Бесплатная электронная книга

УЧУСЬ

# Haskell Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#haskell

. . . 1

1: Haskell . . . 2

. . . 2

: . . . 2

. . . 2

Examples . . . 3

, ! . . . 3

: . . . 4

. . . 5

1 . . . 6

2 . . . 6

, . . . 6

. . . 8

REPL . . . 8

GHC () . . . 8

. . . 10

. . . 10

100 . . . 10

. . . 10

. . . 11

. . . 11

. . . 11

. . . 11

. . . 11

2: Attoparsec . . . 13

. . . 13

. . . 13

Examples . . . 13

. . . 13

Bitmap - . . . 14

3: Data.Aeson - JSON . . . 16

Examples 16

16

Data.Aeson.Value 17

17

4: data.text 18

18

Examples 18

18

18

19

19

, 20

20

5: GHCJS 22

22

Examples 22

«Hello World!» Node.js 22

6: GTK3 23

23

23

Examples 23

, Gtk 23

7: IO 25

Examples 25

25

25

25

26

27

28

IO `main` 28

IO 29

- 30
- 30
Lazy IO 31
IO do 32
«a» «IO a» 32
stdout 33
putChar :: Char -> IO () - char stdout 33
putStr :: String -> IO () - String stdout 33
putStrLn :: String -> IO () - String stdout 33
print :: Show a => a -> IO () - a Show stdout 33
`stdin` 34
getChar :: IO Char - Char stdin 34
getLine :: IO String - String stdin , 34
read :: Read a => String -> a - String 34
8: Monad Transformers 36
Examples 36
36
37
: ! 38
38
9: Monoid 40
Examples 40
Monoid 40
40
40
Monoid for () 41
10: Profunctor 42
42
42
42
Examples 43

(->) Profunctor 43

**11: Reader / ReaderT 44**

44

Examples 44

44

**12: XML 46**

46

Examples 46

`xml` 46

**13: zipWithM 47**

47

47

Examples 47

47

**14: 48**

Examples 48

48

48

Quicksort 49

49

49

49

**15: HTML 51**

Examples 51

div 51

51

**16: 53**

53

53

53

Examples 53

53

54

**54**

**54**

**zip-** **54**

**55**

**17:** **57**

57

57

**57**

Examples 59

59

` (Fractional Int) ...` 59

60

**18:** **61**

Examples 61

Postgres 61

61

61

**19:** **62**

Examples 62

62

63

foldFree iterM? 63

64

**20:** **67**

67

67

Examples 67

67

**67**

Either 67
68
68
21: Google 69
69
Examples 69
, .proto 69
22: 71
Examples 71
71
71
71
72
(==>) 72
73
23: - 74
Examples 74
74
75
24: 77
77
Examples 77
Data.Vector 77
78
(`map`) (` fold`) 78
78
25: 79
Examples 79
Bang 79
79
79

**80**

80

82

**26: 83**

83

83

Examples 83

83

**84**

**84**

postIncrement 84

85

Traversable 85

Traversable 85

**27: 87**

Examples 87

87

- 87

**28: 89**

89

89

Examples 89

89

, 89

**29: 91**

Examples 91

91

« » 91

- () 92

**30: 93**

93

93

Examples....93

C Haskell....93

Haskell C....94

**31:** ....**96**

....96

Examples....97

....97

....97

**32: GHCi**....**99**

....99

Examples....99

GHCi....99

GHCi....99

GHCi....99

....99

GHCi....100

....100

GHCi....100

....101

**33: - Data.Map**....**102**

Examples....102

....102

....102

....102

....103

....103

....103

....103

**34: (, , ...)**....**105**

....105

Examples....105

....

105

. . . . . . . . . . 106

. . . . . . . . . . 106

. . . . . . . . . . 106

() . . . . . . . . . . 107

(currying) . . . . . . . . . . 107

- . . . . . . . . . . 108

. . . . . . . . . . 108

**35: . . . . . . . . . . 109**

. . . . . . . . . . 109

Examples . . . . . . . . . . 109

hslogger . . . . . . . . . . 109

**36: . . . . . . . . . . 110**

. . . . . . . . . . 110

. . . . . . . . . . 110

Examples . . . . . . . . . . 110

. . . . . . . . . . 110

. . . . . . . . . . 111

. . . . . . . . . . 111

. . . . . . . . . . 111

. . . . . . . . . . 112

. . . . . . . . . . 112

**37: . . . . . . . . . . 114**

. . . . . . . . . . 114

Examples . . . . . . . . . . 114

, . . . . . . . . . . 114

IO monad . . . . . . . . . . 116

. . . . . . . . . . 117

. . . . . . . . . . 118

. . . . . . . . . . 118

- . . . . . . . . . . 119

. . . . . . . . . . 120

38: ........ 121
Examples ........ 121
........ 121
39: ........ 123
Examples ........ 123
~~ Identity ........ 123
Free Identity ~~ (Nat,) ~~ Writer Nat ........ 123
~~ MaybeT (Writer Nat) ........ 124
Free (Writer w) ~~ Writer [w] ........ 124
Free (Const c) ~~ c ........ 125
Free (Reader x) ~~ Reader ( x) ........ 125
40: GHC ........ 126
........ 126
Examples ........ 126
MultiParamTypeClasses ........ 126
FlexibleInstances ........ 126
OverloadedStrings ........ 127
TupleSections ........ 127
N- ........ 128
........ 128
UnicodeSyntax ........ 128
BinaryLiterals ........ 129
ExistentialQuantification ........ 129
LambdaCase ........ 131
RankNTypes ........ 131
OverloadedLists ........ 132
FunctionalDependencies ........ 133
GADTs ........ 133
ScopedTypeVariables ........ 134
PatternSynonyms ........ 135
RecordWildCards ........ 136

**41: cofree comonads ..... 137**

Examples ..... 137

Cofree Empty ~~ Empty ..... 137

Cofree (Const c) ~~ Writer c ..... 137

Cofree Identity ~~ Stream ..... 137

Cofree ~~ NonEmpty ..... 138

Cofree (Writer w) ~~ WriterT w Stream ..... 138

Cofree ( e) ~~ NonEmptyT (Writer e) ..... 138

Cofree (Reader x) ~~ Moore x ..... 139

**42: ..... 140**

..... 140

..... 140

**? ..... 140**

**..... 140**

..... 140

..... 141

..... 141

Examples ..... 142

..... 142

..... 142

**..... 142**

**..... 142**

..... 143

**& ..... 143**

**..... 144**

Haskell ..... 144

..... 145

..... 146

..... 146

..... 146

makeFields ..... 147

43: ........ 150

........ 150

........ 150

........ 150

Examples ........ 151

........ 151

........ 151

........ 152

........ 152

44: Infix ........ 153

........ 153

Examples ........ 153

........ 153

........ 153

........ 153

........ 154

........ 154

........ 155

........ 155

45: ........ 157

Examples ........ 157

........ 157

........ 158

46: ........ 159

........ 159

Examples ........ 159

........ 159

Traversable ........ 160

........ 161

Traversable ........ 161

........ 162

........

163

.....164

47: .....165

.....165

.....165

Examples.....166

.....166

RPAR.....166

rseq.....167

48: .....169

.....169

.....169

.....169

.....169

.....169

Examples.....170

.....170

.....170

.....170

.....170

.....170

.....171

.....171

.....171

.....171

.....172

49: (GHC).....173

Examples.....173

.....173

50: RankNTypes.....174

.....174

.....

174

Examples....174

RankNTypes....174

51: IO....175

Examples....175

IO....175

52: -....176

Examples....176

....176

....177

....177

EventNetworks....178

53: ....179

....179

Examples....179

....179

....180

....180

, , ....180

....181

....181

54: ....182

....182

....182

Examples....182

....182

....182

....183

55: ....184

Examples....184

....184

....184

184
, 184
185
185
186
186
56: 187
Examples 187
187
187
188
57: 190
190
190
Examples 190
190
191
- 1 191
- 2 192
58: 193
Examples 193
193
194
194
RecordWildCards 195
195
196
NamedFieldPuns 196
RecordWildcards 196
196
59: 197

197

.......... 197

Examples .......... 197

.......... 197

.......... 197

Foldable .......... 198

.......... 199

.......... 200

.......... 201

.......... 201

, .......... 201

**60: .......... 203**

.......... 203

Examples .......... 203

`forkIO` .......... 203

MVARD .......... 203

.......... 204

atomically :: STM a -> IO a .......... 205

readTVar :: TVar a -> STM a .......... 205

writeTVar :: TVar a -> a -> STM () .......... 205

**61: .......... 206**

Examples .......... 206

.......... 206

.......... 206

.......... 206

.......... 207

.......... 207

.......... 207

.......... 207

**62: .......... 209**

.......... 209

Examples .......... 210

..........

210
.....210
.....210
63: .....212
.....212
.....212
Examples.....213
.....213
.....213
.....213
.....214
.....215
.....215
.....216
foldl.....217
foldr.....217
`map`.....218
`filter`.....218
.....219
64: .....220
Examples.....220
.....220
.....220
.....221
.....221
.....221
.....222
.....222
65: .....223
Examples.....223
.....223
.....223
.....

**223**

**223**

Stackage LTS (resolver) 224

**224**

224

225

225

225

**66: 227**

Examples 227

227

**67: 229**

Examples 229

229

**229**

**229**

230

Haskell 231

231

231

232

232

233

233

233

234

234

235

235

235

235

Haskell .....236

68: .....237

Examples.....237

SmallCheck, QuickCheck HUnit.....237

69: .....239

.....239

Examples.....239

.....239

.....240

.....240

.....241

70: .....243

.....243

Examples.....243

.....243

.....243

.....244

.....244

71: .....247

Examples.....247

.....247

.....247

.....247

.....248

.....248

.....249

.....249

.....249

.....250

.....250

.....

250

**72: ........ 252**

........ 252

........ 252

Examples ........ 252

Functor ........ 252

: Ord ........ 253

........ 254

**........ 254**

**........ 254**

**........ 255**

**........ 255**

Ord ........ 255

**........ 255**

**........ 255**

**........ 255**

Monoid ........ 256

**........ 256**

**........ 256**

Num ........ 256

........ 257

**73: ........ 259**

Examples ........ 259

: ........ 259

**74: ........ 260**

........ 260

Examples ........ 260

........ 260

........ 261

........ 261

runEffect ........ 261

.......... 261
Proxy .......... 262
.......... 262
75: .......... 265
.......... 265
.......... 265
.......... 265
.......... 265
Examples .......... 265
Functor .......... 265
.......... 265
.......... 266
.......... 267
.......... 268
.......... 268
.......... 268
.......... 269
.......... 269
Functor .......... 269
.......... 270
functor .......... 270
.......... 270
.......... 271
.......... 272
76: .......... 273
.......... 273
Examples .......... 273
.......... 273
- .......... 274
.......... 275
77: .......... 276

.......276

Examples.......276

.......276

.......277

.......277

.......**278**

**78: Haskell & QuasiQuotes.......279**

.......279

**Haskell?.......279**

**? (, ?).......279**

**Haskell .......279**

Examples.......280

Q.......280

n-arity.......281

Haskell Quasiquotes.......283

.......**283**

**(: QuasiQuotation).......283**

.......**284**

**QuasiQuotes.......284**

.......**284**

.......**286**

# Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: haskell-language

It is an unofficial and free Haskell Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Haskell Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# глава 1: Начало работы с языком Haskell

## замечания

Haskell - это усовершенствованный чисто функциональный язык программирования.

# Особенности:

- **Статически типизировано:** каждое выражение в Haskell имеет тип, который определяется во время компиляции. Проверка статического типа - это процесс проверки безопасности типа программы на основе анализа текста программы (исходного кода). Если программа проходит проверку статического типа, то гарантируется, что программа удовлетворит некоторый набор свойств безопасности типа для всех возможных входов.
- **Чисто функциональный** : каждая функция в Haskell является функцией в математическом смысле. Нет инструкций или инструкций, только выражений, которые не могут мутировать переменные (локальные или глобальные), а не состояния доступа, такие как временные или случайные числа.
- **Параллельно:** его флагманский компилятор GHC поставляется с высокопроизводительным параллельным сборщиком мусора и небольшой библиотекой параллелизма, содержащим множество полезных примитивов параллелизма и абстракций.
- **Ленькая оценка:** функции не оценивают свои аргументы. Задерживает оценку выражения до тех пор, пока не потребуется его значение.
- **Универсальный:** Haskell построен для использования во всех контекстах и средах.
- **Пакеты:** вклад с открытым исходным кодом в Haskell очень активен с широким спектром пакетов, доступных на серверах общих пакетов.

Последний стандарт Haskell - Haskell 2010. По состоянию на май 2016 года группа работает над следующей версией Haskell 2020.

Официальная документация Haskell также является исчерпывающим и полезным ресурсом. Отличное место для поиска книг, курсов, учебных пособий, руководств, руководств и т. Д.

## Версии

| Версия | Дата выхода |
|---|---|
| Haskell 2010 | 2012-07-10 |
| Haskell 98 | 2002-12-01 |

# Examples

## Привет, мир!

Базовый «Привет, мир!» программа в Haskell может быть выражена кратко только в одной или двух строках:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Первая строка представляет собой необязательную аннотацию типа, указывающую, что `main` является значением типа `IO ()` , представляющим собой операцию ввода-вывода, которая «вычисляет» значение типа `()` (читает «единица», пустой кортеж не передает никакой информации) помимо выполнения некоторых побочных эффектов для внешнего мира (здесь, печатая строку на терминале). Аннотации этого типа обычно опущены для `main` потому что это его *единственный* возможный тип.

Поместите это в файл `helloworld.hs` и скомпилируйте его с помощью компилятора Haskell, такого как GHC:

```
ghc helloworld.hs
```

Выполнение скомпилированного файла приведет к выводу `"Hello, World!"` печатается на экране:

```
./helloworld
Hello, World!
```

В качестве альтернативы `runhaskell` или `runghc` позволяют запускать программу в интерпретируемом режиме без необходимости ее компиляции:

```
runhaskell helloworld.hs
```

Интерактивный REPL также может использоваться вместо компиляции. Он поставляется с большинством сред Haskell, таких как `ghci` который поставляется вместе с компилятором GHC:

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

Кроме того, загрузите скрипты в ghci из файла с помощью `load` (или `:l` ):

```
ghci> :load helloworld
```

`:reload` (или `:r` ) перезагружает все в ghci:

```
Prelude> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )

<some time later after some edits>

*Main> :r
Ok, modules loaded: Main.
```

---

## Объяснение:

Эта первая строка является сигнатурой типа, объявляющей тип `main` :

```
main :: IO ()
```

Значения типа `IO ()` описывают действия, которые могут взаимодействовать с внешним миром.

Поскольку Haskell имеет полноценную систему типа Hindley-Milner, которая позволяет автоматически вводить тип, титровые подписи технически необязательны: если вы просто опустите `main :: IO ()` , компилятор сможет вывести тип самостоятельно анализируя *определение* `main` . Тем не менее, очень часто считается, что плохой стиль не записывать сигнатуры типов для определений верхнего уровня. Причины включают:

- Типовые подписи в Haskell - очень полезная часть документации, потому что система типов настолько выразительна, что вы часто можете видеть, какая функция удобна для просто, глядя на ее тип. Эту «документацию» можно легко получить с помощью таких инструментов, как GHCi. И в отличие от обычной документации, контролер типа компилятора убедится, что он действительно соответствует определению функции!

- Подписи типов *сохраняют ошибки локально* . Если вы допустили ошибку в определении без предоставления его сигнатуры типа, компилятор может не сразу сообщить об ошибке, а вместо этого просто вывести для него бессмысленный тип, с которым он фактически выглядит. При *использовании* этого значения вы можете получить критическое сообщение об ошибке. С подписью компилятор очень хорошо разбирается в ошибках, где они происходят.

Эта вторая строка выполняет фактическую работу:

```
main = putStrLn "Hello, World!"
```

Если вы исходите из императивного языка, может быть полезно отметить, что это определение также можно записать в виде:

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

Или, что то же самое, (Haskell имеет разводку на основе макета, но *будьте осторожны, смешивая вкладки и пробелы непоследовательно,* что будет путать этот механизм):

```
main = do
    putStrLn "Hello, World!"
    return ()
```

Каждая строка в блоке `do` представляет собой некоторое монадическое (здесь, I / O) *вычисление* , так что весь блок `do` представляет общее действие, состоящее из этих подэтапов, путем комбинирования их способом, определенным для данной монады (для ввода / вывода это означает просто выполнение их один за другим).

Синтаксис `do` сам по себе является синтаксическим сахаром для монад, например `IO` , и `return` - это действие no-op, создающее его аргумент без каких-либо побочных эффектов или дополнительных вычислений, которые могут быть частью определенного определения монады.

Вышеупомянутое то же самое, что и определение `main = putStrLn "Hello, World!"` , потому что значение `putStrLn "Hello, World!"` уже имеет тип `IO ()` . Рассматривается как «заявление», `putStrLn "Hello, World!"` можно рассматривать как полную программу, и вы просто определяете `main` ссылку на эту программу.

Вы можете посмотреть подпись `putStrLn` онлайн :

```
putStrLn :: String -> IO ()
-- thus,
putStrLn (v :: String) :: IO ()
```

`putStrLn` - это функция, которая принимает строку в качестве аргумента и выводит действие ввода-вывода (то есть значение, представляющее программу, которую может выполнять среда выполнения). Время выполнения всегда выполняет действие с именем `main` , поэтому нам просто нужно определить его равным `putStrLn "Hello, World!"` ,

## Факториал

Факториальная функция - Haskell «Hello World!» (и для функционального программирования в целом) в том смысле, что он лаконично демонстрирует основные принципы языка.

## Вариант 1

```
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

Демо-версия

- `Integral` - это класс целочисленных типов чисел. Примеры включают `Int` и `Integer` .
- `(Integral a) =>` помещает ограничение на тип `a` находящийся в указанном классе
- `fac :: a -> a` говорит, что `fac` - это функция, которая принимает `a` и возвращает `a`
- `product` - это функция, которая накапливает все числа в списке, умножая их вместе.
- `[1..n]` - это специальное обозначение, которое desugars для `enumFromTo 1 n` и является диапазоном чисел `1 ≤ x ≤ n` .

## Вариант 2

```
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

Демо-версия

Эта вариация использует сопоставление образцов, чтобы разбить определение функции на отдельные случаи. Первое определение вызывается, если аргумент равен `0` (иногда это называется условием останова), а второе определение - иначе (порядок определений значителен). Это также иллюстрирует рекурсию, поскольку `fac` ссылается на себя.

---

Стоит отметить, что из-за правил перезаписи обе версии `fac` будут скомпилированы с идентичным машинным кодом при использовании GHC с активированными оптимизациями. Таким образом, с точки зрения эффективности эти два будут эквивалентны.

### Фибоначчи, используя ленивую оценку

Lazy оценка означает, что Haskell будет оценивать только элементы списка, значения которых необходимы.

Основным рекурсивным определением является:

```
f (0)   <-   0
f (1)   <-   1
f (n)   <-   f (n-1) + f (n-2)
```

Если оценивать напрямую, это будет *очень* медленно. Но, представьте, у нас есть список, который записывает все результаты,

```
fibs !! n  <-  f (n)
```

затем

```
                    ┌────┐    ┌────┐    ┌────┐
                    │ f(0) │    │ f(1) │    │ f(2) │
fibs  ->  0 : 1 :   │  +   │ :  │  +   │ :  │  +   │ :  .....
                    │ f(1) │    │ f(2) │    │ f(3) │
                    └────┘    └────┘    └────┘

                    ┌──────────────────────────────────────┐
                    │ f(0)   :   f(1)   :   f(2)   :  ..... │
                    └──────────────────────────────────────┘
      ->  0 : 1 :                 +
                    ┌──────────────────────────────────────┐
                    │ f(1)   :   f(2)   :   f(3)   :  ..... │
                    └──────────────────────────────────────┘
```

Это кодируется как:

```
fibn n = fibs !! n
    where
    fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

Или даже как

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`zipWith` создает список, применяя данную двоичную функцию к соответствующим элементам из двух перечисленных ему списков, поэтому `zipWith (+) [x1, x2, ...] [y1, y2, ...]` равно `[x1 + y1, x2 + y2, ...]` .

Другой способ написания `fibs` это с `scanl` функции :

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`scanl` создает список частичных результатов, которые `foldl` бы `foldl` , работая слева направо по списку ввода. То есть `scanl f z0 [x1, x2, ...]` равно `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...`

Благодаря ленивой оценке обе функции определяют бесконечные списки, не вычисляя их полностью. То есть, мы можем написать функцию `fib` , извлекая n-й элемент неограниченной последовательности Фибоначчи:

```
GHCi> let fib n = fibs !! n  -- (!!) being the list subscript operator
-- or in point-free style:
```

```
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

### Начиная

## Онлайн REPL

Самый простой способ начать писать Haskell - это, вероятно, переход на веб-сайт Haskell или Try Haskell и использование онлайн-REPL (read-eval-print-loop) на главной странице. Онлайновый REPL поддерживает большинство базовых функций и даже некоторых IO. Существует также базовый учебник доступен , который может быть запущен, набрав команду `help` . Идеальный инструмент, чтобы начать изучать основы Haskell и попробовать некоторые вещи.

## *GHC (я)*

Для программистов, которые готовы заняться немного больше, есть *GHCi* , интерактивная среда, которая поставляется вместе с *компилятором Glorious / Glasgow Haskell* . *GHC* можно установить отдельно, но это только компилятор. Чтобы иметь возможность устанавливать новые библиотеки, необходимо также установить такие инструменты, как *Cabal* и *Stack* . Если вы используете Unix-подобную операционную систему, проще всего установить *Stack,* используя:

```
curl -sSL https://get.haskellstack.org/ | sh
```

Это устанавливает GHC, изолированный от остальной части вашей системы, поэтому его легко удалить. Однако всем командам должен предшествовать `stack` . Еще один простой подход - установить платформу Haskell . Платформа существует в двух вариантах:

1. **Минимальное** распределение содержит только *GHC* (для компиляции) и *Cabal / Stack* (для установки и сборки пакетов)
2. **Полный** дистрибутив дополнительно содержит инструменты для разработки проектов, профилирования и анализа охвата. Также включен дополнительный набор широко используемых пакетов.

Эти платформы можно установить, загрузив установщик и следуя инструкциям или используя диспетчер пакетов вашего дистрибутива (обратите внимание, что эта версия не гарантируется как актуальная):

- Ubuntu, Debian, Mint:

```
sudo apt-get install haskell-platform
```

- Fedora:

```
sudo dnf install haskell-platform
```

- Красная шляпа:

```
sudo yum install haskell-platform
```

- Arch Linux:

```
sudo pacman -S ghc cabal-install haskell-haddock-api \
               haskell-haddock-library happy alex
```

- Gentoo:

```
sudo layman -a haskell
sudo emerge haskell-platform
```

- OSX с Homebrew:

```
brew cask install haskell-platform
```

- OSX с MacPorts:

```
sudo port install haskell-platform
```

После установки должно быть возможно запустить *GHCi* , вызвав команду `ghci` любом месте терминала. Если установка прошла успешно, консоль должна выглядеть примерно так:

```
me@notebook:~$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

возможно, с дополнительной информацией о том, какие библиотеки были загружены перед `Prelude>` . Теперь консоль стала Haskell REPL, и вы можете выполнить код Haskell, как в онлайн-REPL. Чтобы выйти из этой интерактивной среды, можно ввести `:q` или `:quit` . Для получения дополнительной информации о том , что команды доступны в *GHCi,* типа `:?` как показано на начальном экране.

Поскольку писать одни и те же вещи снова и снова на одной строке не всегда так практически, может быть хорошей идеей написать код Haskell в файлах. Обычно эти файлы имеют `.hs` для расширения и могут быть загружены в REPL с помощью `:l` или `:load` .

Как упоминалось ранее, *GHCi* является частью *GHC* , который на самом деле является компилятором. Этот компилятор может быть использован для преобразования файла `.hs` с кодом Haskell в запущенную программу. Поскольку файл `.hs` может содержать множество функций, в файле должна быть определена `main` функция. Это будет отправной точкой для программы. Файл `test.hs` можно скомпилировать с помощью команды

```
ghc test.hs
```

это создаст объектные файлы и исполняемый файл, если ошибок не было, и `main` функция была определена правильно.

# Дополнительные инструменты

1. Это уже упоминалось ранее как диспетчер пакетов, но *стек* может быть полезным инструментом для разработки Haskell совершенно по-разному. После установки он способен

   - установка (несколько версий) *GHC*
   - создание проекта и строительные леса
   - управление зависимостями
   - проекты по строительству и тестированию
   - бенчмаркинг

2. IHaskell - это ядро haskell для IPython и позволяет комбинировать (runnable) код с уценкой и математической нотацией.

### Штрихи

Несколько *наиболее важных* вариантов:

### Ниже 100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

### неограниченный

Сито Эратосфена, использующее пакет данных :

```
import qualified Data.List.Ordered
```

```
ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))

_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

## традиционный

(субоптимальное пробное деление)

```
ps = sieve [2..]
     where
     sieve (x:xs) = [x] ++ sieve [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

## Оптимальное пробное деление

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

## переходный

От пробного деления до сита Эратосфена:

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

## Самый короткий код

```
nubBy (((>1).).gcd) [2..]          -- i.e., nubBy (\a b -> gcd a b > 1) [2..]
```

`nubBy` также из `Data.List` , например `(\\)` .

## Объявление значений

Мы можем объявить серию выражений в REPL следующим образом:

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> result
150
```

Чтобы объявить те же значения в файле, мы пишем следующее:

```
-- demo.hs

module Demo where
-- We declare the name of our module so
-- it can be imported by name in a project.

x = 5

y = 2 * 5 + x

result = y * 10
```

Имена модулей капитализируются, в отличие от имен переменных.

Прочитайте Начало работы с языком Haskell онлайн:
https://riptutorial.com/ru/haskell/topic/251/начало-работы-с-языком-haskell

# глава 2: Attoparsec

## Вступление

Attoparsec - это библиотека комбинаторного анализатора, которая «нацелена, в частности, на эффективную работу с сетевыми протоколами и сложными форматами текстовых / двоичных файлов».

Attoparsec предлагает не только скорость и эффективность, но и обратную трассировку и инкрементный вход.

Его API тесно связан с API-интерфейсом другой библиотеки парсеров-парсеров Parsec.

Существуют подмодули для совместимости с `ByteString` , `Text` и `Char8` . Рекомендуется использовать расширение языка `OverloadedStrings` .

## параметры

| Тип | подробность |
| --- | --- |
| `Parser ia` | Основной тип представления синтаксического анализатора. `i` - тип строки, например `ByteString` . |
| `IResult ir` | Результат анализа, с `Fail i [String] String` , `Partial (i -> IResult ir)` и `Done ir` как конструкторы. |

## Examples

### Комбинаторы

Разбор ввода лучше всего достигается за счет более крупных функций парсера, состоящих из меньших одноцелевых.

Предположим, мы хотели разобрать следующий текст, который представляет собой рабочее время:

> Понедельник: 0800 1600.

Мы могли бы разделить их на два «токена»: название дня - «понедельник» - и часть времени «0800» - «1600».

Чтобы проанализировать имя дня, мы могли бы написать следующее:

```
data Day = Day String

day :: Parser Day
day = do
  name <- takeWhile1 (/= ':')
  skipMany1 (char ':')
  skipSpace
  return $ Day name
```

Чтобы проанализировать временную часть, мы могли бы написать:

```
data TimePortion = TimePortion String String

time = do
    start <- takeWhile1 isDigit
    skipSpace
    end <- takeWhile1 isDigit
    return $ TimePortion start end
```

Теперь у нас есть два парсера для наших отдельных частей текста, мы можем объединить их в «большом» синтаксическом анализаторе для чтения рабочего дня всего дня:

```
data WorkPeriod = WorkPeriod Day TimePortion

work = do
    d <- day
    t <- time
    return $ WorkPeriod d t
```

а затем запустите синтаксический анализатор:

```
parseOnly work "Monday: 0800 1600"
```

## Bitmap - Разбор двоичных данных

Attoparsec делает синтаксический анализ двоичных данных тривиальным. Предполагая эти определения:

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import           Data.ByteString            (ByteString, readFile)
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                   hiding (readFile, take)

-- The DIB section from a bitmap header
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- The entire bitmap header
data Header = Header DIB Int Reserved Reserved Int
```

```
            deriving (Show)
```

Мы можем легко разобрать заголовок из растрового файла. Здесь у нас есть 4 функции парсера, которые представляют секцию заголовка из файла растрового изображения:

Во-первых, раздел DIB можно прочитать, взяв первые 2 байта

```
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

Аналогично, размер растрового изображения, зарезервированные разделы и смещение пикселей можно легко прочитать:

```
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: Parser Int
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

которые затем могут быть объединены в большую функцию синтаксического анализа для всего заголовка:

```
bitmapHeader :: Parser Header
bitmapHeader = do
    dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

Прочитайте Attoparsec онлайн: https://riptutorial.com/ru/haskell/topic/9681/attoparsec

# глава 3: Data.Aeson - JSON в Хаскелле

## Examples

### Интеллектуальное кодирование и декодирование с использованием дженериков

Самый простой и быстрый способ кодирования типа данных Haskell для JSON с помощью Aeson использует дженерики.

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

Сначала создадим тип данных Person:

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

Чтобы использовать функцию `encode` и `decode` из пакета `Data.Aeson` , нам нужно сделать `Person` экземпляром `ToJSON` и `FromJSON` . Поскольку мы получаем `Generic` для `Person` , мы можем создавать пустые экземпляры для этих классов. Определения по умолчанию для методов определяются в терминах методов, предоставляемых классом типа `Generic` .

```
instance ToJSON Person
instance FromJSON Person
```

Готово! Чтобы улучшить скорость кодирования, мы можем немного изменить экземпляр `ToJSON` :

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

Теперь мы можем использовать функцию `encode` для преобразования `Person` в (ленивый) Bytestring:

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

И для декодирования мы можем просто использовать `decode` :

```
> encodeNewPerson "Hans" "Wurst" 30
"{\"lastName\":\"Wurst\",\"age\":30,\"firstName\":\"Hans\"}"


> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

## Быстрый способ создания Data.Aeson.Value

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Data.Aeson

main :: IO ()
main = do
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer)
] :: Value
  print . encode $ example
```

## Дополнительные поля

Иногда мы хотим, чтобы некоторые поля в строке JSON были необязательными. Например,

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

Это может быть достигнуто

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

Прочитайте Data.Aeson - JSON в Хаскелле онлайн:
https://riptutorial.com/ru/haskell/topic/4525/data-aeson---json-в-хаскелле

# глава 4: data.text

## замечания

`Text` является более эффективной альтернативой стандартным `String` типам Haskell. `String` определяется как связанный список символов в стандартной прелюдии в отчете Haskell :

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

`Text` представлен как упакованный массив символов Юникода. Это похоже на то, как большинство других языков высокого уровня представляют строки, и дает гораздо лучшую эффективность времени и пространства, чем версия списка.

`Text` должен быть предпочтительнее для `String` для всех видов использования. Заметное исключение зависит от библиотеки, которая имеет `String` API, но даже в этом случае может быть полезно использовать `Text` внутри и преобразовать в `String` непосредственно перед взаимодействием с библиотекой.

Во всех примерах в этом разделе используется расширение языка `OverloadedStrings` .

# Examples

### Текстовые литералы

Расширение языка `OverloadedStrings` позволяет использовать обычные строковые литералы для обозначения значений `Text` .

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

### Удаление пробелов

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "\n\r\t   leading and trailing whitespace   \t\r\n"
```

`strip` удаляет пробелы с начала и конца значения `Text` .

```
ghci> T.strip myText
"leading and trailing whitespace"
```

`stripStart` удаляет пробелы только с самого начала.

```
ghci> T.stripStart myText
"leading and trailing whitespace   \t\r\n"
```

`stripEnd` удаляет пробелы только с конца.

```
ghci> T.stripEnd myText
"\n\r\t   leading and trailing whitespace"
```

`filter` можно использовать для удаления пробелов или других символов с середины.

```
ghci> T.filter /=' ' "spaces in the middle of a text string"
"spacesinthemiddleofatextstring"
```

## Разделение текстовых значений

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "mississippi"
```

`splitOn` разбивает `Text` на список `Texts` на вхождения подстроки.

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

`splitOn` является инверсным для `intercalate` .

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

`split` разбивает `Text` значение на куски на символы, которые удовлетворяют булевому предикату.

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

## Текст кодирования и декодирования

Функции кодирования и декодирования для различных кодировок Unicode можно найти в модуле `Data.Text.Encoding` .

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

Обратите внимание, что `decodeUtf8` будет генерировать исключение из недопустимого ввода. Если вы хотите обрабатывать недействительный UTF-8 самостоятельно, используйте `decodeUtf8With` .

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

## Проверка того, является ли текст подстрокой другого текста

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

`isInfixOf :: Text -> Text -> Bool` проверяет, содержится ли `Text` в любом другом `Text` .

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

`isPrefixOf :: Text -> Text -> Bool` проверяет, появляется ли `Text` в начале другого `Text` .

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

`isSuffixOf :: Text -> Text -> Bool` проверяет, появляется ли `Text` в конце другого `Text` .

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

## Текст индексирования

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "mississippi"
```

Символы в определенных индексах могут быть возвращены функцией `index` .

```
ghci> T.index myText 2
's'
```

Функция `findIndex` выполняет функцию типа `(Char -> Bool)` и Text и возвращает индекс первого вхождения данной строки или Nothing, если она не встречается.

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

Функция `count` возвращает количество раз, когда `Text` запроса встречается в другом `Text` .

```
ghci> count ("miss"::T.Text) myText
1
```

Прочитайте data.text онлайн: https://riptutorial.com/ru/haskell/topic/3406/data-text

# глава 5: GHCJS

## Вступление

GHCJS является компилятором Haskell для JavaScript, который использует API GHC.

## Examples

### Запуск «Hello World!» с Node.js

`ghcjs` может быть вызван с теми же аргументами командной строки, что и `ghc` . Сгенерированные программы могут запускаться непосредственно из оболочки с помощью Node.js и SpiderMonkey jsshell . например:

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
Hello world!
```

Прочитайте GHCJS онлайн: https://riptutorial.com/ru/haskell/topic/9260/ghcjs

# глава 6: GTK3

## Синтаксис

- obj <- <widgetName> Новое - как создаются виджеты (например, Windows, кнопки, сетки)
- set <widget> [<attributes>] - Установить атрибуты, определенные как атрибут Attr в документации виджета (например, buttonLabel)
- on <widget> <event> <IO action> - Добавление действия IO к виджетам Signal self ( например, buttonActivated)

## замечания

Во многих дистрибутивах Linux библиотека Haskell Gtk3 доступна в виде пакета в диспетчере системных пакетов (например, `libghc-gtk` в APT Ubuntu). Тем не менее, для некоторых разработчиков , возможно , было бы предпочтительнее использовать инструмент , как `stack` для управления изолированных сред, и установлен GTK3 через `cabal` вместо через глобальную установку менеджер системы упаковки. Для этого параметра `gtk2hs-buildtools` . Запустите `cabal install gtk2hs-buildtools` перед добавлением `gtk` , `gtk3` или любых других библиотек Haskell на основе Gtk в вашу запись, `build-depends` от проекта, в вашем файле cabal.

## Examples

### Привет, мир в Gtk

В этом примере показано, как можно создать простой «Hello World» в Gtk3, настроив виджеты окон и кнопок. Пример кода также продемонстрирует, как устанавливать различные атрибуты и действия в виджетах.

