Классы типов в Haskell

Кольцо вычетов

Создадим несколько учебных примеров, для лучшего понимания использования алгебраических типов на практике. Допустим, мы хотим рассмотреть системы вычетов по модулю 5. С точки зрения математики, мы хотели бы реализовать на Haskell кольцо вычетов, т.е. возможность складывать, умножать и т.п. классы вычетов.

```
[2] + [2] = [4];
[2] + [3] = [0];
[2] * [2] = [4];
[2] * [3] = [1];
```

(Иное представление — операции по модулю 5)

В математике, чтобы ввести понятие *вычетов*, начинают со сравнения по модулю.

Два целых числа a и b будем называть сравнимыми по модулю $m \in \mathbb{N}$, $m \neq 0$, если их разность делится на m, иными словами

$$a \equiv b \pmod{m} \quad \Leftrightarrow \quad (a - b) \vdots m.$$

Два целых числа a и b будут сравнимы по модулю $m \in \mathbb{N}$, $m \neq 0$ тогда и только тогда, когда они имеют одинаковые остатки при делении на m.

Отношение сравнения по модулю $(\equiv)$ будет для целых чисел *рефлексивно*, *симметрично* и *транзитивно*, иными словами, будет являться *эквивалентностью* на множестве $\mathbb{Z}$.

Домашнее задание: найти математические определения введённых понятий и проверить, что это так.

Исходя из этого, множество целых чисел $\mathbb{Z}$ *факторизуется*, т.е. разбивается на непересекающиеся подмножества, все элементы которых имеют одинаковые остатки при делении на заданное число, напр., на 5.

$$\mathbb{Z} = [0] \cup [1] \cup [2] \cup [3] \cup [4], \quad [i] \cap [j] = \varnothing \text{ при } i \neq j.$$

Теперь, можно индуцировать арифметические операции на этих множествах, как на единых объектах, например $(+)$:

$$[k] := [i] + [j],$$

следующим образом. Пусть x, y — произвольные числа такие, что

$$x \in [i], \quad y \in [j], \quad z := x + y,$$

и k равен остатку при делении z на 5, т.е.

$$k \equiv (x + y) \pmod{5} \quad \& \quad 0 \leqslant k < 5.$$

Множество $[k]$ будет искомое.

Аналогично вводим умножение. Таким образом, отношение сравнения по модулю $(\equiv)$ для целых чисел будет *конгруэнцией* и мы можем задать *модель* (алгебраическую систему) $\mathfrak{M} = \langle \mathbb{Z}/\equiv; +, *, 0, 1 \rangle$. Здесь мы выделяем *носитель* (множество элементов) $\mathbb{Z}/\equiv$,

вводимые операции $(+)$ и $(*)$ на элементах носителя, и некоторые выделенные константы (0 и 1).

Алгебраические системы

Зададим тройку $\langle \sigma_F, \sigma_P, \sigma_C \rangle$, состоящую из попарно непересекающихся множеств, где множество функциональных символов σ_F определяет имена основных операций, множество предикатных символов σ_P определяет множество основных отношений, множество σ_C — множество выделенных элементов или, как ещё их называют, множество символов констант (или просто констант).

Зададим отображение $\rho : \sigma_F \cup \sigma_P \to \mathbb{N}^+$, определяющее арность или местность функциональных и предикатных символов. Тогда $\sigma = \langle \langle \sigma_F, \sigma_P, \sigma_C \rangle, \rho \rangle$ называется *сигнатурой*. Если множества символов сигнатуры σ конечные, то часто будем писать

$$\sigma \;=\; \langle F_0^{n_0}, \dots, F_k^{n_k};\; P_0^{m_0}, \dots, P_r^{m_r};\; c_0, \dots, c_s \rangle,$$

где $\sigma_F = \{F_0^{n_0}, \dots, F_k^{n_k}\}$, $\sigma_P = \{P_0^{m_0}, \dots, P_r^{m_r}\}$, $\sigma_C = \{c_0, \dots, c_s\}$, а $n_i = \rho(F_i)$, $m_j = \rho(P_j)$.

