Типы в Haskell

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 27 ноября 2017 года

План занятия

1. Ещё про Haskell
   - Функции высшего порядка
   - Статический полиморфизм функций
2. Алгебраические типы данных
   - Откуда берётся тип-сумма
   - Что такое тип-сумма
   - Примеры типов-сумм
   - Использование типов-сумм
3. Классы типов
   - Что и зачем
   - Для параметризованных типов
   - Прочие плюшки
4. Бонус
   - Отладка
   - Резюме

Напоминание

- Функция является *функцией высшего порядка*, если она в качестве одного из аргументов принимает другую функцию.
- Пример: `map`. Он первым параметром принимает функцию, которая преобразует элементы списка.
- Функции высшего порядка является основными кирпичиками в функциональном программировании.

Ещё один паттерн

```
sum (x:xs) = x + sum xs
sum _ = 0

prod (x:xs) = x * prod xs
prod _ = 1

max (x:xs) = max x (max xs)
max x = -1

concat (x:xs) = x ++ (concat xs)
concat _ = ""
```

Что общего?

Ещё один паттерн

```
sum (x:xs) = x + sum xs
sum _ = 0

prod (x:xs) = x * prod xs
prod _ = 1

max (x:xs) = max x (max xs)
max x = -1

concat (x:xs) = x ++ (concat xs)
concat _ = ""
```

Что общего?

- Все эти функции считают функцию от множества элементов.
- Для пересчёта требуется знать только текущее значение и очередной элемент.

Правая свёртка

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a _ = a

sum    xs = foldr (+)  0    xs
prod   xs = foldr (*)  1    xs
max    xs = foldr max  (-1) xs
concat xs = foldr (++) ""   xs
```

Ещё одна популярная функция высшего порядка.

Упражнение на понимание

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a _ = a
```

А что такое `foldr (:) [4,5] xs`?

Упражнение на понимание

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a _ = a
```

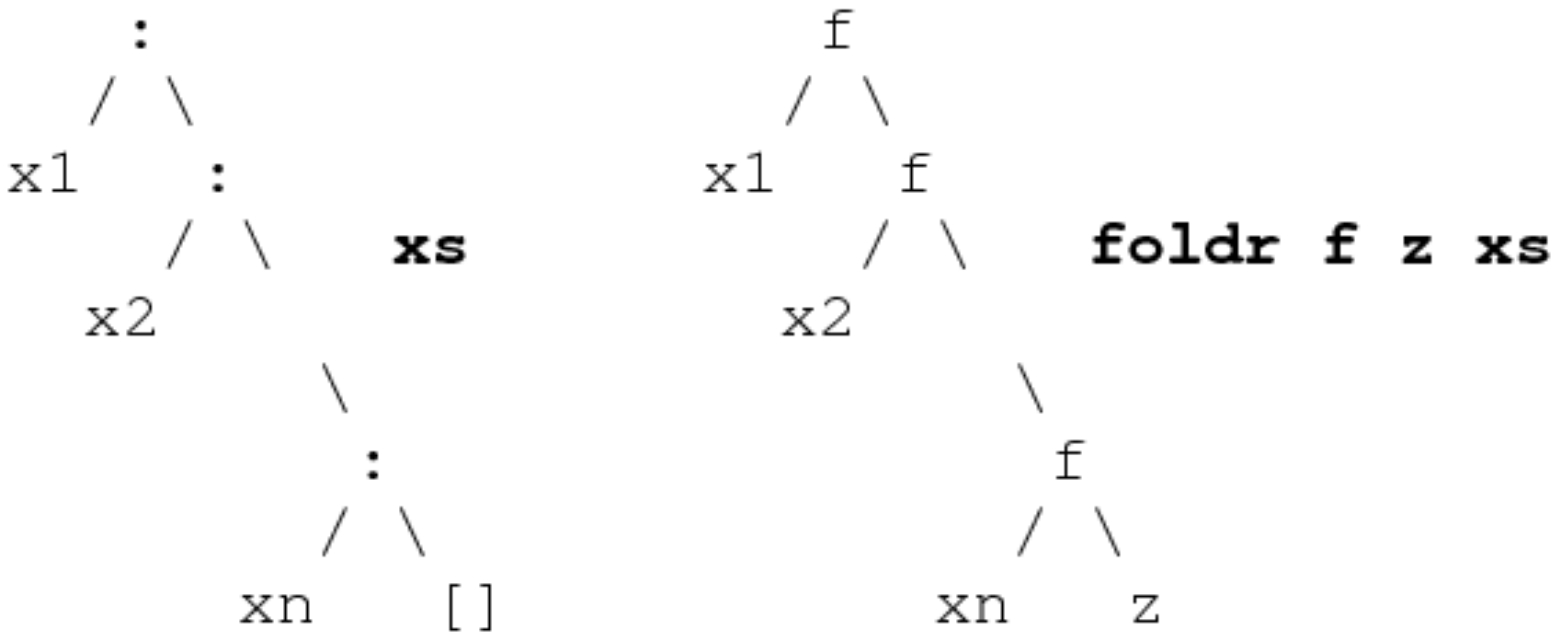
А что такое `foldr (:) [4,5] xs`?