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  on window objectDestroy mainQuit
  set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
  set button [ buttonLabel := "Hello World" ]
  on button buttonActivated $ do
```

```
    putStrLn "A \"clicked\"-handler to say \"destroy\""
    widgetDestroy window
  set window [ containerChild := button ]
  widgetShowAll window
  mainGUI -- main loop
```

Прочитайте GTK3 онлайн: https://riptutorial.com/ru/haskell/topic/7342/gtk3

# глава 7: IO

## Examples

### Чтение всего содержимого стандартного ввода в строку

```
main = do
    input <- getContents
    putStr input
```

Входные данные:

```
This is an example sentence.
And this one is, too!
```

Выход:

```
This is an example sentence.
And this one is, too!
```

Примечание. Эта программа будет фактически печатать части вывода до того, как все входные данные будут полностью прочитаны. Это означает, что, если, например, вы используете `getContents` поверх файла 50MiB, ленивая оценка Haskell и сборщик мусора гарантируют, что только части файла, которые в настоящее время необходимы (чтение: незаменим для дальнейшего выполнения), будут загружены в память. Таким образом, файл 50MiB не будет загружен сразу в память.

### Чтение строки со стандартного ввода

```
main = do
    line <- getLine
    putStrLn line
```

Входные данные:

```
This is an example.
```

Выход:

```
This is an example.
```

### Анализ и построение объекта со стандартного ввода

```
readFloat :: IO Float
```

```
readFloat =
    fmap read getLine


main :: IO ()
main = do
    putStr "Type the first number: "
    first <- readFloat

    putStr "Type the second number: "
    second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

Входные данные:

```
Type the first number: 9.5
Type the second number: -2.02
```

Выход:

```
9.5 + -2.02 = 7.48
```

## Чтение из файлов

Как и в нескольких других частях библиотеки ввода-вывода, функции, которые неявно используют стандартный поток, имеют аналог в `System.IO` который выполняет одно и то же задание, но с дополнительным параметром слева, типа `Handle` , который представляет поток, являющийся обрабатываются. Например, `getLine :: IO String` имеет экземпляр `hGetLine :: Handle -> IO String` .

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- show file contents line by line
    end <- hIsEOF handle
    when ( not end ) $ do
        line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
    hPutStr stderr "Type a filename: "
    filename <- getLine
    handle <- openFile filename ReadMode
    dumpFile handle filename 1
```

```
    hClose handle
```

Содержимое файла `example.txt` :

```
This is an example.
Hello, world!
This is another example.
```

Входные данные:

```
Type a filename: example.txt
```

Выход:

```
example.txt:1: This is an example.
example.txt:2: Hello, world!
example.txt:3: This is another example
```

## Проверка условий окончания файла

Немного интуитивно понятный способ использования стандартных библиотек ввода / вывода большинства других языков, `isEOF` не требует выполнения операции чтения перед проверкой состояния EOF; время выполнения сделает это за вас.

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
    end <- isEOF
    if end then
        putStrLn $ "End-of-file reached at line " ++ show line ++ "."
    else do
        getLine
        eofTest $ line + 1


main :: IO ()
main =
    eofTest 1
```

Входные данные:

```
Line #1.
Line #2.
Line #3.
```

Выход:

```
End-of-file reached at line 4.
```

## Чтение слов из целого файла

В Haskell часто имеет смысл *не беспокоить файловые дескрипторы* вообще, а просто читать или записывать весь файл прямо с диска на память † и выполнять всю секцию / обработку текста с чистой строковой структурой данных. Это позволяет избежать смешивания IO и логики программ, что может значительно помочь избежать ошибок.

```
-- | The interesting part of the program, which actually processes data
--   but doesn't do any IO!
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | A simple wrapper that only fetches the data from disk, lets
--   'reverseWords' do its job, and puts the result to stdout.
main :: IO ()
main = do
   content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

Если `loremipsum.txt` содержит

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

то программа выведет

```
elit
adipiscing
consectetur
amet,
sit
dolor
ipsum
Lorem
```

Здесь `mapM_` просмотрел список всех слов в файле и напечатал каждую из них в отдельной строке с помощью `putStrLn` .

---

† Если вы считаете, что это бесполезно в памяти, у вас есть точка. Фактически, лента Haskell часто может избежать того, что весь файл должен постоянно находиться в памяти ... но будьте осторожны, этот ленивый IO вызывает свой собственный набор проблем. Для критически важных приложений часто имеет смысл принудительно читать весь файл, строго; вы можете сделать это с помощью `Data.Text` версии `readFile` .

## IO определяет действие `main` вашей программы

Чтобы выполнить исполняемый файл программы Haskell, вы должны предоставить файл с `main` функцией типа `IO ()`

```
main :: IO ()
```

```
main = putStrLn "Hello world!"
```

Когда Haskell скомпилирован, он анализирует данные `IO` здесь и превращает его в исполняемый файл. Когда мы запустим эту программу, она будет печатать `Hello world!` ,

Если у вас есть значения типа `IO a` отличные от `main` они ничего не сделают.

```
other :: IO ()
other = putStrLn "I won't get printed"

main :: IO ()
main = putStrLn "Hello world!"
```

Компиляция этой программы и ее запуск будут иметь тот же эффект, что и последний пример. Код в `other` игнорируется.

Для того чтобы код в `other` имел эффекты времени исполнения, вы должны *составить* его в `main` . Никакие значения `IO` конечном итоге не состоят в `main` будут иметь никакого эффекта времени исполнения. Чтобы составить два значения `IO` последовательно, вы можете использовать `do` -notation:

```
other :: IO ()
other = putStrLn "I will get printed... but only at the point where I'm composed into main"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

Когда вы компилируете и запускаете *эту* программу, она выводит

```
Hello world!
I will get printed... but only at the point where I'm composed into main
```

Обратите внимание, что порядок операций описывается тем, как `other` составлялись в `main` а не в порядке определения.

## Роль и цель IO

Haskell - это чистый язык, что означает, что выражения не могут иметь побочных эффектов. Побочным эффектом является то, что выражение или функция выполняет не только значение, например, изменение глобального счетчика или печать на стандартный вывод.

В Haskell боковые эффекты (в частности, те, которые могут влиять на реальный мир) моделируются с использованием `IO` . Строго говоря, `IO` является конструктором типа, беря тип и создавая тип. Например, `IO Int` - это тип вычисления ввода-вывода, производящий значение `Int` . Тип `IO` является *абстрактным* , а интерфейс, предоставляемый для `IO`

гарантирует, что некоторые незаконные значения (то есть функции с нечувствительными типами) не могут существовать, гарантируя, что все встроенные функции, которые выполняют IO, имеют тип возврата, заключенный в `IO` .

Когда запускается программа Haskell, выполняется вычисление, представленное значением Haskell с именем `main` , тип которого может быть `IO x` для любого типа `x` .

## Манипулирование значениями ввода-вывода

В стандартной библиотеке имеется множество функций, обеспечивающих типичные действия `IO` которые должен выполнять язык программирования общего назначения, например чтение и запись в дескрипторы файлов. Общие действия `IO` создаются и объединяются в основном с двумя функциями:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Эта функция (обычно называемая *связыванием* ) принимает `IO` действие и функцию, которая возвращает действие `IO` , и производит действие `IO` которое является результатом применения функции к значению, полученному первым действием `IO` .

```
return :: a -> IO a
```

Эта функция принимает любое значение (т. Е. Чистое значение) и возвращает вычисление IO, которое не делает IO и не дает заданное значение. Другими словами, это операция ввода-вывода без операции.

Существуют дополнительные общие функции, которые часто используются, но все они могут быть записаны в терминах двух выше. Например, `(>>) :: IO a -> IO b -> IO b` аналогичен `(>>=)` но результат первого действия игнорируется.

Простая программа приветствует пользователя, используя следующие функции:

```
main :: IO ()
main =
  putStrLn "What is your name?" >>
  getLine >>= \name ->
  putStrLn ("Hello " ++ name ++ "!")
```

Эта программа также использует `putStrLn :: String -> IO ()` и `getLine :: IO String` .

---

Примечание: типы определенных выше функций на самом деле более общие, чем те, которые указаны (а именно: `>>=` , `>>` и `return` ).

# Семантика ввода-вывода

Тип `IO` в Haskell имеет очень схожую семантику с языком императивных языков программирования. Например, когда записывается `s1 ; s2` на императивном языке, чтобы указать исполняющий оператор `s1` , затем оператор `s2` , можно написать `s1 >> s2` чтобы моделировать одно и то же в Haskell.

Тем не менее, семантика `IO` немного отличается от того, что можно ожидать от императивного фона. Функция `return` *не* прерывает поток управления - она не влияет на программу, если другое действие `IO` выполняется последовательно. Например, `return () >> putStrLn "boom"` корректно выводит «стрелу» на стандартный вывод.

---

Формальная семантика `IO` может быть задана в терминах простых равенств, связанных с функциями в предыдущем разделе:

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

Эти законы обычно называются левой идентичностью, правильной идентичностью и составом, соответственно. Их можно более естественным образом выразить с точки зрения функции

```
(>=>) :: (a -> IO b) -> (b -> IO c) -> a -> IO c
(f >=> g) x = (f x) >>= g
```

следующее:

```
return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
(f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

---

# Lazy IO

Функции, выполняющие вычисления ввода-вывода, как правило, строгие, что означает, что все предшествующие действия в последовательности действий должны быть завершены до начала следующего действия. Обычно это полезное и ожидаемое поведение - `putStrLn "X" >> putStrLn "Y"` должен печатать «XY». Однако некоторые функции библиотеки выполняют лёгкие операции ввода-вывода, что означает, что действия ввода-вывода, необходимые для получения значения, выполняются только тогда, когда фактически используется значение. Примерами таких функций являются `getContents` и `readFile` . Lazy I / O может резко снизить производительность программы Haskell, поэтому при

использовании библиотечных функций следует обратить внимание на то, какие функции являются ленивыми.

## IO и `do` нотации

Haskell обеспечивает более простой способ объединения различных значений ввода-вывода в большие значения IO. Этот специальный синтаксис известен как `do` обозначение * и это просто синтаксический сахар для использований в `>>=` , `>>` и `return` функций.

Программа в предыдущем разделе , можно записать двумя различными способами , используя `do` нотации, первый из которых расположение чувствительных и второе существо расположение нечувствительны:

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")


main = do {
  putStrLn "What is your name?" ;
  name <- getLine ;
  putStrLn ("Hello " ++ name ++ "!")
  }
```

Все три программы в точности эквивалентны.

---

* Обратите внимание , что `do` обозначение также применимо к более широкому классу конструкторов типа под названием *монада.*

### Получение «а» из «IO a»

Общий вопрос : « У меня есть значение `IO a` , но я хочу сделать что - то , что `a` стоимость:? Как я могу получить доступ к нему» Как можно работать с данными, поступающими из внешнего мира (например, увеличивая число, набранное пользователем)?

Дело в том, что если вы используете чистую функцию для данных, полученных нечисто, тогда результат все еще нечист. Это зависит от того, что сделал пользователь! Значение типа `IO a` стендах для «вычисления бокового осуществления , в результате чего значения типа `a` » , который может быть запущен *только* (а) слагающим его в `main` и (б) составление и выполнение программы. По этой причине, нет никакого способа , в чистом Haskell мире «получить `a` аут».

Вместо этого мы хотим построить новое вычисление, новое значение `IO` , которое использует значение `a` *во время выполнения* . Это еще один способ *составления* значений `IO` и поэтому мы можем использовать `do` -notation:

```
-- assuming
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "My computation resulted in: " ++ show int

newComputation :: IO ()
newComputation = do
  int <- myComputation       -- we "bind" the result of myComputation to a name, 'int'
  putStrLn $ getMessage int   -- 'int' holds a value of type Int
```

Здесь мы используем чистую функцию ( `getMessage` ) , чтобы превратить `Int` в `String` , но мы используем `do` запись , чтобы сделать его можно применять в результате `IO` вычисления `myComputation` *когда* (после) гласит , что вычисление. В результате получается более крупное вычисление `IO newComputation` . Этот метод использования чистых функций в нечистом контексте называется *подъемом* .

## Письмо на stdout

В соответствии с спецификацией языка Haskell 2010 следующие стандартные функции ввода-вывода доступны в Prelude, поэтому импорт не требуется для их использования.

#### `putChar :: Char -> IO ()` - записывает `char` в `stdout`

```
Prelude> putChar 'a'
aPrelude>  -- Note, no new line
```

#### `putStr :: String -> IO ()` - записывает `String` в `stdout`

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- Note, no new line
```

#### `putStrLn :: String -> IO ()` - записывает `String` в `stdout` и добавляет новую строку

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

#### `print :: Show a => a -> IO ()` - пишет `a` экземпляр `Show` на `stdout`

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe is an instance of the `Show` type class
Just 'a'
Prelude> print Nothing
Nothing
```

Напомним, что вы можете создавать экземпляры `Show` для ваших собственных типов, используя `deriving` :

```
-- In ex.hs
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- Person is an instance of `Show`, thanks to `deriving`
```

Загрузка и запуск в GHCi:

```
Prelude> :load ex.hs
[1 of 1] Compiling ex             ( ex.hs, interpreted )
Ok, modules loaded: ex.
*Main> main  -- from ex.hs
Person {name = "Alex"}
*Main>
```

## Чтение из `stdin`

В соответствии с спецификацией языка Haskell 2010 следующие стандартные функции ввода-вывода доступны в Prelude, поэтому импорт не требуется для их использования.

#### `getChar :: IO Char` - читать `Char` от `stdin`

```
-- MyChar.hs
main = do
  myChar <- getChar
  print myChar

-- In your shell

runhaskell MyChar.hs
a -- you enter a and press enter
'a'  -- the program prints 'a'
```

#### `getLine :: IO String` - чтение `String` из `stdin` , новый символ строки

```
Prelude> getLine
Hello there!  -- user enters some text and presses enter
"Hello there!"
```

#### `read :: Read a => String -> a` - преобразовать String в значение

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

Другими, менее распространенными функциями являются:

- `getContents :: IO String` - возвращает все входные данные пользователя как одну строку, которая читается лениво, поскольку это необходимо
- `interact :: (String -> String) -> IO ()` - принимает в качестве аргумента функцию типа String-> String. Весь вход со стандартного устройства ввода передается этой функции в качестве аргумента

Прочитайте IO онлайн: https://riptutorial.com/ru/haskell/topic/1904/io

# глава 8: Monad Transformers

## Examples

### Монадический счетчик

Пример того, как составить читатель, писатель и государственную монаду, используя монадные трансформаторы. Исходный код можно найти в этом репозитории

Мы хотим реализовать счетчик, который увеличивает его значение на заданную константу.

Начнем с определения некоторых типов и функций:

```
newtype Counter = MkCounter {cValue :: Int}
  deriving (Show)

-- | 'inc c n' increments the counter by 'n' units.
inc :: Counter -> Int -> Counter
inc (MkCounter c) n = MkCounter (c + n)
```

Предположим, что мы хотим выполнить следующее вычисление с использованием счетчика:

- установите счетчик на 0
- установить постоянную приращения 3
- увеличивайте счетчик 3 раза
- установить постоянную инкремента равной 5
- увеличивать счетчик в 2 раза

Государственная монада обеспечивает абстракции для прохождения состояния вокруг. Мы можем использовать государственную монаду и определить нашу функцию приращения как трансформатор состояния.

```
-- | CounterS is a monad.
type CounterS = State Counter

-- | Increment the counter by 'n' units.
incS :: Int-> CounterS ()
incS n = modify (\c -> inc c n)
```

Это уже позволяет нам более четко и лаконично вычислять вычисления:

```
-- | The computation we want to run, with the state monad.
mComputationS :: CounterS ()
mComputationS = do
  incS 3
  incS 3
  incS 3
```

```
  incS 5
  incS 5
```

Но мы все равно должны передавать константу приращения при каждом вызове. Мы бы хотели этого избежать.

# Добавление среды

Монада-читатель обеспечивает удобный способ передачи окружения. Эта монада используется в функциональном программировании для выполнения того, что в мире ОО известно как *инъекция зависимости* .

В своей простейшей версии монада-читатель требует двух типов:

- тип считываемого значения (т.е. наша среда, `r` ниже),
- значение, возвращаемое монада-читателя ( `a` ниже).

  Читатель ra

Однако нам нужно также использовать государственную монаду. Таким образом, нам необходимо использовать трансформатор `ReaderT` :

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

Используя `ReaderT` , мы можем определить наш счетчик со средой и состоянием следующим образом:

```
type CounterRS = ReaderT Int CounterS
```

Мы определяем функцию `incR` которая принимает константу инкремента из среды (используя `ask` ), и для определения нашей инкрементной функции в терминах нашей монады `CounterS` мы используем функцию `lift` (которая принадлежит классу трансформатора монады ).

```
-- | Increment the counter by the amount of units specified by the environment.
incR :: CounterRS ()
incR = ask >>= lift . incS
```

Используя монаду-читателю, мы можем определить наши вычисления следующим образом:

```
-- | The computation we want to run, using reader and state monads.
mComputationRS :: CounterRS ()
mComputationRS = do
  local (const 3) $ do
    incR
    incR
```

```
    incR
    local (const 5) $ do
      incR
      incR
```

# Требования изменились: нам нужна регистрация!

Теперь предположим, что мы хотим добавить журнал в наши вычисления, чтобы мы могли видеть эволюцию нашего счетчика во времени.

У нас также есть монада для выполнения этой задачи, монада-писателя . Как и в случае с монадой-читателем, поскольку мы их составляем, нам необходимо использовать монадный трансформатор считывателя:

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

Здесь `w` представляет тип выхода аккумулировать (который должен быть моноид, что позволило нам накопить это значение), `m` является внутренней монадой и типа вычислений. `a`

Затем мы можем определить наш счетчик с протоколированием, средой и состоянием следующим образом:

```
type CounterWRS = WriterT [Int] CounterRS
```

И используя `lift` мы можем определить версию функции приращения, которая регистрирует значение счетчика после каждого приращения:

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

Теперь вычисление, содержащее запись, может быть записано следующим образом:

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
  local (const 3) $ do
    incW
    incW
    incW
    local (const 5) $ do
      incW
      incW
```

# Делать все за один раз

Этот пример предназначен для демонстрации трансформаторов монады. Тем не менее, мы можем добиться такого же эффекта, составив все аспекты (среду, состояние и протоколирование) за одну операцию приращения.

Для этого мы используем ограничения типа:

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

Здесь мы приходим к решению, которое будет работать для любой монады, которая удовлетворяет вышеприведенным ограничениям. Вычислительная функция определяется таким образом:

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

так как в его теле мы используем inc '.

Мы могли бы выполнить это вычисление, например, в `ghci` например:

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

Прочитайте Monad Transformers онлайн: https://riptutorial.com/ru/haskell/topic/7752/monad-transformers

# глава 9: Monoid

## Examples

### Экземпляр Monoid для списков

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

Проверка законов `Monoid` для этого экземпляра:

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- prepending an empty list is a no-op

x `mappend` mempty = x   <->   xs ++ [] = xs  -- appending an empty list is a no-op

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs           -- appending lists is associative
```

### Свертывание списка моноидов в одно значение

`mconcat :: [a] -> a` это еще один метод `Monoid` класса типов :

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

Его определение по умолчанию - `mconcat = foldr mappend mempty` .

### Числовые моноиды

Числа являются моноидальными двумя способами: *добавлением* с 0 в качестве единицы измерения и *умножением* на 1 в качестве единицы. Оба они одинаково важны и полезны в разных обстоятельствах. Поэтому вместо того, чтобы выбирать предпочтительный экземпляр для чисел, есть два типа `newtypes` , `Sum` и `Product` чтобы пометить их для разных функций.

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
```

```
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

Это позволяет разработчику выбирать, какую функциональность использовать, обертывая значение в соответствующем `newtype` .

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

## Экземпляр Monoid for ()

`()` является `Monoid` . Поскольку существует только одно значение типа `()` , есть только одна вещь `mempty` и `mappend` могли бы сделать:

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

Прочитайте Monoid онлайн: https://riptutorial.com/ru/haskell/topic/2211/monoid

# глава 10: Profunctor

## Вступление

`Profunctor` это класс типов обеспечивается `profunctors` пакета в `Data.Profunctor` .

Подробное объяснение см. В разделе «Примечания».

## Синтаксис

- dimap :: Profunctor p => (a -> b) -> (c -> d) -> pbc -> pad
- lmap :: Profunctor p => (a -> b) -> pbc -> pac
- rmap :: Profunctor p => (b -> c) -> pab -> pac
- dimap id id = id
- lmap id = id
- rmap id = id
- dimap fg = lmap f. rmap g
- lmap f = dimap f id
- rmap f = dimap id f

## замечания

Профессора, как описано в документах по Hackage, «бифунтор, где первый аргумент контравариантен, а второй аргумент ковариантен».

Так что это значит? Ну, бифунтор похож на нормальный функтор, за исключением того, что он имеет два параметра вместо одного, каждый со своей собственной функцией `fmap` like для отображения на нем.

Быть «ковариантным» означает, что второй аргумент для профинансора - это как нормальный функтор: его функция отображения ( `rmap` ) имеет сигнатуру типа `Profunctor p => (b -> c) -> pab -> pac` . Он просто отображает функцию во втором аргументе.

Быть «контравариантным» делает первый аргумент немного более странным. Вместо отображения как нормального функтора его функция отображения ( `lmap` ) имеет сигнатуру типа `Profunctor p => (a -> b) -> pbc -> pac` . Это, казалось бы, обратное отображение имеет наибольшее значение для входов в функцию: вы должны запустить `a -> b` на входе, а затем свою другую функцию, оставив новый вход как `a` .

**Примечание:** Именование для нормального функтора с одним аргументом немного вводит в заблуждение: класс-указатель `Functor` реализует «ковариантные» функторы, а «контравариантные» функторы реализуются в `Contravariant` классе типов в `Data.Functor.Contravariant` и ранее (ошибочно названный) `Cofunctor` стеллаж в `Data.Cofunctor` .

## Examples

### (->) Profunctor

(->) - простой пример профинансора: левый аргумент является входом в функцию, а правый аргумент совпадает с аргументом функтора-читателя.

```
instance Profunctor (->) where
    lmap f g = g . f
    rmap f g = g . g
```

Прочитайте Profunctor онлайн: https://riptutorial.com/ru/haskell/topic/9694/profunctor

# глава 11: Reader / ReaderT

## Вступление

Reader предоставляет функциональность для передачи значения по каждой функции. Полезный справочник с некоторыми диаграммами можно найти здесь: http://adit.io/posts/2013-06-10-three-useful-monads.html

## Examples

### Простая демонстрация

Ключевой частью монады-читателя является функция `ask` ( https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask) , которая определена для иллюстративных цели:

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
  x <- ask
  return $ "The value is: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
  x <- reader id
  lift $ print "Hello from within"
  liftAnnotated $ print "Hello from within..."
  return $ "The value is: " ++ show x
```

Вышеприведенное будет распечатываться:

```
"The value is: 100"
"Hello from within"
```

```
"Hello from within..."
"The value is: 200"
```

Прочитайте Reader / ReaderT онлайн: https://riptutorial.com/ru/haskell/topic/9320/reader---readert

# глава 12: XML

## Вступление

Кодирование и декодирование XML-документов.

## Examples

### Кодирование записи с использованием библиотеки `xml`

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | Create XML from a Package
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

Прочитайте XML онлайн: https://riptutorial.com/ru/haskell/topic/9264/xml

# глава 13: zipWithM

## Вступление

`zipWithM` - это `zipWith` поскольку `mapM` должен `map` : он позволяет объединять два списка, используя монадическую функцию.

Из модуля `Control.Monad`

## Синтаксис

- zipWithM :: Аппликация m => (a -> b -> mc) -> [a] -> [b] -> m [c]

## Examples

### Вычисление отпускных цен

Предположим, вы хотите узнать, имеет ли определенный набор цен продажи смысл для магазина.

Первоначально изначально стоимость составляла 5 долларов США, поэтому вы не хотите принимать продажу, если цена продажи меньше для любого из них, но вы хотите знать, что означает новая цена.

Вычисление одной цены легко: вы рассчитываете продажную цену и возвращаете `Nothing` если вы не получаете прибыль:

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

Чтобы вычислить его для всей продажи, `zipWithM` делает это очень простым:

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

Это не вернет `Nothing` если какая-либо из продажных цен будет ниже 5 долларов США.

Прочитайте zipWithM онлайн: https://riptutorial.com/ru/haskell/topic/9685/zipwithm

# глава 14: Алгоритмы сортировки

## Examples

### Вставка Сортировка

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

**Пример использования:**

```
> isort [5,4,3,2,1]
```

**Результат:**

```
[1,2,3,4,5]
```

### Сортировка слиянием

**Закаленное слияние двух упорядоченных списков**

Сохранение дубликатов:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

**Версия сверху вниз:**

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))

firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

Он определен таким образом для ясности, а не для эффективности.

**Пример использования:**

```
> msort [3,1,4,5,2]
```

**Результат:**

```
[1,2,3,4,5]
```

**Нижняя версия:**

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
    go [a] = a
    go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

## Quicksort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                      ++ [x] ++
               qsort [b | b <- xs, b >= x]
```

## Сортировка пузырьков

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
               t | t == s    -> t
                 | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2    = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

## Перестановка Сортировать

Также известен как богосор .

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

Чрезвычайно неэффективен (на современных компьютерах).

## Выбор сортировки

Сортировка выбора выбирает минимальный элемент повторно, пока список не будет пуст.

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

Прочитайте Алгоритмы сортировки онлайн: https://riptutorial.com/ru/haskell/topic/2300/алгоритмы-сортировки

# глава 15: Анализ HTML с объективом и объективом

## Examples

### Извлечь текстовое содержимое из div с определенным идентификатором

Taggy-lens позволяет нам использовать объективы для анализа и проверки документов HTML.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package
taggy-lens

{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "\
    \<!doctype html><html><body>\
    \<div>first div</div>\
    \<div id=\"thediv\">second div</div>\
    \<div id=\"not-thediv\">third div</div>"

main :: IO ()
main = do
    T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

### Фильтрация элементов из дерева

Найдите div с `id="article"` и разделите все теги внутреннего скрипта.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package
taggy-lens --package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer
```

```
somehtmlSmall :: Text
somehtmlSmall =
    "<!doctype html><html><body>\
    \<div id=\"article\"><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>\
    \</body></html>"

renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
            allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                     (children %~
                      filter (\n -> n ^? element . name /= Just "script")))
                mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- outputs:
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

Вклад, основанный на ответе @ duplode's SO

Прочитайте Анализ HTML с объективом и объективом онлайн:
https://riptutorial.com/ru/haskell/topic/6962/анализ-html-с-объективом-и-объективом

# глава 16: Аппликативный метод

## Вступление

`Applicative` - это класс типов `f :: * -> *` который позволяет использовать приложение отмененной функции над структурой, в которой функция также встроена в эту структуру.

## замечания

## Определение

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Обратите внимание на ограничение `Functor` на `f` . `pure` функция возвращает свой аргумент, встроенный в `Applicative` структуру. Функция infix `<*>` (произносится как «apply») очень похожа на `fmap` за исключением функции, встроенной в `Applicative` структуру.

Правильный экземпляр `Applicative` должен удовлетворять *прикладным законам* , хотя они не применяются компилятором:

```
pure id <*> a = a                              -- identity
pure (.) <*> a <*> b <*> c = a <*> (b <*> c)   -- composition
pure f <*> pure a = pure (f a)                 -- homomorphism
a <*> pure b = pure ($ b) <*> a                -- interchange
```

## Examples

### Альтернативное определение

Поскольку каждый прикладной функтор является функтором , `fmap` всегда можно использовать на нем; таким образом, сущность Аппликативного - это сопряжение несущего содержимого, а также способность его создавать:

```
class Functor f => PairingFunctor f where
  funit :: f ()                  -- create a context, carrying nothing of import
  fpair :: (f a,f b) -> f (a,b)  -- collapse a pair of contexts into a pair-carrying context
```

Этот класс изоморфен `Applicative` .

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

Наоборот,

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

### Общие примеры применения

# Может быть

`Maybe` , это аппликативный функтор, содержащий возможно отсутствующее значение.

```
instance Applicative Maybe where
    pure = Just

    Just f <*> Just x = Just $ f x
    _ <*> _ = Nothing
```

`pure` поднимает данное значение в `Maybe` , применяя `Just` к нему. Функция `(<*>)` применяет функцию, завернутую в « `Maybe` к значению в « `Maybe` . Если присутствуют как функция, так и значение (построено с помощью `Just` ), функция применяется к значению, и возвращается завернутый результат. Если либо отсутствует, вычисление не может продолжаться, и вместо него `Nothing` не возвращается.

# Списки

Один из способов, чтобы списки соответствовали сигнатуре типа `<*> :: [a -> b] -> [a] -> [b]` - это декартово произведение двух списков, объединяющее каждый элемент первого списка с каждым элемент второго:

```
fs <*> xs = [f x | f <- fs, x <- xs]
         -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

Обычно это интерпретируется как эмулирующий недетерминизм со списком значений, стоящих для недетерминированного значения, возможные значения которого находятся над этим списком; поэтому комбинация двух недетерминированных значений охватывает все возможные комбинации значений в двух списках:

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

# Бесконечные потоки и zip-списки

Существует класс `Applicative` который «застегивает» свои два входа вместе. Один простой пример - бесконечные потоки:

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

`Applicative` экземпляр `Stream` применяет поток функций к потоку аргументов по-разному, сопоставляя значения в двух потоках по положению. `pure` возвращает постоянный поток - бесконечный список одного фиксированного значения:

```
instance Applicative Stream where
    pure x = let s = Stream x s in s
    Stream f fs <*> Stream x xs = Stream (f x) (fs <*> xs)
```

Списки также допускают «zippy» `Applicative` экземпляр, для которого существует `ZipList` :

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

Поскольку `zip` обрезает свой результат в соответствии с самым коротким входом, единственная реализация `pure` которая удовлетворяет `Applicative` законам, - это та, которая возвращает бесконечный список:

```
    pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

Например:

```
ghci> getZipList $ ZipList [(+1),(+2)] <*> ZipList [3,30,300]
[4,32]
```

Две возможности напоминают нам внешний и внутренний продукт, аналогичный умножению матрицы 1 столбца ( `nx 1` ) с 1-строчной ( `1 xm` ) в первом случае, в результате получается матрица `nxm` (но сплющенная ); или умножения 1-строчной и 1-столбцовой матриц (но без суммирования) во втором случае.

# функции

Когда они специализированы по функциям `(->) r` , сигнатуры типа `pure` и `<*>` соответствуют `S` комбинаторов `K` и `S` соответственно:

```
pure :: a -> (r -> a)
```

```
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

`pure` должен быть `const` , а `<*>` принимает пару функций и применяет их каждый к фиксированному аргументу, применяя два результата:

```
instance Applicative ((->) r) where
    pure = const
    f <*> g = \x -> f x (g x)
```

Функции - прототипный «zippy» аппликативный. Например, поскольку бесконечные потоки изоморфны `(->) Nat` , ...

```
-- | Index into a stream
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | List all the return values of the function in order
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

... представление потоков по более высокому порядку создает автоматически `Applicative` экземпляр zippy.

Прочитайте Аппликативный метод онлайн: https://riptutorial.com/ru/haskell/topic/8162/аппликативный-метод

# глава 17: арифметика

## Вступление

В Haskell все выражения (включая числовые константы и функции, действующие на них) имеют разрешимый тип. Во время компиляции тип-checker определяет тип выражения из типов элементарных функций, которые его составляют. Так как данные по умолчанию неизменяемы, то нет операций типа «casting», но есть функции, которые копируют данные и обобщают или специализируют типы в пределах причины.

## замечания

# Числовая иерархия типов

`Num` сидит в корневой строке иерархии типов. Его характерные операции и некоторые общие примеры показаны ниже (те, которые загружаются по умолчанию с Prelude плюс те, что у `Data.Complex` ):

```
λ> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- Defined in 'GHC.Num'
instance RealFloat a => Num (Complex a) -- Defined in 'Data.Complex'
instance Num Word -- Defined in 'GHC.Num'
instance Num Integer -- Defined in 'GHC.Num'
instance Num Int -- Defined in 'GHC.Num'
instance Num Float -- Defined in 'GHC.Float'
instance Num Double -- Defined in 'GHC.Float'
```

Мы уже видели класс `Fractional` , который требует `Num` и вводит понятия «деления» `(/)` и обратного числа:

```
λ> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
      -- Defined in 'GHC.Real'
instance RealFloat a => Fractional (Complex a) -- Defined in 'Data.Complex'
```

```
instance Fractional Float -- Defined in 'GHC.Float'
instance Fractional Double -- Defined in 'GHC.Float'
```

`Real` классы моделей .. действительные числа. Он требует `Num` и `Ord` , поэтому он моделирует упорядоченное числовое поле. В качестве контрпримера комплексные числа *не* являются упорядоченным полем (т. Е. Не имеют естественного отношения упорядочения):

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- Defined in 'GHC.Real'
instance Real Word -- Defined in 'GHC.Real'
instance Real Integer -- Defined in 'GHC.Real'
instance Real Int -- Defined in 'GHC.Real'
instance Real Float -- Defined in 'GHC.Float'
instance Real Double -- Defined in 'GHC.Float'
```

`RealFrac` представляет числа, которые могут быть округлены

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- Defined in 'GHC.Real'
instance RealFrac Float -- Defined in 'GHC.Float'
instance RealFrac Double -- Defined in 'GHC.Float'
```

`Floating` (что подразумевает `Fractional` ) представляет собой константы и операции, которые могут не иметь конечного десятичного разложения.

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
```

```
  GHC.Float.expm1 :: a -> a
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- Defined in ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- Defined in ‘Data.Complex’
instance Floating Float -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
```

Внимание: в то время как выражения, такие как `sqrt . negate :: Floating a => a -> a` абсолютно верны, они могут возвращать `NaN` («не-число»), что может быть не предполагаемым поведением. В таких случаях мы, возможно, захотим работать над полем Комплекс (показано ниже).

# Examples

## Основные примеры

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

В приведенных выше примерах, типа проверка выводит типа вируса *класса* , а не конкретный типа для двух констант. В Haskell класс `Num` является наиболее общим численным (поскольку он охватывает целые числа и числа), но `pi` должен принадлежать к более специализированному классу, поскольку он имеет ненулевую дробную часть.

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

Конкретные типы, указанные выше, были выведены GHC. Более общие типы, такие как `list0 :: Num a => [a]` , сработали бы, но было бы еще сложнее сохранить (например, если бы один из них содержал `Double` в списке `Num` s) из-за описанных выше предостережений.

## `Не удалось вывести (Fractional Int) ...`

Сообщение об ошибке в заголовке является общей ошибкой начинающего. Давайте посмотрим, как это происходит и как это исправить.

Предположим, нам нужно вычислить среднее значение списка чисел; следующее заявление, похоже, сделает это, но оно не будет компилироваться:

```
averageOfList ll = sum ll / length ll
```

Проблема заключается в функции деления `(/)` : ее сигнатура есть `(/) :: Fractional a => a -> a -> a` , но в случае выше знаменатель (заданный `length :: Foldable t => ta -> Int` ) имеет тип `Int` (и `Int` не принадлежит классу `Fractional` ), следовательно, сообщение об ошибке.

Мы можем исправить сообщение об ошибке с `fromIntegral :: (Num b, Integral a) => a -> b` . Можно видеть, что эта функция принимает значения любого `Integral` типа и возвращает соответствующие в классе `Num` :

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

## Примеры функций

Какой тип `(+)` ?

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

Какой тип `sqrt` ?

```
λ> :t sqrt
sqrt :: Floating a => a -> a
```

Каков тип `sqrt . fromIntegral` ?

```
sqrt . fromIntegral :: (Integral a, Floating c) => a -> c
```

Прочитайте арифметика онлайн: https://riptutorial.com/ru/haskell/topic/8616/арифметика

# глава 18: Базы данных

## Examples

### Postgres

Postgresql-simple - это библиотека Haskell среднего уровня для связи с базой данных PostgreSQL. Он очень прост в использовании и предоставляет API-интерфейс типа для чтения / записи в БД.

Выполнение простого запроса так же просто, как:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- execute with no parameter substitution
  print i
```

### Замена параметров

PostreSQL-Simple поддерживает подстановку параметров для безопасных параметризованных запросов с использованием `query` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

### Выполнение вставок или обновлений

Вы можете запускать вставки / обновление SQL-запросов с помощью `execute` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

Прочитайте Базы данных онлайн: https://riptutorial.com/ru/haskell/topic/4444/базы-данных

# глава 19: Бесплатные монады

## Examples

### Свободные монады разделяют монадические вычисления на структуры данных и интерпретаторы

Например, вычисление, включающее команды для чтения и записи из приглашения:

Сначала мы описываем «команды» нашего вычисления как типа данных «Фунтор»

```
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

Затем мы используем `Free` для создания «Free Monad over `TeletypeF` » и создаем некоторые основные операции.

