Типы в Haskell

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Среда, 9 ноября года

План занятия

1. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Замечания из алгебры
   - Использование типов-сумм
2. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки

1 Алгебраические типы данных
- Откуда берётся тип-сумма
- Что такое тип-сумма
- Примеры типов-сумм
- Замечания из алгебры
- Использование типов-сумм

2 Классы типов
- Что и зачем
- Для параметризованных типов
- Прочие плюшки

Тип-произведение

- `int x;` принимает значения: $0, 1, -1, 2, -2, \dots$
- `struct { int x, y; }` принимает значения: $(0,0), (0,1), (1,0), (-1,0)\dots$
- Множество значений `struct` есть декартово произведение значений составных элементов.
- Поэтому `struct` иногда называют типом-произведением.
- Тип-произведение — *составной тип*, собирается из более маленьких.
- Есть практически во всех языках.

Упражнение

- Пусть у интернет-магазина есть три способа оплаты:
  1. Банковской картой, нужно знать её данные.
  2. Наличными при получении, ничего дополнительно знать не нужно.
  3. Выставление счёта на QIWI-кошелёк, нужно знать номер телефона.
- Требуется создать тип данных «способ оплаты», который можно хранить и обрабатывать.
- Иногда требуется преобразовывать способ оплаты в строку.
- Иногда требуется понимать, надо ли что-то делать с сервере для проведения оплаты (если да — положить в очередь).

Упражнение (C-подход)

```
enum PaymentMethodType { CARD, CASH, QIWI_BILL };
struct PaymentMethod {
  PaymentMethodType type;
  CardInfo card_info;
  char phone[20];
};
```

- Надо везде явно смотреть на поле `type` и городить if'ы.
- Для обработки пишем функции вроде `to_string`, которые разбирают случаи.
- Можем случайно обратиться к `card_info`, если не проверим способ оплаты.
- Храним больше байт, чем реально надо (можно `union`, но там есть свои проблемы).

Упражнение (ООП-подход)

- Вводим интерфейс `PaymentMethod`, а сами методы делаем подклассами.
- Общие функции вроде `to_string` вносим в интерфейс.
- Специфичные функции либо руками разбирают случаи, либо используют Visitor.
- Так обычно и делают.
- Можно добавлять как новые классы, так и новые операции с объектами.

Упражнение (if'ы)

```
bool need_processing(PaymentMethod *m) {
  if (dynamic_cast<CardPayment*>(m)) {
    return true;
  }
  if (dynamic_cast<CashPayment*>(m)) {
    return false;
  }
  assert(false);
}
```

- Можно обрабатывать несколько случаев одинаково.
- Узнаем об отсутствующем `if` только во время выполнения, если есть тест.
- Компилятор не проверит, что мы ничего не забыли.

Упражнение (Visitor)

```
bool result;
class NeedProcessingVisitor {
  void visit(CardPayment *p) { result = true; }
  void visit(CashPayment *p) { result = false; }
}
bool need_processing(PaymentMethod *m) {
  m->accept(NeedProcessingVisitor());
  return result;
}
```

- Очень много кода не по делу.
- Создаётся целый объект в памяти и идут пляски с возвращаемым значением.

Тип-сумма

- Можно ввести *тип-сумму*: множество его допустимых значений равно *дизъюнктному объединению*[1] допустимых значений составных частей.
- Чтобы обобщить до суммы произвольных типов, можно каждому значению составной части добавить «тэг».
- Пример: тип «способ оплаты»:

```
data PaymentMethod = BankCard String | Cash | Qiwi String
a = BankCard "1234 5678 9012 3456"
b = Cash
c = Qiwi "+7 812 000 00 00"
```

- Обычно встречается в функциональных языках.
- Именно его наличие обычно подразумевают под «наличием алгебраических типов данных».

[1] объединение попарно непересекающихся множеств

Тип-сумма: подробности

```
data PaymentMethod = BankCard String | Cash | Qiwi String
```

- PaymentMethod называется *конструктором типа*.
- BankCard, Cash, Qiwi называются «конструкторами данных», являются теми самыми «тэгами».
- Не путать с конструкторами в ООП!
- И конструктор типа, и конструктор данных долнжы начинаться с большой буквы.
- Работает с pattern matching:

```
to_string (BankCard num) = "BankCard " ++ num
to_string Cash           = "Cash"
to_string (Qiwi phone)   = "Qiwi " ++ phone
```

- Можно дописать в конец строки с data слова deriving Show, чтобы GHCI мог выводить значения типа PaymentMethod.

1. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Замечания из алгебры
   - Использование типов-сумм
2. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки

CharOrNotFound

Поиск элемента по номеру:

```
data CharOrNotFound = NotFound | Found Char deriving Show

getItem :: [Char] -> Int -> CharOrNotFound
getItem (x:_ ) 0           = Found x
getItem (x:xs) n | n > 0 = getItem xs (n - 1)
getItem _      _           = NotFound
```

- Не требуются «магические значения» для ситуации «элемент не найден».
- Компилятор проверяют, что мы всегда обрабатываем оба случая.
- По типу функции сразу понятно, что она может вернуть.
- Нет исключений; функции чистые.

Maybe

Можно обобщить до *параметризованного типа*:

```
data GetResult a = NotFound | Found a deriving Show

getItem :: [a] -> Int -> GetResult a
getItem (x:_ ) 0           = Found x
getItem (x:xs) n | n > 0 = getItem xs (n - 1)
getItem _ _                = NotFound
```

- `GetResult` — это не тип, это *конструктор типа*.
- `a` — единственный параметр этого конструктора.
- А вот `GetResult Char` — уже конкретный тип:
  `data GetResult Char = NotFound | Found Char`
- В Haskell такой тип называется `Maybe`.
- А в Java есть generic-тип `Optional<>`).
- На самом деле `[Int]` — это сахар для `[] Int`.

Упражнение

- Напишите тип для функции getItem, если бы она использовала Maybe:

```
-- Уже объявлен в языке, писать не надо.
data Maybe a = Nothing | Just a

getItem :: [a] -> Int -> ???
```

- Напишите функцию getItem.
- Удалите явное указание типа, проверьте, какой тип вывелся автоматически (`:t getItem` в GHCI).

Упражнение

- Напишите тип для функции getItem, если бы она использовала Maybe:

```
-- Уже объявлен в языке, писать не надо.
data Maybe a = Nothing | Just a

getItem :: [a] -> Int -> ???
```

- Напишите функцию getItem.
- Удалите явное указание типа, проверьте, какой тип вывелся автоматически (:t getItem в GHCI).

```
getItem :: [a] -> Int -> Maybe a
getItem (x:_)  0           = Just x
getItem (x:xs) n | n > 0 = getItem xs (n - 1)
getItem _ _                = Nothing
```

Either

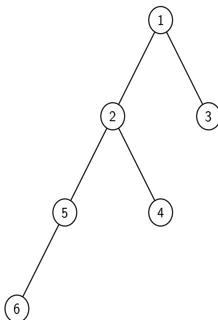
На случай, если хотим сообщить об ошибке:

```
-- Уже объявлен в языке, писать не надо.
data Either a b = Left a | Right b

parseBool :: String -> Either String Bool
parseBool "true"  = Right True
parseBool "false" = Right False
parseBool x       = Left ("Invalid value: " ++ x)
```

- Обычно за `Right` принимает успешное вычисление («правильный» результат).
- А за `Left` — сообщение об ошибке («левый» результат).

Двоичная куча

```
data Heap = Nil | Node Int Heap Heap deriving Show

Node 1 (Node 2 (Node 5 (Node 6 Nil Nil) Nil)
               (Node 4 Nil Nil))
       (Node 3 Nil Nil)
```



Min-Max куча

Так как вершина всегда хранит либо минимум, либо максимум, это можно указать прямо в типе. Тогда точно не запутаемся, где какая вершина, и не надо это явно считать и передавать.

```
data MinMaxHeap = Nil | MinNode Int MinMaxHeap MinMaxHeap
                      | MaxNode Int MinMaxHeap MinMaxHeap
```

Можно даже строже, у нас есть взаимная рекурсия:

```
data MinMaxHeap = Nil1 | MinNode Int MaxMinHeap MaxMinHeap
data MaxMinHeap = Nil2 | MaxNode Int MinMaxHeap MinMaxHeap
```

К сожалению, назвать оба конструктора данных `Nil` нельзя.