```
foldr (:) [4,5] [1,2,3] =
1:(foldr (:) [4,5] [2,3]) =
1:(2:(foldr (:) [4,5] [3])) =
1:(2:(3:(foldr (:) [4,5] []))) =
1:(2:(3:[4,5])) =
[1,2,3,4,5]
```

Дописывание `[4,5]` в конец списка.

```
(++) a b = foldr (:) b a
```

Картинка



Бамбук растёт вправо, поэтому *правая* свёртка.

Упражнения

Как узнать, все ли элементы равны `True`?

Упражнения

Как узнать, все ли элементы равны `True`?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

Упражнения

Как узнать, все ли элементы равны `True`?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

```
foldr (+) 0 (map (^2) xs)
```

Как выразить `map` через `foldr`?

Упражнения

Как узнать, все ли элементы равны `True`?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

```
foldr (+) 0 (map (^2) xs)
```

Как выразить `map` через `foldr`?

```
map f xs = foldr (\a x -> (f a):x) [] xs
```

Вывод: в теории почти всё есть `foldr`. На практике лучше использовать готовые функции.

- Мы нигде не указывали типы ни аргументов функций, ни возвращаемых значений.
- Свободно использовали функции для разных типов (вроде `map`).
- Если набрать `:t map` в GHCI, увидим её тип:

$$\underbrace{(a -> b)}_{\text{функция}} -> \underbrace{[a]}_{\text{исходный список}} -> \underbrace{[b]}_{\text{результат}} .$$

- Справа от последней `->` — возвращаемое значение, до этого — аргументы.
- Тут `a` и `b` — типовые переменные. На их месте может стоять любой тип.
- Естественным образом получаем, что `map` вообще всё равно, с какими списками работать.
- Haskell автоматически выводит наиболее общие типы для практически всех функций.
- Все проверки типов — *на этапе компиляции*.

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.
- А функции с типом `a -> b` не бывает — она в общем случае не может создать что-то типа `b`.
- Что может делать `a -> [a]`?

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.
- А функции с типом `a -> b` не бывает — она в общем случае не может создать что-то типа `b`.
- Что может делать `a -> [a]`?
- Только создавать список из одинаковых элементов *фиксированной* длины, которая не зависит от аргумента.

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	`dropWhile`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	`dropWhile`
`(a -> Int) -> [a] -> Int`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	`dropWhile`
`(a -> Int) -> [a] -> Int`	`sum (map f xs)`

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	`dropWhile`
`(a -> Int) -> [a] -> Int`	`sum (map f xs)`
`(a -> b -> b) -> b -> [a] -> b`	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
`a -> a`	`id x = x`
`a -> b -> a`	`fst x y = x`
`a -> b -> b`	`snd x y = y`
`(a -> b) -> a -> b`	`apply f x = f x`
`[a] -> a`	`get xs = xs !! c`
`(a -> Bool) -> [a] -> [a]`	`filter`
`(a -> Bool) -> [a] -> [a]`	`dropWhile`
`(a -> Int) -> [a] -> Int`	`sum (map f xs)`
`(a -> b -> b) -> b -> [a] -> b`	`foldr`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y` `foo x y z = x y z`	`(a -> b) -> a -> b`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	`(a -> Int) -> a -> a -> Int`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	`(a -> Int) -> a -> a -> Int`
`foo x y = (x y) + y`	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	`(a -> Int) -> a -> a -> Int`
`foo x y = (x y) + y`	`(Int -> Int) -> Int -> Int`

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	`(a -> Int) -> a -> a -> Int`
`foo x y = (x y) + y`	`(Int -> Int) -> Int -> Int`
`foo x y = (x y):y`	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
`foo x y = x y`	`(a -> b) -> a -> b`
`foo x y z = x y z`	`(a -> b -> c) -> a -> b -> c`
`foo x y z = (x y) + (x z)`	`(a -> Int) -> a -> a -> Int`
`foo x y = (x y) + y`	`(Int -> Int) -> Int -> Int`
`foo x y = (x y):y`	`([a] -> a) -> [a] -> [a]`

Резюме

- Без полиморфизма функции высшего порядка были бы бесполезны.
- Часто по типу полиморфной функции можно догадаться, что она делает.
- Есть специальный поисковик Hoogle, который ищет функции по их типу.
- Hoogle — полезная штука, если вам нужна какая-то «очевидно полезная» функция. Найдётся всё.

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.

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 должны 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.

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 (`ghci -W file.hs`).
- 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
```

