Если вы видите что-то необычное, просто сообщите мне.

# Делая свой тип.

Вновь, добро пожаловать на серию Отрыв Понедельнечного Хаскельного Утра! Заключительная часть. На случай, если вы прпоустили 2 прошлые главы. В первой части мы обсудили базовую установку Haskell платформы. Затем окунулись в написание базовых выражений на Haskell в интерпретаторе. Во второй части, мы начали с написания нашей собственной функции в модуле Haskell. Так же изучили всяких синтаксических уловок для построения больших и улучшенных функций.

В третьей части мы собираемся углубиться в системы типов. Изучим как создавать свои типы данных, а так же хитрости для упрощения описания наших типов.

# Созание нового типа данных

Вперед, к типам данных! Помните, что у нас есть github репозиторий где вы можете получить код для этой части. Если вы хотите реализовать его самостоятельно, вы можете перейти к модулю DataTypes. Но если вы просто хотите посмотреть на завершенный код, вы можете взглянут на DataTypesComplete.

Для этой статьи, предскавим. что мы пытаемся смоделировать некий TODO список. В этой статье создадим несколько различных `Task` типов данных для отражения отдельных задач в списке. Создадим тип данных сначала у которого будет ключевое слово и затем имя типа. Затем добавим оператор присваивания `=`.

```
module DataTypes where


data Task1 = ...
```

В отличии от выражения и функции именя которые мы использовали в ранее, наши типы начинаются с заглавной буквы. Это то что отличает типы от обычных выражений в Haskell. Теперь собираемся создать наш первый конструктор. Это специальный тип выражения, который позволяет нам создавать объект нашего типа `Task`. Они имеют схожесть с конструкторами скажем на Java. Но они они так же очень сложны. Конструкторы имеют Заглавные буквы а так же список типов. Этот список типов содержит информацию которую хранит конструктор. В нашем случае, мы хотим, чтобы наша задача имела имя и ожидаемоее время выполнения в минутах, отражены как String, и Int соответственно.

```
data Task1 = BasicTask1 String Int
```

Вот так, теперь мы можем начать создавать Task объекты. Например, давайте определим пару простых задач как выражения в нашем модуле.

```
assignment1 :: Task1
assignment1 = BasicTask1 "Do assignment 1" 60

laundry1 :: Task1
laundry1 = BasicTask1 "Do Laundry" 45
```

Мы можем загрузить наш код в интерпретатор, чтоы проверить что он собирается и имеет смысл:

```
>> :l MyData.hs
>> :t assignment1
assignment1 :: Task1
>> :t laundry1
laundry1 :: Task
```

Отметим, что тип нашего выражения `Task1` даже не смотря, что мы собираемся объекты используя `BasicTask1Constructor`. В Java, можно иметь множество конструкторов для одного типа. Мы можем сделать так же и в Haskell, но выглядит это по сложнее. Давайте определим другой тип для различных мест, где мы можем работать над задачами. Мы можем производить работу над задачами в школе, офисе, дома. Отразим это создава конструктор для каждого из них. Разделим конструктор используя вертикальную черту `|`:

```
data Location =
  School |
  Office |
  Home
```

В этом случае, каждый из конструкторов простая отметка, которая не имеет параметров или данных хранящихся в нем. Это пример `Enum` типа. Мы можем технически сделать различные типы выражения отражающими каждый из них.

```
schoolLocation :: Location
schoolLocation = School

officeLocation :: Location
officeLocation = Office

homeLocation :: Location
homeLocation = Home
```

Но эти выражения не более полезны чем использовать сами конструкторы.

Теперь, имея пару типов, мы можем сделать так, что один из наших типов будет содержать другие! Добавим новый конструктор в наш тип задач. Это будет еще сложнее чем просто список мест.

```
data Task1 =
  BasicTask1 String Int |
  ComplexTask1 String Int Location
...

complexTask :: Task1
complexTask = ComplexTask1 "Write Memo" 30 Office
```

Это сильно отличается от конструктора в других языках. Мы можем иметь различные поля для различных отображений типов. Можно обернуть совершенно отличающийся тип зависящий от конструктора который мы используем. Это отлично, так как дает нам гибкость, которую ругие языке не могут.