Дерево разбора выражения

```
data BinOp = Add | Sub | Mul | Div deriving Show
data Tree = Number Int
          | Reference String
          | BinaryOperation BinOp Tree Tree
          deriving Show
-- Глубокий pattern matching
fold' Sub (Reference a) (Reference b) | a == b = Number 0
fold' Mul (Reference _) (Number 0)              = Number 0
fold' Mul (Number 0)    (Reference _)           = Number 0
fold' op a b = BinaryOperation op a b
-- Рекурсивное сворачивание выражений
fold (BinaryOperation op a b) =
    fold' op (fold a) (fold b)
fold x = x
```

Односвязные списки

```
data List a = Empty | Cons a (List a) deriving Show

head' (Cons x _ ) = x
tail' (Cons _ xs) = xs
```

- Выше написано почти определение встроенного списка.
- `[]` — это сахар для конструктора `Empty`.
- `:` — это сахар для конструктора `Cons`.
- Конкретно в Haskell любые структуры бывают бесконечными из-за ленивости, не только списки.
- Например, бесконечное двоичное дерево имеет право на жизнь.

Упражнение

1. Напишите тип «двоичное дерево» (`Tree`), в котором у каждой вершины либо 0 детей, либо 2, а каждая вершина содержит значение типа `Int`.
2. Добавьте `deriving Show`.
3. Напишите функцию `tree_sum`, которая считает сумму в данном ей дереве.
4. Удалите разбор какого-нибудь случая и запустите GHCI так: `ghci -W file.hs`
5. Убедитесь, что выпало предупреждение о неразобранном случае.
6. Напишите функцию, которая возвращает бесконечное дерево `Int`'ов, где каждая вершина содержит номер своего уровня.
7. Выведите результат на экран, объясните увиденное.

Промежуточные итоги

- Под «алгебраическими типами данных» обычно подразумевают поддержку типов-сумм вместе с типами-произведениями *на уровне языка*. Такая поддержка даёт:
  1. Более наглядные типы.
  2. Невозможность обратиться к данным из другого «случая».
  3. Pattern matching и сильное упрощение кода.
  4. Предупреждения компилятора о нерассмотренных случаях (ключ `-W` для GHC/GHCI).
- Добавлять случаи в тип-сумму обычно после объявления нельзя.
- В языках без типов-сумм, но с ООП, обычно используется:
  - Наследование от общего предка вместо типов-сумм.
  - Visitor вместо pattern mactching.
- Типы-суммы очень часто возникают при работе с AST.
- В Haskell любой пользовательский тип является типом-суммой (возможно, из одного слагаемого).
- В Haskell можно параметризовать пользовательские типы.

1. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Замечания из алгебры
   - Использование типов-сумм
2. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки

Мощь типов-сумм

- Любой вообразимый тип без стрелок (т.е. без функций) можно представить, как тип-сумму:

```
data Bool = True | False
data Int = 0 | 1 | -1 | 2 | -2 | ...
data (Int, Int) = (0,0) | (0,1) | (1,0) | ...
data [Int] = [] | [0] | [0,0] | [1] | ...
```

- Когда мы пишем параметрирозованный тип, мы на самом деле пишем лишь его «шаблон» или функцию, которая возвращает «реальный» тип: нельзя определить множество значений `Maybe a`, не зная множество значений `a`.
- Это называется «тип высшего порядка». Этакие шаблоны из C++/generic'и из Java.
- Напоминание: отсюда возникло название «*конструктор типа*».