```
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

Поскольку `Free f` является `Monad` , когда `f` является `Functor` , мы можем использовать стандартную `Monad` комбинатор (включая `do` запись) , чтобы построить `Teletype` вычисления.

```
import Control.Monad -- we can use the standard combinators

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

Наконец, мы пишем «интерпретатор», который превращает `Teletype a` ценности, которые мы знаем, как работать с `IO a`

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
  run (PrintLine str x) = putStrLn *> return x
```

```
  run (ReadLine f) = fmap f getLine
```

Что мы можем использовать для «запуска» `Teletype a` вычисления в `IO`

```
> interpretTeletype mockingbird
hello
hello
goodbye
goodbye
this will go on forever
this will go on forever
```

## Свободные монады похожи на неподвижные точки

Сравните определение `Free` с определением `Fix` :

```
data Free f a = Return a
              | Free (f (Free f a))

newtype Fix f = Fix { unFix :: f (Fix f) }
```

В частности, сравните тип конструктора `Free` с типом конструктора `Fix` . `Free` слои создают функтор так же, как `Fix` , за исключением того, что `Free` имеет дополнительный `Return a` .

## Как работают foldFree и iterM?

Есть несколько функций, которые помогут свести `Free` вычисления, интерпретируя их в другую монаду `m` : `iterM :: (Functor f, Monad m) => (f (ma) -> ma) -> (Free fa -> ma)` и `foldFree :: Monad m => (forall x. fx -> mx) -> (Free fa -> ma)` . Что они делают?

Сначала давайте посмотрим, что потребуется, чтобы скрыть интерпретацию функции `Teletype a` в `IO` вручную. Мы можем видеть, что `Free fa` определяется

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

`Pure` случай прост:

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

Теперь, как интерпретировать вычисление `Teletype` которое было построено с помощью конструктора `Free` ? Мы хотели бы получить значение типа `IO a` , исследуя `teletypeF :: TeletypeF (Teletype a)` . Для начала напишем функцию `runIO :: TeletypeF a -> IO a` которая отображает один слой свободной монады в действие `IO` :

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
runIO (ReadLine k) = fmap k getLine
```

Теперь мы можем использовать `runIO` для заполнения остальной `interpretTeletype` . Напомним, что `teletypeF :: TeletypeF (Teletype a)` является слоем функтора `TeletypeF` который содержит остальные вычисления `Free` . Мы будем использовать `runIO` интерпретировать внешний слой (таким образом , мы имеем `runIO teletypeF :: IO (Teletype a)` ) , а затем использовать `IO` монады `>>=` комбинатор интерпретировать Возвращенный `Teletype a` .

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

Определение `foldFree` - это просто `interpretTeletype` , за исключением того, что функция `runIO` была `runIO` . В результате `foldFree` работает независимо от любого конкретного базового функтора и любой целевой монады.

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

`foldFree` имеет тип ранга-2: `eta` - естественное преобразование. Мы могли бы дать `foldFree` типа `Monad m => (f (Free fa) -> m (Free fa)) -> Free fa -> ma` , но это дает `eta` свободу осматривая `Free` вычисления внутри `f` слоя. Предоставление `foldFree` этого более ограничительного типа гарантирует, что `eta` может обрабатывать только один слой за раз.

`iterM` дает функции свертывания возможность изучить подкоманду. Результат (monadic) предыдущей итерации доступен для следующего параметра внутри `f` . `iterM` аналогичен *параморфизму,* тогда как `foldFree` подобен *катаморфизму* .

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

## Свободная монада

Существует альтернативная формулировка свободной монады, называемой монадой фрира (или приглашением, или оперативной). Мост Freer не требует экземпляра Functor для своего базового набора команд, и он имеет более узнаваемую структуру списка, чем стандартная свободная монада.

Freer monad представляет программы как последовательность атомных *инструкций,* принадлежащих набору команд `i :: * -> *` . Каждая команда использует свой параметр для объявления своего типа возврата. Например, набор базовых инструкций для `State`

монады выглядит следующим образом:

```
data StateI s a where
    Get :: StateI s s  -- the Get instruction returns a value of type 's'
    Put :: s -> StateI s ()  -- the Put instruction contains an 's' as an argument and returns
()
```

Секвенирование этих инструкций происходит с помощью `:>>=` constructor. `:>>=` принимает одну инструкцию, возвращающую `a` и добавляет ее к остальной части программы, передавая ее возвращаемое значение в продолжение. Другими словами, учитывая инструкцию, возвращающую `a` и функцию, чтобы превратить `a` в программу, возвращающую `b` `:>>=` создаст программу, возвращающую `b` .

```
data Freer i a where
    Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

Заметим, что `a` существует в количественном выражении в конструкторе `:>>=` . Единственный способ для интерпретатора узнать, что такое `a` - это сопоставление шаблонов на GADT `i` .

> **Кроме того,** лемма совместной Йонеды говорит нам, что `Freer` изоморфен `Free` . Напомним определение функтора `CoYoneda` :
>
> ```
> data CoYoneda i b where
>   CoYoneda :: i a -> (a -> b) -> CoYoneda i b
> ```
>
> `Freer i` эквивалентен `Free (CoYoneda i)` . Если вы берете конструкторы `Free` и устанавливаете `f ~ CoYoneda i` , вы получаете:
>
> ```
> Pure :: a -> Free (CoYoneda i) a
> Free :: CoYoneda i (Free (CoYoneda i) b) -> Free (CoYonda i) b ~
>         i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
> ```
>
> из которого мы можем восстановить конструкторы `Freer i` , просто установив `Freer i ~ Free (CoYoneda i)` .

Поскольку `CoYoneda i` является `Functor` для любого `i` , `Freer` является `Monad` для любого `i` , даже если `i` не является `Functor` .

```
instance Monad (Freer i) where
    return = Return
    Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- using `(->) r`'s instance of Functor, so fmap
= (.)
```

Интерпретаторы могут быть созданы для `Freer` путем сопоставления инструкций некоторым монахам-обработчикам.

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

Например, мы можем интерпретировать монаду `Freer (StateI s)` использующую монаду обычного `State s` в качестве обработчика:

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI s a -> State s a
          toState Get = State.get
          toState (Put x) = State.put x
```

Прочитайте Бесплатные монады онлайн: https://riptutorial.com/ru/haskell/topic/1290/бесплатные-монады

# глава 20: бифункторa

## Синтаксис

- bimap :: (a -> b) -> (c -> d) -> pac -> pbd
- first :: (a -> b) -> pac -> pbc
- второй :: (b -> c) -> pab -> pac

## замечания

`Functor` мельницы `Functor` ковариантен в *одном* типе параметра. Например, если `f` является `Functor` , то, учитывая `fa` и функцию вида `a -> b` , можно получить `fb` (используя `fmap` ).

`Bifunctor` ковариантен по *двум* типам параметров. Если `f` - `Bifunctor` , то, учитывая функцию `fab` и две функции, одну из `a -> c` , а другую из `b -> d` , можно получить `fcd` (используя `bimap` ).

`first` следует рассматривать как `fmap` по первому типу параметра, `second` как `fmap` над вторым, а `bimap` следует понимать как ковариационное отображение двух функций по параметрам первого и второго типа соответственно.

## Examples

### Общие случаи Бифунтера

# Двухэлементные кортежи

`(,)` является примером типа, имеющего экземпляр `Bifunctor` .

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

`bimap` выполняет пару функций и применяет их к соответствующим компонентам кортежа.

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

## `Either`

`Either` экземпляр `Bifunctor` выбирает одну из двух функций для применения в зависимости от того, является ли значение « `Left` или « `Right` .

```
instance Bifunctor Either where
    bimap f g (Left x) = Left (f x)
    bimap f g (Right y) = Right (g y)
```

## первый и второй

Если сопоставление ковариантно только по первому аргументу или только по второму аргументу, то нужно использовать `first` или `second` (вместо `bimap` ).

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

Например,

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

## Определение бифунтера

`Bifunctor` - это класс типов с двумя параметрами типа ( `f :: * -> * -> *` ), оба из которых могут ковариантно отображаться одновременно.

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

`bimap` можно рассматривать как применение пары операций `fmap` к типу данных.

Правильный экземпляр `Bifunctor` для типа `f` должен удовлетворять *бифунторным законам* , аналогичным *законам функтора* :

```
bimap id id = id  -- identity
bimap (f . g) (h . i) = bimap f h . bimap g i  -- composition
```

Класс `Bifunctor` находится в модуле `Data.Bifunctor` . Для версий GHC> 7.10 этот модуль поставляется вместе с компилятором; для более ранних версий вам необходимо установить пакет `bifunctors` .

Прочитайте бифунктора онлайн: https://riptutorial.com/ru/haskell/topic/8020/бифунктора

# глава 21: Буферы протокола Google

## замечания

Чтобы использовать протокольные буферы с Haskell, вы должны установить пакет `htprotoc` :

1. Клонировать проект от Github
2. Использовать Stack для сборки и установки

Теперь вы должны найти исполняемый файл `hprotoc` в `$HOME/.local/bin/` .

# Examples

### Создание, создание и использование простого файла .proto

Давайте сначала создадим простой `.proto` файл `person.proto`

```
package Protocol;

message Person {
    required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

После сохранения мы можем теперь создавать файлы Haskell, которые мы можем использовать в нашем проекте, запустив

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

Мы должны получить аналогичный результат:

```
Loading filepath: "/<path-to-project>/person.proto"
All proto files loaded
Haskell name mangling done
Recursive modules resolved
./Protocol/Person.hs
./Protocol.hs
Processing complete, have a nice day.
```

`hprotoc` создаст новую папку `Protocol` в текущем каталоге с `Person.hs` которую мы можем просто импортировать в наш проект haskell:

```
import Protocol (Person)
```

В качестве следующего шага, если использовать Stack add

```
  protocol-buffers
, protocol-buffers-descriptor
```

`build-depends:` и

```
Protocol
```

для `exposed-modules` в вашем файле `.cabal` .

Если мы получим входящее сообщение из потока, сообщение будет иметь тип `ByteString` .

Чтобы преобразовать `ByteString` (который, очевидно, должен содержать закодированные данные Person) в наш тип данных Haskell, нам нужно вызвать функцию `messageGet` которую мы импортируем по

```
import Text.ProtocolBuffers (messageGet)
```

который позволяет создать значение типа `Person` используя:

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
    Left   _           -> Nothing
    Right  (person, _)  -> Just person
```

Прочитайте Буферы протокола Google онлайн: https://riptutorial.com/ru/haskell/topic/5018/буферы-протокола-google

# глава 22: Быстрая проверка

## Examples

### Объявление собственности

В своем простейшем случае *свойство* является функцией, которая возвращает `Bool` .

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

Свойство объявляет высокоуровневый инвариант программы. Тестер QuickCheck проверит функцию со 100 случайными входами и проверит, что результат всегда `True` .

По соглашению функции, которые являются свойствами, имеют имена, которые начинаются с `prop_` .

### Проверка одного объекта

Функция `quickCheck` проверяет свойство на 100 случайных входах.

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK, passed 100 tests.
```

Если свойство некорректно для некоторого ввода, `quickCheck` распечатывает контрпример.

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- plainly not true for all xs

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** Failed! Falsifiable (after 2 tests):
[()]
```

### Проверка всех свойств в файле

`quickCheckAll` - это помощник шаблона Haskell, который находит все определения в текущем файле, имя которого начинается с `prop_` и проверяет их.

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

idempotent :: Eq a => (a -> a) -> a -> Bool
idempotent f x = f (f x) == f x

prop_sortIdempotent = idempotent sort

-- does not begin with prop_, will not be picked up by the test runner
```

```
sortDoesNotChangeLength xs = length (sort xs) == length xs


return []
main = $quickCheckAll
```

Обратите внимание, что требуется строка `return []` . Он делает текстовые надписи над этой строкой видимыми для Template Haskell.

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent from tree.hs:7 ===
+++ OK, passed 100 tests.
```

## Произвольное генерирование данных для пользовательских типов

Класс `Arbitrary` предназначен для типов, которые могут быть случайно сгенерированы QuickCheck.

Минимальная реализация `Arbitrary` - это `arbitrary` метод, который выполняется в монаде `Gen` для получения случайного значения.

Вот пример `Arbitrary` для следующего типа данных непустых списков.

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- randomly select one of the cases from the list
        End <$> arbitrary,  -- call a's instance of Arbitrary
        Cons <$>
            arbitrary <*>  -- call a's instance of Arbitrary
            arbitrary  -- recursively call NonEmpty's instance of Arbitrary
        ]
```

## Использование импликации (==>) для проверки свойств с предварительными условиями

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

Если вы хотите проверить, что свойство выполнено, если выполнено предварительное условие, вы можете использовать оператор `==>` . Обратите внимание: если очень маловероятно, чтобы произвольные входы соответствовали предварительному условию, QuickCheck может отказаться раньше.

```
prop_overlySpecific x y = x == 0 ==> x * y == 0
```

```
ghci> quickCheck prop_overlySpecific
*** Gave up! Passed only 31 tests.
```

## Ограничение размера тестовых данных

Трудно проверить функции с плохой асимптотической сложностью, используя quickcheck, поскольку случайные входы обычно не ограничены размером. Добавив верхнюю границу размера ввода, мы все еще можем проверить эти дорогостоящие функции.

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
    longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

Используя `quickCheckWith` с модифицированной версией `stdArgs` мы можем ограничить размер входов не более 10. В этом случае, когда мы создаем списки, это означает, что мы генерируем списки размером до 10. Наша функция перестановок не занимать слишком много времени для выполнения этих коротких списков, но мы все же можем быть достаточно уверенными в правильности нашего определения.

Прочитайте Быстрая проверка онлайн: https://riptutorial.com/ru/haskell/topic/1156/быстрая-проверка

# глава 23: Веб-разработка

## Examples

### служитель

Servant - это библиотека для декларирования API на уровне типа, а затем:

- (эта часть слуги может считаться веб-картой),
- получить клиентские функции (в haskell),
- генерировать клиентские функции для других языков программирования,
- создавать документацию для ваших веб-приложений
- и больше...

У Servant есть сжатый, но мощный API. Простой API может быть написан в очень немногих строках кода:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int
} deriving (Eq, Show, Generic)

instance ToJSON User  -- automatically convert User to JSON
```

Теперь мы можем объявить наш API:

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
```

в котором говорится, что мы хотим разоблачить `/users` для запросов `GET` с помощью `sortby` параметров `sortby` типа `SortBy` и вернуть JSON типа `User` в ответ.

Теперь мы можем определить наш обработчик:

```
-- This is where we'd return our user data, or e.g. do a database lookup
server :: Server UserAPI
server = return [User "Alex" 31]

userAPI :: Proxy UserAPI
userAPI = Proxy
```

```
app1 :: Application
app1 = serve userAPI server
```

И основной метод, который прослушивает порт `8081` и обслуживает наш пользовательский API:

```
main :: IO ()
main = run 8081 app1
```

Примечание. У Stack есть шаблон для генерации базовых API-интерфейсов в Servant, что полезно для быстрого и быстрого запуска.

## Йесод

Проект Yesod может быть создан с использованием `stack new` с использованием следующих шаблонов:

- `yesod-minimal` . Возможны простейшие леса Эсода.
- `yesod-mongo` . Использует MongoDB как механизм БД.
- `yesod-mysql` . Использует MySQL как механизм БД.
- `yesod-postgres` . Использует PostgreSQL как механизм БД.
- `yesod-postgres-fay` . Использует PostgreSQL как механизм БД. Использует язык Fay для front-end.
- `yesod-simple` . Рекомендуемый шаблон для использования, если вам не нужна база данных.
- `yesod-sqlite` . Использует SQlite как механизм БД.

`yesod-bin` пакет предоставляет `yesod` исполняемый файл, который можно использовать для запуска сервера разработки. Обратите внимание, что вы также можете запускать приложение напрямую, поэтому инструмент `yesod` является необязательным.

`Application.hs` содержит код, который отправляет запросы между обработчиками. Он также настраивает параметры базы данных и ведения журнала, если вы их использовали.

`Foundation.hs` определяет тип `App` , который можно рассматривать как среду для всех обработчиков. Будучи в монаде `HandlerT` , вы можете получить это значение, используя функцию `getYesod` .

`Import.hs` - это модуль, который просто реэкспортирует часто используемые материалы.

`Model.hs` содержит шаблон Haskell, который генерирует типы кода и данных, используемые для взаимодействия с БД. Присутствует только в том случае, если вы используете БД.

`config/models` - это то, где вы определяете свою схему БД. Используется `Model.hs` .

`config/routes` определяет URI веб-приложения. Для каждого метода HTTP маршрута вам

нужно создать обработчик с именем `{method}{RouteR}` .

`static/` directory содержит статические ресурсы сайта. Они скомпилируются в двоичный файл с `Settings/StaticFiles.hs` модуля `Settings/StaticFiles.hs` .

`templates/` содержит шаблоны Шекспира , которые используются при обслуживании запросов.

Наконец, `Handler/` directory содержит модули, которые определяют обработчики маршрутов.

Каждый обработчик является `HandlerT` монады `HandlerT` на основе IO. Вы можете проверить параметры запроса, его тело и другую информацию, сделать запросы к БД с помощью `runDB` , выполнить произвольный ввод-вывод и вернуть пользователю различные типы контента. Для обслуживания HTML `defaultLayout` функция `defaultLayout` которая позволяет использовать аккуратный состав шаблонов шекспира.

Прочитайте Веб-разработка онлайн: https://riptutorial.com/ru/haskell/topic/4721/веб-разработка

# глава 24: векторы

## замечания

> Он [Data.Vector] делает упор на очень высокую производительность благодаря слиянию циклов, сохраняя при этом богатый интерфейс. Основными типами данных являются массивы в коробке и распакованные массивы, а массивы могут быть неизменными (чистыми) или изменяемыми. Массивы могут содержать элементы Storable, подходящие для перехода на C и из C, и вы можете конвертировать между типами массивов. Массивы индексируются неотрицательными значениями Int.

У Haskell Wiki есть следующие рекомендации :

> В общем:
>
> - Конечные пользователи должны использовать Data.Vector.Unboxed для большинства случаев
> - Если вам нужно хранить более сложные структуры, используйте Data.Vector
> - Если вам нужно перейти на C, используйте Data.Vector.Storable
>
> Для библиотекарей;
>
> - Используйте общий интерфейс, чтобы обеспечить максимально гибкую библиотеку: Data.Vector.Generic

## Examples

### Модуль Data.Vector

Модуль Data.Vector, предоставляемый вектором, представляет собой высокопроизводительную библиотеку для работы с массивами.

После того, как вы импортировали `Data.Vector` , легко начать использовать `Vector` :

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

Вы даже можете иметь многомерный массив:

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]

Prelude Data.Vector> :t x
x :: Vector (Vector Integer)
```

## Фильтрация вектора

Фильтровать нечетные элементы:

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

## Отображение (`map`) и уменьшение (` fold`) вектора

Векторы могут быть `map` 'd и `fold'd`, filter `'d and` zip`'d:

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

Уменьшить до одного значения:

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

## Работа с несколькими векторами

Замените два массива на массив пар:

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

Прочитайте векторы онлайн: https://riptutorial.com/ru/haskell/topic/4738/векторы

# глава 25: взыскательность

## Examples

### Шаблоны Bang

Шаблоны, аннотированные с ударом ( `!` ), Оцениваются строго, а не лениво.

```
foo (!x, y) !z = [x, y, z]
```

В этом примере `x` и `z` оба будут вычисляться до нормальной нормальной формы головы, прежде чем возвращать список. Это эквивалентно:

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

Шаблоны Bang активируются с использованием языкового расширения Haskell 2010 `BangPatterns` .

### Нормальные формы

В этом примере приводится краткий обзор - для более подробного объяснения *нормальных форм* и примеров см. Этот вопрос .

# Уменьшенная нормальная форма

Приведенная нормальная форма (или просто нормальная форма, когда контекст ясен) выражения является результатом оценки всех приводимых подвыражений в данном выражении. Из-за нестрогой семантики Haskell (обычно называемой *лень* ) подвыражение не может быть приводимым, если оно находится под связующим (т.е. абстракция лямбда - `\x -> ..` ). Нормальная форма выражения обладает тем свойством, что если оно существует, оно уникально.

Другими словами, это не имеет значения (в терминах денотационной семантики), в каком порядке вы уменьшаете подвыражения. Однако ключ к написанию исполняемых программ Haskell часто гарантирует, что правильное выражение оценивается в нужное время, т. Е. Понимание оперативной семантики.

Выражение, нормальная форма которого сама по себе, называется *нормальной* .

Некоторые выражения, например, `let x = 1:x in x` , не имеют нормальной формы, но все же продуктивны. Выражение выражения все еще имеет *значение* , если оно допускает бесконечные значения, которые представляют собой список `[1,1, ...]` . Другие

выражения, такие как `let y = 1+y in y` , не имеют значения или их значение не `undefined` .

# Нормальная форма слабой головы

RNF соответствует полной оценке выражения - аналогично, нормальная форма слабой головки (WHNF) соответствует оценке *головы* выражения. Глава выражения `e` полностью оценивается, если `e` - приложение `Con e1 e2 .. en` и `Con` - конструктор; или абстракции `\x -> e1` ; или частичное приложение `f e1 e2 .. en` , где частичное приложение означает, что `f` принимает больше `n` аргументов (или, что эквивалентно, тип `e` является типом функции). В любом случае подвыражения `e1..en` могут быть оценены или не оценены для выражения в WHNF - они могут даже быть `undefined` .

Семантика оценки Haskell может быть описана в терминах WHNF - для оценки выражения `e` , сначала оцените его в WHNF, а затем рекурсивно оцените все его подвыражения слева направо.

Первоначальная функция `seq` используется для оценки выражения WHNF. `seq xy` денотационно равно `y` (значение `seq xy` точно равно `y` ); кроме того, `x` оценивается как WHNF, когда `y` оценивается WHNF. Выражение также может быть оценено в WHNF с шаблоном помех (активируется расширением `-XBangPatterns` ), синтаксис которого следующий:

```
f !x y = ...
```

В котором `x` будет оцениваться в WHNF, когда `f` оценивается, а `y` не (обязательно) оценивается. Шаблон помех может также отображаться в конструкторе, например

```
data X = Con A !B C .. N
```

в этом случае конструктор `Con` считается строгим в поле `B` , что означает, что поле `B` оценивается как WHNF, когда конструктор применяется к достаточным (здесь, двум) аргументам.

## Ленивые узоры

Ленивые или *неопровержимые* шаблоны (обозначаемые синтаксисом `~pat` ) - это шаблоны, которые всегда совпадают, даже не рассматривая сопоставимое значение. Это означает, что ленивые шаблоны будут соответствовать даже нижним значениям. Однако последующее использование переменных, связанных в подструктурах неопровержимого шаблона, заставит совпадение шаблонов происходить, оценивая до дна, если совпадение не будет выполнено.

Следующая функция в своем аргументе ленива:

```
f1 :: Either e Int -> Int
f1 ~(Right 1) = 42
```

и поэтому мы получаем

```
λ» f1 (Right 1)
42
λ» f1 (Right 2)
42
λ» f1 (Left "foo")
42
λ» f1 (error "oops!")
42
λ» f1 "oops!"
*** type mismatch ***
```

Следующая функция написана с ленивым шаблоном, но на самом деле использует переменную шаблона, которая заставляет совпадение, поэтому не будет `Left` аргументом:

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
3
λ» f2 (Right (error "oops!"))
*** Exception: oops!
λ» f2 (Left "foo")
*** Exception: lazypat.hs:5:1-21: Irrefutable pattern failed for pattern (Right x)
λ» f2 (error "oops!")
*** Exception: oops!
```

`let` привязки ленивы, ведут себя как неопровержимые шаблоны:

```
act1 :: IO ()
act1 = do
    ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
    ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

Здесь `act1` работает на входах, которые анализируются на любой список строк, тогда как в `act2` для `putStrLn s1` требуется значение `s1` которое заставляет совпадение шаблонов для `[s1, s2]` , поэтому оно работает только для списков ровно двух строк:

```
λ» act1
> ["foo"]
Done
```

```
λ» act2
> ["foo"]
*** readIO: no parse ***
```

## Строгие поля

В объявлении `data` префикс типа с bang ( `!` ) Делает поле *строгим полем* . Когда применяется конструктор данных, эти поля будут вычисляться до нормальной нормальной формы головы, поэтому данные в полях гарантированно всегда будут в слабой нормальной форме головы.

Строгие поля могут использоваться как для записей, так и для не-записей:

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

Прочитайте взыскательность онлайн: https://riptutorial.com/ru/haskell/topic/3798/взыскательность

# глава 26: Государственная Монада

## Вступление

Государственные монады - это своего рода монада, несущая состояние, которое может измениться во время каждого прогона вычисления в монаде. Реализации обычно имеют форму `State sa` которая представляет собой вычисление, которое переносит и потенциально модифицирует состояние типа `s` и производит результат типа `a` , но термин «монада государства» обычно может относиться к любой монаде, которая несет состояние. Пакет `mtl` и `transformers` обеспечивает общие реализации государственных монадов.

## замечания

Новички в Хаскелле часто уклоняются от `State` монады и рассматривают ее как табу, как заявленное преимущество функционального программирования - это избегание государства, так что вы не теряете это, когда используете `State` ? Более тонкий взгляд на то, что:

- Государство может быть полезно в *небольших контролируемых дозах* ;
- Тип `State` обеспечивает возможность очень точно контролировать дозу.

Причины, что если у вас есть `action :: State sa` , это говорит вам, что:

- `action` особенное, потому что оно зависит от состояния;
- Состояние имеет тип `s` , поэтому на `action` не может влиять никакое старое значение в вашей программе - только `s` или некоторое значение, доступное из некоторых `s` ;
- `runState :: State sa -> s -> (a, s)` ставит «барьер» вокруг действия с состоянием, так что его эффективность *не* может наблюдаться извне этого барьера.

Таким образом, это хороший набор критериев для использования `State` в конкретном сценарии. Вы хотите видеть, что ваш код *сводит к минимуму область состояния* , как путем выбора узкого типа для `s` и путем установки `runState` как можно ближе к «дну» (так что на ваши действия может влиять как минимум возможный.

## Examples

### Нумерация узлов дерева с помощью счетчика

У нас есть такой тип данных дерева:

```
data Tree a = Tree a [Tree a] deriving Show
```

И мы хотим написать функцию, которая присваивает номер каждому узлу дерева, от счетчика:

```
tag :: Tree a -> Tree (a, Int)
```

# Длинный путь

Сначала мы сделаем это очень далеко, так как он очень хорошо иллюстрирует механику низкого уровня `State` монады.

```
import Control.Monad.State

-- Function that numbers the nodes of a `Tree`.
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep is where the action happens.  This just gets the ball
    -- rolling, with `0` as the initial counter value.
    evalState (tagStep tree) 0

-- This is one monadic "step" of the calculation.  It assumes that
-- it has access to the current counter value implicitly.
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a subtrees) = do
    -- The `get :: State s s` action accesses the implicit state
    -- parameter of the State monad.  Here we bind that value to
    -- the variable `counter`.
    counter <- get

    -- The `put :: s -> State s ()` sets the implicit state parameter
    -- of the `State` monad.  The next `get` that we execute will see
    -- the value of `counter + 1` (assuming no other puts in between).
    put (counter + 1)

    -- Recurse into the subtrees.  `mapM` is a utility function
    -- for executing a monadic actions (like `tagStep`) on a list of
    -- elements, and producing the list of results.  Each execution of
    -- `tagStep` will be executed with the counter value that resulted
    -- from the previous list element's execution.
    subtrees' <- mapM tagStep subtrees

    return $ Tree (a, counter) subtrees'
```

# Рефакторинг

## Разделите счетчик на действие postIncrement

Бит, где мы `get` текущий счетчик, а затем `put` счетчик ting + 1, можно разделить на действие `postIncrement` , аналогичное тому, что предоставляют языки C-стиля:

```
postIncrement :: Enum s => State s s
```

```
postIncrement = do
    result <- get
    modify succ
    return result
```

## Разделите дерево в функцию более высокого порядка

Логику древовидной ходьбы можно разделить на свою собственную функцию, например:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
    a' <- action a
    subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

С помощью этой и функции `postIncrement` мы можем переписать `tagStep` :

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

## Использовать класс `Traversable`

Решение `mapTreeM` выше может быть легко переписано в экземпляр [класса `Traversable`](https://hackage.haskell.org/package/base/docs/Data-Traversable.html) :

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

Обратите внимание, что для этого требовалось использовать вместо `Monad` `Applicative` ( оператор `<*>` ).

Таким образом, теперь мы можем написать `tag` как pro:

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

Обратите внимание, что это работает для любого типа `Traversable` , а не только для нашего типа `Tree` !

## Избавление от шаблона `Traversable`

GHC имеет расширение `DeriveTraversable` которое устраняет необходимость записи

экземпляра выше:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}

data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

Прочитайте Государственная Монада онлайн: https://riptutorial.com/ru/haskell/topic/5740/государственная-монада

# глава 27: Графика с блеском

## Examples

### Установка блеска

Gloss легко устанавливается с помощью инструмента Cabal. Установив Cabal, можно `cabal install gloss` инсталлятор для установки Gloss.

В качестве альтернативы пакет можно создать из источника, загрузив исходный код из Hackage или GitHub и выполнив следующие действия:

1. Введите каталог `gloss/gloss-rendering/` и выполните `cabal install`
2. Войдите в каталог `gloss/gloss/` и еще раз выполните `cabal install`

### Получение чего-то на экране

В Gloss можно использовать функцию `display` для создания очень простой статической графики.

Для этого нужно сначала `import Graphics.Gloss` . Затем в коде должно быть указано следующее:

```
main :: IO ()
main = display window background drawing
```

`window` типа `Display` которое может быть построено двумя способами:

```
-- Defines window as an actual window with a given name and size
window = InWindow name (width, height) (0,0)

-- Defines window as a fullscreen window
window = FullScreen
```

Здесь последний аргумент `(0,0)` в `InWindow` обозначает местоположение верхнего левого угла.

**Для версий старше 1.11:** В более старых версиях Gloss `FullScreen` принимает еще один аргумент, который предназначен для размера кадра, который нарисован, на котором, в свою очередь, растягивается до полноэкранного размера, например: `FullScreen (1024,768)`

`background` имеет тип `Color` . Он определяет цвет фона, поэтому он прост как:

```
background = white
```

Затем мы добираемся до самого чертежа. Рисунки могут быть очень сложными. Как их указать, они будут рассмотрены в другом месте ([на этот раз можно ссылаться] [1]), но это может быть так же просто, как следующий круг с радиусом 80:

```
drawing = Circle 80
```

---

### Пример суммирования

Как более или менее указано в документации по Hackage, получить что-то на экране так же просто, как:

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    where
      window = InWindow "Nice Window" (200, 200) (0, 0)
      background = white
      drawing = Circle 80
```

Прочитайте Графика с блеском онлайн: https://riptutorial.com/ru/haskell/topic/5570/графика-с-блеском

# глава 28: Дата и время

## Синтаксис

- addDays :: Integer -> День -> День
- diffDays :: День -> День -> Целое число
- fromGregorian :: Integer -> Int -> Int -> Day

  ```
  convert from proleptic Gregorian calendar. First argument is year, second month number
  (1-12), third day (1-31). Invalid values will be clipped to the correct range, month
  first, then day.
  ```

- getCurrentTime :: IO UTCTime

## замечания

Модуль `Data.Time` из `time` пакета обеспечивает поддержку для извлечения и управления значениями даты и времени:

# Examples

### Поиск сегодняшней даты

Текущую дату и время можно найти с помощью `getCurrentTime` :

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

В качестве альтернативы, только дата возвращается по `fromGregorian` :

```
fromGregorian 1984 11 17  -- yields a Day
```

### Добавление, вычитание и сравнение дней

В течение `Day` мы можем выполнять простые арифметические и сопоставления, такие как добавление:

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
```

```
addDays 1 (fromGregorian 2000 12 31)
-- 2001-01-01
```

Вычесть:

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

и даже найти разницу:

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

обратите внимание, что порядок имеет значение:

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

Прочитайте Дата и время онлайн: https://riptutorial.com/ru/haskell/topic/4950/дата-и-время

# глава 29: Доверенные

## Examples

### Использование прокси

Тип `Proxy :: k -> *` , найденный в `Data.Proxy` , используется, когда вам нужно предоставить компилятору некоторую информацию о типе - например, выбрать экземпляр класса типа, который, тем не менее, не имеет значения во время выполнения.