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)
```

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.

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.

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

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!

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 сранивать.
- Надо добавить *контекст* — сказать, что списки можно сравнивать только если можно сранивать элементы:

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

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).

Автовывод контекста

```haskell
-- Ord a => a -> a -> a
max' a b = if a > b then a else b
-- Eq, кстати, тоже будет:
-- class Eq a => Ord a where
-- ...

-- (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.

Как отлаживать

- Ваш лучший друг — чтение кода, выписывание типов, тестирование.
- Haskell обычно выдаёт точный символ, в котором произошла ошибка. Там и надо смотреть.
- Начинать лучше с самой верхней ошибки.
- Можно закомментировать кусок кода, а у оставшегося явно написать нужный тип:

```
map' :: (a -> b) -> [a] -> [b]
map' f a b = f a  -- Ошибка компиляции
```

- Проверяйте код при помощи hlint.
- Он заодно проверяет соответствие некоторому стилю.
- Не все рекомендации hlint жизненно необходимо выполнять, так как единого стиля нет.

Trace

```haskell
import Debug.Trace
-- traceShow :: Show a => a -> b -> b
sum' xs = sum'' 0 xs
    where
        sum'' a [] = traceShow (a, []::[Int]) a
        sum'' a (x:xs) = traceShow (a, x:xs)
                           (sum'' (a + x) xs)
```

- Использовать только для отладки
- У `traceShow` два параметра — один она печатает, второй возвращает
- `traceShow` нарушает чистоту

HSpec

Можно и нужно писать тесты для функций:

```
import Test.Hspec   -- cabal install hspec
                    -- apt-get install haskell-hspec
main = hspec $ do
  describe "length function" $ do
    it "works on empty list" $ do
      (length []) `shouldBe` 0
    it "works on single-item list" $ do
      (length [10]) `shouldBe` 1
      (length ["foo"]) `shouldBe` 1
```

Особенности функционального стиля

Необязательно, но обычно:

- Алгоритм разбивается не на «шаги», а на мелкие функции с чёткими контрактами
- Вместо циклов — рекурсия или встроенные функции (коих много)
- Очень мощная система типов:
  - Функции обощаются получше
  - Алгебраические типы данных, позволяют писать *везде определённые* функции
  - Вместо `if` — *pattern matching*
- Очень компактный код
- Нет никаких изменяемых переменных, захотели изменить объект — создали копию (*иммутабельность*)
- Частичное применение функций (или каррирование, из одного другое в каком-то смысле следует)

Мои наблюдения-1

- На функциональных языках обычно очень компактные программы и много синтаксического сахара.
- Обычно функциональные языки (в том числе Haskell) умеют очень сильно расширять свой синтаксис до неузнаваемости.
- Иногда всё это превращается в сахарную вату.
- Элементы ФП в разной степени поддерживаются в разных языках, в том числе в «императивных»: C++, Python, Java.
- Есть модные смеси императивного и функционального программирования вроде Scala или OCaml.
- Некоторые функциональные языки используются в реальной жизни: Erlang.
- Чисто функциональные программы может быть сложнее отлаживать, так как нет «состояния программы».

Мои наблюдения-2

- Многие идеи из ФП полезны и в повседневной жизни:
  - Неизменяемое состояние.
  - Функции высшего порядка (где есть поддержка в языке).
  - Чистые функции без побочных эффектов.
- Если язык поддерживают хотя бы `map`, лямбда-функции или list comprehension, на нём уже намного приятнее писать.
- Функциональные элементы могут сильно упростить код императивной программы без потери скорости.
- Дополнительных проблем эти элементы не вносят.
- Надо быть аккуратными и не мешать их с изменяемым состоянием.