Количество значений

- Если заменить каждый тип на количество возможных значений, то названия «тип-сумма» и «тип-произведение» отображают операции, которые надо производить с этими количествами:

```
data Bool = True | False  -- Два значения
data Foo = Bool | (Bool, Bool) -- 2 + 2 * 2 = 6 значений
```

- В типах тоже есть дистрибутивность умножения и сложения:

```
-- Названия конструктора данных и типа могут совпадать
data Foo1 = Foo1 (Bool, Maybe Int)
data Foo2 = FalseNoInt
          | FalseWithInt Int
          | TrueNoInt
          | TrueWithInt Int
```

- Enum'ы (перечисления) — это типы-суммы, в которых каждое слагаемое имеет ровно одно значение:

```
data BinOp = Add | Sub | Mul | Div
```

Алгебры

Для понимания этимологии словосочетания «алгебраический тип»:

- *Алгебра* — это множество A, на котором ввели какие-то функции.
- Каждая функция принимает некоторое число аргументов из A и возвращает элемент из A.
- Разные функции могут принимать разное число аргументов.
- Никаких дополнительных требований нет.
- Рациональные числа являются алгеброй с операциями:
  1. $+\colon \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
  2. $-\colon \mathbb{Q} \to \mathbb{Q}$ (унарный минус)
- Также алгебрами являются все группы:
  1. Функция-константа: $e\colon G$
  2. Взятие обратного элемента: $x^{-1}\colon G \to G$
  3. Умножение элементов: $\cdot\colon G \times G \to G$
  4. Дополнительно требуются свойства группы.
- Мораль: если сказали «введём операцию с элементами» — уже появилась алгебра.

Алгебра типов

- Типы данных тоже образуют алгебру (обозначим их множество за T):
  1. `Int`: T
  2. `Char`: T
  3. Операция «сделать список»: $[]\colon T \to T$
  4. Операция «сделать кортеж (тип-произведение)»: $(,)\colon T \times T \to T$
  5. Операция «тип-сумма»: $|\colon T \times T \to T$
- *Алгебраический тип* — это тип, собранный из более простых.
- Активно используются и в императивных языках: `vector<int>`, `int[]`, `struct {}`
- Операцию «сделать кортеж» можно обозначить как умножение типов (так сделано в OCaml).
- В императивных языках алгебра типов обычно более скудная.
- Типы высшего порядка (параметризованные) — это на самом деле функции над типами, то есть в алгебру в Haskell входят не только типы; подробностей не будет.

1. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Замечания из алгебры
   - Использование типов-сумм
2. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки

Архиватор

- Мы пишем консольный архиватор для одного файла (вроде gzip).
- Хотим написать тип, который хранит параметры командной строки: входной файл, выходной файл, сжатие/расжатие.

Попытка 1:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out  :: String,
                   op   :: Operation }
                 deriving Show
```

Замечание: тут у нас тип-сумма с одним-конструктором, простых типов-произведений в Haskell нет.

Архиватор

- Мы пишем консольный архиватор для одного файла (вроде gzip).
- Хотим написать тип, который хранит параметры командной строки: входной файл, выходной файл, сжатие/расжатие.

Попытка 1:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out  :: String,
                   op   :: Operation }
                 deriving Show
```

Замечание: тут у нас тип-сумма с одним-конструктором, простых типов-произведений в Haskell нет. А теперь хотим добавить пароль для запаковки/распаковки (нужен не всегда)

Архиватор

Попытка 2:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out  :: String,
                   op   :: Operation,
                   pwd  :: Maybe String }
                deriving Show
```

Архиватор

Попытка 2:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out  :: String,
                   op   :: Operation,
                   pwd  :: Maybe String }
                 deriving Show
```

Новые параметры: уровень логирования (`Int`), флаг «не перезаписывать файл при создании», они всегда заданы.

Архиватор

Попытка 3:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp :: String,
                   out :: String,
                   op  :: Operation,
                   pwd :: Maybe String,
                   logLevel  :: Int,
                   overwrite :: Bool }
                 deriving Show
```

Архиватор

Попытка 3:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp :: String,
                   out :: String,
                   op  :: Operation,
                   pwd :: Maybe String,
                   logLevel  :: Int,
                   overwrite :: Bool }
                 deriving Show
```

- Новые параметры: формат архива, степень сжатия.

Архиватор

Попытка 3:

```
data Operation = Compress | Decompress deriving Show
data Args = Args { inp :: String,
                   out :: String,
                   op  :: Operation,
                   pwd :: Maybe String,
                   logLevel  :: Int,
                   overwrite :: Bool }
                 deriving Show
```

- Новые параметры: формат архива, степень сжатия.
- Если просто добавить поле `Maybe (Format, Int)`, то появится инвариант и структура может стать некорректной.
- Можно убрать `op` и заменить его на новый `Maybe`.

