Расширения системы типов Haskell

Развитая система типов в Haskell далека от идеала, и существует ряд задач и проблем, которые она стандартным образом решить не может. Для ghc есть полезные расширения, которые помогают для их решения. Но, возможно, в будущем, в новой итерации языка мы увидим важные изменения системы типов, которые кардинально изменят ситуацию со многими такими проблемами.

Многопараметрические классы типов и функциональные зависимости

Многопараметрические классы типов используются при написании класса, который должен быть параметризован для двух (или более) разных типов. *Отметим, что стандартный Haskell нам не позволяет этого сделать*.

Напомним, что есть параллели между системой типов и классов в Haskell и подходами в ООП. Так, классы Haskell похожи на классы в других объектно-ориентированных языках, таких как C++ и Java. Метод класса, определённый в классе Haskell, соответствует виртуальной функции в классе языка C++. Каждое воплощение класса обеспечивает своё собственное определение для каждого метода, умолчания в классе соответствуют реализации по умолчанию виртуальной функции в базовом классе C++. Классы Haskell в грубом приближении похожи на интерфейсы в Java. Как и объявления интерфейсов, объявления классов в Haskell задают протокол использования объектов, а не определение собственно объекта.

Более подробно:

Мягкое введение в Haskell. Часть-1 (сравнение с другими языками)

Однако понятие класса типов с двумя и более параметрами отличается от понятия интерфейса в объектно-ориентированном языке, больше напоминая контракт (договор) между двумя различными типами.

Multi-parameter Type Classes in Haskell

Haskell/Advanced type classes

Начнём с наводящего примера.

Предположим, мы хотим создать класс типов «Collection», который мог бы использоваться с различными конкретными типами данных. Пусть он на первом этапе поддерживает две операции: «вставка» для добавления элементов и предикат «быть элементом» для тестирования членства.

Первая попытка может выглядеть так:

```
class Collection c where
    insert :: e -> c -> c
    member :: e -> c -> Bool

-- Make lists an instance of Collection:
instance Collection [a] where
    insert x xs = x:xs
    member = elem
```

Этот пример не скомпилируется. Проблема в том, что переменная типа `e` взялась из «ниоткуда», и в типе воплощения (экземпляра) `Collection` нет ничего такого, что могло бы сказать нам, что на самом деле представляет собой `e`, поэтому мы никогда не сможем определить реализацию этих методов. Многопараметрические классы типов решают эту проблему, позволяя нам помещать `e` в тип класса. Вот пример, который компилируется и может использоваться:

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}

class Eq e => Collection e c where
    insert :: e -> c -> c
    member :: e -> c -> Bool

instance Eq a => Collection a [a] where
    insert = (:)
    member = elem
```

Можем добавить поддержку для массивов:

```
import Data.Array

-- ...

instance Eq a => Collection a (Array Int a) where
    insert e c = listArray (i,j+1) (e:es)
        where
          es = elems c
          (i,j) = bounds c
    member e c = e `elem` (elems c)
```

Проверяем:

```
*Main> sq :: Array Int Int; sq = listArray (1,10) [i^2 | i <- [1..10]]
*Main> :t sq
sq :: Array Int Int
*Main> (20::Int) `insert` sq
array (1,11) [(1,20),(2,1),(3,4),(4,9),(5,16),(6,25),(7,36),(8,49),
  (9,64),(10,81),(11,100)]
*Main> (20::Int) `member` sq
False
```

Можем добавить поддержку для собственного типа данных, в которых коллекции (возможно, бесконечные) задаются характеристическими функциями:

```
instance Eq a => Collection a (a -> Bool) where
    insert e c = \t -> (t == e || (c t))
    member e c = c e
```

В данном случае, любая функция на каком типе `a`, возвращающая булево значения, однозначно задаёт некоторое подмножество элементов типа, где она возвращает `True`. Тогда, реализация метода `member` означает применение этой характеристической функции (она будет задана переменной `c`) к элементу `e`.

Реализация метода **insert** расширяет действующую характеристическую функцию c: либо это и есть добавляемый элемент e (проверяем равенством), либо в противном случае используем действующую характеристическую функцию.

Проверяем:

```
*Main> col = \x -> (x `mod` 2 == 1)
*Main> col 1
True
*Main> col 3
True
*Main> col 5
True
*Main> col (-1)
True
*Main> col 0
False
*Main> (0 `insert` col) 0
True
*Main> 0 `member` col
False
```

Вроде бы всё работает. Однако, что будет, если мы захотим добавить в класс `Collection` новую функцию, создающую пустую коллекцию:

```
    empty c :: c
```

например, для воплощения списков:

```
    empty  = []
```

Мы получим сообщение системы типов об ошибке! Оказывается, если класс определён для двух переменных типов, то Haskell требует использования их обоих.

Эта проблема решаема, если мы заведём класс-прослойку:

```
class CollE s where
  empty  :: s

class CollE c => Collection e c where
  insert :: e -> c -> c
```

Но есть ещё более трудноуловимые ошибки. Рассмотрим такие две функции:

```
f x y coll = insert x (insert y coll)
g coll = f True 'a' coll
```

Их типы выводятся в следующем виде:

```
f :: (Collection e1 c, Collection e2 c) => e1 -> e2 -> c -> c
g :: (Collection Bool c, Collection Char c) => c -> c
```

Обратите внимание, что сигнатура типа для функции f позволяет параметрам x и y назначаться с различными типами, даже если пытаться вставить каждое из двух значений,

одно за другим, в одну и ту же коллекцию `coll`. Если мы надеемся смоделировать коллекции, содержащие только один тип значения, то это явно неподходящий способ. Что ещё хуже, определение для функции `g` принято, не вызывая ошибку типа. Таким образом, ошибка в этом коде не будет обнаружена в точке определения, но только в точке использования, которая может даже не быть в том же модуле. Очевидно, что мы предпочли бы избежать этих проблем, устраняя неясности, выводя более точные типы и обеспечивая более раннее обнаружение ошибок типов [MPJones, C.5].

Попытка исправить ситуацию

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleInstances #-}

class Collection e c where
  empty  :: c e
  insert :: e -> c e -> c e
  member :: e -> c e -> Bool
```

Фактически, это именно тот подход, который был применен «отцами-основателями» Окасаки и Пейтоном Джонсом для более реалистичных попыток создать библиотеку такого рода.

Ключевым отличием здесь является то, что мы абстрагируемся над конструктором типа `c`, который используется для формирования коллекции типа `c e`, а не над самим типом коллекции, представленной как `c` в оригинальном объявлении класса. Таким образом, `Collection` является примером *класса конструкторов* (мы говорили в Лекции-13), в котором второй параметр является унарным конструктором типа, заменяя нульместный параметр типа `c`, который использовался в исходном определении. Это изменение позволяет избежать непосредственных проблем, о которых мы упоминали выше:

– функция `empty` имеет тип `Collection e c =>c e`, и вполне проходит проверку.

– функция `f` получила более аккуратный тип:

```
f :: (Collection e c) => e -> e -> c e -> c e
```

– функция `g` отбраковывается, как нам и хотелось, с ошибкой типа, так как теперь тип `f` не позволяет два аргумента разного типа.

К сожалению, эта версия класса `Collection` не такая общая, каким был первоначальный класс. Только один из трёх экземпляров, перечисленных в предыдущем разделе, может использоваться с этой версией `Collection`, потому что только один из них — экземпляр для списков — имеет тип коллекции, который можно записать в форме `c e`, для некоторого конструктора типа `c` и элементарного типа `e` [MPJones, C.235].

```
instance Eq a => Collection a [] where
    insert = (:)
    member = elem
    empty  = []
```

Используем функциональную зависимость (Functional Dependency)

Анализ ситуации с коллекциями приводит к выводу, что в общей ситуации по базому типу `e` мы не сможем определить тип коллекции, а вот по типу коллекции `c` мы можем

вывести тип базового типа e. Это соображение мы могли бы передать компилятору для помощи с определением типов.