# Параметризированные типы

Еще использовать параметризированные типы с другими определениями типов. Это значит, что один или более полей зависят от типа, который был выбран человеком который писал код. Давайте предположим, у нас есть тип, который имеет несколько базовых конструкторов для различных видов времени. Это ограничит наше описание для простоты.

```
data TaskLength =
  QuarterHour |
  HalfHour |
  ThreeQuarterHour |
  Hour |
  HourAndHalf |
  TwoHours |
  ThreeHours
```

Теперь мы хотим описать задачу где время задачи будет выражатся в Int. Но так же хотим, чтобы была возможность описать с помощью нового типа. Давайте сделаем вторую верси нашего `Task` типа, который может использовать оба типа для времени выполнения. Мы можем сделать это с помощью параметризованного типа:

```
data Task2 a =
  BasicTask2 String a |
  ComplexTask2 String a Location
```

Тип стал мистическим, и теперь мы можем его заполнять как хотим. Но теперь при выводе `Task2` типа в сигнатуре, мы должны будет заполнить правильное определение.

```
assignment2 :: Task2 Int
assignment2 = BasicTask2 "Do assignment 2" 60

assignment2' :: Task2 TaskLength
assignment2' = BasicTask2 "Do assignment 2" Hour
```

```
laundry2 :: Task2 Int
laundry2 = BasicTask2 "Do Laundry" 45

laundry2' :: Task2 TaskLength
laundry2' = BasicTask "Do Laundry" ThreeQuarterHour

complexTask2 :: Task2 TaskLength
complexTask2 = ComplexTask2 "Write Memo" HalfHour Office
```

К этом нужно относится с осторожностью, так как это может ограничить нашу возможность делать определнные вещи. Например, мы не можем создать список, который содержит оба и `assignment2` и `complexTask2`. Это потому, что два выражения теперь различные типы.

```
-- THIS WILL CAUSE A COMPILER ERROR
badTaskList :: [Task2 a]
badTaskList = [assignment2, complexTask2]
```

# Пример списка

Говоря о списках, мы можем приоткрыть завесу тайны о том, как списки реализованны.

Большое количество синтаксического сахара меняют способ написания списка на практике. Но на уровне кода, списки определяются двумя конструкторами, `Nil` и `Cons`.

```
data List a =
  Nil |
  Cons a (List a)
```

Как мы ожидаем, тип `List` имеет один параметр. Это то что позволяет нам одновременно иметь `Int` или `String`. Конструктор `Nil` это пустой список. Не содержит объектов. Поэтому в любое время, в которое вы будете использовать выражение [], занайте вы используете `Nil`. Второй конструктор складывает один элемент с другим списком. Тип элемента и списка должны, конечно же совпадать. При использовании `:` оператора для добавления элемента в список, вы уже используете `Cons` конструктор.

```
emptyList :: [Int]
emptyList = [] -- Actually Nil

fullList :: [Int]
-- Equivalent to Cons 1 (Cons 2 (Cons 3 Nil))
-- More commonly written as [1,2,3]
fullList = 1 : 2 : 3 : []
```

Еще одна вещь, то что наша структура данных рекурсивна. Мы можем увидеть в `Cons` конструкторе как список содержит другой список с параметрами. Это Работает отлично, покоа есть какой-то базовый случай! Тогда, у нас будет `Nil`. Представьте если у нас есть один конструктор и он принимает рекурсивный параметр. У нас возникает затруднительное положение, из-за того, что мы не знаем как создать любойс писок на первом месте.

# Синтаксическая записи

Давайте вернемся к основам, непараметризированному типу данных `Task`. Предположим, нас не волнует в целом объект `Task`. Скорее, мы хотим один из его кусочков, напиример имя или время. Так как наш код - единственный способ сделать это использовать сопоставление с образцом который явит нужное поле.

```
import Data.Char (toUpper)

...

twiceLength :: Task1 -> Int
twiceLength (BasicTask1 name time) = 2 * time

capitalizedName :: Task1 -> String
capitalizedName (BasicTask1 name time) = map toUpper name

tripleTaskLength :: Task1 -> Task1
tripleTaskLength (BasicTask1 name time) = BasicTask1 name (3 * time)
```

Теперь слегка упрощаем. Вы можете использовать нижнее подчеркивание вместо параметра, который вы не хотите исопльзовать. Но несмотря на это, может получится