```
{-# LANGUAGE PolyKinds #-}

data Proxy a = Proxy
```

Функции , которые используют `Proxy` - `ScopedTypeVariables a Proxy` обычно используют `ScopedTypeVariables` , чтобы выбрать экземпляр класса типа на основе `a` типа.

Например, классический пример неоднозначной функции,

```
showread :: String -> String
showread = show . read
```

что приводит к ошибке типа, поскольку разработчик не знает, какой экземпляр `Show` или `Read` использовать, может быть разрешен с помощью `Proxy` :

```
{-# LANGUAGE ScopedTypeVariables #-}

import Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

При вызове функции с `Proxy` , вам нужно использовать аннотацию типа , чтобы объявить , какие вы имели в виду. `a`

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- attempt to parse a char literal as a Bool
"*** Exception: Prelude.read: no parse
```

### Идиома «полиморфный прокси»

Поскольку `Proxy` содержит информации о времени выполнения, никогда не нужно сопоставлять шаблоны в конструкторе `Proxy` . Таким образом, общая идиома заключается в абстрактном `Proxy` типа `Proxy` с использованием переменной типа.

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

Теперь, если у вас есть `fa` в области для некоторого `f` , вам не нужно выписывать `Proxy :: Proxy a` при вызове `f` .

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

## Прокси - это как ()

Поскольку `Proxy` содержит информации о времени выполнения, вы всегда можете написать естественное преобразование `fa -> Proxy a` для любого `f` .

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

Это похоже на то, как любое значение всегда можно стереть до `()` :

```
unit :: a -> ()
unit _ = ()
```

Технически, `Proxy` - это конечный объект в категории функторов, так же как `()` - это конечный объект в категории значений.

Прочитайте Доверенные онлайн: https://riptutorial.com/ru/haskell/topic/8025/доверенные

# глава 30: Интерфейс внешней функции

## Синтаксис

- foreign import ccall unsafe «foo» hFoo :: Int32 -> IO Int32 {- импортирует функцию с именем `foo` в некоторый объектный файл и определяет символ `hFoo` который можно вызвать с кодом Haskell. -}

## замечания

Хотя у cabal есть поддержка для включения библиотек C и C ++ в пакет Haskell, есть несколько ошибок. Во-первых, если у вас есть данные (а не функция), определенные в `bo` которые используются в `ao` , и перечислите `C-sources: ac, bc` , то cabal не сможет найти данные. Это подтверждено в # 12152 . Обходной путь при использовании cabal состоит в том, чтобы переупорядочить список `C-sources` как `C-sources: bc, ac` . Это может не работать при использовании стека, потому что стек всегда связывает `C-sources` алфавитном порядке, независимо от того, в каком порядке вы их перечисляете.

Еще одна проблема заключается в том, что вы должны окружать любой код C ++ в файлах заголовка (.h) с помощью `#ifdef __cplusplus` . Это связано с тем, что GHC не понимает код C ++ в файлах заголовков. Вы все еще можете писать код C ++ в заголовочных файлах, но вы должны окружить его защитой.

`ccall` относится к *вызывающему соглашению* ; в настоящее время `ccall` и `stdcall` ( соглашение Pascal). `unsafe` ключевое слово необязательно; это уменьшает накладные расходы для простых функций, но может вызвать взаимоблокировки, если посторонняя функция блокирует бесконечно или имеет недостаточное разрешение для выполнения 1 .

## Examples

### Вызов C из Haskell

По соображениям производительности или из-за наличия зрелых библиотек C вы можете вызвать код C из программы Haskell. Вот простой пример того, как вы можете передавать данные в библиотеку C и получать ответ.

foo.c:

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs:

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

`unsafe` ключевое слово генерирует более эффективный вызов, чем «безопасный», но требует, чтобы код C никогда не вызывал обратную связь с системой Haskell. Так как `foo` полностью в C и никогда не вызовет Haskell, мы можем использовать `unsafe` .

Нам также необходимо дать инструкции cabal для компиляции и ссылки в источнике C.

foo.cabal:

```
name:                foo
version:             0.0.0.1
build-type:          Simple
extra-source-files:  *.c
cabal-version:       >= 1.10

executable foo
  default-language: Haskell2010
  main-is:       Foo.hs
  C-sources:     foo.c
  build-depends: base
```

Затем вы можете запустить:

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

## Передача Haskell выполняет функции обратного вызова кода C.

Для функций C очень часто принимают указатели на другие функции в качестве аргументов. Наиболее популярным примером является установка действия, которое будет выполняться при нажатии кнопки в некоторой библиотеке инструментов GUI. Функции Haskell можно передавать как C-обратные вызовы.

Чтобы вызвать эту функцию C:

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

мы сначала импортируем его в код Haskell:

```
foreign import ccall "header.h event_callback_add"
```

```
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

Теперь, посмотрев, как `Object_Event_Cb` определен в заголовке C, определите, что `Callback` находится в Haskell:

```
type Callback = Ptr () -> Ptr () -> IO ()
```

Наконец, создайте специальную функцию, которая будет обертывать функцию Haskell типа `Callback` в указатель `FunPtr Callback` :

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

Теперь мы можем зарегистрировать обратный вызов с кодом C:

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- callback code
    return ()
callbackAdd cpPtr
```

Важно освободить выделенный `FunPtr` только вы `FunPtr` обратный вызов:

```
freeHaskellFunPtr cbPtr
```

Прочитайте Интерфейс внешней функции онлайн: https://riptutorial.com/ru/haskell/topic/7256/интерфейс-внешней-функции

# глава 31: интрига

## Синтаксис

- cabal <command>, где <command> является одним из следующих:
- **[Глобальный]**
  - Обновить
    - Обновляет список известных пакетов
  - устанавливать
    - Установка пакетов
  - Помогите
    - Справка о командах
  - Информация
    - Отобразить подробную информацию о конкретном пакете
  - список
    - Список пакетов, соответствующих строке поиска.
  - получать
    - Пакеты загрузок для последующей установки
  - пользователем конфигурации
    - Отображение и обновление глобальной конфигурации шлюза пользователя
- **[пакет]**
  - получить
    - Загрузить / Извлечь исходный код пакета (репозиторий)
  - в этом
    - Создайте новый файл пакета .cabal (в интерактивном режиме)
  - конфигурировать
    - Подготовьтесь к созданию пакета
  - строить
    - Компилировать все / определенные компоненты
  - чистый
    - Очистка после сборки
  - бежать
    - Создает и запускает исполняемый файл
  - РЕПЛ
    - Откройте сеанс интерпретатора для данного компонента
  - тестовое задание
    - Запуск всех / конкретных тестов в наборе тестов
  - скамейка
    - Запуск всех / конкретных тестов
  - проверять

        - Проверьте пакет на наличие распространенных ошибок
    - sdist
        - Создайте исходный файл рассылки (.tar.gz)
    - загружать
        - Загружает исходные пакеты или документацию в Hackage
    - доклад
        - Загружать отчеты о создании на удаленный сервер
    - замерзать
        - Замораживание
    - генераторных границы
        - Создание ограничений зависимости
    - пикша
        - Создание HTML-документации Haddock
    - hscolour
        - Сгенерировать HsColour colourised code, в формате HTML
    - копия
        - Скопируйте файлы в места установки
    - регистр
        - Зарегистрируйте этот пакет с помощью компилятора
- **[песочница]**
    - песочница
        - Создание / изменение / удаление песочницы
            - cabal sandbox init [FLAGS]
            - cabal sandbox удалить [FLAGS]
            - добавочный источник песочницы cabal [FLAGS] PATHS
            - cabal sandbox delete-source [FLAGS] PATHS
            - list-sources из песочницы cabal [FLAGS]
            - cabal sandbox hc-pkg [FLAGS] [-] COMMAND [-] [ARGS]
    - Exec
        - Предоставить команду доступа к хранилищу пакетов песочницы
    - РЕПЛ
        - Открытый интерпретатор с доступом к пакетам песочницы

# Examples

## Установка пакетов

Чтобы установить новый пакет, например, aeson:

```
cabal install aeson
```

## Работа с песочницами

Проект Haskell может использовать системные пакеты или использовать песочницу. Песочница представляет собой изолированную базу данных пакетов и может предотвращать конфликты зависимостей, например, если несколько проектов Haskell используют разные версии пакета.

Чтобы инициализировать песочницу для пакета Haskell, перейдите в его каталог и запустите:

```
cabal sandbox init
```

Теперь пакеты можно установить, просто выполнив `cabal install`.

Листинг пакетов в песочнице:

```
cabal sandbox hc-pkg list
```

Удаление песочницы:

```
cabal sandbox delete
```

Добавить локальную зависимость:

```
cabal sandbox add-source /path/to/dependency
```

Прочитайте интрига онлайн: https://riptutorial.com/ru/haskell/topic/4740/интрига

# глава 32: Использование GHCi

## замечания

GHCI - это интерактивный REPL, который поставляется вместе с GHC.

## Examples

### Запуск GHCi

Введите `ghci` в командной строке для запуска GHCI.

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

### Изменение приглашения по умолчанию GHCi

По умолчанию приглашение GHCI отображает все модули, которые вы загрузили в свой интерактивный сеанс. Если у вас много загруженных модулей, это может затянуться:

```
Prelude Data.List Control.Monad> -- etc
```

Команда `:set prompt` изменяет приглашение для этого интерактивного сеанса.

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

Чтобы изменить приглашение навсегда, добавьте `:set prompt "foo> "` в файл конфигурации GHCi .

### Файл конфигурации GHCi

GHCi использует файл конфигурации в `~/.ghci` . Конфигурационный файл состоит из последовательности команд, которые GHCi будет выполнять при запуске.

```
$ echo ":set prompt \"foo> \"" > ~/.ghci
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from ~/.ghci
foo>
```

### Загрузка файла

Команда `:l` или `:load` - проверяет и загружает файл.

```
$ echo "f = putStrLn \"example\"" > example.hs
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
ghci> :l example.hs
[1 of 1] Compiling Main               ( example.hs, interpreted )
Ok, modules loaded: Main.
ghci> f
example
```

## Выход из GHCi

Вы можете оставить GHCi просто с помощью `:q` или `:quit`

```
ghci> :q
Leaving GHCi.

ghci> :quit
Leaving GHCi.
```

Альтернативно, сочетание клавиш `CTRL` + `D` ( `Cmd` + `D` для OSX) имеет тот же эффект, что и `:q` .

## Перезагрузка уже загруженного файла

Если вы загрузили файл в GHCi (например, с помощью `:l filename.hs` ), и вы изменили файл в редакторе вне GHCi, вы должны перезагрузить файл с помощью `:r` или `:reload` , чтобы использовать изменения, следовательно вам больше не нужно вводить имя файла.

```
ghci> :r
OK, modules loaded: Main.

ghci> :reload
OK, modules loaded: Main.
```

## Точки останова с GHCi

GHCi поддерживает точки прерывания по требованию из коробки с интерпретированным кодом (код, который был `:loaded` ).

Со следующей программой:

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

загружен в GHCi:

```
Prelude> :load mySum.hs
[1 of 1] Compiling Main             ( mySum.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Теперь мы можем установить точки останова, используя номера строк:

```
*Main> :break 2
Breakpoint 0 activated at mySum.hs:2:3-39
```

и GHCi остановится в соответствующей строке, когда мы запустим функцию:

```
*Main> doSum 12
Stopped at mySum.hs:2:3-39
_result :: IO () = _
n :: Integer = 12
[mySum.hs:2:3-39] *Main>
```

Это может сбивать с толку, где мы находимся в программе, поэтому мы можем использовать `:list` для уточнения:

```
[mySum.hs:2:3-39] *Main> :list
1  doSum n = do
2    putStrLn ("Counting to " ++ (show n))   -- GHCi will emphasise this line, as that's where
we've stopped
3    let v = sum [1..n]
```

Мы можем печатать переменные и продолжать выполнение:

```
[mySum.hs:2:3-39] *Main> n
12
:continue
Counting to 12
sum to 12 = 78
*Main>
```

## Многострочные операторы

Команда `:{` начинает *многострочный режим* и `:}` завершает ее. В многострочном режиме GHCi интерпретирует новые строки как точки с запятой, а не как конец инструкции.

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Прочитайте Использование GHCi онлайн: https://riptutorial.com/ru/haskell/topic/3407/использование-ghci

# глава 33: Контейнеры - Data.Map

## Examples

### строительство

Мы можем создать карту из списка кортежей следующим образом:

```
Map.fromList [("Alex", 31), ("Bob", 22)]
```

Карта также может быть построена с одним значением:

```
> Map.singleton "Alex" 31
fromList [("Alex",31)]
```

Существует также `empty` функция.

```
empty :: Map k a
```

Data.Map также поддерживает типичные операции с множеством, такие как `union` , `difference` и `intersection` .

### Проверка пустого

Мы используем `null` функцию для проверки, является ли данная Карта пустой:

```
> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)]
False

> Map.null $ Map.empty
True
```

### Поиск значений

На картах много запросов.

`member :: Ord k => k -> Map ka -> Bool` дает `True` если ключ типа `k` находится в `Map ka` :

```
> Map.member "Alex" $ Map.singleton "Alex" 31
True
> Map.member "Jenny" $ Map.empty
False
```

`notMember` аналогичен:

```
> Map.notMember "Alex" $ Map.singleton "Alex" 31
False
> Map.notMember "Jenny" $ Map.empty
True
```

Вы также можете использовать `findWithDefault :: Ord k => a -> k -> Map ka -> a` чтобы получить значение по умолчанию, если ключ отсутствует:

```
Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x'
Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'
```

## Вставка элементов

Вставка элементов проста:

```
> let m = Map.singleton "Alex" 31
fromList [("Alex",31)]

> Map.insert "Bob" 99 m
fromList [("Alex",31),("Bob",99)]
```

## Удаление элементов

```
> let m = Map.fromList [("Alex", 31), ("Bob", 99)]
fromList [("Alex",31),("Bob",99)]

> Map.delete "Bob" m
fromList [("Alex",31)]
```

## Импорт модуля

Модуль `Data.Map` в пакете `containers` предоставляет структуру `Map` которая имеет как строгие, так и ленивые реализации.

При использовании `Data.Map` обычно обычно импортируется, чтобы избежать столкновений с функциями, уже определенными в Prelude:

```
import qualified Data.Map as Map
```

Таким образом , мы бы тогда перед именем `Map` вызовов функций с `Map.` , например

```
Map.empty -- give me an empty Map
```

## Экземпляр моноида

`Map kv` предоставляет экземпляр Monoid со следующей семантикой:

- `mempty` - это пустая `Map` , то есть такая же, как `Map.empty`
- `m1 <> m2` - левое смещение объединения `m1` и `m2` , т. е. если какой-либо ключ присутствует как в `m1` и `m2` , то значение из `m1` выбирается для `m1 <> m2` . Эта операция также доступна за пределами `Monoid` , например , как `Map.union` .

Прочитайте Контейнеры - Data.Map онлайн: https://riptutorial.com/ru/haskell/topic/4591/контейнеры---data-map

# глава 34: Кортежи (пары, тройки, ...)

## замечания

- Haskell не поддерживает кортежи с одним компонентом изначально.
- Единицы (написанные `()` ) можно понимать как кортежи с нулевыми компонентами.
- Не существует предопределенных функций для извлечения компонентов кортежей с более чем двумя компонентами. Если вы считаете, что вам нужны такие функции, рассмотрите возможность использования пользовательского типа данных с метками записей вместо типа кортежа. Затем вы можете использовать метки записей в качестве функций для извлечения компонентов.

## Examples

### Построить значения кортежа

Для создания кортежей используйте скобки и запятые. Используйте одну запятую, чтобы создать пару.

```
(1, 2)
```

Используйте больше запятых для создания кортежей с большим количеством компонентов.

```
(1, 2, 3)

(1, 2, 3, 4)
```

Обратите внимание, что также можно объявлять кортежи, используя в их unsugared форме.

```
(,) 1 2     -- equivalent to (1,2)
(,,) 1 2 3  -- equivalent to (1,2,3)
```

Кортежи могут содержать значения разных типов.

```
("answer", 42, '?')
```

Кортежи могут содержать сложные значения, такие как списки или больше кортежей.

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

## Написание типов кортежей

Используйте круглые скобки и запятые для написания типов кортежей. Используйте одну запятую, чтобы написать тип пары.

```
(Int, Int)
```

Используйте больше запятых, чтобы писать типы кортежей с большим количеством компонентов.

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

Кортежи могут содержать значения разных типов.

```
(String, Int, Char)
```

Кортежи могут содержать сложные значения, такие как списки или больше кортежей.

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

## Образец матча по кортежам

Сравнение шаблонов на кортежах использует конструкторы кортежей. Для сопоставления пары, например, мы использовали бы конструктор `(,)` :

```
myFunction1 (a, b) = ...
```

Мы используем больше запятых для соответствия кортежей с большим количеством компонентов:

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

Шаблоны кортежей могут содержать сложные шаблоны, такие как шаблоны списков или другие шаблоны кортежей.

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

## Извлечь компоненты кортежа

Используйте функции `fst` и `snd` (из `Prelude` или `Data.Tuple` ), чтобы извлечь первый и второй компоненты пар.

```
fst (1, 2) -- evaluates to 1

snd (1, 2) -- evaluates to 2
```

Или используйте сопоставление шаблонов.

```
case (1, 2) of (result, _) => result -- evaluates to 1

case (1, 2) of (_, result) => result -- evaluates to 2
```

Согласование шаблонов также работает для кортежей с более чем двумя компонентами.

```
case (1, 2, 3) of (result, _, _) => result -- evaluates to 1

case (1, 2, 3) of (_, result, _) => result -- evaluates to 2

case (1, 2, 3) of (_, _, result) => result -- evaluates to 3
```

Haskell не предоставляет стандартные функции, такие как `fst` или `snd` для кортежей с более чем двумя компонентами. Библиотека `tuple` в Hackage предоставляет такие функции в модуле `Data.Tuple.Select` .

## Примените двоичную функцию к кортежу (неуправляемому)

Используйте функцию `uncurry` (из `Prelude` или `Data.Tuple` ), чтобы преобразовать двоичную функцию в функцию на кортежах.

```
uncurry (+) (1, 2) -- computes 3

uncurry map (negate, [1, 2, 3]) -- computes [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- computes 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- computes [3, 7, 11]

uncurry (curry f) -- computes the same as f
```

## Примените функцию кортежа к двум аргументам (currying)

Используйте функцию `curry` (из `Prelude` или `Data.Tuple` ), чтобы преобразовать функцию, которая берет кортежи для функции, которая принимает два аргумента.

```
curry fst 1 2 -- computes 1

curry snd 1 2 -- computes 2

curry (uncurry f) -- computes the same as f
```

```
import Data.Tuple (swap)
curry swap 1 2 -- computes (2, 1)
```

## Компоненты своп-пары

Используйте `swap` (из `Data.Tuple` ) для замены компонентов пары.

```
import Data.Tuple (swap)
swap (1, 2) -- evaluates to (2, 1)
```

Или используйте сопоставление шаблонов.

```
case (1, 2) of (x, y) => (y, x) -- evaluates to (2, 1)
```

## Строгость соответствия кортежа

Шаблон `(p1, p2)` строгий в самом внешнем конструкторе кортежа, что может привести к неожиданному поведению строгости . Например, следующее выражение расходится (используя `Data.Function.fix` ):

```
fix $ \(x, y) -> (1, 2)
```

так как совпадение по `(x, y)` строго в конструкторе кортежа. Однако следующее выражение, использующее неопровержимый шаблон , оценивает `(1, 2)` как и ожидалось:

```
fix $ \ ~(x, y) -> (1, 2)
```

Прочитайте Кортежи (пары, тройки, ...) онлайн: https://riptutorial.com/ru/haskell/topic/5342/кортежи--пары--тройки------

# глава 35: логирование

## Вступление

Регистрация в Haskell достигается обычно через функции в монаде `IO` и поэтому ограничена нечистыми функциями или «действиями IO».

Существует несколько способов записи информации в программу Haskell: из `putStrLn` (или `print` ) в библиотеки, такие как hslogger или через `Debug.Trace` .

## Examples

### Регистрация с помощью hslogger

`hslogger` модуль обеспечивает аналогичный API для Питона `logging` базы, и поддерживает иерархически имени лесорубов, уровни и перенаправление на ручках вне `stdout` и `stderr` .

По умолчанию все сообщения уровня `WARNING` и выше отправляются в stderr, и все остальные уровни журналов игнорируются.

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                     updateGlobalLogger, warningM)

main = do
  debugM "MyProgram.main" "This won't be seen"
  infoM "MyProgram.main" "This won't be seen either"
  warningM "MyProgram.main" "This will be seen"
```

Мы можем установить уровень регистратора по его имени с помощью `updateGlobalLogger` :

```
  updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "This will now be seen"
```

У каждого регистратора есть имя, и они расположены иерархически, поэтому `MyProgram` является родителем `MyParent.Module` .

Прочитайте логирование онлайн: https://riptutorial.com/ru/haskell/topic/9628/логирование

# глава 36: Модули

## Синтаксис

- module Name где - экспортировать все имена, объявленные в этом файле
- module name (functionOne, Type (..)) где - экспортировать только функции One, Type и Type
- import Module - импортировать все экспортированные имена модуля
- импортировать квалифицированный модуль как MN - квалифицированный импорт
- import module (justThisFunction) - импортировать только определенные имена из модуля
- import Module hiding (functionName, Type) - импортировать все имена из модуля, кроме функцииName и Type

## замечания

Haskell поддерживает модули:

- модуль может экспортировать все или подмножество его типов и функций
- модуль может «реэкспортировать» имена, импортированные из других модулей

На потребительском конце модуля можно:

- импортировать все или подмножество членов модуля
- скрыть импорт определенного члена или набора членов

У haskell.org есть отличная глава по определению модуля.

## Examples

### Определение собственного модуля

Если у нас есть файл под названием `Business.hs` , мы можем определить `Business` модуль, который можно `import` , например:

```
module Business (
    Person (..), -- ^ Export the Person type and all its constructors and field names
    employees    -- ^ Export the employees function
```

```
) where
-- begin types, function definitions, etc
```

Возможно, более глубокая иерархия; см. пример Иерархический список модулей .

## Экспорт конструкторов

Чтобы экспортировать тип и все его конструкторы, необходимо использовать следующий синтаксис:

```
module X (Person (..)) where
```

Итак, для следующих определений верхнего уровня в файле `People.hs` :

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

Это объявление модуля вверху:

```
module People (Person (..)) where
```

будет экспортировать только `Person` и его конструкторов `Friend` and `Foe` .

Если список экспорта, следующий за ключевым словом модуля, опущен, все имена, привязанные к верхнему уровню модуля, будут экспортированы:

```
module People where
```

будет экспортировать `Person` , его конструкторы и функцию `isFoe` .

## Импорт отдельных членов модуля

Haskell поддерживает импорт подмножества элементов из модуля.

```
import qualified Data.Stream (map) as D
```

будет импортировать `map` только из `Data.Stream` , и для вызова этой функции потребуется `D.` .:

```
D.map odd [1..]
```

иначе компилятор попытается использовать функцию `map` `Prelude` .

## Скрыть импорт

Prelude часто определяет функции, имена которых используются в другом месте. Не скрывая такого импорта (или используя квалифицированный импорт там, где происходят столкновения) вызовет ошибки компиляции.

`Data.Stream` определяет функции с именем `map`, `head` и `tail` которые обычно сталкиваются с теми, которые определены в Prelude. Мы можем скрыть импорт из Prelude, используя `hiding` :

```
import Data.Stream -- everything from Data.Stream
import Prelude hiding (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- etc
```

На самом деле для этого потребуется слишком много кода, чтобы скрыть конфликты Prelude, подобные этому, поэтому вместо этого вы бы использовали `qualified` импорт `Data.Stream`.

## Квалификационный импорт

Когда несколько модулей определяют одни и те же функции по имени, компилятор будет жаловаться. В таких случаях (или для повышения удобочитаемости) мы можем использовать `qualified` импорт:

```
import qualified Data.Stream as D
```

Теперь мы можем предотвратить ошибки компилятора неоднозначности, когда мы используем `map`, которая определена в `Prelude` и `Data.Stream`:

```
map (== 1) [1,2,3] -- will use Prelude.map
D.map (odd) (fromList [1..]) -- will use Data.Stream.map
```

Также возможно импортировать модуль с только идентифицирующими именами, которые можно получить с помощью `import Data.Text as T`, что позволяет иметь `Text` вместо `T.Text` и т. Д.

## Иерархические имена модулей

Имена модулей соответствуют иерархической структуре файловой системы. Со следующим макетом файла:

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
Bar.hs
```

заголовки модулей будут выглядеть так:

```
-- file Foo.hs
module Foo where

-- file Bar.hs
module Bar where

-- file Foo/Bar.hs
module Foo.Bar where

-- file Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

Обратите внимание, что:

- имя модуля основано на пути файла, объявляющего модуль
- Папки могут совместно использовать имя с модулем, который дает иерархическую структуру именования модулей

Прочитайте Модули онлайн: https://riptutorial.com/ru/haskell/topic/5234/модули

# глава 37: Монады

## Вступление

Монада - это тип данных составных действий. `Monad` - это класс конструкторов типов, значения которых представляют такие действия. Возможно , `IO` является самым узнаваемым один: значение `IO a` является «рецептом для получения `a` значении от реального мира».

Мы говорим, что конструктор типа `m` (такой как `[]` или `Maybe` ) *образует монаду,* если есть `instance Monad m` удовлетворяющий некоторым законам о составе действий. Тогда мы можем рассуждать о `ma` как «действие, результат которого имеет тип `a` ».

## Examples

### Возможно, монада

`Maybe` , используется для представления возможных пустых значений - аналогично `null` на других языках. Обычно он используется как выходной тип функций, которые могут каким-то образом сбой.

Рассмотрим следующую функцию:

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

Подумайте о `halve` как действия, в зависимости от `Int` , который пытается вдвое уменьшить целое число, если нечетно.

Как мы `halve` число целых чисел?

```
takeOneEighth :: Int -> Maybe Int            -- (after you read the 'do' sub-section:)
takeOneEighth x =
  case halve x of                              --  do {
    Nothing -> Nothing
    Just oneHalf ->                            --     oneHalf    <- halve x
      case halve oneHalf of
        Nothing -> Nothing
        Just oneQuarter ->                     --     oneQuarter <- halve oneHalf
          case halve oneQuarter of
            Nothing -> Nothing                 --     oneEighth  <- halve oneQuarter
            Just oneEighth ->
              Just oneEighth                   --     return oneEighth }
```

- `takeOneEighth` - это *последовательность* из трех `halve` шагов, соединенных вместе.

- Если шаг на `halve` не удался, мы хотим, чтобы вся композиция `takeOneEighth` потерпела неудачу.
- Если шаг с `halve` успеха завершен, мы хотим передать его результат вперед.

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                            -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                         -- also, f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

и теперь мы можем написать:

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve             -- or,
    -- return x >>= halve >>= halve >>= halve             -- which is parsed as
    -- (((return x) >>= halve) >>= halve) >>= halve       -- which can also be written as
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x    -- or, equivalently, as
    --  halve <=<     halve <=<     halve      $        x
```

*Состав Клейсли* `<=<` определяется как `(g <=< f) x = g =<< fx` или эквивалентно как `(f >=> g) x = fx >>= g` . При этом указанное выше определение становится

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve               -- infixr 1 <=<
        -- or, equivalently,
        --      halve >=> halve >=> halve               -- infixr 1 >=>
```

---

Существует три закона монады, которые должны соблюдаться каждой монадой, то есть каждый тип, который является экземпляром класса `Monad` :

```
1.  return x >>= f  =  f x
2.    m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

где `m` - монада, `f` имеет тип `a -> mb` и `g` имеет тип `b -> mc` .

Или, что то же самое, используя оператор состава `>=>` Клейсли, определенный выше:

```
1.    return >=> g  =  g                     -- do { y <- return x ; g y } == g x
2.    f >=> return  =  f                     -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h  =  f >=> (g >=> h)       -- do { z <- do { y <- f x; g y } ; h z }
                                             --  == do { y <- f x ; do { z <- g y; h z } }
```

Соблюдение этих законов значительно облегчает рассуждение о монаде, потому что оно гарантирует, что использование монадических функций и составление их ведет себя разумно, подобно другим монадам.

Давайте проверим, `Maybe` монада `Maybe` подчиняться трем законам монады.

1. **Закон левого тождества** - `return x >>= f = fx`

```
return z >>= f
= (Just z) >>= f
= f z
```

2. **Правильный закон тождества** - `m >>= return = m`

- `Just` конструктор данных

```
Just z >>= return
= return z
= Just z
```

- `Nothing` конструктора данных

```
Nothing >>= return
= Nothing
```

3. **Закон ассоциативности** - `(m >>= f) >>= g = m >>= (\x -> fx >>= g)`

- `Just` конструктор данных

```
-- Left-hand side
((Just z) >>= f) >>= g
= f z >>= g

-- Right-hand side
(Just z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- `Nothing` конструктора данных

```
-- Left-hand side
(Nothing >>= f) >>= g
= Nothing >>= g
= Nothing

-- Right-hand side
Nothing >>= (\x -> f x >>= g)
= Nothing
```

## IO monad

Невозможно получить значение типа `a` из выражения типа `IO a` и его не должно быть. На самом деле это большая часть того, почему монады используются для моделирования `IO` .

Выражение типа `IO a` можно представить как представляющее действие, которое может взаимодействовать с реальным миром и, если оно выполняется, приведет к чему-то типа `a` . Например, функция `getLine :: IO String` из прелюдии не означает, что под `getLine` есть

определенная строка, которую я могу извлечь - это означает, что `getLine` представляет действие получения строки со стандартного ввода.

Неудивительно, что `main :: IO ()` поскольку программа Haskell представляет собой вычисление / действие, которое взаимодействует с реальным миром.

То, что вы *можете* сделать с выражениями типа `IO a` потому что `IO` является монадой:

- Последовательность двух действий с использованием `(>>)` для создания нового действия, выполняющего первое действие, отбрасывает любое значение, которое оно произвело, а затем выполняет второе действие.

```
-- print the lines "Hello" then "World" to stdout
putStrLn "Hello" >> putStrLn "World"
```

- Иногда вы не хотите отбрасывать значение, которое было создано в первом действии, - вам действительно хотелось бы, чтобы его подавали во второе действие. Для этого имеем `>>=` . Для `IO` он имеет тип `(>>=) :: IO a -> (a -> IO b) -> IO b` .

```
-- get a line from stdin and print it back out
getLine >>= putStrLn
```

- Возьмите нормальное значение и преобразуйте его в действие, которое сразу же возвращает значение, которое вы ему дали. Эта функция менее очевидно , полезно , пока вы не начнете использовать `do` запись.

```
-- make an action that just returns 5
return 5
```

Больше из Haskell Wiki на монаде IO здесь .

## Список Монад

Списки образуют монаду. У них есть экземпляр монады, эквивалентный этому:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

Мы можем использовать их для подражания недетерминированности в наших вычислениях. Когда мы используем `xs >>= f` , функция `f :: a -> [b]` отображается в списке `xs` , получая список списков результатов каждого применения `f` по каждому элементу `xs` и всех списков результаты затем объединяются в один список всех результатов. В качестве примера мы вычисляем сумму двух недетерминированных чисел, используя **do-notation** , причем сумма представлена списком сумм всех пар целых чисел из двух списков, каждый из которых представляет все возможные значения недетерминированного числа:

```
sumnd xs ys = do
  x <- xs
  y <- ys
  return (x + y)
```

Или, что то же самое, используя `liftM2` в `Control.Monad` :

```
sumnd = liftM2 (+)
```

мы получаем:

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

## Монад как подкласс прикладной

Начиная с GHC 7.10, `Applicative` является суперклассом `Monad` (т. `Monad` Каждый тип, который является `Monad` также должен быть `Applicative` ). Все методы `Applicative` ( `pure` , `<*>` ) могут быть реализованы с точки зрения методов `Monad` ( `return` , `>>=` ).

Очевидно, что `pure` и `return` служат эквивалентным целям, поэтому `pure = return` . Определение для `<*>` слишком относительно ясно:

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
       -- = mf >>= (\f -> mx >>= (\x -> return (f x)))
       -- = [r   | f <- mf, x <- mx, r <- return (f x)]   -- with MonadComprehensions
       -- = [f x | f <- mf, x <- mx]
```

Эта функция определяется как `ap` в стандартных библиотеках.

Таким образом, если вы уже определили экземпляр `Monad` для типа, вы можете получить экземпляр `Applicative` для него «бесплатно», указав

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

Как и в случае с монадскими законами, эти эквивалентности не применяются, но разработчики должны следить за тем, чтобы они всегда поддерживались.

## Нет общего способа извлечь значение из монадического вычисления

Вы можете переносить значения в действия и обрабатывать результат одного вычисления в другом:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

Однако определение Монады не гарантирует существования функции типа `Monad m => ma -> a` .

Это означает, что вообще **нет способа извлечь значение из вычисления** (т. Е. «Развернуть» его). Это касается многих случаев:

```
extract :: Maybe a -> a
extract (Just x) = x          -- Sure, this works, but...
extract Nothing  = undefined  -- We can’t extract a value from failure.
```

В частности, нет функции `IO a -> a` , которая часто путает новичков; см. этот пример .

## делать-нотация

`do` -notation - синтаксический сахар для монад. Вот правила:

```
do x <- mx                               do x <- mx
   y <- my       is equivalent to           do y <- my
   ...                                         ...
```

```
do let a = b                             let a = b in
   ...           is equivalent to          do ...
```

```
do m                                     m >> (
   e             is equivalent to          e)
```

```
do x <- m                                m >>= (\x ->
   e             is equivalent to          e)
```

```
do m             is equivalent to        m
```

Например, эти определения эквивалентны:

```
example :: IO Integer
example =
  putStrLn "What's your name?" >> (
    getLine >>= (\name ->
      putStrLn ("Hello, " ++ name ++ ".") >> (
        putStrLn "What should we return?" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

---

```
example :: IO Integer
example = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ".")
  putStrLn "What should we return?"
  line <- getLine
```

```
  let n = (read line :: Integer)
  return (n + n)
```

## Определение Монады

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Наиболее важной функцией для работы с монадами является **оператор связывания `>>=`** :

```
(>>=) :: m a -> (a -> m b) -> m b
```

- Подумайте о `ma` , как *«действия с `a` результате».*
- Подумайте `a -> mb` *«действия (* в *зависимости от* как `a` *параметре) с `b` . Результат».*

`>>=` **последовательно выполняет два действия, связывая результат от первого действия ко второму.**

Другая функция, определенная `Monad` :

```
return :: a -> m a
```

Его имя несчастливо: это `return` имеет ничего общего с ключевым словом `return` найденным на императивных языках программирования.

`return x` **- это тривиальное действие, приводящее к результату `x` .** (Это тривиально в следующем смысле :)

```
return x >>= f        ≡  f x      --  "left identity" monad law
       x >>= return   ≡  x        -- "right identity" monad law
```

Прочитайте Монады онлайн: https://riptutorial.com/ru/haskell/topic/2968/монады

# глава 38: Обобщенные типы алгебраических данных

## Examples

### Основное использование

Когда расширение `GADTs` включено, помимо регулярных объявлений данных, вы также можете объявить обобщенные алгебраические типы данных следующим образом:

```
data DataType a where
    Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
    Constr3 :: DataType Int
```

В объявлении GADT перечислены типы всех конструкторов, тип данных которых явно указан. В отличие от обычных деклараций типов данных тип конструктора может быть любой N-арной (включая нулевую) функцию, которая в конечном итоге приводит к типу данных, применяемому к некоторым аргументам.

В этом случае мы объявили, что тип `DataType` имеет три конструктора: `Constr1` , `Constr2` и `Constr3` .

Конструктор `Constr1` не отличается от объявленного с использованием регулярного объявления `data DataType a = Constr1 Int a (Foo a) | ...` : `data DataType a = Constr1 Int a (Foo a) | ...`

Однако `Constr2` требует, чтобы `a` имел экземпляр `Show` , и поэтому при использовании конструктора экземпляр должен существовать. С другой стороны, при сопоставлении шаблонов на нем появляется тот факт, что `a` является экземпляром `Show` , поэтому вы можете написать:

```
foo :: DataType a -> String
foo val = case val of
    Constr2 x -> show x
    ...
```

Обратите внимание, что параметр `Show a` ограничение не отображается в типе функции и отображается только в коде справа от `->` .

`Constr3` имеет тип `DataType Int` , что означает, что всякий раз, когда значение типа `DataType a` является `Constr3` , известно, что `a ~ Int` . Эта информация также может быть восстановлена с совпадением шаблонов.

Прочитайте Обобщенные типы алгебраических данных онлайн:
https://riptutorial.com/ru/haskell/topic/2971/обобщенные-типы-алгебраических-данных

# глава 39: Общие монады как свободные монады

## Examples

### Бесплатная пустая ~~ Identity

Дано

```
data Empty a
```

у нас есть

```
data Free Empty a
     = Pure a
-- the Free constructor is impossible!
```

которая изоморфна

```
data Identity a
     = Identity a
```

### Free Identity ~~ (Nat,) ~~ Writer Nat

Дано

```
data Identity a = Identity a
```

у нас есть

```
data Free Identity a
     = Pure a
     | Free (Identity (Free Identity a))
```

которая изоморфна

```
data Deferred a
     = Now a
     | Later (Deferred a)
```

или эквивалентно (если вы обещаете сначала оценить первый элемент) `(Nat, a)` , aka `Writer Nat a` , с

```
data Nat = Z | S Nat
```

```
data Writer Nat a = Writer Nat a
```

## Бесплатно Может быть ~~ MaybeT (Writer Nat)

Дано

```
data Maybe a = Just a
             | Nothing
```

у нас есть

```
data Free Maybe a
     = Pure a
     | Free (Just (Free Maybe a))
     | Free Nothing
```

что эквивалентно

```
data Hopes a
     = Confirmed a
     | Possible (Hopes a)
     | Failed
```

или эквивалентно (если вы обещаете сначала оценить первый элемент) `(Nat, Maybe a)` , aka `MaybeT (Writer Nat) a` с

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

## Free (Writer w) ~~ Writer [w]

Дано

```
data Writer w a = Writer w a
```

у нас есть

```
data Free (Writer w) a
     = Pure a
     | Free (Writer w (Free (Writer w) a))
```

которая изоморфна

```
data ProgLog w a
     = Done a
     | After w (ProgLog w a)
```

или, что то же самое, (если вы обещаете сначала оценить журнал), `Writer [w] a` .

## Free (Const c) ~~ Либо c

Дано

```
data Const c a = Const c
```

у нас есть

```
data Free (Const c) a
     = Pure a
     | Free (Const c)
```

которая изоморфна

```
data Either c a
     = Right a
     | Left c
```

## Free (Reader x) ~~ Reader (поток x)

Дано

```
data Reader x a = Reader (x -> a)
```

у нас есть

```
data Free (Reader x) a
     = Pure a
     | Free (x -> Free (Reader x) a)
```

которая изоморфна

```
data Demand x a
     = Satisfied a
     | Hungry (x -> Demand x a)
```

или эквивалентно `Stream x -> a` с

```
data Stream x = Stream x (Stream x)
```

Прочитайте Общие монады как свободные монады онлайн:
https://riptutorial.com/ru/haskell/topic/8256/общие-монады-как-свободные-монады

# глава 40: Общие расширения языка GHC

## замечания

Эти языковые расширения обычно доступны при использовании компилятора Glasgow Haskell (GHC), поскольку они не являются частью утвержденного отчета о языке Haskell 2010 . Чтобы использовать эти расширения, нужно либо проинформировать компилятор, используя флаг, либо поместить программу `LANGUAGE` перед ключевым словом `module` в файле. Официальную документацию можно найти в разделе 7 руководства пользователя GCH.

Формат программы `LANGUAGE` - `{-# LANGUAGE ExtensionOne, ExtensionTwo ... #-}` . Это буквальный `{-#` за которым следует `LANGUAGE` за которым следует список расширений, разделенных запятыми, и, наконец, закрытие `#-}` . Несколько программ `LANGUAGE` могут быть помещены в один файл.

# Examples

### MultiParamTypeClasses

Это очень распространенное расширение, которое позволяет использовать классы типов с несколькими параметрами типа. Вы можете думать о MPTC как о связи между типами.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
    convert i = fromIntegral i
```

Порядок параметров имеет значение.

Иногда MPTC можно заменить семействами типов.

### FlexibleInstances

Регулярные экземпляры требуют:

```
All instance types must be of the form (T a1 ... an)
where a1 ... an are *distinct type variables*,
and each type variable appears at most once in the instance head.
```

Это означает, что, например, пока вы можете создать экземпляр для `[a]` вы не можете создать экземпляр специально для `[Int]` .; `FlexibleInstances` расслабляет:

```
class C a where

-- works out of the box
instance C [a] where

-- requires FlexibleInstances
instance C [Int] where
```

## OverloadedStrings

Обычно строковые литералы в Haskell имеют тип `String` (который является псевдонимом типа для `[Char]` ). Хотя это не проблема для небольших образовательных программ, в реальных приложениях часто требуется более эффективное хранилище, например `Text` или `ByteString` .