Определение.
Алгебраической системой (моделью) $\mathfrak{A}$ сигнатуры σ называется пара $\mathfrak{A} = \langle A, \mathrm{int}_\sigma \rangle$, состоящая из непустого множества A и интерпретации int_σ, которая функциональному символу $f \in \sigma_F$ сопоставляет отображение $\mathrm{int}_\sigma(f)$, действующее из $A^{\rho(f)}$ в A, предикатному символу $P \in \sigma_P$ — подмножество $\mathrm{int}_\sigma(P)$ множества $A^{\rho(P)}$, константному символу $c \in \sigma_C$ — элемент $\mathrm{int}_\sigma(c)$ множества A.

Множество A называется *основным множеством* или *носителем* системы и обозначается как $|\mathfrak{A}|$. Отображение int_σ называется *интерпретацией* алгебраической системы $\mathfrak{A}$. Если из контекста ясно, как определены int_σ и ρ и как заданы сами операции и отношения на $\mathfrak{A}$, то эти обозначения опускаем и в случае конечной сигнатуры обычно, чтобы задать систему $\mathfrak{A}$, будем писать:

$$\mathfrak{A} \;=\; \langle A; F_0^{n_0}, \dots, F_k^{n_k}; P_0^{m_0}, \dots, P_r^{m_r}; c_0, \dots, c_s \rangle,$$

или даже писать $\mathfrak{A} = \langle A; \sigma \rangle$, если σ известна или несущественна.

Пример.
Так, введенное раннее исчисление высказываний формирует простейшую двухэлементную булеву алгебру

$$\mathfrak{B} = \langle B; \&, \vee, \neg \rangle, \quad \text{где} \quad B = \{\text{л}, \text{и}\}.$$

wikipedia: Алгебраическая система

Введённое определение алгебраической системы чрезвычайно важно как для математики (прежде всего для дискретных её областей) так и для концепций Computer Science, где оно находит много общего с понятием объектов в объектно-ориентированном программировании или с полиморфными перегружаемыми классами в языках, подобных Haskell.

Необходимые определения термов, логических формул, значений термов и истинности (выполнимости) утверждений на модели можно найти в монографии (Ершов Ю.Л., Палютин Е.А. Математическая логика: Учеб. пособие. 3-е изд., стер. СПб.: Издательство «Лань», 2004.).

Определение.
Алгебраическая система $\mathfrak{A} = \langle A, +, \cdot, 0 \rangle$ называется *кольцом*, если на данной системе выполнены следующие аксиомы:

1. $\forall x \forall y \forall z \; (\; (x + y) + z \; = \; x + (y + z) \;)$;
2. $\forall x \forall y \; (\; x + y = y + x \;)$;
3. $\forall x \; (0 + x = x + 0 = x)$;
4. $\forall x \exists y \; (x + y = y + x = 0)$;
5. $\forall x \forall y \forall z \; (\; x \cdot (y + z) \; = \; x \cdot y \; + \; x \cdot z \;)$;
6. $\forall x \forall y \forall z \; (\; (x + y) \cdot z \; = \; x \cdot y \; + \; x \cdot z \;)$.

Если на кольце дано отношение эквивалентности $\sim$, устойчивое относительно операций:

$$\forall x_1 \forall x_2 \forall y_1 \forall y_2 \; (\; (x_1 \sim y_1 \wedge x_2 \sim y_2) \; \rightarrow \; (x_1 + x_2 \sim y_1 + y_2) \wedge (x_1 \cdot x_2 \sim y_1 \cdot y_2)),$$

то говорят об отношении конгруэнтности, при котором возможно задать фактор-кольцо. Для целых чисел $\mathbb{Z}$ таким фактор-кольцом будет кольцо вычетов по некоторому модулю $\mathbb{Z}/\equiv_m$. С другой стороны, можно задать изоморфную алгебру вычетов $\mathfrak{K}$ для множества некоторых объектов, например $\{\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}\}$ явным указанием таблицы операций (т.е. интерпретацией сигнатурных символов на данном множестве):