громоздко если у ваш тип имеет ножество полей. Мы можем написать нашу функцию позволяющую иметь доступ к отдельным полям. Под капотом, конечно же, будет сопоставление с образом.

```
taskName :: Task1 -> String
taskName (BasicTask1 name _) = name

taskLength :: Task1 -> Int
taskLength (BasicTask1 _ time) = time

twiceLength :: Task1 -> Int
twiceLength task = 2 * (taskLength task)

capitalizedName :: Task1 -> String
capitalizedName task = map toUpper (taskName task)

tripleTaskLength :: Task1 -> Task1
tripleTaskLength task = BasicTask1 (taskName task) (3 * (taskLength task))
```

Но это применение нельзя масштабировать, так как нам нужно писать эту функцию для каждого поля, которое мы будем создавать. Теперь представьте насколько легко, использовать метод `setter` в Java. Сравним это с `tripleTaskLength` выше. Нужно протись по всем полям, что не есть хорошо. Отличная новость, в том, что мы можем заставить Haskell написать функцию для нас использовать синтаксис записи. Для этого, всё, что нам нужно это назначить каждому полю в определении нашего типа. Давайте сделаем новую версию `Task`.

```
data Task3 = BasicTask3
  { taskName :: String
  , taskLength :: Int }
```

Теперь можно писать тот же код без `getter` функции которую мы писали выше.

```
-- These will now work WITHOUT our separate definitions for "taskName" and
-- "taskLength"
twiceLength :: Task3 -> Int
twiceLength task = 2 * (taskLength task)
```

```
capitalizedName :: Task3 -> String
capitalizedName task = map toUpper (taskName task)
```

Теперь можно создать задачу, мы всё еще можем использвать `BasicTask3` сам по себе. Но для чистоты кода, мы можем так же создать объект используя синтаксическую запись, где мы называли поле:

```
-- BasicTask3 "Do assignment 3" 60 would also work
assignment3 :: Task3
assignment3 = BasicTask3
  { taskName = "Do assignment 3"
  , taskLength = 60 }

laundry3 :: Task3
laundry3 = BasicTask3
  { taskName = "Do Laundry"
  , taskLength = 45 }
```

Мы так же можем написать `setter` еще проще используя синтаксическую запись. Вопользуемся прошлой задачей и затем списоком изменений "changes" чтобы поместить их в скобки.

```
tripleTaskLength :: Task3 -> Task3
tripleTaskLength task = task { taskLength = 3 * (taskLength task) }
```

В общем, мы используем только синтаксическую запись, когда есть один конструктор для типа данных. Мы можем использовать различные поля для различных конструкторов, но только наш код чуток безопаснее. Давайте посмотрим на еще один пример определения `Task`:

```
data Task4 =
  BasicTask4
    { taskName4 :: String,
      taskLength4 :: Int }
  |
  ComplexTask4
    { taskName4 :: String,
      taskLength4 :: Int,
      taskLocation4 :: Location }
```

Проблема текущей системы, в том. что компилятор будет создавать `taskLocation4` функцию, которая будет собираться для любой задачи. Но функция отработает правильно, только когда вызывается `ComplexTask4`. Следующий код, будет собираться даже если будет причиной падения, и чтобы этого избежать:

```
causeError :: Location
causeError = taskLocation4 (BasicTask4 "Cause error" 10)
```

В добавок, в наших различных конструкторах используются различные типы, мы не можем использовать то же имя для них. Это может выглядить странно, когда мы хотим отразить ту же идею с различными типами. Этот пример не соберется потому что GHC не может определять тип функции `taskLength4`. Она даже может иметь тип `Task -> Int` или `Task -> TaskLength`.

```
data Task4 =
  BasicTask4
    { taskName4 :: String,
      taskLength4 :: Int }
  |
  ComplexTask4
    { taskName4 :: String,
      taskLength4 :: TaskLength, -- Note we use "TaskLength" and not an Int here!
      taskLocation4 :: Location }
```

# Ключевое слово типа.

Теперь, мы знаем, что большинство входных и выходных типов данных самодельные. Но бывают случаю когда вам не нужно делать этого. Мы можем создать новый тип без создавания полностью нового типа структур. Есть два способа сделать это. Первое это ключевое слово. Оно позволяет вам создавать синонимы для типов, таких как `typedef` ключевое слово в C++. Самое распространенное, как мы видели это `String` это список символов.

```
type String = [Char]
```

Распространенный способ использования для него, это когда вы объединяете множество различных типов в кортеж. Это может быть довольно нужно писать кортеж несколько раз в коде.

```
makeTupleBigger :: (Int, String, Task) -> (Int, String, Task)
makeTupleBigger (intValue, stringValue, (BasicTask name time) =
  (2 * intValue, map toUpper stringValue, (BasicTask (map toUpper name) (2 * time)))
```