`OverloadedStrings` просто меняет тип литералов на

```
"test" :: Data.String.IsString a => a
```

Предоставление им прямого доступа к функциям, ожидающим такого типа. Многие библиотеки реализуют этот интерфейс для своих строковых типов, включая Data.Text и Data.ByteString, которые обеспечивают определенные преимущества по времени и пространству по сравнению с `[Char]` .

Есть также некоторые уникальные применения `OverloadedStrings` подобные тем, что из простой библиотеки Postgresql, которая позволяет SQL-запросам записываться в двойных кавычках, таких как обычная строка, но обеспечивает защиту от неправильной конкатенации, пресловутого источника атак SQL-инъекций.

Чтобы создать экземпляр класса `IsString` вам необходимо внедрить функцию `fromString` . Пример †:

```
data Foo = A | B | Other String deriving Show

instance IsString Foo where
  fromString "A" = A
  fromString "B" = B
  fromString xs  = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

---

† Этот пример любезно Линдона Maydwell ( `sordina` на GitHub) нашел здесь .

## TupleSections

Синтаксическое расширение, которое позволяет применять конструктор кортежа (который является оператором) в виде раздела:

```
(a,b) == (,) a b

-- With TupleSections
(a,b) == (,) a b == (a,) b == (,b) a
```

# N-кортежи

Он также работает для кортежей с арностью более двух

```
(,2,) 1 3 == (1,2,3)
```

# картографирование

Это может быть полезно в других местах, где используются разделы:

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

Вышеприведенный пример без этого расширения будет выглядеть так:

```
map (\a -> (a, "tag")) [1,2,3]
```

## UnicodeSyntax

Расширение, позволяющее использовать символы Unicode вместо определенных встроенных операторов и имен.

| ASCII | Unicode | Используйте (s) |
|---|---|---|
| `::` | `∷` | имеет тип |
| `->` | `→` | типы функций, лямбда, ветви `case` и т. д. |
| `=>` | `⇒` | ограничения класса |
| `forall` | `∀` | явный полиморфизм |
| `<-` | `←` | `do` запись |
| `*` | `★` | тип (или вид) типов (например, `Int :: ★`) |
| `>-` | `□` | `proc` для `Arrows` |
| `-<` | `□` | `proc` для `Arrows` |

| ASCII | Unicode | Используйте (s) |
|---|---|---|
| `>>-` | ▯ | `proc` для `Arrows` |
| `-<<` | ▯ | `proc` для `Arrows` |

Например:

```
runST :: (forall s. ST s a) -> a
```

станет

```
runST :: (∀ s. ST s a) → a
```

---

Обратите внимание, что пример `*` vs. `★` немного отличается: поскольку `*` не зарезервирован, `★` также работает так же, как `*` для умножения или любой другой функции с именем `(*)` и наоборот. Например:

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

## BinaryLiterals

Стандартный Haskell позволяет записывать целочисленные литералы в десятичных (без префикса), шестнадцатеричные (с предшествующими `0x` или `0X` ) и восьмеричные (с предшествующими `0o` или `0O` ). Расширение `BinaryLiterals` добавляет возможность двоичного кода (которому предшествуют `0b` или `0B` ).

```
0b1111 == 15     -- evaluates to: True
```

## ExistentialQuantification

Это системное расширение типа, которое позволяет типы, которые экзистенциально квантуются, или, другими словами, иметь переменные типа, которые получают только экземпляр во время выполнения † .

Значение экзистенциального типа похоже на ссылку на абстрактный базовый класс в языках ОО: вы не знаете точный тип в составе, но можете ограничить *класс* типов.

```
data S = forall a. Show a => S a
```

или эквивалентно, с синтаксисом GADT:

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

Экзистенциальные типы открывают дверь для вещей, таких как почти гетерогенные контейнеры: как сказано выше, на самом деле могут быть разные типы в значении `S` , но все они могут быть `show` n, следовательно, вы также можете сделать

```
instance Show S where
    show (S a) = show a    -- we rely on (Show a) from the above
```

Теперь мы можем создать коллекцию таких объектов:

```
ss = [S 5, S "test", S 3.0]
```

Что также позволяет использовать полиморфное поведение:

```
mapM_ print ss
```

Экзистенциалы могут быть очень мощными, но обратите внимание, что они на самом деле не нужны очень часто в Haskell. В приведенном выше примере все, что вы действительно можете сделать с экземпляром `Show` - это показать (duh!) Значения, т. Е. Создать строковое представление. Таким образом, весь `S` тип содержит точно столько же информации, сколько строка, которую вы получаете при ее показе. Поэтому обычно лучше просто сохранить эту строку сразу, тем более, что Haskell ленив, и поэтому строка вначале будет только неоценимым thunk.

С другой стороны, экзистенции вызывают некоторые уникальные проблемы. Например, способ, которым информация типа «скрыта» в экзистенциальном. Если вы сопоставляете шаблон по значению `S` , у вас будет скрытый тип в области видимости (точнее, его экземпляр `Show` ), но эта информация никогда не сможет выйти из сферы действия, поэтому становится немного «секретным обществом»: компилятор не позволяет *чему-либо* избежать рамки, кроме значений, тип которых уже известен извне. Это может привести к появлению странных ошибок, таких как `Couldn't match type 'a0' with '()' 'a0' is untouchable` .

---

† Контрастируйте это с обычным параметрическим полиморфизмом, который обычно разрешается во время компиляции (разрешая полное стирание типа).

---

Экзистенциальные типы отличаются от типов Rank-N - эти расширения, грубо говоря, двойственны друг к другу: чтобы фактически использовать значения экзистенциального типа, вам нужна (возможно, ограниченная) полиморфная функция, как `show` в примере. Полиморфная функция *универсально* квантифицирована, т. Е. Работает *для любого* типа в данном классе, тогда как экзистенциальная квантификация означает, что она работает

*для* определенного типа, который является априорным неизвестным. Если у вас есть полиморфная функция, этого достаточно, однако, чтобы передавать полиморфные функции, такие как аргументы, вам нужно `{-# LANGUAGE Rank2Types #-}` :

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

## LambdaCase

Синтаксическое расширение, которое позволяет вам писать `\case` вместо `\arg -> case arg of` .

Рассмотрим следующее определение функции:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

Если вы хотите избежать повторения имени функции, вы можете написать что-то вроде:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

Используя расширение LambdaCase, вы можете написать это как выражение функции, не указывая аргумент:

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

## RankNTypes

Представьте себе следующую ситуацию:

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

Здесь мы хотим передать функцию, которая преобразует значение в строку, применить эту функцию как к параметру строки, так и к параметру int и распечатать их оба. На мой взгляд, нет причин, по которым это должно потерпеть неудачу! У нас есть функция, которая работает с обоими типами параметров, которые мы передаем.

К сожалению, это не будет проверять! GHC выводит `a` тип , основанный от его первого появления в теле функции. То есть, как только мы ударим:

```
putStrLn (show' string)
```

GHC выведет, что `show' :: String -> String` , так как `string` является `String` . Он будет продолжать взрываться, пытаясь `show' int` .

`RankNTypes` позволяет вместо этого писать сигнатуру типа следующим образом, количественно определяя все функции, которые удовлетворяют типу `show'` :

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

Это ранг 2 полиморфизм: Мы утверждаем , что `show'` функция должна работать для всех `a` е *в* нашей функции, а предыдущая реализация в настоящее время работает.

Расширение `RankNTypes` позволяет произвольно `RankNTypes` блоки `forall ...` в сигнатуры типов. Другими словами, он допускает полиморфизм ранга N.

## OverloadedLists

*добавлено в GHC 7.8* .

OverloadedLists, аналогичный OverloadedStrings , позволяет выводить литералы списков следующим образом:

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

Это удобно при работе с такими типами, как `Set` , `Vector` и `Map` s.

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

Класс `IsList` в `GHC.Exts` предназначен для использования с этим расширением.

`IsList` оснащен одной функцией типа, `Item` и тремя функциями: `fromList :: [Item l] -> l` , `toList :: l -> [Item l]` и `fromListN :: Int -> [Item l] -> l` где `fromListN` является обязательным. Типичными реализациями являются:

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
  toList   = Set.toList
```

*Примеры, взятые из OverloadedLists - GHC* .

## FunctionalDependencies

Если у вас есть многопараметрический тип-класс с аргументами a, b, c и x, это расширение позволяет вам выразить, что тип x может быть однозначно идентифицирован из a, b и c:

```
class SomeClass a b c x | a b c -> x where ...
```

При объявлении экземпляра такого класса он будет проверяться на всех других экземплярах, чтобы убедиться, что функциональная зависимость выполняется, то есть не существует другого экземпляра с тем же `abc` но отличается `x` .

Вы можете указать несколько зависимостей в списке, разделенном запятыми:

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

Например, в MTL мы видим:

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

Теперь, если у вас есть значение типа `MonadReader a ((->) Foo) => a` , компилятор может сделать вывод, что `a ~ Foo` , так как второй аргумент полностью определяет первый и упростит тип соответственно.

Класс `SomeClass` можно рассматривать как функцию аргументов `abc` что приводит к `x` . Такие классы могут использоваться для выполнения вычислений в системе типов.

## GADTs

Обычные алгебраические типы данных являются параметрическими в своих переменных

типа. Например, если мы определим ADT как

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

с надеждой, что это будет статически исключать не-типизированные условные `IntLit :: Int -> Expr a` , это не будет вести себя так, как ожидалось, так как тип `IntLit :: Int -> Expr a` : для *любого* выбора `a` он производит значение типа `Expr a` . В частности, для `a ~ Bool` у нас есть `IntLit :: Int -> Expr Bool` , что позволяет нам построить что-то вроде `If (IntLit 1) e1 e2` что и пыталось исключить тип конструктора `If` .

Обобщенные алгебраические типы данных позволяют нам управлять результирующим типом конструктора данных, чтобы они были не просто параметрическими. Мы можем переписать наш тип `Expr` как GADT следующим образом:

```
data Expr a where
  IntLit :: Int -> Expr Int
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Здесь тип конструктора `IntLit` - `Int -> Expr Int` , и поэтому `IntLit 1 :: Expr Bool` не будет проверяться typecheck.

Согласование шаблонов по значению GADT вызывает уточнение типа возвращаемого термина. Например, можно написать оценщик для `Expr a` следующим образом:

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- Here we can use `(+)` because x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- Here we can use `not` because b :: Bool
   not b
crazyEval (If b thn els) =
  -- Because b :: Expr Bool, we can use `crazyEval b :: Bool`.
  -- Also, because thn :: Expr a and els :: Expr a, we can pass either to
  -- the recursive call to `crazyEval` and get an a back
  crazyEval $ if crazyEval b then thn else els
```

Обратите внимание, что мы можем использовать `(+)` в приведенных выше определениях, потому что, когда, например, `IntLit x` соответствует шаблону, мы также узнаем, что `a ~ Int` (а также для `not` и `if_then_else_` когда `a ~ Bool` ).

## ScopedTypeVariables

`ScopedTypeVariables` позволяет ссылаться на универсально квантифицированные типы внутри декларации. Чтобы быть более явным:

```
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

Важно то, что мы можем использовать `a` , `b` и `c` чтобы проинструктировать компилятор в подвыражениях объявления (кортеж в предложении `where` и первый `a` в конечном результате). На практике `ScopedTypeVariables` помогает записывать сложные функции в виде суммы частей, позволяя программисту добавлять сигнатуры типов к промежуточным значениям, которые не имеют конкретных типов.

## PatternSynonyms

Синонимы шаблонов - это абстракции шаблонов, аналогичные тем, как функции являются абстракциями выражений.

В этом примере давайте посмотрим на интерфейс `Data.Sequence` , и давайте посмотрим, как его можно улучшить с помощью синонимов шаблонов. Тип `Seq` - это тип данных, который внутренне использует сложное представление для достижения хорошей асимптотической сложности для различных операций, в первую очередь как O (1) (un) consing, так и (un) snocing.

Но это представление громоздко и некоторые его инварианты не могут быть выражены в системе типов Хаскелла. Из-за этого тип `Seq` подвергается воздействию пользователей как абстрактного типа, наряду с сохраняющими инварианты функциями доступа и конструкторами, среди которых:

```
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

Но использование этого интерфейса может быть немного громоздким:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
    x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

Мы можем использовать шаблоны просмотра, чтобы немного почистить его:

```
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
```

```
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

Используя `PatternSynonyms` языка `PatternSynonyms` , мы можем дать еще более приятный интерфейс, позволяя сопоставлять шаблоны, чтобы притворяться, что у нас есть cons- или snoc-list:

```
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

Это позволяет нам писать `uncons` в очень естественном стиле:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

## RecordWildCards

См. RecordWildCards

Прочитайте Общие расширения языка GHC онлайн:
https://riptutorial.com/ru/haskell/topic/1274/общие-расширения-языка-ghc

# глава 41: Общие функторы как основа cofree comonads

## Examples

### Cofree Empty ~~ Empty

Дано

```
data Empty a
```

у нас есть

```
data Cofree Empty a
   --  = a :< ...  not possible!
```

### Cofree (Const c) ~~ Writer c

Дано

```
data Const c a = Const c
```

у нас есть

```
data Cofree (Const c) a
     = a :< Const c
```

которая изоморфна

```
data Writer c a = Writer c a
```

### Cofree Identity ~~ Stream

Дано

```
data Identity a = Identity a
```

у нас есть

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

которая изоморфна

```
data Stream a = Stream a (Stream a)
```

## Cofree Может быть ~~ NonEmpty

Дано

```
data Maybe a = Just a
             | Nothing
```

у нас есть

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
     | a :< Nothing
```

которая изоморфна

```
data NonEmpty a
     = NECons a (NonEmpty a)
     | NESingle a
```

## Cofree (Writer w) ~~ WriterT w Stream

Дано

```
data Writer w a = Writer w a
```

у нас есть

```
data Cofree (Writer w) a
     = a :< (w, Cofree (Writer w) a)
```

что эквивалентно

```
data Stream (w,a)
     = Stream (w,a) (Stream (w,a))
```

который может быть правильно написан как `WriterT w Stream` с

```
data WriterT w m a = WriterT (m (w,a))
```

## Cofree (или e) ~~ NonEmptyT (Writer e)

Дано

```
data Either e a = Left e
                | Right a
```

у нас есть

```
data Cofree (Either e) a
     = a :< Left e
     | a :< Right (Cofree (Either e) a)
```

которая изоморфна

```
data Hospitable e a
     = Sorry_AllIHaveIsThis_Here'sWhy a e
     | EatThis a (Hospitable e a)
```

или, если вы обещаете оценивать журнал только после полного результата, `NonEmptyT (Writer e) a` с

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

## Cofree (Reader x) ~~ Moore x

Дано

```
data Reader x a = Reader (x -> a)
```

у нас есть

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

которая изоморфна

```
data Plant x a
     = Plant a (x -> Plant x a)
```

aka Moore .

Прочитайте Общие функторы как основа cofree comonads онлайн: https://riptutorial.com/ru/haskell/topic/8258/общие-функторы-как-основа-cofree-comonads

# глава 42: объектив

## Вступление

Объект представляет собой библиотеку для Haskell, которая предоставляет объективы, изоморфизмы, складки, обходы, геттеры и сеттеры, которые предоставляют единый интерфейс для запросов и манипулирования произвольными структурами, в отличие от концепций аксессуаров Java и мутаторов.

## замечания

# Что такое объектив?

Объективы (и другая оптика) позволяют нам разделить описание *того, как* мы хотим получить доступ к некоторым данным из *того, что* мы хотим с ним делать. Важно различать абстрактное понятие объектива и конкретную реализацию. Понимание абстрактно делает программирование с `lens` намного проще в долгосрочной перспективе. Существует много изоморфных представлений линз, поэтому для этого обсуждения мы избегаем какого-либо конкретного обсуждения реализации и вместо этого дадим обзор концепций на высоком уровне.

# фокусирование

Важной концепцией в абстрактном понимании является понятие *фокусировки* . Важная оптика *фокусируется* на определенной части большей структуры данных, не забывая о более широком контексте. Например, объектив `_1` фокусируется на первом элементе кортежа, но не забывает о том, что было во втором поле.

После того, как мы сосредоточимся, мы можем поговорить о том, какие операции нам позволяют выполнять с объективом. Учитывая объект `Lens sa` который при заданном типе типа `s` фокусируется на конкретном `a` , мы можем либо

1. Извлеките `a` , забыв о дополнительном контексте или
2. Замените `a` , предоставив новое значение

Они соответствуют хорошо известным операциям `get` и `set` которые обычно используются для характеристики объектива.

## Другие оптика

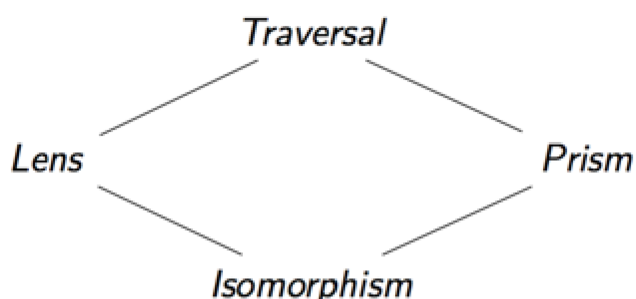
Аналогичным образом мы можем говорить о других оптиках.

| оптический | Фокусируется на... |
|---|---|
| объектив | Одна часть продукта |
| призма | Одна часть суммы |
| пересечение | Нулевые или более части структуры данных |
| изоморфизм | ... |

Каждая оптика фокусируется по-другому, как таковая, в зависимости от того, какой тип оптики у нас есть, мы можем выполнять разные операции.

## Состав

Более того, мы можем составить любую из двух опций, которые мы так обсуждали, чтобы определить сложные обращения к данным. Четыре типа оптики, которые мы обсуждали, образуют решетку, результатом объединения двух оптических элементов является их верхняя граница.




Например, если мы собрали вместе линзу и призму, мы получим обход. Причиной этого является то, что по их (вертикальной) композиции мы сначала фокусируемся на одной части продукта, а затем на одной части суммы. Результатом является оптическая система, которая фокусируется на нулевой или одной части наших данных, которая является частным случаем обхода. (Это также иногда называют аффинным обходом).

## В Хаскелле

Причиной популярности в Haskell является то, что есть очень сжатое представление оптики. Вся оптика - это просто функции определенной формы, которые могут быть составлены вместе с использованием композиции функций. Это приводит к очень легкому

внедрению, что упрощает интеграцию оптики в ваши программы. В дополнение к этому, из-за особенностей кодирования, состав функций также автоматически вычисляет верхнюю границу двух оптических элементов, которые мы составляем. Это означает, что мы можем повторно использовать одни и те же комбинаторы для разных оптических элементов без явного литья.

# Examples

## Манипулирование кортежами с помощью объектива

Получение

```
("a", 1) ^. _1 -- returns "a"
("a", 1) ^. _2 -- returns 1
```

настройка

```
("a", 1) & _1 .~ "b" -- returns ("b", 1)
```

Изменение

```
("a", 1) & _2 %~ (+1) -- returns ("a", 2)
```

`both` обхода

```
(1, 2) & both *~ 2 -- returns (2, 4)
```

## Линзы для записей

# Простая запись

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

Создаются объективы `x` и `y` .

```
let p = Point 5.0 6.0
p ^. x     -- returns 5.0
set x 10 p -- returns Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- returns Point { _x = 6.0, _y = 6.0 }
```

# Управление записями с именами повторяющихся полей

```
data Person = Person { _personName :: String }
makeFields ''Person
```

Создает класс `HasName` класса, `name` объектива для `Person` и делает `Person` экземпляром `HasName` . Последующие записи также будут добавлены в класс:

```
data Entity = Entity { _entityName :: String }
makeFields ''Entity
```

Расширение шаблона Haskell требуется для работы `makeFields` . Технически, вполне возможно создать линзы, сделанные таким образом с помощью других средств, например, вручную.

### Головные линзы

Операторы объектива имеют полезные варианты, которые работают в контекстах состояния. Они получаются заменой `~` на `=` в имени оператора.

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

> Примечание. Варианты с сохранением состояния не должны изменять тип, поэтому у них есть подписи `Lens'` или `Simple Lens'` .

# Избавление от `&` цепочек

Если операции с объективом необходимо сковать, он часто выглядит так:

```
change :: A -> A
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

Это работает благодаря ассоциативности `&` . Однако версия с сохранением состояния более ясна.

```
change a = flip execState a $ do
    lensA %= operationA
    lensB %= operationB
    lensC %= operationC
```

Если `lensX` фактически является `id` , вся операция, конечно, может выполняться напрямую, просто поднимите ее с `modify` .

# Императивный код со структурированным состоянием

Предположим, что этот пример:

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses ''Point
makeLenses ''Entity
makeLenses ''World
```

Мы можем написать код, похожий на классические императивные языки, но при этом позволяя нам использовать преимущества Haskell:

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- move the first entity
    entities . ix 0 . position . x += 1

    -- do some operation on all of them
    entities . traversed . position %= \p -> p `pointAdd` ...

    -- or only on a subset
    entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

## Написание объектива без шаблона Haskell

Чтобы демистифицировать шаблон Haskell, предположим, что у вас есть

```
data Example a = Example { _foo :: Int, _bar :: a }
```

затем

```
makeLenses 'Example
```

производит (более или менее)

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

Тем не менее, ничего особенного не происходит. Вы можете написать их сами:

```
foo :: Lens' (Example a) Int
--  :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; expand the alias
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
--  :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; expand the alias
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

По сути, вы хотите «посетить» ваш объектив «фокус» с помощью функции `wrap` а затем перестроить «весь» тип.

## Объективы и призмы

`Lens' sa` означает, что вы *всегда* можете найти `a` в любом `s` . А `Prism' sa` означает , что *иногда* вы можете обнаружить , что `s` на самом деле просто но иногда это что - то другое. `a`

Чтобы быть более понятным, мы имеем `_1 :: Lens' (a, b) a` потому что у любого набора *всегда* есть первый элемент. У нас есть `_Just :: Prism' (Maybe a) a` , `Maybe a a Just Nothing` `_Just :: Prism' (Maybe a) a` , потому что *иногда* `Maybe a` на самом деле является значение , завернутые в `Just` , но *иногда* это не `Nothing` .

С этой интуицией некоторые стандартные комбинаторы можно интерпретировать параллельно друг другу

- `view :: Lens' sa -> (s -> a)` "получает" `a` из `s`
- `set :: Lens' sa -> (a -> s -> s)` "наборы" в `a` слот `s`
- `review :: Prism' sa -> (a -> s)` «понимает», что `a` может быть `s`
- `preview :: Prism' sa -> (s -> Maybe a)` «пытается» превратить `s` в `a` .

Другой способ подумать о том, что значение типа `Lens' sa` показывает, что `s` имеет ту же структуру, что и `(r, a)` для некоторого неизвестного `r` . С другой стороны, `Prism' sa` показывает, что `s` имеет ту же структуру, что и `Either ra` для некоторого `r` . Мы можем написать эти четыре функции выше с этими знаниями:

```
-- `Lens' s a` is no longer supplied, instead we just *know* that `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- `Prism' s a` is no longer supplied, instead we just *know* that `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

## обходов

`Traversal' sa` показывает, что `s` имеет 0-to-many `a` s внутри него.

```
toListOf :: Traversal' s a -> (s -> [a])
```

Любой тип `t` который является `Traversable` автоматически имеет этот `traverse :: Traversal (ta) a`.

Мы можем использовать `Traversal`, чтобы установить или отобразить по всем из них `a` значения

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

`f :: Lens' sa` говорит , что именно один внутри `a s`. А `g :: Prism' ab` говорит, что в `a` есть 0 или 1 `b` s. Составление `f . g` дает нам `Traversal' sb` потому что после `f` а затем `g` показывает, как там есть 0-to-1 `b` s в `s`.

## Линзы составляют

Если у вас есть `f :: Lens' ab` и `g :: Lens' bc` то `f . g` - это `Lens' ac` полученная следующим образом `f` а затем `g`. В частности:

- Линзы представляют собой функции (на самом деле они просто *являются* функциями)
- Если вы думаете о `view` функциональности `Lens`, похоже , потоки данных «слева направо» -за может чувствовать назад к нормальной интуиции для композиции функций. С другой стороны, это должно быть естественно, если вы думаете `.` -нотация, как это происходит на языках ОО.

Более чем просто компоновка `Lens` с `Lens` `(.)` Можно использовать для создания почти любого типа типа « `Lens` ». Не всегда легко понять, что результат, так как тип становится более жестким, но вы можете использовать диаграмму `lens` чтобы понять это. Композиция `x . y` имеет тип наименьшей-верхней границы типов `x` и `y` в этой диаграмме.

## Классные линзы

В дополнение к стандартной функции `makeLenses` для генерации `Lens` es `Control.Lens.TH` также предлагает функцию `makeClassy`. `makeClassy` имеет тот же тип и работает по существу так же, как `makeLenses`, с одним ключевым отличием. В дополнение к созданию стандартных объективов и обходов, если тип не имеет аргументов, он также создаст класс, описывающий все типы данных, которые обладают типом в качестве поля. Например

```
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

создаст

```
class HasFoo t where
   foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

## Поля с makeFields

(Этот пример скопирован из этого ответа StackOverflow )

Допустим, у вас есть несколько разных типов данных, которые должны иметь объектив с таким же названием, в этом случае `capacity` . `makeFields` создаст класс, который выполнит это без конфликтов пространства имен.

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
           , TemplateHaskell
  #-}

module Foo
where

import Control.Lens

data Foo
  = Foo { fooCapacity :: Int }
  deriving (Eq, Show)
$(makeFields ''Foo)

data Bar
  = Bar { barCapacity :: Double }
  deriving (Eq, Show)
$(makeFields ''Bar)
```

Тогда в ghci:

```
*Foo
λ let f = Foo 3
|     b = Bar 7
|
b :: Bar
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
```

```
λ barCapacity b
7.0
it :: Double

*Foo
λ f ^. capacity
3
it :: Int

*Foo
λ b ^. capacity
7.0
it :: Double

λ :info HasCapacity
class HasCapacity s a | s -> a where
  capacity :: Lens' s a
    -- Defined at Foo.hs:14:3
instance HasCapacity Foo Int -- Defined at Foo.hs:14:3
instance HasCapacity Bar Double -- Defined at Foo.hs:19:3
```

Итак, что на самом деле сделано, объявлен класс `HasCapacity sa` , где емкость - это `Lens'` от `s` до `a` ( `a` фиксируется, как только s известен). Он определил имя «емкость», сняв с поля (с нижней) имя типа данных из поля; Мне приятно не использовать знак подчеркивания ни имени поля, ни имени объектива, поскольку иногда синтаксис записи на самом деле является тем, что вы хотите. Вы можете использовать makeFieldsWith и различные lensRules, чтобы иметь несколько разных опций для расчета имен объективов.

В случае, если это помогает, используя ghci -ddump-сращивания Foo.hs:

```
[1 of 1] Compiling Foo              ( Foo.hs, interpreted )
Foo.hs:14:3-18: Splicing declarations
    makeFields ''Foo
  ======>
    class HasCapacity s a | s -> a where
      capacity :: Lens' s a
    instance HasCapacity Foo Int where
      {-# INLINE capacity #-}
      capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: Splicing declarations
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
      {-# INLINE capacity #-}
      capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
Ok, modules loaded: Foo.
```

Итак, первый сплайс сделал класс `HasCapcity` и добавил экземпляр для Foo; второй использовал существующий класс и сделал экземпляр для Bar.

Это также работает, если вы импортируете класс `HasCapcity` из другого модуля; `makeFields` может добавлять дополнительные экземпляры в существующий класс и распространять ваши типы через несколько модулей. Но если вы снова используете его в другом модуле, где вы не импортировали класс, он будет создавать новый класс (с тем же именем), и у вас

будет две отдельные перегруженные объективы, которые несовместимы.

Прочитайте объектив онлайн: https://riptutorial.com/ru/haskell/topic/891/объектив

# глава 43: Объявления о фиксации

## Синтаксис

1. infix [integer] ops
2. infixl [integer] ops
3. infixr [integer] ops

## параметры

| Компонент декларации | Имея в виду |
| --- | --- |
| `infixr` | оператор является право-ассоциативным |
| `infixl` | оператор лево-ассоциативный |
| `infix` | оператор не является ассоциативным |
| необязательная цифра | приоритет привязки оператора (диапазон 0 ... 9, значение по умолчанию 9) |
| `op1, ... , opn` | операторы |

## замечания

Для анализа выражений, связанных с операторами и функциями, Haskell использует декларации четности, чтобы выяснить, куда идут скобки. Для этого

1. обертывает функции приложения в parens
2. использует привязку привязки для объединения групп терминов, разделенных операторами с одинаковым приоритетом
3. использует ассоциативность этих операторов, чтобы выяснить, как добавить парны к этим группам

Заметим, что мы предполагаем здесь, что операторы в любой заданной группе из шага 2 должны иметь одну и ту же ассоциативность. Фактически, Haskell отклонит любую программу, где это условие не выполняется.

В качестве примера приведенного выше алгоритма мы можем шагнуть, хотя процесс добавления скобок к `1 + negate 5 * 2 - 3 * 4 ^ 2 ^ 1` .

```
infixl 6 +
infixl 6 -
infixl 7 *
infixr 8 ^
```

1. `1 + (negate 5) * 2 - 3 * 4 ^ 2 ^ 1`
2. `1 + ((negate 5) * 2) - (3 * (4 ^ 2 ^ 1))`
3. `(1 + ((negate 5) * 2)) - (3 * (4 ^ (2 ^ 1)))`

---

Более подробная информация содержится в разделе 4.4.2 отчета Haskell 98 .

# Examples

## Ассоциативность

`infixl` vs `infixr` vs `infix` описывает, на каких сторонах будут группироваться `infixr` . Например, рассмотрим следующие декларации определения (в базе)

```
infixl 6 -
infixr 5 :
infix  4 ==
```

`infixl` говорит нам, что `-` оставил ассоциативность, а это означает, что `1 - 2 - 3 - 4` анализируется как

```
((1 - 2) - 3) - 4
```

`infixr` говорит нам, что `:` имеет правую ассоциативность, а это означает, что `1 : 2 : 3 : []` анализируется как

```
1 : (2 : (3 : []))
```

`infix` сообщает нам, что `==` не может использоваться без нас, включая скобки, что означает, что `True == False == True` является синтаксической ошибкой. С другой стороны, `True == (False == True)` или `(True == False) == True` в порядке.

Операторы без явного объявления `infixl 9` - это `infixl 9` .

## Приоритет привязки

Число, которое следует за информацией ассоциативности, описывает, в каком порядке применяются операторы. Он должен всегда находиться между `0` и `9` включительно. Это обычно называют тем, насколько тесно связан оператор. Например, рассмотрим следующие декларации определения (в базе)

```
infixl 6 +
```

```
infixl 7 *
```

Так как `*` имеет более высокий приоритет привязки, чем `+` мы читаем `1 * 2 + 3` как

```
(1 * 2) + 3
```

Короче говоря, чем выше число, тем ближе оператор «потянет» парнеров по обе стороны от него.

# замечания

- Функциональное приложение *всегда* привязывается выше операторов, поэтому `` fx `op` gy `` должно интерпретироваться как `(fx)` op `(gy)` независимо от того, что оператор` `` `op` `` и его объявление фиксации.
- Если приоритет привязки опускается в объявлении фиксации (например, мы имеем `infixl *!?` ), по умолчанию используется значение `9` .

## Примеры объявлений

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- ``infixl 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR` ``
- `infix 4 ==, /=, <, <=, >=, >`
- `infix ??`

Прочитайте Объявления о фиксации онлайн: https://riptutorial.com/ru/haskell/topic/4691/объявления-о-фиксации

# глава 44: Операторы Infix

## замечания

Большинство функций Haskell вызывается с именем функции, за которым следуют аргументы (префиксная нотация). Для функций, которые принимают два аргумента типа (+), иногда имеет смысл предоставить аргумент до и после функции (infix).

## Examples

### прелюдия

# логический

`&&` является логическим И, `||` является логическим ИЛИ.

`==` - равенство, `/=` не равенство, `<` / `<=` меньше и `>` / `>=` более крупные операторы.

# Арифметические операторы

Численные операторы `+` , `-` и `/` ведут себя в основном так, как вы ожидали. (Отдел работает только с дробными номерами, чтобы избежать проблем округления - целочисленное деление должно выполняться с помощью `quot` или `div` ). Более необычными являются три оператора возведения в степень Хаскелла:

- `^` берет базу любого типа номера в неотрицательную интегральную мощность. Это работает просто ( быстрым ) повторным умножением. Например

  ```
  4^5  ≡  (4*4)*(4*4)*4
  ```

- `^^` делает то же самое в положительном случае, но и работает для отрицательных показателей. Например

  ```
  3^^(-2)  ≡  1 / (2*2)
  ```

  В отличие от `^` , для этого требуется дробный базовый тип (т. `4^^5 :: Int` не будет работать, только `4^5 :: Int` или `4^^5 :: Rational` ).

- `**` реализует возведение в степень реального числа. Это работает для очень общих аргументов, но более усложнительно дорого, чем `^` или `^^` , и, как правило, берет

небольшие ошибки с плавающей запятой.

```
2**pi  ≡  exp (pi * log 2)
```

# Списки

Существует два оператора конкатенации:

- `:` (произносится cons ) добавляет один аргумент перед списком. Этот оператор фактически является конструктором и поэтому может также использоваться для *сопоставления шаблонов* («инверсная конструкция») списка.
- `++` объединяет целые списки.

```
[1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
```

`!!` является индексирующим оператором.

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

Обратите внимание, что списки индексирования неэффективны (сложность $O(n)$ вместо $O(1)$ для массивов или $O(\log n)$ для карт ); обычно Haskell предпочитает деконструировать списки, сворачивая сопоставление шаблонов от вместо индексации.

# Управляющий поток

- `$` - оператор приложения функции.

```
f $ x  ≡  f x
       ≡  f(x)   -- disapproved style
```

  Этот оператор в основном используется для исключения скобок. Он также имеет строгую версию `$!` , который заставляет аргумент оцениваться перед применением функции.

- `.` Составляет функции.

```
(f . g) x  ≡  f (g x)  ≡  f $ g x
```

- `>>` последовательности монадических действий. Например, `writeFile "foo.txt" "bla" >> putStrLn "Done."` сначала напишите в файл, затем распечатайте сообщение на экране.
- `>>=` делает то же самое, а также принимает аргумент, который должен быть передан

от первого действия к следующему. `readLn >>= \x -> print (x^2)` будет ждать ввода пользователем числа, а затем вывести квадрат этого числа на экран.