Добавим к оригинальному классу аннотацию `c -> e`, которая будет означать, что «тип c задаёт уникальным образом тип e».

```
{-# LANGUAGE FunctionalDependencies,FlexibleInstances #-}

class Collection e c | c -> e where
   empty  :: c
   insert :: e -> c -> c
   member :: e -> c -> Bool
```

Практически же, компилятор ограничивает возможные варианты для типа e первым подходящим воплощением. В случае с функцией `empty` компилятор получает «твёрдые гарантии» однозначного определения типа и ошибки неоднозначности уже не возникает.

Теперь мы можем добавить воплощения для различных коллекций:

```
import Data.Array

instance Eq a => Collection a [a] where
    insert = (:)
    member = elem
    empty  = []

instance Eq a => Collection a (a -> Bool) where
    insert e c = \t -> (t == e || (c t))
    member e c = c e
    empty = \_ -> False

instance Eq a => Collection a (Array Int a) where
    insert e c = listArray (i,j+1) (e:es)
        where
          es = elems c
          (i,j) = bounds c
    member e c = e `elem` (elems c)
    empty  =  listArray (0,0) []
```

Даже поддерживается работа с пустым массивом:

```
*Main> ss :: Array Int Int; ss = empty
*Main> :t ss
ss :: Array Int Int

*Main> ss ! 0
*** Exception: (Array.!): undefined array element
*Main> indices ss
[0]
```

Можем теперь ещё добавить поддержку для множеств:

```
import qualified Data.Set as S
```

```
-- ...
instance (Eq a, Ord a)  => Collection a (S.Set a) where
    insert = S.insert
    member = S.member
    empty  = S.empty
```

Проверяем:

```
Prelude Data.Set> False `member` (True `insert` empty)
```

Но вот попытка создания той самой «плохой функции» теперь не проходит на стадии компиляции:

```
funcdep25.hs:46:17: error:
    * Couldn't match expected type `Bool' with actual type `Char'
    * In the second argument of `f', namely 'a'
      In the expression: f True 'a' coll
      In an equation for `g': g coll = f True 'a' coll
   |
46 | g coll = f True 'a' coll

   |                 ^^^
```

Соответственно и `f` имеет более приличную сигнатуру:

```
f :: Collection e c => e -> e -> c -> c
```

которая на стадии компиляции не допустит использования параметров различного типа.

Вопросы однозначности, которые решает установление функциональной зависимости, рассмотрим в дальнейших примерах.

Примеры с арифметикой

В Haskell, как мы знаем, непривычная ситуация с обычной арифметикой чисел и поведением литералов (которые являются перегруженными функциями).

Вот такая ситуация, вполне «обычная для обычных языков», в Haskell вызывает ошибку:

```
Prelude> let x=2::Int; y=3::Integer
Prelude> x+y

<interactive>:2:3: error:
    * Couldn't match expected type `Int' with actual type `Integer'
    * In the second argument of `(+)', namely `y'
      In the expression: x + y
      In an equation for `it': it = x + y
```

Используя многопараметрические классы типов, мы могли бы как-то исправить ситуацию:

```
{-# LANGUAGE MultiParamTypeClasses #-}

module AddF (plus) where
```

```
class Add a b  where
    plus :: a -> b -> Integer

instance Add Int Int where
    plus x y = fromIntegral(x) + fromIntegral(y)

instance Add Integer Integer where
    plus x y = x + y

instance Add Int Integer where
    plus x y = fromIntegral(x) + y

instance Add Integer Int where
    plus x y = x + fromIntegral(y)
```

В этом случае, сложение будет проходить вполне нормально:

```
*AddF> let x=2::Int; y=3::Integer
*AddF> x `plus` y
5
```

И даже будут нормально работать целые литералы:

```
*AddF> 2 `plus` 3
5
```

Но рассмотрим теперь задачу взаимодействия большего числа типов, таких **Int**, **Integer**, **Float**, **Double** и структур с их участием, типа векторов.