Архиватор

Попытка 4:

```
-- data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out :: String,
                   comprArgs :: Maybe (Format, Int),
                   pwd :: Maybe String,
                   logLevel  :: Int,
                   overwrite :: Bool }
                 deriving Show
```

Архиватор

Попытка 4:

```
-- data Operation = Compress | Decompress deriving Show
data Args = Args { inp  :: String,
                   out :: String,
                   comprArgs :: Maybe (Format, Int),
                   pwd :: Maybe String,
                   logLevel  :: Int,
                   overwrite :: Bool }
                 deriving Show
```

- А если теперь добавим третью операцию, то всё сломается.
- А если она при этом будет только читать архив (но никуда не писать), всё сломается ещё раз.
- При взгляде на структуру неочевидно, как узнать, сжимаем или разжимаем, хотя вся информация в ней есть.

Архиватор

Намного лучше разделить параметры на совсем общие и относящиеся к операции.
Попытка 5, финальная:

```
data CommonArgs = Common { logLevel  :: Int }
data OperationArgs = -- Заодно имена лучше отражают суть.
    Compress   { input     :: String, archive :: String,
                 pwd       :: Maybe String,
                 overwrite :: Bool,
                 format    :: Format, level :: Int }
  | Decompress { archive   :: String, output :: String,
                 pwd       :: Maybe String,
                 overwrite :: Bool }
data Args = Args (CommonArgs, OperationArgs)
```

Мораль: лучше думать в терминах «какие по смыслу бывают ситуации», чем «как хранить всё одновременно»

Хранение URL

URL-адреса бывают:

- Относительные: `../images/facepalm.jpg`.
- Абсолютные, бывают:
  - На том же домене: `sewiki/index.php`.
  - На другом домене, причём:
    - Та же схема (протокол): `google.com/humans.txt`
    - Другая схема: `ftp://mirror.yandex.ru/`

Можно закодировать так[2]:

```
data URL = URL (Maybe (Maybe (Maybe String, String))) String
URL Nothing              "../images/facepalm.jpg"
URL (Just Nothing) "sewiki/index.php"
URL (Just (Just (Nothing    , "google.com"))) "humans.txt"
URL (Just (Just (Just "ftp", "mirror.yandex.ru"))) ""
```

Ужасно, не правда ли?

[2] True story: раздел «Thinking in Sum Types» по ссылке

Хранение URL

URL-адреса бывают:

- Относительные: `../images/facepalm.jpg`.
- Абсолютные, бывают:
  - На том же домене: `sewiki/index.php`.
  - На другом домене, причём:
    - Та же схема (протокол): `google.com/humans.txt`
    - Другая схема: `ftp://mirror.yandex.ru/`

А можно так:

```
data URL = Relative String
         | Absolute String
         | OtherDomain { domain :: String, path :: String }
         | FullUrl     { schema :: String,
                         domain :: String, path :: String }
```

Мораль: иногда может помочь «раскрыть по дистрибутивности».

[2] True story: раздел «Thinking in Sum Types» по ссылке

Физика

```
data Time = Sec  Double | Min  Double
          | Msec Double | Usec Double
          deriving Show
diff (Sec x) (Sec y) = Sec (x - y)
diff x y = diff (to_sec x) (to_sec y)

diff' x y = Sec (x' - y')
  where (Sec x') = to_sec x -- pattern matching
        (Sec y') = to_sec y

to_sec (Sec x) = Sec x
to_sec (Msec x) = Sec (x / 1000)
to_sec (Usec x) = Sec (x / 1000000)
to_sec (Min x) = Sec (x * 60)
```

Физика

```
data Time = Sec  Double | Min  Double
          | Msec Double | Usec Double
          deriving Show
diff (Sec x) (Sec y) = Sec (x - y)
diff x y = diff (to_sec x) (to_sec y)

diff' x y = Sec (x' - y')
  where (Sec x') = to_sec x -- pattern matching
        (Sec y') = to_sec y
```

- Компилятор обяжет вас указать единицы измерения во всех местах.
- Все проверки будут сделаны во время компиляции.
- Можно работать в удобных единицах и автоматически конвертировать из остальных.