## Пользовательские операторы

В Haskell вы можете определить любой инфиксный оператор, который вам нравится. Например, я мог бы определить оператор обертывания списка как

```
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

Вы должны всегда давать этим операторам объявление о фиксации , например

```
infixr 5 >+<
```

(что означает, что `>+<` связывается так же сильно, как `++` и `:` do).

## Поиск информации об инфиксных операторах

Поскольку в Haskell настолько распространены инфиксы, вам регулярно нужно искать их подпись и т. Д. К счастью, это так же просто, как и для любой другой функции:

- Поисковые системы Haskell Hayoo и Hoogle могут использоваться для инфиксных операторов, как и для любого другого, определенного в некоторой библиотеке.
- В GHCi или IHaskell вы можете использовать директивы `:i` и `:t` ( **i** nfo и **t** ype), чтобы узнать основные свойства оператора. Например,

  ```
  Prelude> :i +
  class Num a where
    (+) :: a -> a -> a
    ...
        -- Defined in ‘GHC.Num’
  infixl 6 +
  Prelude> :i ^^
  (^^) :: (Fractional a, Integral b) => a -> b -> a
        -- Defined in ‘GHC.Real’
  infixr 8 ^^
  ```

  Это говорит мне, что `^^` связывается более плотно, чем `+` , оба берут числовые типы в качестве своих элементов, но `^^` требует, чтобы показатель был интегральным, а база была дробной.
  Менее многословный `:t` требует оператора в круглых скобках, например

  ```
  Prelude> :t (==)
  (==) :: Eq a => a -> a -> Bool
  ```

Прочитайте Операторы Infix онлайн: https://riptutorial.com/ru/haskell/topic/6792/операторы-infix

# глава 45: оптимизация

## Examples

### Компиляция вашей программы для профилирования

Компилятор GHC имеет зрелую поддержку для компиляции с аннотациями профилирования.

Использование флагов `-prof` и `-fprof-auto` при компиляции добавит поддержку вашего двоичного `-fprof-auto` для флагов профилирования для использования во время выполнения.

Предположим, что у нас есть эта программа:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Скомпилировал его так:

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

Затем запустили его с параметрами системы времени выполнения для профилирования:

```
./Main +RTS -p
```

Мы увидим, что файл `main.prof` создал пост-исполнение (после выхода программы), и это даст нам все виды профилирующей информации, такие как МВЗ, которая дает нам разбивку стоимости, связанной с запуском различных частей кода :

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                 individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
```

```
CAF        GHC.Conc.Signal        110          0    0.0    0.0     0.0    0.0
CAF        Main                   108          0    0.0    0.0   100.0  100.0
 main      Main                   204          1    0.0    0.0   100.0  100.0
  fib      Main                   205    2692537  100.0  100.0   100.0  100.0
```

## Центры затрат

Центры затрат - это аннотации к программе Haskell, которые могут автоматически добавляться компилятором GHC - с использованием `-fprot-auto` - или программистом с использованием `{-# SCC "name" #-} <expression>` , где "name" is любое имя, которое вы хотите, и `<expression>` - любое действительное выражение Haskell:

```
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
          print $ {-# SCC "print_list" #-} (length l)
```

Компиляция с `-fprof` и работа с `+RTS -p` например, `ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p` будет производить `Main.prof` после выхода программы.

Прочитайте оптимизация онлайн: https://riptutorial.com/ru/haskell/topic/4342/оптимизация

# глава 46: оспоримый

## Вступление

Класс `Traversable` обобщает функцию, ранее известную как `mapM :: Monad m => (a -> mb) -> [a] -> m [b]` для работы с `Applicative` эффектами над структурами, отличными от списков.

## Examples

### Выполнение функции и складки для структуры с возможностью перемещения

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

Каждая `Traversable` структура может быть сделана `Foldable` `Functor` , используя `fmapDefault` и `foldMapDefault` функцию , найденную в `Data.Traversable` .

```
instance Functor MyType where
    fmap = Traversable.fmapDefault

instance Foldable MyType where
    foldMap = Traversable.foldMapDefault
```

`fmapDefault` определяется путем выполнения `traverse` в аппликативном `fmapDefault` `Identity` .

```
newtype Identity a = Identity { runIdentity :: a }

instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity x = Identity (f x)

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

`foldMapDefault` определяется с использованием прикладного функтора `Const` , который игнорирует его параметр при накоплении моноидального значения.

```
newtype Const c a = Const { getConst :: c }

instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
```

```
foldMapDefault f = getConst . traverse (Const . f)
```

## Экземпляр Traversable для двоичного дерева

Реализации `traverse` обычно выглядят как реализация `fmap` в `Applicative` контексте.

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Traversable Tree where
    traverse f Leaf = pure Leaf
    traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

Эта реализация выполняет обход дерева в порядке.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

Расширение `DeriveTraversable` позволяет GHC создавать экземпляры `Traversable` на основе структуры типа. Мы можем изменить порядок машинного обхода путем настройки макета конструктора `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving (Functor, Foldable, Traversable)  -- also using DeriveFunctor and
DeriveFoldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving (Functor, Foldable, Traversable)

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
```

```
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> traverse print (preorder myTree)
'b'
'a'
'c'
ghci> traverse print (postorder myTree)
'a'
'c'
'b'
```

## Перемещение структуры в обратном направлении

Обход может выполняться в обратном направлении с помощью `Backwards` аппликативного функтора , который переворачивает существующий аппликативный так, чтобы скомпонованные эффекты имели место в обратном порядке.

```
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
    Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

`Backwards` можно использовать в «обратном `traverse` ». Когда лежащий в основе аппликативный из `traverse` вызова переворачиваются с `Backwards` , результирующий эффект происходит в обратном порядке.

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

`Reverse` новый тип найден в разделе Data.Functor.Reverse.

## Определение Traversable

```
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
```

```
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    sequence :: Monad m => t (m a) -> m (t a)
    sequence = sequenceA
```

`Traversable` структуры `t` являются финишными контейнерами элементов `a` которые могут работать с эффектной «посетительской» операцией. Функция посетителя `f :: a -> fb` выполняет побочный эффект для каждого элемента структуры, а `traverse` составляет эти побочные эффекты с использованием `Applicative` . Другой способ взглянуть на это состоит в том, что `sequenceA` говорит, что `Traversable` структуры коммутируют с `Applicative` s.

## Преобразование структуры с пересечением с помощью накопительного параметра

Две функции `mapAccum` объединяют операции складывания и отображения.

```
--                                                       A Traversable structure
--                                                                      |
--                                                     A seed value   |
--                                                                |      |
--                                                              |-|   |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
--                                      |------------------|              |--------|
--                                               |                                |
--                         A folding function which produces a new mapped        |
--                           element 'c' and a new accumulator value 'a'         |
--                                                                                 |
--                                                              Final accumulator value
--                                                               and mapped structure
```

Эти функции обобщают `fmap` в том, что они позволяют отображаемым значениям зависеть от того, что произошло ранее в сгибе. Они обобщают `foldl` / `foldr` на то, что они отображают структуру на месте, а также уменьшают ее до значения.

Например, `tails` могут быть реализованы с использованием `mapAccumR` и его сестра `inits` может быть реализована с использованием `mapAccumL` .

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
["", "a", "ab", "abc"]
```

`mapAccumL` осуществляются путем обхода в `State` аппликативном функторе.

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
    pure x = State $ \s -> (s, x)
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)

mapAccumL f z t = runState (traverse (State . flip f) t) z
```

`mapAccumR` работает путем запуска `mapAccumL` в обратном порядке .

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

## Трассируемые структуры как формы с содержанием

Если тип `t` является `Traversable` то значения `ta` можно разбить на две части: их «форма» и их «содержимое»:

```
data Traversed t a = Traversed { shape :: t (), contents :: [a] }
```

где «содержимое» совпадает с тем, что вы «посещаете» с помощью экземпляра `Foldable` .

Переход в одно направление, от `ta` до `Traversed ta` не требует ничего, кроме `Functor` и `Foldable`

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

но возвращение назад использует функцию `traverse`

```
import Control.Monad.State

-- invariant: state is non-empty
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

В законах `Traversable` требуется `break . recombine` и `recombine . break` - идентичность. Примечательно, что это означает, что в `contents` есть только правильные элементы количества, чтобы полностью заполнить `shape` без остатков.

`Traversed t` является `Traversable` . Реализация `traverse` работает, посещая элементы, используя экземпляр списка « `Traversable` а затем снова привязывая инертную форму к результату.

```
instance Traversable (Traversed t) where
```

```
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

## Транспонирование списка списков

Отметив, что `zip` переносит кортеж списков в список кортежей,

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

и сходство между типами `transpose` и `sequenceA` ,

```
-- transpose exchanges the inner list with the outer list
--           +---+-->--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+-->--+---+

-- sequenceA exchanges the inner Applicative with the outer Traversable
--                                               +------>------+
--                                               |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                                  |       |
--                                                  +--->---+
```

идея состоит в том, чтобы использовать `[]` ' `Traversable` and `Applicative` structure для развертывания `sequenceA` как своего рода *n-ary* `zip` , скрепляя вместе все внутренние списки вместе поточечно.

`[]` умолчанию «приоритетный выбор». `Applicative` экземпляр не подходит для нашего использования - нам нужен «zippy» `Applicative` . Для этого мы используем `ZipList` , который находится в `Control.Applicative` .

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

Теперь мы получаем `transpose` бесплатно, `ZipList` в `ZipList Applicative` .

```
transpose :: [[a]] -> [[a]]
transpose = getZipList . traverse ZipList

ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

Прочитайте оспоримый онлайн: https://riptutorial.com/ru/haskell/topic/754/оспоримый

# глава 47: параллелизм

## параметры

| Тип / Функция | подробность |
|---|---|
| `data Eval a` | Eval - это Монада, которая упрощает определение параллельных стратегий |
| `type Strategy a = a -> Eval a` | функция, которая воплощает параллельную стратегию оценки. Функция проходит (части) своего аргумента, оценивая подвыражения параллельно или последовательно |
| `rpar :: Strategy a` | искры его аргумента (для параллельной оценки) |
| `rseq :: Strategy a` | оценивает свой аргумент на слабую головную нормальную форму |
| `force :: NFData a => a -> a` | оценивает всю структуру своего аргумента, сводя его к нормальной форме, прежде чем возвращать сам аргумент. Он обеспечивается модулем Control.DeepSeq |

## замечания

Книга Саймона Марлоу « Параллельное и параллельное программирование в Хаскелле» является выдающейся и охватывает множество концепций. Он также очень доступен даже для самого нового программиста Haskell. Он настоятельно рекомендуется и доступен в формате PDF или онлайн бесплатно.

**Параллельные и параллельные**

Саймон Марлоу делает это лучше всего :

> Параллельная программа - это программа, которая использует множество вычислительных аппаратных средств (например, несколько процессорных ядер) для более быстрого вычисления вычислений. Цель состоит в том, чтобы прийти к ответу ранее, делегируя разные части вычислений различным процессорам, которые выполняются одновременно.
>
> Напротив, параллелизм - это метод структурирования программ, в котором есть несколько потоков управления. Понятно, что потоки управления выполняются «одновременно»; то есть пользователь видит, что их эффекты чередуются. Выполняются ли они в одно и то же время или нет, это деталь реализации;

параллельная программа может выполняться на одном процессоре посредством выполнения с чередованием или на нескольких физических процессорах.

#### Нормальная форма слабой головы

Важно знать, как работает ленивая оценка. Первый раздел этой главы даст сильное введение в WHNF и то, как это относится к параллельному и параллельному программированию.

# Examples

## Эвальская Монада

Параллельность в Haskell может быть выражена с помощью `Eval` Monad из `Control.Parallel.Strategies`, используя функции `rpar` и `rseq` (среди прочего).

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  return (a,b)
```

Выполнение `main` выше будет выполняться и «немедленно возвращаться», в то время как два значения `a` и `b` вычисляются в фоновом режиме через `rpar`.

Примечание: убедитесь, что вы `-threaded` компиляцию с `-threaded` для параллельного выполнения.

## RPAR

`rpar :: Strategy a` выполняет данную стратегию (напомните: `type Strategy a = a -> Eval a`) параллельно:

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  where
    loop = do
      putStrLn "Enter a number"
      n <- getLine

      let lim = read n :: Int
          hf  = quot lim 2
```

```
          result = runEval $ do
            -- we split the computation in half, so we can concurrently calculate primes
            as <- rpar (force (primesBtwn 2 hf))
            bs <- rpar (force (primesBtwn (hf + 1) lim))
            return (as ++ bs)

      forkIO $ putStrLn ("\nPrimes are: " ++ (show result) ++ " for " ++ n ++ "\n")
      loop

-- Compute primes between two integers
-- Deliberately inefficient for demonstration purposes
primesBtwn n m = eratos [n..m]
  where
    eratos []     = []
    eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

Выполнение этого будет демонстрировать одновременное поведение:

```
Enter a number
12
Enter a number

Primes are: [2,3,5,7,8,9,10,11,12] for 12

100
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,7
for 100

200000000
Enter a number
-- waiting for 200000000
200
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107,10
for 200

-- still waiting for 200000000
```

### rseq

Мы можем использовать `rseq :: Strategy a` чтобы заставить аргумент «Нормальная форма слабой головы»:

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
```

```
rseq a
return (a,b)
```

Это тонко изменяет семантику `rpar` примера; в то время как последний будет немедленно вернуться в то время вычисления значений в фоновом режиме, этот пример будет ждать , пока не может быть оценена в WHNF. `a`

Прочитайте параллелизм онлайн: https://riptutorial.com/ru/haskell/topic/6887/параллелизм

# глава 48: Перегруженные литералы

## замечания

### Целочисленные литеры

представляет собой цифру **без** десятичной точки

например, `0` , `1` , `42` , ...

неявно применяется к `fromInteger` который является частью класса `Num` type, поэтому он действительно имеет тип `Num a => a` - то есть он может иметь любой тип, являющийся экземпляром `Num`

---

### Дробные литералы

представляет собой цифру **с** десятичной точкой

например, `0.0` , `-0.1111` , ...

неявно применяется к `fromRational` который является частью класса типа `Fractional` поэтому он действительно имеет тип `a => a` - то есть он может иметь любой тип, являющийся экземпляром `Fractional`

---

### Строковые литералы

Если вы добавите расширение языка `OverloadedStrings` в *GHC,* вы можете иметь то же самое для `String` -literals, которые затем применяются к `fromString` из класса `Data.String.IsString`

Это часто используется для замены `String` на `Text` или `ByteString` .

---

### Список литературы

Списки могут быть определены с помощью синтаксиса `[1, 2, 3]` literal. В GHC 7.8 и выше это также можно использовать для определения других типов списка с расширением `OverloadedLists` .

По умолчанию тип `[]` :

```
> :t []
```

```
[] :: [t]
```

С `OverloadedLists` это становится:

```
[] :: GHC.Exts.IsList l => l
```

# Examples

### Целочисленная цифра

# Тип литерала

```
Prelude> :t 1
1 :: Num a => a
```

# выбор конкретного типа с аннотациями

Вы можете указать тип до тех пор, пока целевой тип `Num` с *аннотацией* :

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
it :: Double
Prelude> 1 :: Word
1
it :: Word
```

если не компилятор будет жаловаться

Прелюдия> 1 :: String

```
<interactive>:
    No instance for (Num String) arising from the literal `1'
    In the expression: 1 :: String
    In an equation for `it': it = 1 :: String
```

### Плавающая цифра

# Тип литерала

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

# Выбор конкретного типа с аннотациями

Вы можете указать тип с *аннотацией типа* . Единственное требование - тип должен иметь экземпляр `Fractional` .

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

если не компилятор будет жаловаться

```
Prelude> 1.0 :: Int
<interactive>:
    No instance for (Fractional Int) arising from the literal `1.0'
    In the expression: 1.0 :: Int
    In an equation for `it': it = 1.0 :: Int
```

### Струны

# Тип литерала

Без каких-либо расширений тип строкового литерала, т. Е. Что-то между двойными кавычками, - это просто строка, также называемая списком символов:

```
Prelude> :t "foo"
"foo" :: [Char]
```

Однако, когда расширение `OverloadedStrings` включено, строковые литералы становятся полиморфными, аналогичными литералам чисел :

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

Это позволяет нам определять значения типа типа string без необходимости каких-либо явных преобразований. По сути, расширение `OverloadedStrings` просто обертывает каждый строковый литерал в общей `fromString` преобразования `fromString` , поэтому, если контекст требует, например, более эффективного `Text` вместо `String` , вам не нужно об этом беспокоиться.

# Использование строковых литералов

```
{-# LANGUAGE OverloadedStrings #-}
```

```haskell
import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"

-- The following two examples are only allowed with OverloadedStrings

withText :: Text
withText = "Hello Text"      -- instead of: withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- instead of: withBS = Data.ByteString.pack "Hello ByteString"
```

Обратите внимание на то, как мы смогли построить значения `Text` и `ByteString` же, как мы строим обычные значения `String` (или `[Char]` ), вместо того, чтобы использовать каждую функцию типа `pack` чтобы явно кодировать строку.

Дополнительные сведения о расширении языка `OverloadedStrings` см. В документации по расширению .

## Список литературы

Расширение GHC OverloadedLists позволяет создавать структуры данных, подобные спискам, с синтаксисом списка литералов.

Это позволяет вам Data.Map следующим образом:

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

Вместо этого (обратите внимание на использование дополнительного M.fromList ):

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

Прочитайте Перегруженные литералы онлайн: https://riptutorial.com/ru/haskell/topic/369/перегруженные-литералы

# глава 49: Переписать правила (GHC)

## Examples

### Использование правил перезаписи для перегруженных функций

В этом вопросе @Viclib попросил об использовании правил перезаписи для использования законов типа typeclass для устранения некоторых перегруженных вызовов функций:

> Обратите внимание на следующий класс:
>
> ```
> class ListIsomorphic l where
>     toList    :: l a -> [a]
>     fromList  :: [a] -> l a
> ```
>
> Я также требую, чтобы `toList . fromList == id`. Как написать правила перезаписи, чтобы сообщить GHC сделать эту замену?

Это несколько сложный вариант для механизма правил перезаписи GHC, поскольку перегруженные функции переписываются в их конкретные методы экземпляра правилами, которые неявно создаются за кулисами GHC (так что что-то вроде `fromList :: Seq a -> [a]` будет переписан в `Seq$fromList` и т. д.).

Однако, сначала переписывая `toList` и `fromList` в не-встроенные методы без класса, мы можем защитить их от преждевременного переписывания и сохранить их до тех пор, пока правило для композиции не будет срабатывать:

```
{-# RULES
  "protect toList"   toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"  forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

Прочитайте Переписать правила (GHC) онлайн: https://riptutorial.com/ru/haskell/topic/4914/переписать-правила--ghc-

# глава 50: Полиморфизм произвольного ранга с RankNTypes

## Вступление

Система типов GHC поддерживает произвольное количественное определение произвольного уровня в типах посредством использования языковых расширений `Rank2Types` и `RankNTypes` .

## Синтаксис

- Произвольное количественное определение ранжирования разрешено с расширением языка `Rank2Types` или `RankNTypes` .
- Если это расширение включено, ключевое слово `forall` можно использовать для добавления количественной оценки более высокого ранга.

## Examples

### RankNTypes

StackOverflow заставляет меня иметь один пример. Если эта тема одобрена, мы должны переместить этот пример здесь.

Прочитайте Полиморфизм произвольного ранга с RankNTypes онлайн: https://riptutorial.com/ru/haskell/topic/8984/полиморфизм-произвольного-ранга-с-rankntypes

# глава 51: Потоковое IO

## Examples

### Потоковое IO

`io-streams` - это библиотека, основанная на `io-streams` которая фокусируется на абстракции Stream, но на IO. Он раскрывает два типа:

- `InputStream` : интеллектуальный дескриптор только для чтения
- `OutputStream` : интеллектуальный дескриптор только для записи

Мы можем создать поток с `makeInputStream :: IO (Maybe a) -> IO (InputStream a)` . Чтение из потока выполняется с использованием `read :: InputStream a -> IO (Maybe a)` , где `Nothing` обозначает EOF:

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
  is <- S.makeInputStream $ randomInt  -- create an InputStream
  forever $ printStream =<< S.read is  -- forever read from that stream
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
  r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "Nada!"
printStream (Just a) = putStrLn $ show a
```

Прочитайте Потоковое IO онлайн: https://riptutorial.com/ru/haskell/topic/4984/потоковое-io

# глава 52: Реактивный-банан

## Examples

### Внедрение внешних событий в библиотеку

Этот пример не привязан к какому-либо конкретному набору инструментов GUI, например, реактивный банан-wx. Вместо этого он показывает, как вводить арбитационные операции `IO` в машины FRP.

Модуль `Control.Event.Handler` предоставляет функцию `addHandler` которая создает пару значений `AddHandler a` и `a -> IO ()` . Первый используется самим реактивным бананом, чтобы получить значение `Event a` , а последнее - это простая функция, которая используется для запуска соответствующего события.

```
import Data.Char (toUpper)

import Control.Event.Handler
import Reactive.Banana

main = do
    (inputHandler, inputFire) <- newAddHandler
```

В нашем случае параметр обработчика имеет тип `a String` , , но код , который позволяет компилятору сделать вывод , что будет написано позже.

Теперь мы определяем `EventNetwork` которая описывает нашу систему, основанную на FRP. Это делается с помощью функции `compile` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
```

Функция `fromAddHandler` преобразует значение `AddHandler a` в `Event a` , которое рассматривается в следующем примере.

Наконец, мы запускаем наш «цикл событий», который запускает события на входе пользователя:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
    forever $ do
        input <- getLine
        inputFire input
```

## Тип события

В реактивном банане тип `Event` представляет собой поток некоторых событий во времени. `Event` похоже на аналоговый импульсный сигнал в том смысле, что он не является непрерывным во времени. В результате `Event` является экземпляром `Functor` класса типов только. Вы не можете объединить два `Event` вместе, потому что они могут срабатывать в разное время. Вы можете сделать что-то со значением [current] `Event` и отреагировать на него с помощью некоторых `IO` .

Преобразования по значению `Event` s выполняются с использованием `fmap` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
```

Реагирование на `Event` осуществляется таким же образом. Сначала вы `fmap` его с помощью действия типа `a -> IO ()` а затем передадите его в функцию `reactimate` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
        let inputEventReaction = fmap putStrLn inputEvent' -- this has type `Event (IO ())
        reactimate inputEventReaction
```

Теперь , когда `inputFire "something"` - `"SOMETHING"` `inputFire "something"` называется, `"SOMETHING"` будет напечатано.

## Тип поведения

Чтобы представить непрерывные сигналы, реакционно-банановые функции `Behavior a` типа. В отличие от `Event` , `Behavior` является `Applicative` , которое позволяет комбинировать n `Behavior` s с помощью n-ary чистой функции (используя `<$>` и `<*>` ).

Чтобы получить `Behavior a` от `Event a` существует функция `accumE` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

`accumE` принимает начальное значение `Behavior` и `Event` , содержащее функцию, которая устанавливает ее в новое значение.

Как и в случае с `Event` s, вы можете использовать `fmap` для работы с текущим значением `Behavior` s, но вы также можете комбинировать их с `(<*>)` .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

Чтобы реагировать на изменения `Behavior` есть функция `changes` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
        inputChanged <- changes inputBehavior
```

Единственное, что следует отметить, это то, что `changes` возвращают `Event (Future a)` вместо `Event a` . Из-за этого `reactimate'` следует использовать вместо того, чтобы `reactimate` . Обоснование этого можно получить из документации.

## Приведение в действие EventNetworks

`EventNetwork` s, возвращаемая `compile` должна быть активирована до того, как активируются события, связанные с реакцией.

```
main = do
    (inputHandler, inputFire) <- newAddHandler

    eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
        reactimate inputEventReaction

    inputFire "This will NOT be printed to the console!"
    actuate eventNetwork
    inputFire "This WILL be printed to the console!"
```

Прочитайте Реактивный-банан онлайн: https://riptutorial.com/ru/haskell/topic/4186/реактивный-банан

# глава 53: Рекурсивные схемы

## замечания

Функции, упомянутые здесь в примерах, определяются с различной степенью абстракции в нескольких пакетах, например, в `recursion-schemes` `data-fix` и `recursion-schemes` (здесь больше функций). Вы можете просмотреть более полный список, выполнив поиск на Hayoo .

# Examples

### Фиксированные очки

`Fix` принимает тип «шаблон» и связывает рекурсивный узел, накладывая шаблон как лазанью.

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Внутри `Fix f` мы найдем слой шаблона `f` . Для того, чтобы заполнить `f` параметра «s, `Fix f` пробки в *себе.* Поэтому, когда вы заглядываете в шаблон `f` вы обнаруживаете рекурсивное возникновение `Fix f` .

Вот как типичный рекурсивный тип данных можно перевести в нашу структуру шаблонов и неподвижных точек. Мы удаляем рекурсивные вхождения типа и отмечаем их позиции с использованием параметра `r` .

```
{-# LANGUAGE DeriveFunctor #-}

-- natural numbers
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: Nat
zero = Fix Zero_
suc :: Nat -> Nat
suc n = Fix (Suc_ n)


-- lists: note the additional type parameter a
-- data List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)
```

```
-- binary trees: note two recursive occurrences
-- data Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

## Складывание структуры по одному слою за раз

*Катаморфизм* или *складки* , модель примитивной рекурсии. `cata` разрывает фиксированную точку за слоем, используя функцию *алгебры* (или *функцию сгибания* ) для обработки каждого слоя. `cata` требует экземпляр `Functor` для типа шаблона `f` .

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix

-- list example
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    where alg Nil_ = z
          alg (Cons_ x acc) = f x acc
```

## Развертывание структуры по одному слою за раз

*Анаморфизмы* , или *разворачиваются* , моделируют примитивную короберизацию. `ana` создает фиксированную точку за слоем, используя функцию *коалгебры* (или *функцию разворачивания* ) для создания каждого нового слоя. `ana` требует экземпляр `Functor` для типа шаблона `f` .

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- list example
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                          Nothing -> Nil_
                          Just (x, y) -> Cons_ x y
```

Обратите внимание, что `ana` и `cata` являются *двойственными* . Типы и реализации являются зеркальными изображениями друг друга.

## Развертывание, а затем складывание, сплавление

Обычно формировать программу как создание структуры данных, а затем сворачивать ее на одно значение. Это называется *hylomorphism* или *refold* . Для повышения эффективности можно полностью исключить промежуточную структуру.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- no mention of Fix!
```

Вывод:

```
hylo f g = cata g . ana f
         = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- definition of cata and ana
         = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
         = g . fmap (cata g . ana f) . f  -- Functor law
         = g . fmap (hylo f g) . f  -- definition of hylo
```

### Примитивная рекурсия

*Параморфизмы* моделируют примитивную рекурсию. На каждой итерации складки функция складывания получает поддерево для дальнейшей обработки.

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

`tails` Prelude можно моделировать как параморфизм.

```
tails :: List a -> List (List a)
tails = para alg
    where alg Nil_ = cons nil nil  -- [[]]
          alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

### Примитивная обработка

*Апоморфизмы* моделируют примитивную короберизацию. На каждой итерации развертки функция разворачивания может возвращать либо новое семя, либо целое поддерево.

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

Обратите внимание, что `apo` и `para` являются *двойственными* . Стрелки типа перевернуты; кортеж в `para` является двойным по отношению к `Either` в `apo` , а реализации - зеркальными отображениями друг друга.

Прочитайте Рекурсивные схемы онлайн: https://riptutorial.com/ru/haskell/topic/2984/рекурсивные-схемы

# глава 54: Роль

## Вступление

`TypeFamilies` языка `TypeFamilies` позволяет программисту определять функции уровня. Что отличает функции типа от конструкторов типа не-GADT, так это то, что параметры функций типа могут быть непараметрическими, тогда как параметры конструкторов типов всегда являются параметрическими. Это различие важно для правильности расширения `GeneralizedNewTypeDeriving` . Чтобы объяснить это различие, роли внедряются в Haskell.

## замечания

См. Также `SafeNewtypeDeriving` .

## Examples

### Номинальная роль

Haskell Wiki имеет пример непараметрического параметра функции типа:

```
type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool
```

Здесь `x` непараметрический, поскольку для определения результата применения параметра `Inspect` к аргументу типа функция типа должна проверять `x` .

В этом случае роль `x` является номинальной. Мы можем явно объявить эту роль с расширением `RoleAnnotations` :

```
type role Inspect nominal
```

### Представительская роль

Пример параметрического параметра функции типа:

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

type family DoNotInspect x
type instance DoNotInspect x = List x
```

Здесь `x` является параметрическим, потому что для определения результата применения `DoNotInspect` к аргументу типа функция типа не нуждается в проверке `x` .

В этом случае роль x является представительной. Мы можем явно объявить эту роль с расширением `RoleAnnotations` :

```
type role DoNotInspect representational
```

## Фантомная роль

Параметр фантомного типа имеет фантомную роль. Фантомные роли нельзя объявить явно.

Прочитайте Роль онлайн: https://riptutorial.com/ru/haskell/topic/8753/роль

# глава 55: Семейство типов

## Examples

### Типовые синонимы

Семейство синонимов типов - это только функции на уровне шрифта: они связывают типы параметров с типами результатов. Они представлены в трех разных вариантах.

# Закрытые семейства синонимов

Они работают так же, как обычные функции Haskell на уровне значений: вы указываете некоторые предложения, сопоставляя некоторые типы с другими:

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
    Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

# Открытые семейства синонимов типа

Они больше похожи на экземпляры typeclass: каждый может добавлять дополнительные предложения в другие модули.

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- Other instances might appear in other modules, but two instances cannot overlap
-- in a way that would produce different results.
```

# Синонимы, связанные с классом

Семейство открытого типа также можно комбинировать с фактическим классом. Обычно это делается, когда, как и в случае связанных семейств данных , некоторый метод класса нуждается в дополнительных вспомогательных объектах, и эти вспомогательные объекты *могут* быть разными для разных экземпляров, но могут также совместно использоваться. Хорошим примером является класс `VectorSpace` :

```
class VectorSpace v where
  type Scalar v :: *
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

Обратите внимание, что в первых двух случаях реализация `Scalar` одинакова. Это было бы невозможно с ассоциированным семейством данных: семейства данных инъективны , семейства синонимов типа нет.

Хотя неинъективность открывает некоторые возможности, подобные описанным выше, это также затрудняет определение типа. Например, следующее правило не будет выглядеть:

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

В этом случае компилятор не может знать, какой экземпляр использовать, потому что аргумент `bar` является собственно полиморфным литералом `Num` . И `Bar` функции типа не может быть разрешен в «обратном направлении», именно потому, что он не является инъективным [†] и, следовательно, не обратим (может быть более одного типа с `Bar a = String` ).

---

[†] Только с этими двумя экземплярами он фактически *является* инъективным, но компилятор не может знать, что кто-то не добавит больше экземпляров позже и тем самым нарушит поведение.

### Типы данных

Семейства данных могут использоваться для создания типов данных, которые имеют разные реализации на основе их аргументов типа.

# Автономные семейства данных

```
{-# LANGUAGE TypeFamilies #-}
data family List a
data instance List Char = Nil | Cons Char (List Char)
data instance List () = UnitList Int
```

В приведенной выше декларации `Nil :: List Char` и `UnitList :: Int -> List ()`

# Связанные семейства данных

Семейства данных также могут быть связаны с классами. Это часто полезно для типов с «вспомогательными объектами», которые требуются для универсальных методов класса, но должны содержать различную информацию в зависимости от конкретного экземпляра. Например, для индексирования местоположений в списке требуется только одно число, тогда как в дереве вам нужно число, указывающее путь на каждом узле:

```
class Container f where
  data Location f
  get :: Location f -> f a -> Maybe a

instance Container [] where
  data Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | otherwise      = Nothing

instance Container Tree where
  data Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
  get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

### приемистости

Тип Семьи не обязательно инъективны. Поэтому мы не можем вывести параметр из приложения. Например, в `servant` , заданном `Server a` типа `Server a` мы не можем вывести тип `a` . Чтобы решить эту проблему, мы можем использовать `Proxy` . Например, в `servant` функция `serve` имеет тип `... Proxy a -> Server a -> ...` Мы можем сделать вывод , из `a Proxy a` потому , что `Proxy` определяются `data` , которые инъективны.

Прочитайте Семейство типов онлайн: https://riptutorial.com/ru/haskell/topic/2955/семейство-типов

# глава 56: Синтаксис в функциях

## Examples

### гвардия

Функция может быть определена с помощью защитных устройств, которые можно рассматривать как классифицирующие поведение в соответствии с вводом.

Возьмем следующее определение функции:

```
absolute :: Int -> Int  -- definition restricted to Ints for simplicity
absolute n = if (n < 0) then (-n) else n
```

Мы можем переставить его с помощью охранников:

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

В этом контексте в `otherwise` это значимый псевдоним для `True` , поэтому он всегда должен быть последним стражем.

### Соответствие шаблону

Haskell поддерживает выражения соответствия шаблону как в определении функции, так и в операторах `case` .

Заявление о ситуации очень похоже на переключатель на других языках, за исключением того, что он поддерживает все типы Haskell.

Начнем с простого:

```
longName :: String -> String
longName name = case name of
                   "Alex"  -> "Alexander"
                   "Jenny" -> "Jennifer"
                   _       -> "Unknown"   -- the "default" case, if you like
```

Или мы могли бы определить нашу функцию как уравнение, которое было бы сопоставлением шаблонов, просто без использования `case` :

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

Более распространенным примером является тип `Maybe` :

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe can only take `Just a` or `Nothing`, so this wildcard suffices
```

Соответствие шаблону также можно использовать в списках:

```
isEmptyList :: [a] -> Bool
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

Собственно, сопоставление шаблонов можно использовать для любого конструктора для любого типа. Например, конструктор для списков `:` и для кортежей `,`

## Использование где и охранники

Учитывая эту функцию:

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
  | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "Poor child, try to get another job"
  | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
  | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
```

Мы можем использовать `where` , чтобы избежать повторений и сделать наш код более читаемым. Смотрите альтернативную функцию ниже, с помощью `where` :

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "Poor child, try to get another job"
  | annualSalary <= mediumSalary = "Money, Money, Money!"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
  where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

Как было отмечено, мы использовали то, `where` в конце функционального тела устранялось повторение вычисления ( `hourlyRate * (weekHoursOfWork * 52)` ), и мы также использовали, `where` организовать диапазон зарплаты.

Именование общих подвыражения также может быть достигнуто с `let` выражениями, но

только `where` синтаксис позволяет *охранникам* ссылаться на эти именованные подвыражения.

Прочитайте Синтаксис в функциях онлайн: https://riptutorial.com/ru/haskell/topic/3799/синтаксис-в-функциях

# глава 57: Синтаксис вызова функции

## Вступление

Синтаксис вызова функции Haskell, объясненный с помощью сравнений с языками языка C, где это применимо. Это направлено на людей, которые приезжают в Haskell с фона на языках C-стиля.

## замечания

В общем случае правило для преобразования вызова функции C-стиля в Haskell в любом контексте (назначение, возврат или встроенный в другой вызов) заключается в замене запятых в списке аргументов стиля C пробелом и перемещении открытия скобки из вызова стиля C, чтобы содержать имя функции и ее параметры.

Если какие-либо выражения полностью заключены в круглые скобки, эти (внешние) пары круглых скобок могут быть удалены для удобочитаемости, поскольку они не влияют на значение выражения.
Существуют и другие обстоятельства, при которых скобки могут быть удалены, но это влияет только на читаемость и удобство обслуживания.

## Examples

### Скобки в вызове базовой функции

Для вызова функции стиля C, например

```
plus(a, b); // Parentheses surrounding only the arguments, comma separated
```

Тогда эквивалентный код Haskell будет

```
(plus a b) -- Parentheses surrounding the function and the arguments, no commas
```

В Haskell круглые скобки явно не требуются для приложения-приложения и используются только для устранения неоднозначности выражений, например, в математике; поэтому в тех случаях, когда скобки окружают весь текст в выражении, скобки на самом деле не нужны, и следующее также эквивалентно:

```
plus a b -- no parentheses are needed here!
```

Важно помнить, что в языках C-стиля функция

## Круглые скобки во встроенных вызовах функций

В предыдущем примере мы не нуждались в круглых скобках, потому что они не влияли на смысл выражения. Однако они часто необходимы в более сложном выражении, как показано ниже.
В C:

```
plus(a, take(b, c));
```

В Haskell это становится:

```
(plus a (take b c))
-- or equivalently, omitting the outermost parentheses
plus a (take b c)
```

Обратите внимание, что это не эквивалентно:

```
plus a take b c -- Not what we want!
```

Можно подумать, что, поскольку компилятор знает, что функция `take` является функцией, она сможет узнать, что вы хотите применить ее к аргументам `b` и `c` и передать ее результат в `plus` .
Однако в Haskell функции часто принимают другие функции в качестве аргументов, и между фактическими различиями между функциями и другими значениями мало фактического различия; и поэтому компилятор не может принять ваше намерение просто потому, что `take` - это функция.

Итак, последний пример аналогичен следующему вызову функции C:

```
plus(a, take, b, c); // Not what we want!
```

## Частичное применение - часть 1

В Haskell функции могут быть частично применены; мы можем рассматривать все функции как принимающие один аргумент и возвращающие измененную функцию, для которой этот аргумент является постоянным. Чтобы проиллюстрировать это, мы можем скопировать функции следующим образом:

```
(((plus) 1) 2)
```

Здесь функция `(plus)` применяется к `1` давая функцию `((plus) 1)` , которая применяется к `2` , что дает функцию `(((plus) 1) 2)` . Поскольку `plus 1 2` - это функция, которая не принимает аргументов, вы можете считать ее простой; однако в Haskell существует небольшое различие между функциями и значениями.

Чтобы подробнее остановиться, функция `plus` - это функция, которая добавляет свои аргументы.
Функция `plus 1` - это функция, которая добавляет `1` к ее аргументу.
Функция `plus 1 2` - это функция, которая добавляет `1` к `2` , что всегда является значением `3` .

