1 Типы-записи

Мы уже знаем синтаксис вида

```
data MyData = MyDataConstructor Int String
```

Он неудобен тем, что у нас нет говорящих названий для элементов, которые лежат внутри. Мы можем вместо этого сделать так:

```
data MyData = MyDataConstructor
            { mydata1 :: Int
            , mydata2 :: String
            }
```

Теперь у полей этого типа данных есть названия. У них есть применение: если у нас есть экземпляр `MyData` под названием `a`, то мы можем достать из него лежащий внутри `Int` вызовом `mydata1 a`. То есть `mydata1 :: MyData -> Int`.

Такой способ определения типов называется *record*'ами (записями).

Задать экземпляр MyData можно двумя способами:

```
v = MyDataConstructor 3 "s"
v' = MyDataConstructor { mydata1 = 3, mydata2 = "s"}
```

Иногда встречается нужда получить экземпляр, в котором от имеющегося отличие только в нескольких полях. Для этого существует специальный синтаксис:

```
v'' = v { mydata1 = 4 }        -- MyDataConstructor 4 "s"
v''' = v'' { mydata2 = "e" } -- MyDataConstructor 4 "e"
```

Это удобно в случаях, когда не хочется сопоставлять с образцом крупную структуру данных a потом склеивать всё назад; ещё можно применять эту конструкцию для того, чтобы создавать "значения по умолчанию", которые затем требуется только обновить. Например, можно было бы считать, что `v` — это значение по умолчанию, и можно было бы заставить пользователей получать новые экземпляры путём модификации `v`.

2 Классы типов

2.1 Примеры

Класс типов позволяет по общему имени обращаться к какой-то функциональности. К примеру, мы хотим задать оператор сложения так, чтобы он работал и на **Int**-ах, и на **Double**-ах, и на комплексных числах, и на многом другом, однако возникает сложность: реализовано сложение во всех этих случаях явно по-разному, хотя бы в силу различий в представлении в памяти. При этом мы не хотим давать складывать экземпляры тех типов, для которых это бессмысленно: к примеру, что такое сложение функций **Bool** −> **String**[1]?

Для решения этой задачи можно создать класс MyNum и сказать, что (<+>) (так как (+) уже занят) можно вызывать на всём, что является экземпляром MyNum. Заодно можно добавить умножение (<*>).

Тогда можно было бы определить этот класс так:

```
class MyNum a where
    (<+>) :: a -> a -> a
    (<*>) :: a -> a -> a
```

Теперь можно объявить **Integer** экземпляром данного класса, то есть привязать к **Integer** словарь из функций, свойственных MyNum:

```
instance MyNum Integer where
    (<+>) a b = a + b
    (<*>) a b = a * b
```

[1]На самом деле варианты есть: например, \ f g b -> f b ++ g b

Как видим, MyNum можно реализовать для типов с kind'ом $*$.

Создадим класс, который можно реализовать только для типов с kind'ом $* -> *$. Назовём его MyFunctor.

```
class MyFunctor f where
  myFmap :: (a -> b) -> f a -> f b
```

Вспомним, что пример того, что имеет kind $* -> *$ — это функциональная стрелка, которой уже дали один параметр. Сделаем её экземпляром MyFunctor:

```
instance MyFunctor (-> e) where
  myFmap = (.)
```

Здесь точка — композиция функций. Заметим, что мы указали в заголовке **instance** некое e — это свободная типовая переменная, вместо неё может подставляться любой тип.

Рассмотрим пример "наследования" классов. Введём тип чисел, расширенных мнимой единицей:

```
data ExtendedWithIm a = ExtendedWithIm
                        { re :: a -- ^ Real part
                        , im :: a -- ^ Imaginary part
                        }

i :: Num a => ExtendedWithIm a
i = ExtendedWithIm 0 1
```

К примеру, ExtendedWithIm **Double** — это примерно комплексные числа:

```
1.0 + 2.5 * i :: ExtendedWithIm Double
```

Мы можем теперь реализовать **Num** для ExtendedWithIm для любых a, которые уже являются **Num**:

```
instance Num a => Num (ExtendedWithIm a) where
    a + b = ExtendedWithIm (re a + re b) (im a + im b)
    a * b = ExtendedWithIm
        (re a * re b - im a * im b)
        (im a * re b + im b * re a)
    abs a = ExtendedWithIm (re a * re a + im a * im a) 0
    signum a = ExtendedWithIm (signum (re a)) (signum (im a))
    fromInteger a = ExtendedWithIm (fromInteger (re a)) 0
    negate a = ExtendedWithIm (- re a) (- im a)
```

2.2 Интуиция

Зададим структуру данных

```
data MyNum a = MyNum
               { plus :: a -> a -> a
               , mult :: a -> a -> a
               }
```

Создадим экземпляр `MyNum` для `Int`:

```
myNumInt :: MyNum Int
myNumInt = MyNum (+) (*)
```

Посмотрим, какой тип имеет функция `plus`:

```
-- :type plus
plus :: MyNum a -> a -> a -> a
```

Это очень похоже на тип настоящего плюса, только вместо двойной стрелки здесь одинарная.

Сейчас мы можем вызвать `plus` таким образом:

```
plus myNumInt 3 4
```

Примерно такой же смысл и у типовых контекстов: это для нас просто коллекции функций, которые протаскиваются вместе с экзеплярами типов. Разница в том, что типовой контекст для каждого типа задан уникально (нельзя сделать две реализации + для **Int**), а потому его не нужно явно указывать как параметр функции, Haskell сам знает, какую коллекцию функций надо взять.

Итого: контекст **Num** a означает, что к a привязан словарь функций, перечисленных в :info **Num**.

2.3 Отличие от интерфейсов

Часто говорят, что классы типов похожи на интерфейсы в ООП-языках. Это правда:

- Как и интерфейсы, классы типов задают коллекцию функций, которые должны быть реализованы для типа данных, чтобы можно было считать, что он реализует интерфейс/является экземпляром класса типов.
- Классы типов можно наследовать, и если класс типов A унаследован от B, то экземпляром B можно сделать только то, что является экземпляром A, а также в функциях, которые ожидают тип, реализующий B, можно предполагать, что этот тип реализует и A. Всё это точно совпадает с тем, как работает наследование у интерфейсов.

Однако есть и важное отличие: интерфейс ожидает, что его методы будут вызываться у какого-то объекта, в то время как в случае с классами типов это не так: это просто коллекция функций. Например, у какого экземпляра вызывается "метод" сложения в (+) (3 :: **Int**) 4, у 3 или 4? Правильный ответ: ни у какого.

Более очевидный пример заключается в том, что в классах можно задать даже функции, которые *создают* экземпляры заданного типа; тогда для вызова функции из класса типов вообще может не требоваться существование экземпляра.

Например, функция **read** :: **Read** a =>**String** -> a порождает экземпляр `a` из строки, и хотя можно сказать, что тип `a` реализует **read**, аналогия с интерфейсами тут ломается: в интерфейсе нельзя задать метод, который работает без экземпляра, и уж тем более нельзя задать метод, который создаёт экземпляр.