$+$	**0**	**1**	**2**	**3**	**4**
0	$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$
1	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$	$\bar{0}$
2	$\bar{2}$	$\bar{3}$	$\bar{4}$	$\bar{0}$	$\bar{1}$
3	$\bar{3}$	$\bar{4}$	$\bar{0}$	$\bar{1}$	$\bar{2}$
4	$\bar{4}$	$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$

$\cdot$	**0**	**1**	**2**	**3**	**4**
0	$\bar{0}$	$\bar{0}$	$\bar{0}$	$\bar{0}$	$\bar{0}$
1	$\bar{0}$	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$
2	$\bar{0}$	$\bar{2}$	$\bar{4}$	$\bar{1}$	$\bar{3}$
3	$\bar{0}$	$\bar{3}$	$\bar{1}$	$\bar{4}$	$\bar{2}$
4	$\bar{0}$	$\bar{4}$	$\bar{3}$	$\bar{2}$	$\bar{1}$

wikipedia: Классы вычетов

Следующий код даёт представление о том, как можно реализовать модель $\mathfrak{M} = \mathbb{Z}/\equiv$ на Haskell:

```
data MyAlg = O | I | II | III | IV  deriving (Eq, Ord, Read, Show)

o :: MyAlg
o = O

e :: MyAlg
e = I

-- вспомогательные функции
next :: MyAlg ->  MyAlg
next O   = I
next I   = II
next II  = III
next III = IV
next IV  = O
```

```
prev :: MyAlg ->  MyAlg
prev O   = IV
prev I   = O
prev II  = I
prev III = II
prev IV  = III

-- это будет наш +
(<>) ::  MyAlg -> MyAlg -> MyAlg
O <> x = x
n <> x = next  (prev n <> x)

-- это будет наше *
(><) ::  MyAlg -> MyAlg -> MyAlg
O >< _ = O
n >< x = (prev n >< x) <> x

-- образец предиката
chet :: MyAlg -> Bool
chet O  = True
chet II = True
chet IV = True
chet _  = False
```

Здесь конструктор типа `MyAlg` определяет требуемый нам носитель модели $\mathbb{Z}/\equiv$. Конструкторы значений `O`, `I`, `II`, `III` и `IV` будут символизировать вычеты. Функции **`next`** и `prev` носят технический характер, работают как кольцевые сдвиги значений. Двухместная функция `(<>)` будет служить у нас сложением, организована по рекурсии. Двухместная функция `(><)` будет служить у нас умножением, тоже организована по рекурсии. Последние две функции можно было бы организовать как явную таблицу «сложения» и «умножения», вида:

```
O <> O = O
O <> I = I
...
IV <> O = IV
IV <> I = O
IV <> II = I
...
O >< O = O
...
I >< II = II
...
```

но было бы утомительно перебирать все комбинации. Вариант с рекурсией выглядит явно выигрышней.

Разовьём идею этого примера. Вместо непривычных операций `(<>)`, `(><)` организуем перегрузку стандартных операций `(+)`, `(-)`, `(*)` из класса **`Num`**. Но сначала определим функцию из целых в вычеты и наоборот (эпиморфизм из целых в вычеты и выбор канонического представителя; в нашем случае, это будет отображение вычетов в первые пять

целых чисел):

```
-- эпиморфизм из целых в вычеты (классы экв.)
toMyAlg :: Integer -> MyAlg
toMyAlg 0 = O
toMyAlg 1 = I
toMyAlg 2 = II
toMyAlg 3 = III
toMyAlg 4 = IV
toMyAlg x = toMyAlg (x `mymod` 5)

-- определяем канонического представителя
fromMyAlg :: MyAlg -> Integer
fromMyAlg O   = 0
fromMyAlg I   = 1
fromMyAlg II  = 2
fromMyAlg III = 3
fromMyAlg IV  = 4
```

Так как обычно в языках программирования функции остатков математически ошибочны, то здесь `mymod` — это математически правильный остаток от деления:

```
mymod a b | (a >= 0) && (b >= 0) = a `mod` b
          | (a < 0)  && (b >= 0) = a `mod` b
          | (a >= 0) && (b < 0)  = a `rem` b
          | (a < 0)  && (b < 0)  = a `mod` (negate b)
```