## Частичное применение - часть 2

В качестве другого примера у нас есть `map` функций, которая принимает функцию и список значений и применяет функцию к каждому значению списка:

```
map :: (a -> b) -> [a] -> [b]
```

Предположим, мы хотим увеличить каждое значение в списке. Вы можете определить свою собственную функцию, которая добавляет ее в свой аргумент и `map` эту функцию над вашим списком.

```
addOne x = plus 1 x
map addOne [1,2,3]
```

но если у вас есть другой взгляд на определение `addOne` , с добавлением круглых скобок для акцента:

```
(addOne) x = ((plus) 1) x
```

Функция `addOne` , применяемая к любому значению `x` , такая же, как частично примененная функция `plus 1` примененная к `x` . Это означает, что функции `addOne` и `plus 1` идентичны, и мы можем избежать определения новой функции, просто заменив `addOne` на `plus 1` , не забывая использовать круглые скобки для выделения `plus 1` в качестве подвыражения:

```
map (plus 1) [1,2,3]
```

Прочитайте Синтаксис вызова функции онлайн: https://riptutorial.com/ru/haskell/topic/9615/синтаксис-вызова-функции

# глава 58: Синтаксис записи

## Examples

### Основной синтаксис

Записи - это расширение типа алгебраических `data` суммы, которое позволяет называть поля:

```
data StandardType = StandardType String Int Bool --standard way to create a sum type

data RecordType = RecordType { -- the same sum type with record syntax
    aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

Имена полей могут быть использованы для того, чтобы вывести именованное поле из записи

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
  r :: RecordType
> :t aString
  aString :: RecordType -> String
> aString r
  "Foobar"
```

Записи могут быть сопоставлены с шаблонами

```
case r of
  RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

Обратите внимание, что не все поля должны быть названы

Записи создаются путем присвоения имен их полям, но также могут быть созданы как обычные типы сумм (часто полезно, когда количество полей невелико и вряд ли изменится)

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

Если запись создается без именованного поля, компилятор выдаст предупреждение, и результирующее значение не будет `undefined` .

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
     Fields of RecordType not initialized: isTrue
> isTrue r
```

```
  Error 'undefined'
```

Поле записи можно обновить, установив его значение. Неизменяемые поля не меняются.

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

Часто бывает полезно создавать объективы для сложных типов записей.

## Копирование записей при изменении значений полей

Предположим, что у вас есть этот тип:

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

и два значения:

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

новое значение типа `Person` может быть создано путем копирования из `alex` , указав, какие значения изменить:

```
anotherAlex = alex { age = 31 }
```

Теперь значения `alex` и `anotherAlex` будут следующими:

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

## Записи с новым типом

Синтаксис записи может использоваться с `newtype` с ограничением, что существует ровно один конструктор с ровно одним полем. Преимущество здесь заключается в автоматическом создании функции для развертывания нового типа. Эти поля часто называются начиная с `run` для монадов, `get` для моноидов и `un` для других типов.

```
newtype State s a = State { runState :: s -> (s, a) }

newtype Product a = Product { getProduct :: a }

newtype Fancy = Fancy { unfancy :: String }
  -- a fancy string that wants to avoid concatenation with ordinary strings
```

Важно отметить, что синтаксис записи обычно никогда не используется для

формирования значений, а имя поля используется строго для разворачивания

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
-- > 756
```

## RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}

data Client = Client { firstName     :: String
                     , lastName      :: String
                     , clientID      :: String
                     } deriving (Show)

printClientName :: Client -> IO ()
printClientName Client{..} = do
    putStrLn firstName
    putStrLn lastName
    putStrLn clientID
```

Шаблон `Client{..}` включает в себя все поля `Client` конструктора и эквивалентен шаблону

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

Его также можно комбинировать с другими полями:

```
Client { firstName = "Joe", .. }
```

Это эквивалентно

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

## Определение типа данных с метками поля

Можно определить тип данных с метками поля.

```
data Person = Person { age :: Int, name :: String }
```

Это определение отличается от обычного определения записи, так как оно также определяет * *записи,* которые могут использоваться для доступа к частям типа данных.

В этом примере определены два регистратора доступа, `age` и `name` , которые позволяют нам получить доступ к полям `age` и `name` соответственно.

```
age :: Person -> Int
name :: Person -> String
```

Аксессоры записи - это просто функции Haskell, которые автоматически генерируются

компилятором. Таким образом, они используются как обычные функции Haskell.

По наименованию полей мы также можем использовать метки полей в ряде других контекстов, чтобы сделать наш код более читаемым.

## Соответствие шаблону

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

Мы можем привязать значение, расположенное в позиции соответствующей метки поля, в то время как соответствие шаблону соответствует новому значению (в данном случае `x` ), которое может использоваться в RHS определения.

## Совпадение шаблонов с `NamedFieldPuns`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

Расширение `NamedFieldPuns` вместо этого позволяет нам просто указывать метку поля, с которой мы хотим сопоставлять, это имя затем затеняется на RHS определения, поэтому обращение к `name` относится к значению, а не к аксессуару записи.

## Сравнение шаблонов с `RecordWildcards`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

При сопоставлении с использованием `RecordWildCards` все метки полей вводятся в объем. (В этом конкретном примере `name` и `age` )

Это расширение несколько противоречиво, поскольку неясно, как значения вводятся в область видимости, если вы не уверены в определении `Person` .

## Обновление записей

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

Существует также специальный синтаксис для обновления типов данных с помощью меток полей.

Прочитайте Синтаксис записи онлайн: https://riptutorial.com/ru/haskell/topic/1950/синтаксис-записи

# глава 59: складываемый

## Вступление

`Foldable` - это класс типов `t :: * -> *` которые допускают операцию *складывания* . Сложность агрегирует элементы структуры в четко определенном порядке, используя функцию объединения.

## замечания

Если `t` является `Foldable` это означает, что для любого значения `ta` мы знаем, как получить доступ ко всем элементам `a` от «внутри» `ta` в фиксированном линейном порядке. Это значение `foldMap :: Monoid m => (a -> m) -> (ta -> m)` : мы «посещаем» каждый элемент с помощью сводной функции и разбиваем все сводки вместе. Порядок `Monoid` уважения (но инвариантны к разным группировкам).

## Examples

### Подсчет элементов Складной структуры

`length` учитывает вхождения элементов `a` в складную структуру `ta` .

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' may contain zero or one 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' always contains exactly one 'a'
```

`length` определяется как эквивалентная:

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

Обратите внимание, что этот тип возврата `Int` ограничивает операции, которые могут выполняться на значениях, полученных вызовами функции `length` . `fromIntegral` - полезная функция, которая позволяет нам справиться с этой проблемой.

### Складывание структуры в обратном порядке

Любая сгиб может выполняться в противоположном направлении с помощью `Dual` моноида , который переворачивает существующий моноид, чтобы агрегация шла назад.

```
newtype Dual a = Dual { getDual :: a }

instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

Когда основной моноид вызова `foldMap` перевернут с помощью `Dual` , сгиб бежит назад; следующий тип `Reverse` определяется в `Data.Functor.Reverse` :

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

Мы можем использовать эту технику , чтобы написать сжатый `reverse` для списков:

```
reverse :: [a] -> [a]
reverse = toList . Reverse
```

## Экземпляр Foldable для двоичного дерева

Чтобы создать экземпляр `Foldable` вам необходимо предоставить определение, по крайней мере, `foldMap` или `foldr` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Foldable Tree where
    foldMap f Leaf = mempty
    foldMap f (Node l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f acc Leaf = acc
    foldr f acc (Node l x r) = foldr f (f x (foldr f acc r)) l
```

Эта реализация выполняет обход дерева в порядке.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

Расширение `DeriveFoldable` позволяет GHC генерировать `Foldable` экземпляры на основе

структуры типа. Мы можем изменить порядок машинного обхода путем настройки макета конструктора `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving Foldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving Foldable

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving Foldable

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> toList (inorder myTree)
"abc"
ghci> toList (preorder myTree)
"bac"
ghci> toList (postorder myTree)
"acb"
```

## Сглаживание Складной структуры в список

`toList` сглаживается в `Foldable` структуру `ta` в список `a` сек.

```
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

`toList` определяется как эквивалент:

```
class Foldable t where
    -- ...
    toList :: t a -> [a]
    toList = foldr (:) []
```

## Выполнение побочного эффекта для каждого элемента Складной структуры

`traverse_` выполняет `Applicative` действие для каждого элемента в `Foldable` структуре. Он игнорирует результат действия, сохраняя только побочные эффекты. (Для версии, которая не отбрасывает результаты, используйте `Traversable` .)

```
-- using the Writer applicative functor (and the Sum monoid)
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- using the IO applicative functor
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- nothing printed
```

`for_` является `traverse_` с аргументами перевернута. Он похож на цикл `foreach` на императивном языке.

```
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

`sequenceA_` сворачивает `Foldable` полную `Applicative` действия в одно действие, игнорируя результат.

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
two
```

`traverse_` определяется как эквивалент:

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

`sequenceA_` определяется как:

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

Более того, когда `Foldable` также является `Functor` , `traverse_` и `sequenceA_` имеют следующее соотношение:

```
traverse_ f = sequenceA_ . fmap f
```

## Сглаживание сложенной структуры в моноид

`foldMap` отображает каждый элемент Складной структуры в `Monoid` , а затем объединяет их в одно значение.

`foldMap` и `foldr` могут быть определены в терминах друг друга, а это означает, что для экземпляров `Foldable` требуется только определение для одного из них.

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

Пример использования моноидного `Product` :

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

## Определение складных

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- and a number of optional methods
```

Интуитивно (хотя и не технически), `Foldable` структуры представляют собой контейнеры элементов `a` которые обеспечивают доступ к их элементам в четко определенном порядке. Операция `foldMap` сопоставляет каждый элемент контейнера с `Monoid` и сворачивает их с использованием структуры `Monoid` .

## Проверка того, что Складная структура пуста

`null` возвращает `True` если в складной структуре `ta` нет элементов `a` и `False` если есть один или несколько. Структуры, для которых `null` имеет значение `True` имеют `length` 0.

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
```

```
False
ghci> null ('x', 3)
False
```

`null` определяется как эквивалентное:

```
class Foldable t where
    -- ...
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

Прочитайте складываемый онлайн: https://riptutorial.com/ru/haskell/topic/753/складываемый

# глава 60: совпадение

## замечания

Хорошими ресурсами для изучения параллельного и параллельного программирования в Haskell являются:

- Параллельное и параллельное программирование в Haskell
- Haskell Wiki

## Examples

### Нерестные нити с `forkIO`

Haskell поддерживает множество форм параллелизма, и наиболее очевидным является разветвление потока с использованием `forkIO` .

Функция `forkIO :: IO () -> IO ThreadId` принимает действие `IO` и возвращает свой `ThreadId` , в то время как действие будет выполняться в фоновом режиме.

Мы можем продемонстрировать это довольно лаконично с помощью `ghci` :

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..100000000]
ThreadId 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- some time later....
Prelude Control.Concurrent> 50000005000000
```

Оба действия будут выполняться в фоновом режиме, а второй почти гарантированно завершится до последнего!

### Связь между потоками с помощью MVARD

Очень легко передавать информацию между потоками с `MVar a` типа `MVar a` и его сопутствующих функций в `Control.Concurrent` :

- `newEmptyMVar :: IO (MVar a)` - создает новый `MVar a`
- `newMVar :: a -> IO (MVar a)` - создает новый `MVar` с заданным значением
- `takeMVar :: MVar a -> IO a` - извлекает значение из данного `MVar` или **блокирует** до тех пор, пока не будет доступно
- `putMVar :: MVar a -> a -> IO ()` - помещает заданное значение в `MVar` или **блокирует** до тех пор, пока оно не будет пустым

Давайте суммируем числа от 1 до 100 миллионов в потоке и ожидаем результата:

```
import Control.Concurrent
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar will block 'til m is non-empty!
```

Более сложная демонстрация может заключаться в том, чтобы принимать пользовательский ввод и суммирование в фоновом режиме, ожидая большего ввода:

```
main2 = loop
  where
    loop = do
        m <- newEmptyMVar
        n <- getLine
        putStrLn "Calculating. Please wait"
        -- In another thread, parse the user input and sum
        forkIO $ putMVar m $ sum [1..(read n :: Int)]
        -- In another thread, wait 'til the sum's complete then print it
        forkIO $ print =<< takeMVar m
        loop
```

Как было сказано ранее, если вы назовете `takeMVar` и `MVar` пуст, он блокируется до тех пор, пока другая нить не помещает что-то в `MVar` , что может привести к проблеме обеденных `MVar` . То же самое происходит и с `putMVar` : если он заполнен, он блокирует «пока он не будет пустым!

Возьмите следующую функцию:

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

Мы запускаем две функции с некоторыми `MVar` s

```
concurrent ma mb     -- new thread 1
concurrent mb ma     -- new thread 2
```

Что может случиться, так это то, что:

1. Тема 1 читает `ma` и блокирует `ma`
2. Thread 2 читает `mb` и, таким образом, блокирует `mb`

Теперь Thread 1 не может прочитать `mb` поскольку Thread 2 заблокировал его, и Thread 2 не может прочитать `ma` поскольку Thread 1 заблокировал его. Классический тупик!

## Атомные блоки с программной транзакционной памятью

Еще одним мощным и зрелым инструментом параллелизма в Haskell является программная транзакционная память, которая позволяет нескольким потокам записывать в одну переменную типа `TVar a` атомным способом.

`TVar a` является основным типом, связанным с монадой `STM` и выступает за транзакционную переменную. Они используются так же, как `MVar` но в монаде `STM` помощью следующих функций:

**`atomically :: STM a -> IO a`**

Выполнять серию действий STM атомарно.

**`readTVar :: TVar a -> STM a`**

Прочтите значение `TVar` , например:

```
value <- readTVar t
```

**`writeTVar :: TVar a -> a -> STM ()`**

Напишите значение для данного `TVar` .

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

Этот пример взят из Haskell Wiki:

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- Initialise a new TVar
  shared <- atomically $ newTVar 0
  -- Read the value
  before <- atomRead shared
  putStrLn $ "Before: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
  after <- atomRead shared
  putStrLn $ "After: " ++ show after
  where timesDo = replicateM_
        milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

Прочитайте совпадение онлайн: https://riptutorial.com/ru/haskell/topic/4426/совпадение

# глава 61: Создание пользовательских типов данных

## Examples

### Создание простого типа данных

Самый простой способ создать пользовательский тип данных в Haskell - использовать ключевое слово `data` :

```
data Foo = Bar | Biz
```

Имя типа указывается между `data` и `=` и называется **конструктором типа** . После `=` мы указываем все **конструкторы** значений нашего типа данных, разделенные символом `|` знак. В Haskell существует правило, что все конструкторы типов и значений должны начинаться с заглавной буквы. Вышеуказанное заявление можно читать следующим образом:

> Определите тип, называемый `Foo` , который имеет два возможных значения: `Bar` и `Biz` .

## Создание переменных нашего пользовательского типа

```
let x = Bar
```

Вышеприведенный оператор создает переменную с именем `x` типа `Foo` . Давайте проверим это, проверив его тип.

```
:t x
```

печать

```
x :: Foo
```

### Создание типа данных с параметрами конструктора значения

Конструкторы значений - это функции, возвращающие значение типа данных. Из-за этого, как и любая другая функция, они могут принимать один или несколько параметров:

```
data Foo = Bar String Int | Biz String
```

Давайте проверим тип конструктора значений `Bar` .

```
:t Bar
```

печать

```
Bar :: String -> Int -> Foo
```

что доказывает, что `Bar` действительно является функцией.

# Создание переменных нашего пользовательского типа

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

### Создание типа данных с параметрами типа

Конструкторы типов могут принимать один или несколько параметров типа:

```
data Foo a b = Bar a b | Biz a b
```

Параметры типа в Haskell должны начинаться с строчной буквы. Наш пользовательский тип данных еще не является реальным типом. Чтобы создать значения нашего типа, мы должны подставить все параметры типа с фактическими типами. Поскольку `a` и `b` могут быть любого типа, наши конструкторы значений являются полиморфными функциями.

# Создание переменных нашего пользовательского типа

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

### Пользовательский тип данных с параметрами записи

Предположим, мы хотим создать тип данных Person, у которого есть имя и фамилия, возраст, номер телефона, улица, почтовый индекс и город.

Мы могли бы написать

```
data Person = Person String String Int Int String String String
```

Если мы хотим получить номер телефона, нам нужно сделать функцию

```
getPhone :: Person -> Int
```

```
getPhone (Person _ _ _ phone _ _ _) = phone
```

Ну, это не весело. Мы можем лучше использовать параметры:

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
                       , age           :: Int
                       , phone         :: Int
                       , street        :: String
                       , code          :: String
                       , town          :: String }
```

Теперь мы получаем функцию `phone` где

```
:t phone
phone :: Person' -> Int
```

Теперь мы можем делать все, что захотим, например:

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

Мы также можем связать номер телефона с помощью сопоставления с образцом :

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

Для удобства использования параметров см. RecordWildCards

Прочитайте Создание пользовательских типов данных онлайн:
https://riptutorial.com/ru/haskell/topic/4057/создание-пользовательских-типов-данных

# глава 62: Состав функции

## замечания

Оператор композиции функций `(.)` Определяется как

```
(.) :: (b -> c) -> (a -> b) ->  (a -> c)
(.)       f           g          x =  f (g x)      -- or, equivalently,

(.)       f           g      =   \x -> f (g x)
(.)       f      =    \g     ->  \x -> f (g x)
(.) =    \f      ->   \g     ->  \x -> f (g x)
(.) =    \f      ->  (\g     -> (\x -> f (g x) ) )
```

Тип `(b -> c) -> (a -> b) -> (a -> c)` можно записать в виде `(b -> c) -> (a -> b) -> a -> c` так как `->` в подписях типа «связывает» вправо, соответствующие прикладной функции, связанной с левым,

```
 f g x y z ...    ==    (((f g) x) y) z ...
```

Таким образом, «поток данных» находится справа налево: `x` «идет» в `g` , результат которого переходит в `f` , давая конечный результат:

```
(.)       f           g          x =  r
                                      where r = f (g x)
-- g :: a -> b
-- f ::      b -> c
-- x :: a
-- r ::           c

(.)       f           g      =   q
                                 where q = \x -> f (g x)
-- g :: a -> b
-- f ::      b -> c
-- q :: a      -> c

....
```

Синтаксически все одно и то же:

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

который легко понять как «три правила секций оператора », где «отсутствующий аргумент» просто входит в пустой слот рядом с оператором:

```
(.) f g    =  (f . g)    =  (f .) g    =  (. g) f
--         1             2             3
```

`x` , находящийся по обе стороны от уравнения, можно опустить. Это известно как eta-сокращение. Таким образом, простой способ записать определение состава функции - это просто

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

Это, конечно, относится к «аргументу» `x` ; всякий раз, когда мы пишем только `(f . g)` без `x` он известен как стиль без точек.

# Examples

## Композиция справа налево

`(.)` позволяет нам составлять две функции, подавая выход одного в качестве входа в другой:

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

Например, если мы хотим сместить преемника входного числа, мы можем написать

```
((^2) . succ) 1        --    4
```

Существует также `(<<<)` который является псевдонимом `(.)` . Так,

```
(+ 1) <<< sqrt $ 25    --    6
```

## Композиция слева направо

`Control.Category` определяет `(>>>)` , который, когда специализирован для функций, является

```
-- (>>>) :: Category cat => cat a b -> cat b c -> cat a c
-- (>>>) :: (->) a b -> (->) b c -> (->) a c
-- (>>>) :: (a -> b) -> (b -> c) -> (a -> c)
( f >>> g ) x = g (f x)
```

Пример:

```
sqrt >>> (+ 1) $ 25    --    6.0
```

## Композиция с бинарной функцией

Регулярная композиция работает для унарных функций. В случае двоичного кода мы можем определить

```
(f .: g) x y = f (g x y)          -- which is also
```

```
             = f ((g x) y)
             = (f . g x) y         -- by definition of (.)
             = (f .) (g x) y
             = ((f .) . g) x y
```

Таким образом, `(f .: g) = ((f .) . g)` по eta-сокращению, и, кроме того,

```
(.:) f g    = ((f .) . g)
            = (.) (f .) g
            = (.) ((.) f) g
            = ((.) . (.)) f g
```

так что `(.:) = ((.) . (.))` , полуизвестное определение.

Примеры:

```
(map (+1) .: filter) even [1..5]      --  [3,5]
(length   .: filter) even [1..5]      --  2
```

Прочитайте Состав функции онлайн: https://riptutorial.com/ru/haskell/topic/4430/состав-функции

# глава 63: Списки

## Синтаксис

1. пустой конструктор списка

   ```
   [] :: [a]
   ```

2. конструктор непустого списка

   ```
   (:) :: a -> [a] -> [a]
   ```

3. head - возвращает первое значение списка

   ```
   head :: [a] -> a
   ```

4. last - возвращает последнее значение списка

   ```
   last :: [a] -> a
   ```

5. tail - возвращает список без первого элемента

   ```
   tail :: [a] -> [a]
   ```

6. init - возвращает список без последнего элемента

   ```
   init :: [a] -> [a]
   ```

7. xs !! i - вернуть элемент по индексу i в списке xs

   ```
   (!!) :: Int -> [a] -> a
   ```

8. взять n xs - вернуть новый список, содержащий n первых элементов списка xs

   ```
   take :: Int -> [a] -> [a]
   ```

9. map :: (a -> b) -> [a] -> [b]

10. filter :: (a -> Bool) -> [a] -> [a]

11. (++) :: [a] -> [a]

12. concat :: [[a]] -> [a]

## замечания

1. Тип `[a]` эквивалентен `[] a` .
2. `[]` создает пустой список.
3. `[]` в определении функции LHS, например `f [] = ...` , является пустым шаблоном списка.

4. `x:xs` создает список, в котором элемент `x` добавляется в список `xs`
5. `f (x:xs) = ...` является совпадением шаблонов для непустого списка, где `x` - голова, а `xs` - хвост.
6. `f (a:b:cs) = ...` и `f (a:(b:cs)) = ...` - то же самое. Они соответствуют шаблону для списка, по меньшей мере, двух элементов, где первым элементом является `a` , второй - `b` , а остальная часть списка - `cs` .
7. `f ((a:as):bs) = ...` не совпадает с `f (a:(as:bs)) = ...` Первый - это соответствие шаблону для непустого списка списков, где `a` - голова головы, `as` также хвост головы, а `bs` - хвост.
8. `f (x:[]) = ...` и `f [x] = ...` совпадают. Они соответствуют шаблону для списка только одного элемента.
9. `f (a:b:[]) = ...` и `f [a,b] = ...` совпадают. Они соответствуют шаблону для списка ровно двух элементов.
10. `f [a:b] = ...` - соответствие шаблона для списка точно одного элемента, где элемент также является списком. `a` - голова элемента, а `b` - хвост элемента.
11. `[a,b,c]` совпадает с `(a:b:c:[])` . Стандартное обозначение списка - это просто синтаксический сахар для конструкторов `(:)` и `[]` .
12. Вы можете использовать `all@(x:y:ys)` , чтобы ссылаться на весь список, как `all` (или любое другое имя, которое вы выбираете), вместо повторения `(x:y:ys)` .

# Examples

## Список литературы

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

## Конкатенация списка

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

(++) xs     [] = xs
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

## Основы списка

Конструктор типов для списков в Haskell Prelude - `[]` . Объявление типа для списка, содержащего значения типа `Int` , записывается следующим образом:

```
xs :: [Int]    -- or equivalently, but less conveniently,
xs :: [] Int
```

Списки в Haskell являются *однородными* *последовательностями* , то есть все элементы должны быть одного типа. В отличие от кортежей, тип списка не зависит от длины:

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

Списки создаются с использованием двух конструкторов :

- `[]` создает пустой список.
- `(:)` , произносится как «cons», добавляет элементы в список. Заканчивая `x` (значение типа `a` ) на `xs` (список значений одного и того же типа `a` ) создает новый список, *голова* которого (первый элемент) равна `x` , а *хвост* (остальные элементы) - `xs` .

Мы можем определить простые списки следующим образом:

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []     -- ((:) is right-associative)
-- or  = [12, 99, 37]          -- (syntactic sugar for lists)
```

Обратите внимание: `(++)` , который может использоваться для построения списков, определяется рекурсивно в терминах `(:)` и `[]` .

## Списки обработки

Для обработки списков мы можем просто сопоставить шаблон конструкторам типа списка:

```
listSum :: [Int] -> Int
listSum []          = 0
listSum (x:xs) = x + listSum xs
```

Мы можем сопоставить больше значений, указав более сложный шаблон:

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

Обратите внимание, что в приведенном выше примере нам нужно было предоставить более

исчерпывающее соответствие шаблонов для обработки случаев, когда в качестве аргумента приводится список нечетных длин.

Haskell Prelude определяет множество встроенных модулей для обработки списков, таких как `map` , `filter` и т. Д. По возможности, вы должны использовать их вместо написания собственных рекурсивных функций.

## Доступ к элементам в списках

Доступ к *n*- му элементу списка (с нулевым основанием):

```
list = [1 .. 10]

firstElement = list !! 0          -- 1
```

Обратите внимание, что `!!` является частичной функцией, поэтому определенные входы создают ошибки:

```
list !! (-1)     -- *** Exception: Prelude.!!: negative index

list !! 1000     -- *** Exception: Prelude.!!: index too large
```

Там также `Data.List.genericIndex` , перегруженная версия `!!` , который принимает любое `Integral` значение как индекс.

```
import Data.List (genericIndex)

list `genericIndex` 4             -- 5
```

Когда они выполняются в виде односвязных списков, эти операции занимают время *O (n)* . Если вы часто `Data.Vector` к элементам по индексу, вероятно, лучше использовать `Data.Vector` (из векторного пакета) или другие структуры данных.

## Изменяется

Создание списка с 1 по 10 прост с использованием нотации диапазона:

```
[1..10]    -- [1,2,3,4,5,6,7,8,9,10]
```

Чтобы указать шаг, добавьте запятую и следующий элемент после элемента start:

```
[1,3..10]  -- [1,3,5,7,9]
```

Обратите внимание, что Haskell всегда делает шаг как арифметическую разницу между терминами и что вы не можете указать больше, чем первые два элемента и верхнюю границу:

```
[1,3,5..10] -- error
[1,3,9..20] -- error
```

Чтобы создать диапазон в порядке убывания, всегда указывайте отрицательный шаг:

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

Поскольку Haskell не является строгим, элементы списка оцениваются только в том случае, если они необходимы, что позволяет нам использовать бесконечные списки. `[1..]` - это бесконечный список, начинающийся с 1. Этот список может быть привязан к переменной или передан как аргумент функции:

```
take 5 [1..]   -- returns [1,2,3,4,5] even though [1..] is infinite
```

Будьте осторожны при использовании диапазонов с плавающей запятой, потому что он допускает переполнения до половины дельта, чтобы предотвратить проблемы округления:

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5] , though 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002] , though 1.2000000000000002 > 1.2
```

Диапазоны работают не только с числами, но и с любым типом, который реализует класс `Enum` . Учитывая некоторые перечислимые переменные `a` , `b` , `c` , синтаксис диапазона эквивалентен вызову этих методов `Enum` :

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

Например, с `Bool` это

```
 [False ..]      -- [False,True]
```

Обратите внимание на пробел после `False` , чтобы это не анализировалось как квалификация имени модуля (т.е. `False..` будет анализироваться как `.` Из модуля `False` ).

## Основные функции в списках

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]
```

```
length [1 .. 10]   --    10

reverse [1 .. 10]  --    [10, 9 .. 1]

take 5 [1, 2 .. ]  --    [1, 2, 3, 4, 5]

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

### foldl

Так реализуется левая складка. Обратите внимание, как порядок аргументов в шаговой функции перевернут по сравнению с `foldr` (правая сфера):

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     =  acc
foldl f acc (x:xs) =  foldl f (f acc x) xs         -- = foldl f (acc `f` x) xs
```

Левая складка, `foldl` , ассоциируется слева. То есть:

```
foldl (+) 0 [1, 2, 3]     -- is equivalent to ((0 + 1) + 2) + 3
```

Причина в том, что `foldl` оценивается следующим образом (посмотрите на индуктивный шаг `foldl` ):

```
foldl (+) 0 [1, 2, 3]                        --  foldl (+)    0   [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                   --  foldl (+)   (0 + 1)   [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]              --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []       --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                    --            (((0 + 1) + 2) + 3)
```

Последняя строка эквивалентна `((0 + 1) + 2) + 3` . Это потому, что `(fab)` совпадает с `(a `f` b)` вообще, и поэтому `((+) 0 1)` в `(a `f` b)` совпадает с `(0 + 1)` .

### foldr

Так реализуется правильная складка:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)              -- = x `f` foldr f z xs
```

Правая складка, `foldr` , ассоциируется справа. То есть:

```
foldr (+) 0 [1, 2, 3]      -- is equivalent to 1 + (2 + (3 + 0))
```

Причина в том, что `foldr` оценивается следующим образом (посмотрите на индуктивный шаг `foldr` ):

```
foldr (+) 0 [1, 2, 3]                     --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))           -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))    -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                   -- 1 + (2 + (3 +            0   ))
```

Последняя строка эквивалентна `1 + (2 + (3 + 0))` , так как `((+) 3 0)` совпадает с `(3 + 0)` .

## Преобразование с помощью `map`

Часто мы хотим преобразовать или преобразовать содержимое коллекции (список или что-то проходящее). В Haskell мы используем `map` :

```
-- Simple add 1
map (+ 1) [1,2,3]
[2,3,4]

map odd [1,2,3]
[True,False,True]

data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

-- Extract just the age from a list of people
map (\(Person n g a) -> a) [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
[31,29]
```

## Фильтрация с помощью `filter`

Учитывая список:

```
li = [1,2,3,4,5]
```

мы можем отфильтровать список с предикатом, используя `filter :: (a -> Bool) -> [a] -> [a]` :

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- Something slightly more complicated
comfy i = notTooLarge && isEven
  where
     notTooLarge = (i + 1) < 5
     isEven = even i

filter comfy li        -- [2]
```

Конечно, речь идет не только о цифрах:

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
    isFemale (Person _ Female _) = True
    isFemale _ = False

onlyLadies [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
-- [Person "Ellie" Female 29]
```

## Распаковка и распаковка списков

zip принимает два списка и возвращает список соответствующих пар:

```
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs

> zip [1,3,5] [2,4,6]
> [(1,2),(3,4),(5,6)]
```

Закрепление двух списков функцией:

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

Распаковка списка:

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

> unzip [(1,2),(3,4),(5,6)]
> ([1,3,5],[2,4,6])
```

Прочитайте Списки онлайн: https://riptutorial.com/ru/haskell/topic/2281/списки

# глава 64: Список рекомендаций

## Examples

### Основные сведения о списках

В Haskell есть списки , которые очень похожи на набор понятий в математике и аналогичных реализациях на императивных языках, таких как Python и JavaScript. В большинстве своих основных понятий списков принимают следующую форму.

```
[ x | x <- someList ]
```

Например

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

Функции могут быть непосредственно применены к x:

```
[ f x | x <- someList ]
```

Это эквивалентно:

```
map f someList
```

Пример:

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

### Шаблоны выражений генератора

Однако `x` в выражении генератора не просто переменна, но может быть любым шаблоном. В случае несоответствия шаблона сгенерированный элемент пропускается, а обработка списка продолжается следующим элементом, действуя, как фильтр:

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

Генератор с переменной `x` в своем шаблоне создает новую область, содержащую все выражения справа, где `x` определяется как сгенерированный элемент.

Это означает, что охранники могут быть закодированы как

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

### гвардия

Еще одна особенность списков - это защита, которая также действует как фильтры. Охранники являются булевыми выражениями и отображаются в правой части панели в понимании списка.

Их основное использование

```
[x    | p x]   ===   if p x then [x] else []
```

Любая переменная, используемая в охраннике, должна появляться слева в понимании или иначе находиться в области видимости. Так,

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` must be defined in outer scope
```

что эквивалентно

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- or,

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)

-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(оператор `>>=` является `infixl 1` , т. е. сопоставляет (заключен в скобки) влево). Примеры:

```
[ x       | x <- [1..4], even x]           -- [2,4]

[ x^2 + 1 | x <- [1..100], even x ]        -- map (\x -> x^2 + 1) (filter even [1..100])
```

### Вложенные генераторы

В списках можно также нарисовать элементы из нескольких списков, и в этом случае результатом будет список всех возможных сочетаний двух элементов, как если бы два списка были обработаны *вложенным* образом. Например,

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

### Параллельное понимание

С расширением языка Parallel List понимает ,

```
[(x,y) | x <- xs | y <- ys]
```

эквивалентно

```
zip xs ys
```

Пример:

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

## Локальные привязки

В списках можно ввести локальные привязки для переменных для хранения некоторых промежуточных значений:

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]    -- [(1,2),(3,10)]
```

Тот же эффект может быть достигнут с помощью трюка,

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]   -- [(1,2),(3,10)]
```

Способы `let` в список являются рекурсивными, как обычно. Но привязки генераторов нет, что позволяет *затенять* :

```
[x | x <- [1..4], x <- [x*x+1], even x]       -- [2,10]
```

## Обозначать

Любой список понимание может быть соответствующим образом закодированы список монады `do` запись .

```
[f x | x <- xs]                f  <$> xs        do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]       fs <*> xs        do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]      f  =<< xs        do { x <- xs ; y <- f x ; return y }
```

Охранники могут обрабатываться с помощью `Control.Monad.guard` :

```
[x   | x <- xs, even x]                         do { x <- xs ; guard (even x) ; return x }
```

Прочитайте Список рекомендаций онлайн: https://riptutorial.com/ru/haskell/topic/4970/список-рекомендаций

# глава 65: стек

## Examples

### Установка стека

**Mac OS X**

Использование Homebrew :

```
brew install haskell-stack
```

### Создание простого проекта

Чтобы создать проект под названием **helloworld** run:

```
stack new helloworld simple
```

Это создаст каталог `helloworld` с файлами, необходимыми для проекта Stack.

### Состав

# Файловая структура

Простой проект содержит следующие файлы:

```
➔ helloworld ls
LICENSE      Setup.hs      helloworld.cabal src          stack.yaml
```

В папке `src` есть файл с именем `Main.hs` Это «отправная точка» проекта `helloworld` . По умолчанию `Main.hs` содержит простой «Hello, World!». программа.

**Main.hs**

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

# Запуск программы

Убедитесь, что вы находитесь в каталоге `helloworld` и запускаете:

```
stack build # Compile the program
stack exec helloworld # Run the program
# prints "hello world"
```

### Пакеты Stackage и изменение версии LTS (resolver)

Stackage - это хранилище пакетов Haskell. Мы можем добавить эти пакеты в проект стека.

# Добавление объектива в проект.

В проекте стека есть файл с именем `stack.yaml` . В `stack.yaml` есть сегмент, который выглядит так:

```
resolver: lts-6.8
```

Stackage хранит список пакетов для каждой ревизии `lts` . В нашем случае нам нужен список пакетов для `lts-6.8` Чтобы найти эти пакеты, посетите:

```
https://www.stackage.org/lts-6.8 # if a different version is used, change 6.8 to the correct
resolver number.
```

Просматривая пакеты, есть объектив-4.13 .