Теперь нам нужно включить ряд расширений, главное из которых — `FunctionalDependencies`. Используем зависимости для описания того, что тип `c` выводится из типов `a` и `b`.

```
{-# LANGUAGE FunctionalDependencies, FlexibleInstances,
    UndecidableInstances #-}

module AddF (plus) where

class Add a b c | a b -> c where
    plus :: a -> b -> c
```

Укажем все возможные расширения для целых:

```
-- int integer

instance Add Int Int Int where
    plus x y = x + y

instance Add Integer Integer Integer where
    plus x y = x + y

instance Add Int Integer Integer where
    plus x y = fromIntegral(x) + y

instance Add Integer Int Integer where
```

```
    plus x y = x + fromIntegral(y)
```

Указанная в определении класса зависимость не даёт добавить лишние декларации, вроде такой:

```
instance Add Int Int Integer where
    plus x y = fromIntegral(x) + fromIntegral(y)
```

так как для комбинации **`Int Int`** уже есть декларация (самая первая в списке).

Далее, воплощения для **`Float`**:

```
-- float

instance Add Float Float Float where
    plus x y = x + y

instance Add Int Float Float where
    plus x y = fromIntegral(x) + y

instance Add Integer Float Float where
    plus x y = fromIntegral(x) + y

instance Add Float Int Float where
    plus x y = x + fromIntegral(y)

instance Add Float Integer Float where
    plus x y = x + fromIntegral(y)
```

Далее, воплощения для **`Double`**:

```
-- double

instance Add Double Double Double where
    plus x y = x + y

instance Add Float Double Double where
    plus x y = realToFrac(x) + y

instance Add Double Float Double where
    plus x y = x + realToFrac(y)

instance Add Int Double Double where
    plus x y = fromIntegral(x) + y

instance Add Integer Double Double where
    plus x y = fromIntegral(x) + y

instance Add Double Int Double where
    plus x y = x + fromIntegral(y)

instance Add Double Integer Double where
    plus x y = x + fromIntegral(y)
```

И, наконец, самые интересные воплощения для произвольных структур (подобных векторам):

```
-- structures

instance Add Integer (Integer,Integer) (Integer,Integer) where
   plus x (y1,y2) = (x+y1, x+y2)

instance (Add a b1 c1, Add a b2 c2) => Add a (b1,b2) (c1,c2) where
   plus x (y1,y2) = (x `plus` y1, x `plus` y2)

instance (Add a b c) => Add a [b] [c] where
   plus x [] = []
   plus x (y:ys) = (x `plus` y):( x `plus` ys)
```

Пример работы:

```
*AddF> s1::Int; s1 =4
*AddF> s2::[Int]; s2 = [10..16]
*AddF> s1 `plus` s2
[14,15,16,17,18,19,20]
*AddF> s3::[Double]; s3 = [(-4),0,2.2,5]
*AddF> s1 `plus` s3
[0.0,4.0,6.2,9.0]
*AddF> s4::(Float,Double); s4 = (2,3)
*AddF> s1 `plus` s4
(6.0,7.0)
```

И собственно, для векторов:

```
-- parametrized vector data

data Vec a = Vc {vx,vy,vz :: a} deriving (Eq,Read)

instance (Show a) => Show (Vec a) where
  show (Vc x y z) = "(" ++ (show x) ++ "; "
    ++ (show y) ++ "; " ++ (show z) ++ ")"

instance (Add a a a) =>
 Add (Vec a) (Vec a) (Vec a) where
    u `plus` w =
        Vc
           ((vx u) `plus` (vx w))
           ((vy u) `plus` (vy w))
           ((vz u) `plus` (vz w))
```

И пример работы:

```
*AddF> u::Vec Int; u=Vc 1 2 3
*AddF> u
(1; 2; 3)
*AddF> w::Vec Int; w=Vc 10 20 30
*AddF> u `plus` w
(11; 22; 33)
```

Но, к сожалению, эта расширенная версия библиотеки опять не может работать с литералами:

```
*AddF> 1 `plus` 2

<interactive>:3:1: error:•
     Could not deduce (Add a0 b0 c)
      from the context: (Add a b c, Num a, Num b)
        bound by the inferred type for ‘’it:
                   forall a b c. (Add a b c, Num a, Num b) => c
        at <interactive>:3:1-10
      The type variables ‘’a0, ‘’b0 are ambiguous•
     In the ambiguity check for the inferred type for ‘’it
      To defer the ambiguity check to use sites, enable
  AllowAmbiguousTypes
      When checking the inferred type
        it :: forall a b c. (Add a b c, Num a, Num b) => c
```