Резюме-1

- Типы-суммы очень естественны, если про них думать.
- Полезны, когда возможные значения типа разбиваются на фиксированное число групп, которые сильно отличаются.
- Если число групп неизвестно — поможет ООП, наследование и Visitor.
- Если группы одинаковы — возможно, имеет смысл сделать тип-сумму внутри, а не снаружи:

```
data Tree = Sum Tree Tree | Mul Tree Tree
-- против
data BinOp = Sum | Mul | Sub
data Tree = BinaryOperation BinOp Tree Tree
```

- В императивных языках типов-сумм обычно нет, используем наследование и Visitor.

Резюме-2

- Типы-суммы позволяют выражать многие инварианты данных («флаг степени сжатия есть только при сжатии файла») на уровне типа.
- Компилятор может проверять сохранение этих инвариантов.
- Компилятор может проверить то, что все случаи везде разобраны.
- При поддержке типов-сумм на уровне языка код получается намного короче и чище, чем через ООП.
- Pattern matching + алгебраические типы = мощь.

1. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Замечания из алгебры
   - Использование типов-сумм

2. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки

Pattern Matching и ==

```
data IntList = Empty | Cons Int IntList deriving Show
a = Cons 1 (Cons 2 Empty)
b = Cons 1 (Cons 2 Empty)
c = Cons 1 (Cons 3 Empty)

isA :: IntList -> Bool
isA (Cons 1 (Cons 2 Empty)) = True
isA _                       = False

isA a -- True
isA b -- True
isA c -- False
a == b  -- ошибка компиляции?
a == c  -- ошибка компиляции?
```

Eq

- Pattern Matching — конструкция на уровне языка.
- == — просто некоторая функция с таким названием.
- В C++ мы бы написали перегрузку функции/оператора.
- В Haskell пишем так:

```
instance Eq IntList where
  Empty         == Empty         = True
  (Cons x xs)   == (Cons y ys)   = (x == y) && (xs == ys)
  _             == _             = False

a == b   -- True
a == c   -- False
b == c   -- False
Empty == Empty               -- True
Empty /= (Cons 1 Empty)      -- True, /= тоже работает
```

class Eq

- Eq — это *класс типов*, который описывает, что к типам можно применять определённые функции:

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

- Говорим, что тип a лежит в классе Eq тогда и только тогда, когда для него есть функции (==) и (/=)
- Класс типов — это такой «интерфейс» для типов.
- Некоторые функции требуют, чтобы параметры были в определённых классах:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

- Слово `instance` на предыдущем слайде добавляло `IntList` в класс Eq.
- Не путать с классами объектов из ООП!

Реализации по умолчанию

- Для `IntList` мы реализовали только `==`, а `/=` получили автоматом.
- В классе можно указывать реализацию по умолчанию:

```
class Eq a where
  (==) :: a -> a -> Bool
  (==) a b = not (a /= b)
  (/=) :: a -> a -> Bool
  (/=) a b = not (a == b)
```

- Тогда *минимальное полное определение* для `Eq` — это либо `==`, либо `/=`.

Класс Eq для списков

- Пусть есть свой класс для списков:

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

- Разумно считать, что списки равны, если равны элементы:

```
instance Eq (List a) where
  Empty       == Empty       = True
  (Cons x xs) == (Cons y ys) = (x == y) && (xs == ys)
  _ == _                     = False
```

- Не скомпилируется, потому что элементы произвольного типа a нельзя сравнивать.
- Надо добавить *контекст* — сказать, что списки можно сравнивать только если можно сранивать элементы:

```
instance Eq a => Eq (List a) where
```

Классы для структур данных

- Иногда хочется указать, что структура данных обладает некоторым свойством:

```
class Mappable f where
  map' :: (a -> b) -> f a -> f b
```

- Mappable — что-то, содержащее элементы произвольного типа, к чему можно делать map'.
- Можно реализовать:

```
instance Mappable List where
  map' _ Empty = Empty
  map' f (Cons x xs) = Cons (f x) (map' f xs)

map' (+1) (Cons 1 (Cons 10 Empty))
-- Cons 2 (Cons 11 Empty)
```