Или даже проще:

```
mymod a b = (abs a) `rem` (abs b)
```

(пишут, что **mod** определяется в терминах **rem**, поэтому менее оптимален даже только для положительных)

ru.wiki: Деление с остатком

6.4.2 Арифметические и теоретико-числовые операции

Теперь, когда все вспомогательные операции готовы, мы можем организовать воплощение класса **Num** для нашего кольца вычетов. Класс **Num** по документации требует для своего «воплощения» реализации функций `(+)`, `(*)`, **abs**, **signum**, **fromInteger** и **negate** (или `(-)` вычитание)

```
-- добавим к предыдущему:
-- интерпретация сигнатурных символов
-- (в терминах Haskell: перегрузка операций класса Num)

instance Num MyAlg where
  fromInteger = toMyAlg
  x + y =  x <> y
  x * y =  x >< y

-- вводим отрицание (рекурсивно) и др.
  negate O = O
  negate n = prev (negate (prev n))
```

```
  signum 0 = 0
  signum _ = I
  abs = id
```

Для реализации `(+)`, `(*)`, и **`fromInteger`** у нас уже всё готово. Функцию **`negate`** зададим так, чтобы

```
x + (negate x) = 0
```

Для **`abs`** и **`signum`** особых идей нет: пусть **`abs`** ничего не изменяет (может быть стоит сделать потом согласование с **`negate`**), а функция **`signum`** естественным образом для `0` возвращает `0`, а для всех остальных значений: `I`.

Вывод.
Таким образом, мы для алгебраической системы $\mathfrak{M} = \langle \mathbb{Z}/\equiv; +, *, 0, 1\rangle$ создали подходящий носитель, и сделали интерпретацию сигнатурных символов $\langle +, *, 0, 1\rangle$ через введенные ранее операции `(<>)`, `(><)`, `o`, `e`.

Пример построения нового типа из заданного

Следующий пример используем для построения типа, который является ограничением уже существующего типа. Таким примером будет тип `Natural`, являющийся сужением типа **`Integer`** до неотрицательных целых чисел.

Кроме того, нам может потребоваться сделать тип, у которого скрыты некоторые детали организации: конструкторы, те или иные методы. Это может предотвратить нежелательное использование, кроме указанных интерфейсных методов. В Haskell это возможно путем вынесения типа и его методов в отдельный модуль.

Также, иногда возникает необходимость снабдить новый тип функциями, которые заданы именно для него, но не для исходного типа (см. пример в лекции-6 с типом `Email` и функцией стоимости). Для натуральных чисел, например, мы могли бы задать функцию, которая для каждого натурального числа возвращает *конечный* список (теоретически, на практике он может быть очень большим) чисел-прешественников. На целых числах задать такую функцию невозможно, так как множество предшественников заведомо бесконечно.

```
module Naturalische (
Natural (MakeNatural),
toNatural,
fromNatural
) where

newtype Natural = MakeNatural Integer deriving (Eq,Ord)

toNatural :: Integer -> Natural
toNatural x | x < 0 = error "Negative natural numbers?"
            | otherwise = MakeNatural x

fromNatural :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

```
chet x = (x `mod` 2 == 0)

instance Num Natural where
  fromInteger = toNatural
  x + y = toNatural (fromNatural x + fromNatural y)
  x - y = toNatural (fromNatural x - fromNatural y)
  x * y = toNatural (fromNatural x * fromNatural y)
  abs = toNatural . fromNatural
  signum = toNatural . signum . fromNatural

instance Show Natural where
  show = show . fromNatural
```

Детали определения класса **Num** можно найти в документации.

Так как наш тип данных «не слишком настоящий» и не поддержан, например, на уровне литералов, и главное, на уровне библиотечных функций, то нам для работы с теми или иными стандартными арифметическими функциями необходимо сначала переводить наши `Natural`-числа с помощью функции `fromNatural` в **`Integer`**-числа, производить операции, а затем возвращаться обратно в `Natural`-числа с помощью функции `toNatural` (если это будет возможно).

Мягкое введение в Haskell: Объявление newtype

Отметим, экспорт конструктора значений `MakeNatural` возможен только вместе с экспортом конструктора типа, как это и сделано в примере (возможен более краткий вариант):