И для корректной работы нам по-прежнему нужно аннотировать тип:

```
*AddF> (1::Int) `plus` (2::Integer)
3
```

Литература по технике функциональных зависимостей в Haskell

Базовая статья:

Jones M.P., Type Classes with Functional Dependencies

Официальная документация:

6.8.7. Functional dependencies

Семейства типов (Type families)

Схожую функциональность обеспечивает расширение `TypeFamilies` для так называемых *семейств типов*. У них есть свои преимущества и недостатки, когда-то подробно разбираемые haskell-сообществом:

Type Families (TF) vs Functional Dependencies (FD)

wikipedia: Type family - Comparison with Functional dependencies

где указывается, что семейства типов всё-таки более предпочтительны, являясь более проработанным и надёжным (строгим) инструментом.

Вспомним, с классами типов мы можем связать набор функций из класса типов с типом. Семейства типов позволяют нам связывать типы с типом. Они определяют частичные функции от типов к типам путём сопоставления с образцом во входных типах. Семейства типов — это типы данных с индексом типа и функции имени для типов. Типы данных и функции извлекаются предоставленными типами.

Семейства типов бывают трех разновидностей:

- семейства типов данных (data families)

- открытое семейство синонимов типа (open type synonym families)
- закрытое семейство синонимов типа (closed type synonym families)

An Introduction to Type Families

По способу объявления семейства типов бывают с объявлением на верхнем уровне и внутри классов типов (тогда их называют *ассоциированные типы* — кроме замкнутых семейств синонимов типов (таких не бывает).

В этой лекции мы не будем делать аккуратного введения в возможности и в описание семейств типов. Ограничимся только возможностью переписать наши примеры.

Здесь мы используем так называемое «*открытое семейство синонимов типов с объявлением внутри класса типа (ассоциированный тип)*»

Пример с коллекциями

```
{-# LANGUAGE TypeFamilies, FlexibleInstances  #-}

import qualified Data.Set as S

class Collection c where
    type Elem c -- упрощённаязапись , можноуказать : type family Elem c
    member :: Elem c -> c -> Bool
    empty  :: c
    insert :: Elem c -> c -> c
--    toList :: c -> [Elem c]

instance Eq e => Collection [e] where
    type Elem [e] = e
    empty = []
    insert = (:)
    member = elem
--    toList = id

instance (Eq e, Ord e) => Collection (S.Set e) where
    type Elem (S.Set e) = e
    insert = S.insert
    member = S.member
    empty  = S.empty
--    toList = Set.toList

instance Eq e => Collection (e -> Bool) where
    type Elem (e -> Bool) = e
    insert e c = \t -> (t == e || (c t))
    member e c = c e
    empty = \_ -> False
```

(о возможности опустить ключевое слово `family` в этом случае см. документацию 6.4.9.4. Associated data and type families)

Проверим его работу:

```
*Main> s::(Int->Bool); s=empty
```

```
*Main> 1 `member` (1 `insert` s)
True
*Main> 2 `member` (1 `insert` s)
False
```

Ну и, наконец, небольшая библиотека для работы с арифметикой — для демонстрации возможностей. Здесь мы используем вновь ту же самую технику работы с семействами типов.

```
{-# LANGUAGE MultiParamTypeClasses, TypeFamilies,
    FlexibleInstances #-}

class Add a b where
  type SumType a b
  plus :: a -> b -> SumType a b

instance Add Integer Integer where
    type SumType Integer Integer = Integer
    plus x y = x + y

instance Add Double Double where
    type SumType Double Double = Double
    plus x y = x + y

instance Add Integer (Integer,Integer) where
    type SumType Integer (Integer,Integer) = (Integer,Integer)
    plus x (y1,y2) = (x + y1, x + y2)
```

Литература по технике семейств типов в Haskell

Официальная документация:

6.4.9. Type families

Другие источники:

An Introduction to Type Families

wiki.haskell: GHC/Type families

Oleg Kiselyov, Simon Peyton Jones, Chung-chieh Shan. Fun with type functions

stackoverflow: ‘type family’ vs ‘data family’, in brief?