Использование синонима далает запись сигнатуры гораздно чище:

```
type TaskTuple = (Int, String, Task)

makeTupleBigger :: TaskTuple -> TaskTuple
makeTupleBigger (intValue, stringValue, (BasicTask name length) =
  (2 * intValue, map toUpper stringValue, (BasicTask (map toUpper name) (2 * length))
```

Конечно, если коллекция будет большой, то стоит сделать полный тип данных для этого. Так же есть некоторые причины почему синонимы типов не всегда лучший выбор. Они могут привести к ошибкам компиляции, с которыми трудно будет работать. Вы возмжожно прошли через несколько ошибок где компилятор уже говорил, что ожидает [Char]. Это было бы понятнее если бы он говорил про String.

И межет так же вести к неинтуитивному коду. Предположим вы используете базовый кортеж вместо типа данных для отображения `Task`. Кто-то может ожидать, что тип `Task` будет иметь свой собственный тип. Затем они будут запутаны тем, что вы работаете с ним как с кортежем.

```
type Task5 = (String, Int)

twiceTaskLength :: Task5 -> Int
-- "snd task" is confusing here
twiceTaskLength task = 2 * (snd task)
```

# Новые типы

Последнюю тему которую мы обсдудим будет "newtypes". Это как синоными с одной стороны и `ADT` с другой. Но они всё еще имет уникальное место в Haskell и лучше если вы привыкните пользоваться им. Предположим, мы хотим иметь новое подход для отображения `TaskLength`. Мы хотим использовать обычное число, но мы чтобы он имел свой собственный отдельный тип. Мы можем это сделать с помощью "newtype":

```
newtype TaskLength2 = TaskLength2 Int
```

Синтакс для `newtypes` выглядит похожим на ADT. Однако, `newtype` определение может только иметь один коснтруктор. И этот конструктор может только принимать отдельный тип аргументов. Большое отличие между ADT и `newtype` идет после компиляции вашего кода. В этом примере, не будет различий между `TaskLength` и `Int` типы во время выполнения. Это хорошо, так как большая часть кода для `Int` типа специализированна на быстром выполнении. Если мы сделаем настоящим ADT, это не тот случай:

```
-- Not as fast!
data TaskLength2 = TaskLength2 Int
```

Но с другой стороны, мы можем сделать гораздо больше таких трюков с `newtype`, нежели чем с ADT. Мы можем, например, использовать синтаксическую запись в конструктре для наших `newtype`. Это позволяет нам использовать имя чтобы извлечь значение изнутри без сопоставления с образцом. Часто сопоставление с образом при использовании синтаксической записи для какого-нибудь `un-TypeName` значениия в качестве имени поля. Так же отметим, что мы не можем использовать `newtype` значеение с той же фунецией как изначальный тип. Когда у анс синоним, мы должны сделать следующее:

```
data Task6 = BasicTask6 String TaskLength2

newtype TaskLength2 = TaskLength2
  { unTaskLength :: Int }

mkTask :: String -> Int -> Task6
mkTask name time = BasicTask6 name (TaskLength2 time)

twiceLength :: Task6 -> Int
twiceLength (BasicTask6 _ len) = 2 * (unTaskLength len)
-- The following would be WRONG!
```

```
-- 2 *len
```

Теперь, `TaskLength2` это эффективная обертка над `Int`. Это делает его похожим на тип синоним, за исключением того, что мы не можем просто использовать `Int` значение по себе. Как вы видите в примере выше, нам нужно пройти через процесс обертки и разворачивания значения. Это выглядит нудно. Но это очень полезно, так как решает главную проблему использования типа синонима. Теперь если мы делаем ошибки касающиеся `TaskLength`, компилятор скажет нам о `Tasklength`. Мы не будем гадать какои из синонимов мы пропустили!

Есть другой пример. Предположим у нас есть функция с несколькими целочисленными аргументами. Если мы всгде используем `Int` типб мы легко смешаем порядок аргументов. Но если мы используем `newtype`, компилятор будет отлавливать ошибки этих типов за нас.

# Заключение

Это заверешение нашего разговора по поводу создания своего типа данных и заверешение нашей Улетной серии! Если вам нужно освежить знания не забудьте проведать часть 1 и 2.

---

Revision #4
Created 2022-03-11 05:19:47 UTC by gasick
Updated 2022-07-14 18:25:33 UTC by gasick