Functor

- Аналог Mappable в Haskell называется Functor, в нём определена функция fmap.
- fmap должна удовлетворять некоторым аксиомам, но их компилятор сам не проверит:

```
fmap id x == x
fmap (f . g) x == fmap f (fmap g x)
```

- После реализации Functor наш новый тип можно использовать в том числе в старых функциях:

```
a = (Cons 1 (Cons 2 (Cons 3 Empty)))
fmap (+1) [1,2,3]
fmap (+1) a
-- Встроенный оператор замены значений на константу
10 <$ [1,2,3]  -- [10,10,10]
10 <$ a
```

Свои классы типов

Можно создать свой собственный класс:

```
class HasSize a where
  size :: a -> Int

instance HasSize [a] where
  size xs = length xs

data Tree a = Nil | Node a (Tree a) (Tree a)
instance HasSize (Tree a) where
  size Nil = 0
  size (Node _ l r) = 1 + size l + size r
```

- Можно определить даже для старых типов. В Java/C++ такое можно сделать только внешними перегруженными функциями.
- Новые функции можно использовать где угодно с любыми типами.

Стандартные классы

- `Show` — то, что можно вывести на экран.
- `Eq` — операторы == и /=.
- `Ord` — операторы <, <= и прочие.
- `Functor` — структура данных, на которой есть `map`.
- `Foldable` — структура данных, на которой есть `foldr` (по сути, умеет разворачиваться в список).
- Для первых трёх Haskell умеет сам генерировать адекватные реализации, если попросить:

```
data List a = Empty | Cons a (List a)
          deriving (Show, Eq, Ord)
```

- Порядок «лексикографический» (более ранний конструктор меньше).

Зависимости классов

- Как определить `Ord`? Хотим, чтобы тип класса `Ord` автоматически подходил везде, где нужен лишь `Eq`.
- Можно скопировать `==` и `/=` в `Ord`, но тогда:
  1. Можно случайно реализовать и `Ord`, и `Eq`, причём по-разному.
  2. Получаем «утиную типизацию»: один класс вкладывается в другой, если есть функции с такими же названиями.
  3. Это нехорошо: придётся следить, что названия функций вообще нигде не пересекаются.
- Можно сказать, что `Ord` определяет только новые операторы, но тогда мы можем определить тип с `<`, но без `==`, что странно.
- Решение: класс `Ord` требует, чтобы тип также был в классе `Eq`:

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

- Если где-то пишем контекст `Ord a`, то `Eq a` появляется неявно.
- В частности, в реализациях по умолчанию в `Ord a` можно использовать `==` и `/=`.

Автовывод контекста

```
-- Ord a => a -> a -> a
max' a b = if a > b then a else b

-- (Functor f, Eq a) => a -> f a -> f (Maybe a)
removeByValue x ys = fmap f ys
  where
    f y | x == y    = Nothing
        | otherwise = Just y
```

Если в файле не видно разных функций с одинаковым названием из разных классов, то компилятор может автоматически вывести ограничения на типы (контекст).

Резюме

- Альтернатива классам типов — интерфейсы из ООП или перегрузки функций.
- Перегрузки функций не отражают связи между разными функциями (вроде == и /=).
- Интерфейсы из ООП *обычно* надо определять в момент создания каждого типа (не добавить интерфейс к уже существующему).
- Интерфейсы из ООП *обычно* не позволяют делать реализации по умолчанию — надо писать руками.
- Классы типов всё это позволяют.
- Компилятор умеет автоматически выводить нужный контекст.
- В Haskell классов типов используется везде, где есть хотя бы доля обобщаемости.

Упражнение

1. Пусть имеется следующее дерево со значениями в листьях и произвольным числом детей:
```
data Tree a = Leaf a | Node [Tree a] deriving Show
```
2. Добавьте Tree в класс Eq. Подсказка: списки там уже есть.
3. Пусть есть такой класс для структур, в которых можно развернуть порядок элементов:
```
class Reversible f where
   reverse' :: f a -> f a
```
4. Добавьте списки в этот класс:
```
instance Reversible [] where
```
5. Добавьте дерево в этот класс:
```
instance Reversible Tree where
```