Functional dependencies versus type families and beyond

Type Families and Pokemon

Экзистенциальные типы

Давайте сначала мы вспомним немного о том, как мы обычно конструируем простые типы в Haskell:

```
data T = Nil | Mk Int
```

и с помощью команды `:t` в ghci мы можем проверить:

```
Nil :: T
Mk  :: Int -> T
```

И если завёдем деконструктор

```
unMk (Mk x) = x
```

то

```
unMk :: T -> Int
```

(с `Nil` работать не будет, но нас сейчас это не интересует)

Теперь, сделаем тип полиморфным, как обычно мы делали, например

```
data T a = Nil | Mk a
-- Nil :: T a
-- Mk  :: a -> T a

unMk :: T a -> a
unMk (Mk x) = x
```

Более привычно нам иметь дело с типом **`Maybe`**:

```
data Maybe a = Nothing | Just a ...
-- Nothing :: Maybe a
-- Just  :: a -> Maybe a

unJust :: Maybe a -> a
unJust (Just x) = x
```

Мы ранее никогда не говорили о том, что при указании типов в Haskell при наличии переменной типа, например, `a` в декларациях вроде такой

```
data Maybe a = Nothing | Just a ...
```

на самом деле тип будет

```
data Maybe a = forall t. Nothing | Just a ...
```

Это, кстати, можно даже прямо указывать в коде при включённой декларации `ExplicitForAll`:

```
{-# LANGUAGE ExplicitForAll #-}

data T a = forall t. Nil | Mk a
```

А, например, тип конструктора **`Just`**:

```
> :t Just
Just :: a -> Maybe a
```

на самом деле понимается как

$$\texttt{Just} :: \forall a\ (a \to \texttt{Maybe}\ a)$$

(стрелка здесь указывает не на импликацию, а на отображение, хотя некоторая «отсылка», связанная с импликацией, присутствует)

ну или в Haskell стиле как

```
Just :: forall a. a -> Maybe a
```

Даже

```
> :t Nothing
Nothing :: Maybe a
```

на самом деле понимается как

$$\texttt{Nothing} :: \forall a\, (\texttt{Maybe}\ a)$$

ну или в Haskell стиле как

```
Nothing :: forall a. Maybe a
```

что явным образом означает, что тип у **`Nothing`** может быть любой, и при конкретном применении надо либо указывать тип (специализировать), либо он автоматически определяется по месту использования.

И также при объявлениях функций вроде такой

```
f x = x + x
```

её наиболее возможный тип может быть определён как

```
> :t f
f :: Num a => a -> a
```

что на самом деле означает

$$\texttt{f} :: \forall a\, (\texttt{Num}\ a \Rightarrow a \to a)$$

где двойную стрелку $\Rightarrow$ совершенно точно можно рассматривать как «следует», а одинарную стрелку можно рассматривать как «следует» в некотором смысле (напр. как «из того, что $D_f = a$ следует, что $E_f = a$», т.е. указание на то, что область определения и значений функции равны).

Отметим, что в логике предикатов формула

$$\forall x\, (Q(x) \Rightarrow P)$$

и формула

$$(\exists x\, Q(x)) \Rightarrow P$$

равносильны! Здесь в $Q(x)$ есть свободная переменная x, а в P её нет.

Вернёмся к нашему простому типу `T` и вместо переменной произвольного типа `a` рассмотрим гипотетическую квантификацию $\exists t\, (t)$, где t опять переменная типа. В гипотетическом псевдо-Haskell это было бы так:

```
data T = Mk (exists t. t) | Nil
```

Тип его конструкторов будет таким:

```
Mk  :: (exists t. t) -> T
Nil :: T
```

Но выражения `exists t` нет в синтаксисе Haskell! Поэтому, используем логическую равносильность выше и получим

```
Mk :: forall t. t -> T
```

В итоге мы пришли к тому, что в Haskell называют *экзистенциальным типом*:

```
data T = forall t. Mk t | Nil
```

но при этом не использовали квантор существования (и на то есть причины).