```
module Naturalische (
Natural (..),
```

но тогда экспортируются все конструкторы значений для данного типа), см.

stackoverflow: export from module

Кроме того, в Haskell существует некоторый «изъян», так как воплощения (instances) классов (**`Num`**, **`Show`**) для данного типа (`Natual`) экспортируются неявным образом и управлять этим — довольно хитрая «магия»:

How to import just specific instances in haskell

Explicitly import instances

How do I export typeclasses?

И также отметим, что есть вполне стандартный тип данных `Natural`, который впрочем, тоже базируется на типе **`Integer`**:

GHC.Natural.

Хорошая статья по теме в целом, указывает аналоги (и разницу) с интерфейсами из мира ООП:

Matt Parsons. How do type classes differ from interfaces?.

О предназначении классов типов в Haskell

Цели

- построение сложных абстрактных структур и разные их воплощения различными типами данных
- *специальный* полиморфизм (ad hoc) и перегрузка методов

Отметим, что наряду с *параметрическим* полиморфизмом (например, при рассмотрении списков вида `[a]` и способов их обработки), это одна из особенностей языка Haskell.

Примеры специального полиморфизма

- литералы 1, 2 и т.д. часто используются для представления целых как ограниченной, так и неограниченной длины
- числовые операторы, такие как `+`, часто определяют как способные работать со многими различными видами чисел
- оператор равенства (в Haskell это `==`) обычно работает с числами и многими другими (но не всеми) типами.

Мягкое введение в Haskell / Классы типов и перегрузка

Создание классов типов в Haskell

В предыдущем пункте были показаны примеры, когда мы для новых алгебраических типов данных: кольца вычетов и натуральных чисел решили использовать привычные операции сложения, вычитания, умножения и т.п. Для этого мы воспользовались механизмом воплощения стандартных классов для наших типов данных.

Однако, классы можно также создавать «с чистого листа». Идея классов в Haskell родственна идее сигнатурных символов операций, предикатов, констант и их интерпретаций в теории моделей, о которых кратко говорилось выше. В Haskell эта идея сводится к тому, что при организации класса мы выписываем сигнатуры нужных нам типов функций и их простейших связей (взаимных определений), а потом, при разработке новых типов данных, мы указываем воплощение данного класса для требуемого типа.

При этом, мы вправе использовать механизм модулей, который позволяет нам скрыть детали реализации для конечного пользователя. Например, в примере с натуральными числами выше предикат `chet` сокрыт для внешнего мира. А конструктор данных `MakeNatural` разрешён к использованию в явном виде. Если бы мы этого не хотели, то нужно было бы указать:

```
module Naturalische (
Natural,
toNatural,
fromNatural
) where
...
```

stackoverflow: export from module

Абстрактные типы данных и сокрытие конструкторов значений

Таким образом, мы приходим к понятию *абстрактных типов данных*, где пользователю с помощью классов представлен интерфейс доступа и манипуляций к значениям из некоторого типа данных, детали реализации которого намеренно скрыты и могут в будущем намеренно изменены.

Правда, в случае, когда скрыт конструктор значений (а это важная составляющая сокрытия реализации типа данных), возникает специфическая для Haskell проблема невозможности использовать pattern-matching. Дело в в том, что pattern-matching в Haskell как раз и реализован в основном с помощью декомпозиции конструкторов значений.

Есть обходные варианты, например, можно использовать расширение компилятора `ViewPatterns`. Например, вместо использования паттерн-матчинга в таком примере:

```
ftest :: Natural -> String
ftest (MakeNatural 1) = "one"
ftest (MakeNatural 2) = "two"
ftest (MakeNatural x) = "other"
```

с такими тестами:

```
ghci> ftest 2
"two"
ghci> ftest 5
"other"
ghci> ftest (-5)
"*** Exception: Negative natural numbers?
CallStack (from HasCallStack):
  error, called at testnat3.hs:10:23 in main:Naturalische
ghci> ftest (MakeNatural 2)
```

В случае без доступа к конструктору `MakeNatural`, можно было определить таким образом:

```
{-# LANGUAGE ViewPatterns #-}

import Naturalische

fte :: Natural -> String
fte (fromNatural  -> 1) = "one"
fte (fromNatural  -> 2) = "two"
fte (fromNatural  -> x) = "other"
```

аналогично

```
ghci> fte 2
"two"
ghci> fte 5
"other"
ghci> fte (-5)
"*** Exception: There is no negative natural numbers!
CallStack (from HasCallStack):
  error, called at .\Naturalische.hs:10:23 in main:Naturalische
```

```
ghci> fte (MakeNatural 1)

<interactive>:4:6: error:
    Data constructor not in scope: MakeNatural :: t0 -> Natural
```

(при этом работа с целыми литералами происходит за счёт их «перегруженности»)

Или такой, более яркий пример работы по pattern-matching'у со списками, если сокрыт конструктор `(:)`

```
{-# LANGUAGE ViewPatterns #-}

eitherEndIsZero :: [Int] -> Bool
eitherEndIsZero (head -> 0) = True
eitherEndIsZero (last -> 0) = True
eitherEndIsZero          _  = False

> eitherEndIsZero [0,1,8,9]
True
```

В обычном случае, мы писали бы примерно так:

```
eitherEndIsZero :: [Int] -> Bool
eitherEndIsZero (x:xs) = (x == 0) || (last xs == 0)
```

или так:

```
eitherEndIsZero xs =  (head xs == 0) || (last xs == 0)
```

В любом случае, мы видим, что работа вместо прямого использования конструктора `(:)` в левой части заменяется на использование соответствующих функций.

Для правой части, мы вполне могли бы ввести функцию

```
push x xs = x:xs
```

и после этого не экспортировать конструктор `(:)` вообще.

School of Haskell: ViewPatterns

View patterns: lightweight views for Haskell

Syntactic extensions

Refactoring Pattern Matching

Ещё пример…

Создадим собственный класс `Logic` с операциями `&&&, |||, neg` — описывающие сигнатуры и взаимосвязи конъюнкции, дизъюнкции и отрицания. Затем создадим тип данных `MyBool` и воплощения для него класса `Logic`.

Формально класс описывается следующим образом:

```
class Logic a where
   neg  :: a -> a
  (&&&) :: a -> a -> a
```

```
  (|||) :: a -> a -> a
```

Можно указать простейшие связи между этими операциями:

```
x &&& y = neg ( (neg x) ||| (neg y) )
x ||| y = neg ( (neg x) &&& (neg y) )
```

Чтобы уменьшить число скобок, полезно указывать приоритет заданных операций и их ассоциативность:

```
infixl 7 &&&
infixl 5 |||
```

Вот примерный образец минимального решения:

```
class Logic a where
   neg  :: a -> a
  (&&&) :: a -> a -> a
  (|||) :: a -> a -> a

data MyBool = T | F deriving (Eq, Read, Show)
T `kon` T = T
_ `kon` _ = F
F `diz` F = F
_ `diz` _ = T
ot T = F
ot F = T

instance Logic MyBool where
  neg   = ot
  (&&&) = kon
  (|||) = diz
```

Ну или более полный пример:

```
class Logic a where
   neg  :: a -> a
   (&&&) :: a -> a -> a
   (|||) :: a -> a -> a
   (==>) :: a -> a -> a
   (!)   :: a -> a -> a

   neg x   = x ! x -- neg IS NOT MINIMAL DEF!
-- x  y = (x ! y) ! (x ! y)
-- x ||| y = (x ! x) ! (y ! y)
   x  !  y = neg (x &&& y)
   x &&& y = neg ( (neg x) ||| (neg y) )
   x ||| y = neg ( (neg x) &&& (neg y) )
   x ==> y = (neg x) ||| y

infixl 4 `neg`
infixl 7 &&&
infixl 5 |||
```

```
data MyBool = T | F deriving (Eq, Read, Show)
T `kon` T = T
_ `kon` _ = F
F `diz` F = F
_ `diz` _ = T
ot T = F
ot F = T

instance Logic MyBool where
  neg   = ot
  (&&&) = kon
-- (|||) = diz

instance Logic Bool where
  neg   = not
  (&&&) = (&&)
  (|||) = (||)

instance Logic Int where
  neg 0 = 1
  neg _ = 0
  (&&&) = (*)
```

К сожалению, для примеров с числами необходима аннотация типов:

```
{-
*Main> 1 &&& 0

<interactive>:1:1: error:
    * Ambiguous type variable `a0' arising from a use of `print'
      prevents the constraint `(Show a0)' from being solved.
      Probable fix: use a type annotation to specify what `a0' should be.
      These potential instances exist:
        instance Show Ordering -- Defined in `GHC.Show'
        instance Show Integer -- Defined in `GHC.Show'
        instance Show a => Show (Maybe a) -- Defined in `GHC.Show'
        ...plus 22 others
        ...plus 18 instances involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    * In a stmt of an interactive GHCi command: print it
*Main> :t (1 &&& 0)
(1 &&& 0) :: (Logic a, Num a) => a
*Main>   (1::Int) &&& (0::Int)
0
*Main>   (1::Int) &&& (1::Int)
1
-}
```

Как работает АТД

Как уже упоминалось выше, классы типов полезны при создании «АТД» (Абстрактные Типы Данных — Abstract Data Type — англ.) Это позволяет реализовать ситуацию, ко-

гда разные разработчики создают различные уровни абстракции: архитекторы — АТД с абстрактными методами, программисты — либо воплощения этих АТД для конкретных данных, либо использования методов АТД в следующих «слоях абстракции». А воображаемые пользователи могут использовать как прелести реализации, так и возможности создавать методы, независимые от реализации.

В «невоображаемой» реальности ситуация сильно осложняется и, по крайней мере, на Haskell мы порой вынуждены использовать ещё более громоздкие виды абстракции: GADT, Type Family, Type Functional Dependency и т.п.

В этом разделе рассмотрим идею АТД и простое использование этой идеи. Создадим модуль с задекларированной функцией-предикатом:

```
module Sosedy (Sosedy, (~~)) where

class Sosedy a where
     (~~) :: a -> a -> Bool
```

Данный класс и его метод описывает ситуацию «быть соседом» вне зависимости от реализации. Далее, некто, или мы, можем на этой базе создать новые функции, не зависящие от реализации, напр., предикат, описывающий свойство «быть соседом симметрично»:

```
module SimSosedy where

import Sosedy

(~~~) :: (Sosedy a) => a -> a -> Bool
a ~~~ b = (a ~~ b) && (b ~~ a)
```

Ну и, наконец, создадим различные реализации (воплощения) для данного класса и метода:

```
module SosedyInst((~~),(~~~)) where

import Sosedy
import SimSosedy
import Data.Char(ord)

instance Sosedy Int where
  x ~~ y = (abs (x-y) == 1)

instance Sosedy Char where
  x ~~ y = ((ord(x) - ord(y)) == 1)

instance Sosedy Double where
  x ~~ y = (abs (x-y) < 1.0)
```

Указаны воплощения класса `Sosedy` и функции `(~~)` для типов **`Int`**, **`Char`** и **`Double`**.

Заметим, что функция `(~~~)` будет реализована автоматически.

К сожалению, Haskell так просто не позволяет сделать воплощение для других классов, например, таким образом:

```
instance (Integral a) => Sosedy a where
  x ~~ y = (abs(x-y) == 1)

instance (RealFloat a) => Sosedy a where
  x ~~ y = (abs (x-y) < 1.0)
```

Здесь будет ошибка `Duplicate instance declarations...`, компилятор не позволит сделать два воплощения для произвольного типа `a`, хотя и с разными ограничениями — тем не менее, если бы объявление было одно, только для класса `Integral a`, то проверка для типа `Int` была бы обеспечена:

```
> (1::Int) ~~ (2::Int)
True
```

Иными словами, семантика объявления `(RealFloat a) =>` работает не совсем так, как будто мы объявляем метод для какого-то множества типов (в данном случае для `Integral a` или `RealFloat a`), а так, что мы требуем ограничение на типы для каких-то компонентов типов в воплощении (будет потом ещё, подробнее) и тут возникает коллизия.

Ссылки по теме:

wikipedia: Абстрактный тип данных

Abstract Data Types

Предопределённые классы типов в Haskell

Haskell имеет большое предопределённое множество полезных классов типов в своей «экосистеме». О многих из них не нужно заботиться, в большинстве случаев достаточно указания `deriving NameOfClass` и компилятор автоматически выведет нужное воплощение класса для `NameOfClass` создаваемого типа.

Вспомним, у нас было определение типа данных для цвета:

```
data Color = Red | Green | Blue | Black | White
```

В данном случае, многие классы типов могут быть выведены автоматически для этого типа данных, например

```
data Color = Red | Green | Blue | Black | White
  deriving (Eq, Ord, Show)
```

Это означает, если у нас есть значение `x::Color` и `y::Color`, то мы можем сразу вычислить

```
> x = Red
> y = Blue
> x == y
False
> x < y
True
> x
Red
```

Иными словами, класс **Eq** означает возможность сравнения на равенство, **Ord** — сравнения по порядку, **Show** — строковое представление. А **deriving** «просит» компилятор реализовать эти возможности для вводимого типа данных «естественным» образом.

В нестандартных или более сложных типах данных необходимо описывать воплощение методов класса.

Так, например, для цвета мы могли бы задать свое строковое представление:

```
instance Show Color where
  show Red   = "rot"
  show Green = "grün"
  show Blue  = "blau"
  show Black = "schwarz"
  show White = "weiss"
```

и тогда в ghci

```
> Red
rot
```

Здесь всё просто, декларация **instance Show** Color указывает, что мы определяем воплощение для простого типа данных (без параметра).

Определение класса **Eq** и его воплощение для списков более сложно и выглядит примерно так:

```
class Eq a  where
    (==), (/=)           :: a -> a -> Bool

    x /= y               = not (x == y)
    x == y               = not (x /= y)

infix  4  ==, /=

instance (Eq a) => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _xs    == _ys    = False
```

Data.Eq

Заметим, что мы требуем при определении воплощения, чтобы было выполнено (**Eq** a), т.е. тип a, который лежит внутри списков, должен быть доступен для проверки на равенство. Иначе, мы не сможем сделать проверку на равенство и для списков.

Аналогично, для Ord.

```
data Ordering = LT | EQ | GT deriving ...

class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
```

```
    max, min             :: a -> a -> a

    compare x y = if x == y
                  then EQ
                  else if x <= y
                    then LT
                    else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y

infix  4  <, <=, >=, >

instance (Ord a) => Ord [a] where
    compare []     []     = EQ
    compare []     (_:_)  = LT
    compare (_:_)  []     = GT
    compare (x:xs) (y:ys) = case compare x y of
                                EQ    -> compare xs ys
                                other -> other
```

Как обычно, определив всего одну функцию **compare** для списков, мы автоматически получаем воплощение функций <, >, <=, >=, **max**, **min** для списков.

Отметим, что в определении класса (**Ord** a) указание (**Eq** a) показывает, что мы определяем, фактически, подкласс (**Ord** a) для ранее заданного класса (**Eq** a).

Для воплощений классов параметрических типов указания (**Eq** a) или (**Ord** a) показывают, что существует требование контекста, т.е. некоторая составляющая типа данных, для которого мы определяем воплощения методов, должна удовлетворять данному условию. С другой стороны, когда мы смотрим тип, например, функции (+):

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

то это означает, что (+) будет работать с аргументами любого типа данных, которые «подписаны» на класс **Num** (т.е. имеют воплощение для этого класса).

Выводы о типах и классах

Воплощение данного класса для для данного типа данных должно быть единственно. Напомним, что фактически это означает построение алгебраической системы интерпретацией данной сигнатуры для операций внутри данного множества-носителя. Если нам нужно альтернативное воплощение данного класса для для данного типа данных, то с помощью декларации **newtype** мы создаем изоморфный тип данных, для которого можно сделать иное воплощение класса.

Рассмотрение некоторых из множества других предопределенных классов, особенно числовых классов, оставим для следующей лекции.