех структур данных почти всегда видится некоторый шаблон (или *схема*), который не зависит от конкретной структуры данных

Подготовительная информация (1/2)

Напоминание про великий и ужасный Fix

```
data NatF r     = Zero | Succ r
  deriving (Show, Functor)

data ListF a r = Nil  | Cons a r
  deriving (Show, Functor)

data TreeF a r = Empty | Leaf a | Node r r
  deriving (Show, Functor)

type Nat    = Fix NatF
type List a = Fix (ListF a)
type Tree a = Fix (TreeF a)

data Fix f = Fix (f (Fix f))
```

Подготовительная информация (2/2)

Data family для Fix

```
type family Base t :: * -> *
type instance Base (Fix f) = f
```

Класс для "сворачиваемых" типов

```
class Foldable (t :: * -> *) where
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
```

Катаморфизм (catamorphism). Пример для Nat

```
{-# LANGUAGE LambdaCase #-}

natsum :: Nat -> Int
natsum = cata $ \case
  Zero   -> 0
  Succ n -> n + 1

> :t cata
cata :: Foldable t => (Base t b -> b) -> t -> b

> :set -XTypeApplications
> :t cata @Nat
cata @Nat :: (NatF b -> b) -> Nat -> b
```

Первый аргумент включает в себя функцию и начальное состояние foldr'a

Катаморфизм (catamorphism). Пример. Списки

```
data ListF a r = Nil
               | Cons a r
  deriving (Show, Functor)

> :t cata @(List Int)
cata @(List Int)
  :: (ListF Int a -> a) -> List Int -> a
```

Сравните с тем же самым для Nat

```
> :t cata @Nat
cata @Nat :: (NatF a -> a) -> Nat -> a
```

Как связаны cata и foldr?

```
foldr = (a -> b -> b) -> b -> [a] -> b
      ~ b -> (a -> b -> b) -> [a] -> b
      ~ (b, a -> b -> b) -> [a] -> b
      ~ (() -> b, (a, b) -> b) -> [a] -> b
      ~ (Either () (a, b) -> b) -> [a] -> b
      ~ (ListF a b -> b) -> [a] -> b
      = cata @(Fix (ListF a))
```

Катаморфизм могуч!

Фильтрование списка по предикату – это катаморфизм

```
filterL :: (a -> Bool) -> List a -> List a
filterL p = cata alg where
  alg Nil = nil
  alg (Cons x xs)
    | p x       = cons x xs
    | otherwise = xs
```

Внутренняя функция идет от хвоста, если элемент "хороший", то она его оставляет.

Иначе – забывает его сложить в ответ.

```
cons :: a -> List a -> List a
cons x xs = Fix (Cons x xs)

nil :: List a
nil = Fix Nil
```

Параморфизм (paramorphism) (1/2)

```
para :: Foldable t => (Base t (t, a) -> a) -> t -> a
```

Задача: сделать факториал `Nat -> Int` в стиле "свёртки".

```
> :kind! Base Nat (Nat, Int)
Base Nat (Nat, Int) :: *
= NatF (Nat, Int)

natfac :: Nat -> Int
natfac = para alg where
  alg ZeroF = 1
  alg (SuccF (n, f)) = natsum (succ n) * f

natsum :: Nat -> Int
```

Параморфизм (paramorphism) (2/2)

Пример: предыдущее число.

```
natpred :: Nat -> Nat
natpred = para alg where
  alg Zero           = zero
  alg (Succ (n, _)) = n
```

Упражнение: выдать хвост списка через параморфизм

```
tailL :: List a -> List a
tailL = undefined
```

Хиломорфизм (hylomorphism)

```
hylo :: Functor f =>
          (f b -> b) -> (a -> f a) -> a -> b
```

Работает как (корекурсивное) разворачивание, за которым следует (рекурсивное) сворачивание.

Сейчас будет пример:

```
import Data.List.Ordered (merge)

merge :: Ord a => [a] -> [a] -> [a]

data Tree  a r = Empty
               | Leaf a
               | Node r r
  deriving (Show, Functor)
```

Merge Sort

```
mergeSort :: Ord a => [a] -> [a]
mergeSort = hylo alg coalg where
  alg Empty      = []
  alg (Leaf c)   = [c]
  alg (Node l r) = merge l r

  coalg []  = Empty
  coalg [x] = Leaf x
  coalg xs  = Node l r where
    (l, r) = splitAt (length xs `div` 2) xs
```

Merge Sort

```
mergeSort :: Ord a => [a] -> [a]
mergeSort = hylo alg coalg where
  alg Empty      = []
  alg (Leaf c)   = [c]
  alg (Node l r) = merge l r

  coalg []  = Empty
  coalg [x] = Leaf x
  coalg xs  = Node l r where
    (l, r) = splitAt (length xs `div` 2) xs
```

Упражнения: Shell sort, radix sort.

Конец.

Ссылки I


A Mixture of Musings
Associated Types and Haskell
ссылка


Associated Types with Class
ссылка


Origami programming
Jeremy Gibbons
ссылка