Только что созданный тип данных `T` достаточно бесполезен. С ним нельзя сделать «паттерн-матчинг» или построить деконструктор:

```
unMk :: T -> exists t. t
```

так как внимательное рассмотрение условия на тип `exists t. t` показывает, что он невозможен (нечто должно существовать, без конкретного описания; более того, так как под конструктором `Mk` мог быть любой тип, то и это выражение может быть любого типа, т.е. `undef`)

Однако, с этим типом `T` можно сконструировать такую вещь, как *гетерогенный* список:

```
heteroList = [Nil, Mk 5, Mk (), Mk True, Mk map]
```

Правда, в таком виде он совершенно точно бесполезен, так как непонятно, как можно использовать его элементы. Хотя можно посчитать их число.

Усовершенствуем пример

```
{-# LANGUAGE ExistentialQuantification #-}

data T = forall t. Show t => Mk t | Nil

instance Show T where
  show (Mk s) = show s
  show Nil    = "Nil"

heteroList = [Nil, Mk 5, Mk (), Mk True, Mk 5.1]

f xs = map show xs
```

Теперь квантификация идёт не по всем типам, а только в контексте тех, кто «подписан» на класс **`Show`**. Автоматический вывод тут не работает, поэтому воплощение класса **`Show`** для типа `T` делаем в ручном режиме.

```
> f heteroList
["Nil","5","()","True","5.1"]
```

Но вот сделать в этом же стиле фокус для **`Eq`** уже не выйдет, так как базовые элементы лежат в разных типах.

Таким образом, мы применили экзистенциальный тип для некоторого вида «сокрытия» использованных нами типов.

6.4.6. Existentially quantified data constructors

Explicit universal quantification (forall)

wikibooks: Existentially quantified types

What's the theoretical basis for existential types?

Roger Qiu, Existential Types in Haskell

Mark Karpov. Existential quantification

Роман Душкин. Мономорфизм, полиморфизм и экзистенциальные типы

Jonathan Fischoff. Existential Quantification Patterns and Antipatterns

24 Days of GHC Extensions: Existential Quantification

Hengchu Zhang, Taming Heterogeneous Lists in Haskell

Ivan Veselov, Existential types and data abstraction

`forall {..}` in GHC 9

Existential vs. Universally quantified types in Haskell

Existential Haskell

O.Kiselev, Representing existential data types with isomorphic simple types

Обобщённые алгебраические типы (GADTs)

читается: "gad-its"

> Обобщенные алгебраические типы данных, или просто GADT, представляют собой обобщение знакомых вам алгебраических типов данных. По сути, они позволяют явно записывать типы конструкторов.

wikibooks: Haskel/GADT

Попробуем понять, чем они могут быть для нас полезны. В прошлой лекции мы строили некий вид абстрактного дерева синтаксического разбора (AST) для булевых формул простого вида и потом проводили его вычисление. Давайте усложним задачу и рассмотрим немного другой подход к такому дереву.

Пусть нам интересны выражения, которые содержат в себе как простые арифметические выражения с числовыми целыми литерами, так и более сложные булевого типа, которые включают в себя первые с проверкой на равенство. Например,

$$2 + 4 = 2 * 3.$$

Если бы были только арифметические выражения (как булевы в прошлый раз), то мы могли сделать AST так:

```
data Expr = I Int          -- integer constants
          | Add Expr Expr -- add two expressions
          | Mul Expr Expr -- multiply two expressions
```

Тогда выражение `(5+1)*7` будет представлено как

```
(I 5 `Add` I 1) `Mul` I 7 :: Expr
```

Для вычисления на этом дереве (т.е. для трансляции в нормальные выражения с типом **Int**) мы задаём функцию:

```
eval :: Expr -> Int
eval (I n)       = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

и `eval $ (I 5 `Add` I 1) `Mul` I 7` даст 42.

Но нам нужны выражения с равенством! Попытаемся так:

```
data Expr = I Int
          | B Bool           -- boolean constants
          | Add Expr Expr
          | Mul Expr Expr
          | Eq  Expr Expr    -- equality test
```

Тогда выражение `5+1 == 7` будет представлено как `(I 5 `Add` I 1) `Eq` I 7`. Попробуем задать вычисляющую функцию, но теперь она должна работать с двумя типами — целым и булевым. Попробуем так:

```
eval :: Expr -> Either Int Bool
eval (I n) = Left n
eval (B b) = Right b
```

Но дальше проблемы, в слагаемых надо как-то извлечь целую часть из `e1` и `e2`:

```
eval (Add e1 e2) = eval e1 + eval e2
```

и что хуже, что такие выражения будут «правильного» типа:

```
B True `Add` I 5 :: Expr
```

Решение с GADTs:

```
{-# LANGUAGE GADTs #-}

data Expr a where
    I   :: Int  -> Expr Int
    B   :: Bool -> Expr Bool
    Add :: Expr Int -> Expr Int -> Expr Int
    Mul :: Expr Int -> Expr Int -> Expr Int
    Eq  :: Eq a => Expr a -> Expr a -> Expr Bool
```

чем-то напоминает паттерн-матчинг, но только по конструкторам и типам!

Видим, что это работает:

```
*Main> :t (I 1)
(I 1) :: Expr Int
*Main> :t (B True)
(B True) :: Expr Bool
*Main> :t ((B True) `Eq` (B False))
((B True) `Eq` (B False)) :: Expr Bool
*Main> :t ((I 1) `Add` (I 3))
((I 1) `Add` (I 3)) :: Expr Int
*Main> :t ((B True) `Eq` (I 1))

<interactive>:1:17: error:
    * Couldn't match type `Int' with `Bool'
      Expected type: Expr Bool
        Actual type: Expr Int
    * In the second argument of `Eq', namely `(I 1)'
      In the expression: ((B True) `Eq` (I 1))
*Main> :t ( (B True) `Eq` ((I 1) `Eq` (I 2)) )
( (B True) `Eq` ((I 1) `Eq` (I 2)) ) :: Expr Bool
```

Зададим вычисляющую функцию

```
eval :: Expr a -> a
eval (I n) = n
eval (B b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Eq  e1 e2) = eval e1 == eval e2
```

Она замечательно вычисляет:

```
> eval $ I 1
1
> eval $ B True
True
> eval $ I 1 `Add I 2

<interactive>:4:17: error: parse error on input `I'
> eval $ I 1 `Add` I 2
3
> eval $ (I 4 `Add` I 2) `Eq` (I 2 `Mul `I 3)
True
```

GADTs и экзистенциальные типы

И вернёмся теперь к теме экзистенциальных типов. Мы уже обсуждали, что одно из их предназначений — это сокрытие деталей реализации. Но для удобства работы мы можем совместить эти два подхода, точнее просто использовать `GADTs`, вот прошлый пример с гетерогенным списком и их обработкой:

```
{-# LANGUAGE GADTs #-}

data T where
```

```
    Mk :: Show a => a -> T
    Nil :: T

instance Show T where
  show (Mk s) = show s
  show Nil    = "Nil"

heteroList = [Nil, Mk 5, Mk (), Mk True, Mk 5.1]

f xs = map show xs
```

Литература по технике GADTs в Haskell

wikibooks: GADT

6.4.8. Generalised Algebraic Data Types (GADTs)

Roger Qiu, Existential Types in Haskell

James Wilson. Some Awesome Language Extensions Explained. GADTs

Jannis Limperg. A Guide to GHC's Extensions. 4.1.3 GADTs

Real world use of GADT

Makoto Hamana, *Marcelo Fiore*. A Foundation for GADTs and Inductive Families

Matt Parsons. .Basic Type Level Programming in Haskell GADTs

GADTs for dummies

Sandy Maguire. Thinking with Types. C.67

Simple and not-so-simple examples of GADTs and DataKinds

Зоопарк Алгебраических Типов Данных