Теперь мы можем добавить языковой пакет, изменив раздел `helloworld.cabal` :

```
  build-depends: base >= 4.7 && < 5
```

чтобы:

```
  build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

Очевидно, что если мы хотим изменить новый LTS (после его выхода), мы просто изменим номер резольвера, например:

```
resolver: lts-6.9
```

При следующей структуре `stack build` Stack будет использовать версию LTS 6.9 и, следовательно, загружать некоторые новые зависимости.

### Создание и запуск проекта стека

В этом примере имя нашего проекта - «helloworld», который был создан с помощью `stack new`

```
helloworld simple
```

Сначала мы должны построить проект со `stack build` а затем мы можем запустить его с помощью

```
stack exec helloworld-exe
```

## Установка стека

Запустив команду

```
stack install
```

Стек скопирует исполняемый файл в папку

```
/Users/<yourusername>/.local/bin/
```

## Профилирование стеком

Настройте профилирование для проекта через `stack` . Сначала создайте проект с помощью флага `--profile` :

```
stack build --profile
```

Флаги GHC не требуются в файле cabal для работы (например, `-prof` ). `stack` автоматически включит профилирование как для библиотеки, так и для исполняемых файлов в проекте. В следующий раз, когда исполняемый файл запускается в проекте, можно использовать обычные `+RTS` флаги:

```
stack exec -- my-bin +RTS -p
```

## Просмотр зависимостей

Чтобы узнать, от каких пакетов зависит ваш проект, вы можете просто использовать эту команду:

```
stack list-dependencies
```

Таким образом, вы можете узнать, какую версию ваших зависимостей вы фактически вытащили из стека.

Проекты Haskell часто оказываются во многих библиотеках косвенно, и иногда эти внешние зависимости вызывают проблемы, которые необходимо отслеживать. Если вы обнаружите, что вы хотите идентифицировать внешнюю зависимость изгоев, вы можете выполнить grep через весь график зависимостей и определить, какая из ваших зависимостей в конечном

итоге вытягивает нежелательный пакет:

```
stack dot --external | grep template-haskell
```

`stack dot` выводит граф зависимостей в текстовой форме, которую можно искать. Его также можно посмотреть:

```
stack dot --external | dot -Tpng -o my-project.png
```

Вы также можете установить глубину графика зависимостей, если хотите:

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

Прочитайте стек онлайн: https://riptutorial.com/ru/haskell/topic/2970/стек

# глава 66: Стрелы

## Examples

### Функциональные композиции с несколькими каналами

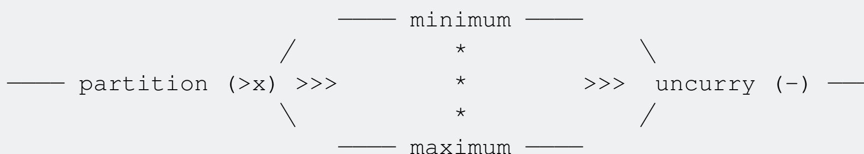
`Arrow` , смутно говоря, является классом морфизмов, которые составляют подобные функции, как с последовательным составом, так и с «параллельным составом». Хотя это наиболее интересно как *обобщение* функций, сам экземпляр `Arrow (->)` уже весьма полезен. Например, следующая функция:

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 where (greater, smaller) = partition (>x) ys
```

также могут быть записаны с помощью комбинаторов стрелок:

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

Этот вид композиции лучше всего визуализировать с помощью диаграммы:




Вот,

- Оператор `>>>` - это просто перевернутая версия обычного `.` (существует также версия `<<<` которая составляет справа налево). Он передает данные с одного этапа обработки на другой.
- исходящий `/` `\` указывает, что поток данных разделен на два «канала». В терминах типов Haskell это реализуется с помощью кортежей:

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  разделяет поток по двум `[Double]` каналам, тогда как

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  объединяет два `Double` канала.
- `***` является оператором параллельной композиции †. Он позволяет `maximum` и `minimum`

работать независимо на разных каналах данных. Для функций сигнатура этого оператора равна

```
(***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
```

---

† По крайней мере, в категории **Hask** (т. **E. B** экземпляре `Arrow (->)` ) `f***g` фактически не вычисляет `f` и `g` параллельно, как в, на разных потоках. Теоретически это было бы возможно.

Прочитайте Стрелы онлайн: https://riptutorial.com/ru/haskell/topic/4912/стрелы

# глава 67: Теория категорий

## Examples

### Теория категорий как система для организации абстракции

Теория категорий - современная математическая теория и ветвь абстрактной алгебры, ориентированная на природу связности и отношения. Он полезен для создания прочных основ и общего языка для многих абстракций с большим количеством повторного использования. Haskell использует теорию категорий как вдохновение для некоторых основных классов, доступных как в стандартной библиотеке, так и в нескольких популярных сторонних библиотеках.

# Пример

`Functor` класс типов говорит , что если тип `F` конкретизирует `Functor` (для которых мы пишем `Functor F` ) , то мы имеем общую операцию

```
fmap :: (a -> b) -> (F a -> F b)
```

который позволяет нам «отображать» по `F` Стандартная (но несовершенная) интуиция заключается в том, что `F a` - контейнер, полный значений типа `a` и `fmap` позволяет применить преобразование к каждому из этих содержащихся элементов. Примером может `Maybe`

```
instance Functor Maybe where
  fmap f Nothing = Nothing     -- if there are no values contained, do nothing
  fmap f (Just a) = Just (f a) -- else, apply our transformation
```

Учитывая эту интуицию, общий вопрос: «Почему бы не назвать `Functor` чем-то очевидным, как `Mappable` ?».

# Подсказка теории категорий

Причина в том, что Functor вписывается в набор общих структур теории категорий, и поэтому, вызывая `Functor` «Functor», мы можем видеть, как он соединяется с этим более глубоким телом знаний.

В частности, Теория категорий очень озабочена идеей стрелок из одного места в другое. В Haskell самым важным набором стрелок являются стрелки функции `a -> b` . Общим для изучения в теории категорий является то, как один набор стрелок относится к другому

набору. В частности, для любого конструктора типов `F` также интересно множество стрелок формы `F a -> F b`.

Таким образом, Functor - любое `F` такое, что существует связь между нормальными стрелками Haskell `a -> b` и `F` специфическими стрелками `F a -> F b`. Соединение определяется `fmap` и мы также признаем несколько законов, которые должны выполняться

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

Все эти закономерности возникают, естественно, из теоретической интерпретации `Functor` и не были бы столь же очевидными, если бы мы только подумали о том, что `Functor` относится к «отображению элементов».

## Определение категории

Категория `C` состоит из:

- Коллекция объектов, называемых `Obj(C)`;
- Коллекция (называемая `Hom(C)`) морфизмов между этими объектами. Если `a` и `b` находятся в `Obj(C)`, то морфизм `f` в `Hom(C)` обычно обозначается `f : a -> b`, а совокупность всех морфизмов между `a` и `b` обозначается `hom(a,b)`;
- Специальный морфизм, называемый *тождественным* морфизмом, для каждого `a : Obj(C)` существует морфизм `id : a -> a`;
- Оператор композиции (`.`), Взяв два морфизма `f : a -> b`, `g : b -> c` и создавая морфизм `a -> c`

которые подчиняются следующим законам:

```
For all f : a -> x, g : x -> b, then id . f = f and g . id = g
```

```
For all f : a -> b, g : b -> c and h : c -> d, then h . (g . f) = (h . g) . f
```

Другими словами, композиция с тождественным морфизмом (слева или справа) не меняет другого морфизма, а композиция ассоциативна.

В Haskell `Category` определяется как класс в Control.Category :

```
-- | A class for categories.
--   id and (.) must form a monoid.
class Category cat where
    -- | the identity morphism
    id :: cat a a

    -- | morphism composition
    (.) :: cat b c -> cat a b -> cat a c
```

В этом случае `cat :: k -> k -> *` объективирует отношение морфизма - существует морфизм `cat ab` тогда и только тогда, когда `cat ab` заселен (т.е. имеет значение). `a` , `b` и `c` все находятся в `Obj(C)` . Сама `Obj(C)` представляется *видом* `k` - например, когда `k ~ *` , как обычно, объекты являются типами.

Каноническим примером категории в Haskell является категория функций:

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

Другим распространенным примером является `Category` стрелок `Kleisli` для `Monad` :

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli f . Kleisli g = Kleisli (f >=> g)
```

### Типы Haskell как категория

## Определение категории

Типы Haskell вместе с функциями между типами образуют (почти †) категорию. Мы имеем тождественный морфизм (функция) ( `id :: a -> a` ) для каждого объекта (типа) `a` ; и состав морфизмов ( `(.) :: (b -> c) -> (a -> b) -> a -> c` ), которые подчиняются законам категорий:

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

Обычно мы называем эту категорию **Hask** .

## Изоморфизм

В теории категорий у нас есть изоморфизм, когда мы имеем морфизм, который имеет обратный, иными словами, существует морфизм, который может быть составлен с ним для создания тождества. В **Hask** это означает наличие пары морфизмов `f` , `g` таких, что:

```
 f . g == id == g . f
```

Если мы найдем пару таких морфизмов между двумя типами, мы будем называть их *изоморфными друг другу* .

Примером двух изоморфных типов будет `((),a)` и `a` для некоторого `a` . Мы можем построить два морфизма:

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

И мы можем проверить, что `f . g == id == g . f` .

# ФУНКТОРЫ

Функтор в теории категорий переходит из категории в другую, отображая объекты и морфизмы. Мы работаем только с одной категорией категории **Hask** типа Haskell, поэтому мы будем видеть только функторы от **Hask** до **Hask** , те функторы, чья начальная и конечная категории одинаковы, называются **endofunctors** . Наши эндофенторы будут полиморфными типами, которые берут тип и возвращают другое:

```
F :: * -> *
```

Послушать законы категориального функтора (сохранить тождества и состав) эквивалентно соблюдению законов функтора Хаскеля:

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

Итак, мы имеем, например, что `[]` , `Maybe` or `(-> r)` являются функторами в **Hask** .

# Монады

Монада в теории категорий является моноидом на **категории эндофунторов** . Эта категория имеет эндофенторы как объекты `F :: * -> *` и естественные преобразования (преобразования между ними для `forall a . F a -> G a` ) как морфизмы.

Моноидный объект может быть определен в моноидальной категории и является типом, имеющим два морфизма:

```
zero :: () -> M
mappend :: (M,M) -> M
```

Мы можем перевести это примерно в категорию эндофунторов Хаска как:

```
return :: a -> m a
join :: m (m a) -> m a
```

И, подчиняясь монадам, законы эквивалентны подчинению категорическим моноидным объектным законам.

---

† На самом деле класс всех типов вместе с классом функций между типами строго *не* образуют категорию в Haskell из-за существования `undefined` . Как правило, это устраняется путем простого определения объектов категории **Hask** как типов без нижних значений, что исключает неисключительные функции и бесконечные значения (codata). Подробное обсуждение этой темы см. Здесь .

### Продукт типов в Хаске

## Категориальные продукты

В теории категорий произведение двух объектов **X** , **Y** является другим объектом **Z** с двумя проекциями: $\pi_1$: **Z → X** и $\pi_2$: **Z → Y** ; так что любые другие два морфизма от другого объекта разлагаются однозначно через эти проекции. Другими словами, если существуют $f_1$: **W → X** и $f_2$: **W → Y** , существует единственный морфизм **g: W → Z** такой, что $\pi_1 \circ g = f_1$ и $\pi_2 \circ g = f_2$ .

## Продукты в Хаске

Это переводит в категорию **Hask** типа Haskell следующим образом: `Z` - произведение `A` , `B` когда:

```
-- if there are two functions
f1 :: W -> A
f2 :: W -> B
-- we can construct a unique function
g  :: W -> Z
-- and we have two projections
p1 :: Z -> A
p2 :: Z -> B
-- such that the other two functions decompose using g
p1 . g == f1
p2 . g == f2
```

Тип **продукта двух типов** `A` , `B` , который следует указанному выше закону, **является кортежем** двух типов `(A,B)` , а обе проекции - `fst` и `snd` . Мы можем проверить, что это следует за приведенным выше правилом, если мы имеем две функции `f1 :: W -> A` и `f2 :: W -> B` мы можем разложить их однозначно следующим образом:

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

И мы можем проверить правильность разложения:

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

## Единственность с точностью до изоморфизма

Выбор `(A,B)` как произведения `A` и `B` не является уникальным. Другим логичным и эквивалентным выбором было бы следующее:

```
data Pair a b = Pair a b
```

Более того, мы могли бы также выбрать `(B,A)` в качестве произведения или даже `(B,A,())` , и мы могли бы найти такую же декомпозиционную функцию, как и выше, также следуя правилам:

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

Это связано с тем, что продукт не уникален, а *уникален до изоморфизма* . Каждые два произведения `A` и `B` не обязательно должны быть равными, но они должны быть изоморфны. В качестве примера два разных продукта, которые мы только что определили, `(A,B)` и `(B,A,())` , изоморфны:

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())

iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

## Единственность разложения

Важно отметить, что функция декомпозиции должна быть единственной. Существуют типы, которые соответствуют всем правилам, необходимым для продукта, но разложение не является уникальным. В качестве примера мы можем попытаться использовать `(A,(B,Bool))` с проекциями `fst fst . snd` как продукт `A` и `B` :

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

Мы можем проверить, что он работает:

```
fst         . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

Но проблема в том, что мы могли бы написать еще одно разложение, а именно:

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

И, поскольку разложение **не** является **уникальным** , `(A,(B,Bool))` **не** является произведением `A` и `B` в **Hask**

**Сопротивление типов в Хаске**

## Интуиция

Категориальный продукт двух типов **A** и **B** должен содержать минимальную информацию, необходимую для хранения внутри экземпляра типа **A** или типа **B.** Теперь мы можем видеть, что интуитивное копроизведение двух типов должно быть `Either ab` . Другие кандидаты, такие как `Either a (b,Bool)` , будут содержать часть ненужной информации, и они не будут минимальными.

Формальное определение получается из категориального определения копроизведения.

## Категориальные сопроводительные материалы

Категориальный копродукт - это двойственное понятие категориального произведения. Он получается непосредственно путем изменения всех стрелок в определении произведения. Копроизведение двух объектов **X** , **Y** является другим объектом **Z** с двумя включениями: **i_1: X → Z** и **i_2: Y → Z** ; так что любые другие два морфизма из **X** и **Y** в другой объект разлагаются однозначно через эти включения. Другими словами, если существуют два морфизма $\mathbf{f_1: X \to W}$ и $\mathbf{f_2: Y \to W}$ , существует единственный морфизм **g: Z → W** такой, что $\mathbf{g \circ i_1 = f_1}$ и $\mathbf{g \circ i_2 = f_2}$

## Копродукт в Хаске

Перевод в категорию « **Хаск** » аналогичен переводу продукта:

```
-- if there are two functions
f1 :: A -> W
f2 :: B -> W
-- and we have a coproduct with two inclusions
i1 :: A -> Z
i2 :: B -> Z
-- we can construct a unique function
g  :: Z -> W
-- such that the other two functions decompose using g
g . i1 == f1
g . i2 == f2
```

Тип сопроцесса двух типов `A` и `B` в **Hask** - это `Either ab` либо любой другой тип, изоморфный ему:

```
-- Coproduct
-- The two inclusions are Left and Right
```

```
data Either a b = Left a | Right b

-- If we have those functions, we can decompose them through the coproduct
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

## Haskell Применительно с точки зрения теории категорий

`Functor` Хаскелла позволяет сопоставить любой тип `a` (объект **Хаска** ) с типом `F a` а также отобразить функцию `a -> b` (морфизм **Хаска** ) на функцию с типом `F a -> F b` . Это соответствует определению теории категорий в том смысле, что функтор сохраняет основную структуру категории.

**Одноидная категория** - это категория, имеющая некоторую *дополнительную* структуру:

- Тензорный продукт (см. Продукт типов в Hask )
- Тензорная единица (единичный объект)

Взяв пару в качестве нашего продукта, это определение можно перевести на Haskell следующим образом:

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

`Applicative` класс эквивалентен этому `Monoidal` и, следовательно, может быть реализован в терминах этого:

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
    f <*> fa = (\(f, a) -> f a) <$> (mcat f fa)
```

Прочитайте Теория категорий онлайн: https://riptutorial.com/ru/haskell/topic/2261/теория-категорий

# глава 68: Тестирование с вкусной

## Examples

### SmallCheck, QuickCheck и HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "String length <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "String length <= 2" $  -- should fail
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "String length <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "String length <= 4" $  -- should fail
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "String comparison 1" $
      assertEqual "description" "OK" "OK"

  , testCase "String comparison 2" $  -- should fail
      assertEqual "description" "fail" "fail!"
  ]
```

Установить пакеты:

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

Выполнить с помощью cabal:

```
cabal exec runhaskell test.hs
```

Прочитайте Тестирование с вкусной онлайн: https://riptutorial.com/ru/haskell/topic/3816/

тестирование-с-вкусной

# глава 69: Тип приложения

## Вступление

`TypeApplications` являются альтернативой *аннотациям* типа, когда компилятор пытается вывести типы для данного выражения.

Эта серия примеров объяснит цель расширения `TypeApplications` и способы его использования

Не забудьте включить расширение, разместив `{-# LANGUAGE TypeApplications #-}` в верхней части исходного файла.

## Examples

### Избегание аннотаций типа

Мы используем аннотации типов, чтобы избежать двусмысленности. Типовые приложения могут использоваться для той же цели. Например

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

Этот код имеет ошибку неоднозначности. Мы знаем, что `a` имеет экземпляр `Num` , и для его печати мы знаем, что ему нужен экземпляр `Show` . Это могло бы работать, если `a` был, например, `Int` , поэтому, чтобы исправить ошибку, мы можем добавить аннотацию типа

```
main = print (x :: Int)
```

Другое решение, использующее приложения типа, будет выглядеть так:

```
main = print @Int x
```

Чтобы понять, что это значит, нам нужно посмотреть на подпись типа `print` .

```
print :: Show a => a -> IO ()
```

Функция принимает один параметр типа `a` , но другой способ взглянуть на это состоит в том, что он фактически принимает два параметра. Первый - это параметр *типа* , второй - значение, тип которого является первым параметром.

Основное различие между параметрами значения и параметрами типа заключается в том, что последние неявно предоставляются функциям, когда мы их называем. Кто их предоставляет? Алгоритм вывода типа! Что позволяют `TypeApplications` , это явно `TypeApplications` эти параметры. Это особенно полезно, когда вывод типа не может определить правильный тип.

Итак, чтобы разбить приведенный выше пример

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

## Тип приложений на других языках

Если вы знакомы с такими языками, как Java, C # или C ++ и концепцией generics / templates, то это сравнение может быть полезно для вас.

Скажем, у нас есть общая функция в C #

```
public static T DoNothing<T>(T in) { return in; }
```

Чтобы вызвать эту функцию с помощью `float` мы можем сделать `DoNothing(5.0f)` или если мы хотим быть явным, мы можем сказать `DoNothing<float>(5.0f)` . Эта часть внутри угловых скобок является типом приложения.

В Haskell это одно и то же, за исключением того, что параметры типа не только неявны на сайтах-узлах, но также и на сайтах определения.

```
doNothing :: a -> a
doNothing x = x
```

Это также можно сделать явным, используя `ScopedTypeVariables` , `Rank2Types` или `RankNTypes` подобные этому.

```
doNothing :: forall a. a -> a
doNothing x = x
```

Затем на сайте вызова мы снова можем либо написать `doNothing 5.0` либо `doNothing @Float 5.0`

## Порядок параметров

Проблема с неявными аргументами типа становится очевидной, если у нас больше нескольких. В какой порядок они входят?

```
const :: a -> b -> a
```

Имеет ли запись `const @Int` значение `a` равно `Int` , или это `b` ? Если мы явно укажем параметры типа, используя `forall` например `const :: forall a b. a -> b -> a` тогда порядок написан так: `a` , тогда `b` .

Если мы этого не сделаем, то порядок переменных будет слева направо. Первой переменной, которая будет упомянута, является параметр первого типа, второй - параметр второго типа и т. Д.

Что делать, если мы хотим указать переменную второго типа, но не первую? Мы можем использовать подстановочный знак для первой переменной, подобной этой

```
const @_ @Int
```

Тип этого выражения

```
const @_ @Int :: a -> Int -> a
```

## Взаимодействие с неоднозначными типами

Предположим, вы вводите класс типов, размер которых в байтах.

```
class SizeOf a where
    sizeOf :: a -> Int
```

Проблема в том, что размер должен быть постоянным для каждого значения этого типа. Мы фактически не хотим, `sizeOf` функция `sizeOf` зависела от `a` , но только от ее типа.

Без приложений типа наилучшим решением было `Proxy` тип, определенный таким образом

```
data Proxy a = Proxy
```

Цель этого типа - переносить информацию о типе, но без информации о значении. Тогда наш класс мог бы выглядеть так

```
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

Теперь вам может быть интересно, почему бы вообще не отказаться от первого аргумента? Тогда тип нашей функции будет просто `sizeOf :: Int` или, если быть более точным, потому что это метод класса `sizeOf :: SizeOf a => Int` или быть еще более явным `sizeOf :: forall a. SizeOf a => Int` .

Проблема заключается в выводе типа. Если я буду писать `sizeOf` где-то, алгоритм вывода только знает, что я ожидаю `Int` . Он не знает, какой тип я хочу заменить `a` . Из-за этого определение будет отклонено компилятором, *если* вы не включили расширение `{-# LANGUAGE AllowAmbiguousTypes #-}` . В этом случае определение компилируется, его просто невозможно

использовать нигде без ошибки двусмысленности.

К счастью, введение типов приложений экономит день! Теперь мы можем написать `sizeOf @Int` , явно говоря, что `a` - `Int` . Приложения типа позволяют нам предоставлять параметр типа, даже если он не отображается в *реальных параметрах функции* !

Прочитайте Тип приложения онлайн: https://riptutorial.com/ru/haskell/topic/10767/тип-приложения

# глава 70: Типированные отверстия

## замечания

Одной из сильных сторон Haskell является способность использовать систему типов для моделирования частей вашего проблемного домена в системе типов. При этом часто встречаются очень сложные типы. При написании программ с этими типами (т. Е. Со значениями, имеющими эти типы), он иногда становится почти невосприимчивым к «жонглированию» всех типов. Начиная с GHC 7.8, появляется новая синтаксическая функция, называемая типизированными отверстиями. Типированные отверстия не изменяют семантику основного языка; они предназначены исключительно для помощи в написании программ.

Подробное объяснение типизированных отверстий, а также обсуждение конструкции типизированных отверстий см. В вики Haskell .

---

Раздел руководства пользователя GHC на типизированных отверстиях .

## Examples

### Синтаксис типизированных отверстий

Типовое отверстие является единственным символом подчеркивания ( _ ) или допустимым идентификатором Haskell, который не находится в области видимости в контексте выражения. Перед существованием типизированных отверстий обе эти вещи вызовут ошибку, поэтому новый синтаксис не будет мешать никакому старому синтаксису.

## Управление поведением типизированных отверстий

Поведение типизированных отверстий по умолчанию - это получение ошибки времени компиляции при столкновении с типизированным отверстием. Тем не менее, существует несколько флагов для точной настройки их поведения. Эти флаги суммируются следующим образом ( GHC trac ):

> По умолчанию GHC имеет введенные отверстия и дает ошибку компиляции, когда он сталкивается с типизированным отверстием.
>
> Когда `-fdefer-type-errors` **или** `-fdefer-typed-holes` , ошибки отверстия преобразуются в предупреждения и приводят к ошибкам во время выполнения.
>
> По `-fwarn-typed-holes` флаг предупреждения `-fwarn-typed-holes` . Без `-fdefer-type-`

`errors` или `-fdefer-typed-holes` этот флаг является не-ор, так как типизированные отверстия являются ошибкой в этих условиях. Если один из флажков отсрочки включен (преобразование введенных ошибок отверстий в предупреждения), `-fno-warn-typed-holes` отключает предупреждения. Это означает, что компиляция полностью завершается успешно, и оценка отверстия приведет к ошибке выполнения.

## Семантика типизированных отверстий

Значение отверстия типа может просто считаться `undefined` , хотя типизированное отверстие вызывает ошибку времени компиляции, поэтому нет необходимости назначать ему значение. Однако типизированное отверстие (когда они включены) создает ошибку времени компиляции (или предупреждение с ошибками отложенного типа), в котором указывается имя типизированного отверстия, его выведенный *наиболее общий* тип и типы любых локальных привязок. Например:

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: Warning:
    Found hole `_var' with type: Int
    Relevant bindings include
      x :: [a] (bound at <interactive>:19:2)
      it :: [a] -> Int (bound at <interactive>:19:1)
    In the first argument of `(+)', namely `_var'
    In the expression: _var + length (drop 1 x)
    In the expression: \ x -> _var + length (drop 1 x)
```

Обратите внимание, что в случае типизированных отверстий в выражениях, введенных в GHCi repl (как указано выше), также указывался тип введенного выражения, как `it` (здесь тип `[a] -> Int` ).

## Использование типизированных отверстий для определения экземпляра класса

Типизированные отверстия могут облегчить определение функций посредством интерактивного процесса.

Предположим, вы хотите определить экземпляр `Foo Bar` класса (для вашего настраиваемого типа `Bar` , чтобы использовать его с некоторой функцией полиморфной библиотеки, для которой требуется экземпляр `Foo` ). Теперь вы традиционно просматриваете документацию `Foo` , выясняете, какие методы вам нужно определять, тщательно изучать их типы и т. Д. - но с типизированными отверстиями вы действительно можете пропустить это!

Сначала просто определите фиктивный экземпляр:

```
instance Foo Bar where
```

Компилятор теперь будет жаловаться

```
Bar.hs:13:10: Warning:
No explicit implementation for
  'foom' and 'quun'
In the instance declaration for 'Foo Bar'
```

Итак, нам нужно определить `foom` для `Bar` . Но что *это* такое? Опять же, мы слишком ленивы, чтобы посмотреть в документации и просто спросить компилятора:

```
instance Foo Bar where
  foom = _
```

Здесь мы использовали типизированное отверстие как простой «запрос документации». Выходы компилятора

```
Bar.hs:14:10:
    Found hole '_' with type: Bar -> Gronk Bar
    Relevant bindings include
      foom :: Bar -> Gronk Bar (bound at Foo.hs:4:28)
    In the expression: _
    In an equation for 'foom': foom = _
    In the instance declaration for 'Foo Bar'
```

Обратите внимание , как компилятор уже заполнил переменный тип класса с конкретным типом `Bar` , который мы хотим создать его экземпляр для. Это может сделать подпись намного понятнее, чем полиморфная, найденная в документации по классу, особенно если вы имеете дело с более сложным методом, например, с классом типа с несколькими параметрами.

Но что, черт возьми, `Gronk` ? На данный момент, вероятно, неплохо спросить Айоу . Однако мы все равно можем уйти без этого: в качестве слепой гипотезы мы предполагаем, что это не только конструктор типов, но и единственный конструктор значений, т. `Gronk a` Он может использоваться как функция, которая каким-то образом приведет к `Gronk a` . Поэтому мы пытаемся

```
instance Foo Bar where
  foom bar = _ Gronk
```

Если нам повезет, `Gronk` на самом деле является ценностью, и компилятор теперь скажет

```
    Found hole '_'
      with type: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
    Where: 'b0' is an ambiguous type variable
```

Хорошо, это уродливо - сначала просто обратите внимание, что у `Gronk` есть два аргумента, поэтому мы можем уточнить нашу попытку:

```
instance Foo Bar where
  foom bar = Gronk _ _
```

И теперь это довольно ясно:

```
    Found hole '_' with type: [(Int, Bar)]
    Relevant bindings include
      bar :: Bar (bound at Bar.hs:14:29)
      foom :: Bar -> Gronk Bar (bound at Foo.hs:15:24)
    In the second argument of 'Gronk', namely '_'
    In the expression: Gronk _ _
    In an equation for 'foom': foom bar = Gronk _ _
```

Теперь вы можете продолжить прогресс, например, деконструируя значение `bar` ( компоненты будут отображаться вместе с типами в разделе « `Relevant bindings` »). Часто в какой-то момент совершенно очевидно, что будет правильным определением, потому что вы видите все доступные аргументы и типы, подходящие друг другу, как головоломка. Или, альтернативно, вы можете видеть, что определение *невозможно* и почему.

Все это лучше всего работает в редакторе с интерактивной компиляцией, например Emacs с haskell-mode. Затем вы можете использовать типизированные отверстия так же, как запросы с загрузкой мыши в среде IDE для интерпретируемого динамического императивного языка, но без ограничений.

Прочитайте Типированные отверстия онлайн: https://riptutorial.com/ru/haskell/topic/4913/типированные-отверстия

# глава 71: Типная алгебра

## Examples

### Естественные числа в алгебре типов

Мы можем провести связь между типами Хаскелла и натуральными числами. Это соединение можно присвоить каждому типу количеству жителей.

## Конечные типы соединений

Для конечных типов достаточно видеть, что мы можем присвоить натуральный тип каждому числу, основанному на числе конструкторов. Например:

```
type Color = Red | Yellow | Green
```

будет **3** . И тип `Bool` будет **2** .

```
type Bool = True | False
```

## Единственность с точностью до изоморфизма

Мы видели, что несколько типов будут соответствовать одному числу, но в этом случае они будут изоморфны. Это означает, что существует пара морфизмов `f` и `g` , состав которых будет тождественным, соединяющим два типа.

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

В этом случае мы будем говорить, что типы **изоморфны** . Мы будем рассматривать два типа, равные в нашей алгебре, если они изоморфны.

Например, два разных представления числа два тривиально изоморфны:

```
type Bit  = I    | O
type Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
```

```
booleanBit False = 0
```

Потому что мы можем видеть `bitValue . booleanBit == id == booleanBit . bitValue`

# Один и ноль

Представление числа **1** , очевидно, является типом только с одним конструктором. В Haskell этот тип является каноническим типом `()` , называемым Unit. Каждый другой тип с одним конструктором изоморфен `()` .

И наше представление **0** будет типом без конструкторов. Это тип **Void** в Haskell, как определено в `Data.Void` . Это будет эквивалентно неживому типу с конструкторами данных:

```
data Void
```

## Добавление и умножение

Сложение и умножение имеют эквиваленты в алгебре этого типа. Они соответствуют **маркированным союзам** и **типам продукции** .

```
data Sum a b = A a | B b
data Prod a b = Prod a b
```

Мы можем видеть, как число жителей каждого типа соответствует операциям алгебры.

Эквивалентно, мы можем использовать `Either` и `(,)` как конструкторы типов для сложения и умножения. Они изоморфны нашим ранее определенным типам:

```
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

Ожидаемым результатам сложения и умножения следуют алгебра типов до изоморфизма. Например, мы можем видеть изоморфизм между 1 + 2, 2 + 1 и 3; как 1 + 2 = 3 = 2 + 1.

```
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())      = Red
f (Right True)   = Green
f (Right False)  = Blue

g :: Color -> Sum () Bool
g Red    = Left ()
g Green = Right True
g Blue   = Right False

f' :: Sum Bool () -> Color
f' (Right ())    = Red
```

```
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

# Правила сложения и умножения

Общие правила коммутативности, ассоциативности и дистрибутивности справедливы, поскольку существуют тривиальные изоморфизмы между следующими типами:

```
-- Commutativity
Sum a b           <=> Sum b a
Prod a b          <=> Prod b a
-- Associativity
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- Distributivity
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

### Рекурсивные типы

# Списки

Списки могут быть определены как:

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

Если мы переведем это в нашу алгебру типов, получим

Список (a) = 1 + a * Список (a)

Но теперь мы можем снова заменить *List (a)* в этом выражении несколько раз, чтобы получить:

Список (a) = 1 + a + a * a + a * a * a + a * a * a + ...

Это имеет смысл, если мы видим список как тип, который может содержать только одно значение, как в `[]` ; или любое значение типа `a` , как в `[x]` ; или два значения типа `a` , как в `[x,y]` ; и так далее. Теоретическое определение списка, которое мы должны получить оттуда, будет:

```
-- Not working Haskell code!
data List a = Nil
            | One a
```

```
            | Two a a
            | Three a a a
            ...
```

## деревья

Например, мы можем сделать то же самое с бинарными деревьями. Если мы определим их как:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Получаем выражение:

> Дерево (a) = 1 + a * Дерево (a) * Дерево (a)

И если мы будем делать те же подстановки снова и снова, мы получим следующую последовательность:

> Дерево (a) = 1 + a + 2 (a * a) + 5 (a * a * a) + 14 (a * a * a * a) + ...

Коэффициенты, которые мы получаем здесь, соответствуют последовательности каталонских чисел, а n-е каталонское число - это точно число возможных бинарных деревьев с n узлами.

### производные

Производная типа является типом своего типа одноярусных контекстов. Это тот тип, который мы получили бы, если бы мы превратили переменную типа в каждую возможную точку и суммировали результаты.

В качестве примера мы можем взять тройной тип `(a,a,a)` и получить его, получив

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

Это согласуется с нашим обычным определением вывода:

> d / da (a * a * a) = 3 * a * a

Подробнее об этой теме можно прочитать в этой статье .

### функции

Функции можно рассматривать как экспоненты в нашей алгебре. Как мы видим, если взять тип `a` с n экземплярами и тип `b` с m экземплярами, тип `a -> b` будет иметь m для мощности n экземпляров.

В качестве примера, `Bool -> Bool` изоморфен `(Bool,Bool)` , как 2 * 2 = $2^2$.

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
iso2 (x,y) = (\p -> if p then x else y)
```

Прочитайте Типная алгебра онлайн: https://riptutorial.com/ru/haskell/topic/4905/типная-алгебра

# глава 72: Типы классов

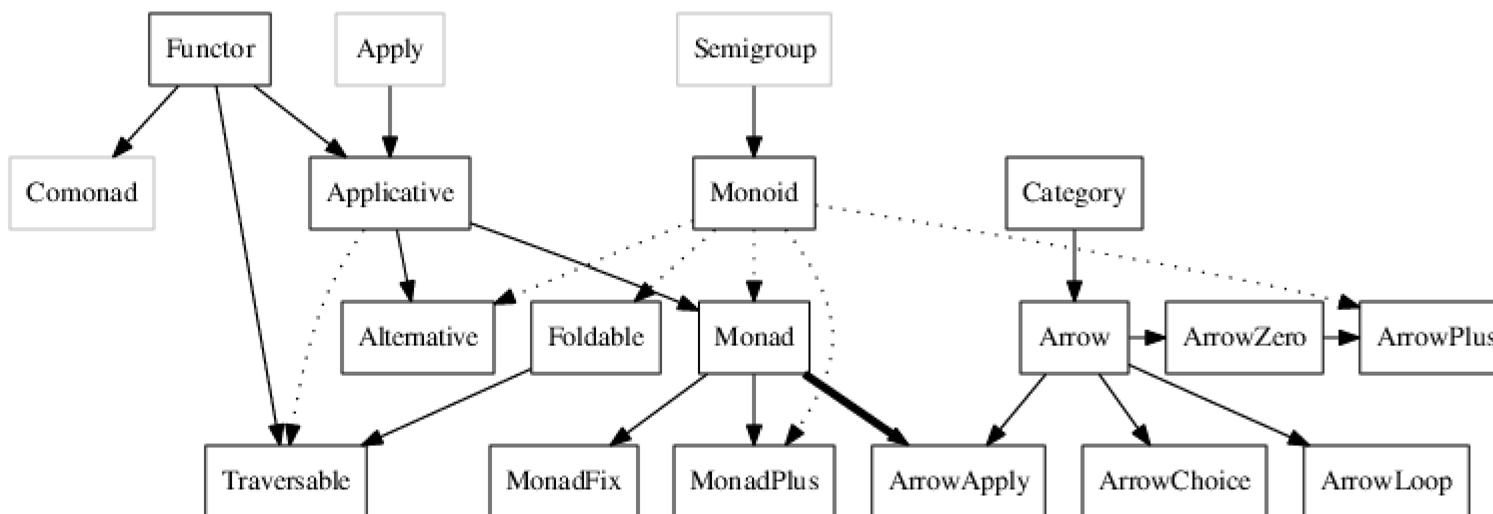
## Вступление

Typeclasses в Haskell - это средство определения поведения, связанного с типом отдельно от определения этого типа. Если, скажем, в Java, вы определяете поведение как часть определения типа - то есть в интерфейсе, абстрактном классе или конкретном классе - Haskell сохраняет эти две вещи отдельно.

В `base` пакете Haskell уже определено несколько типов типов. Связь между ними проиллюстрирована в разделе «Примечания» ниже.

## замечания

Следующая диаграмма, взятая из статьи Typeclassopedia, показывает взаимосвязь между различными классами в Haskell.




## Examples

### Возможно и класс Functor

В Haskell типы данных могут иметь аргументы, подобные функциям. Возьмите тип `Maybe` например.

`Maybe` , это очень полезный тип, который позволяет нам представить идею неудачи или ее возможности. Другими словами, если есть вероятность, что вычисление не удастся, мы будем использовать тип `Maybe` . `Maybe` похож на обертку для других типов, предоставляя им дополнительную функциональность.

Его фактическое заявление довольно просто.

```
Maybe a = Just a | Nothing
```

Это говорит о том, что « `Maybe` происходит в двух формах: « `Just` , который представляет успех, и « `Nothing` , который представляет собой неудачу. `Just` принимает один аргумент, который определяет тип `Maybe` , и `Nothing` принимает. Например, значение `Just "foo"` будет иметь тип `Maybe String` , который является строковым типом, завернутым в дополнительную функциональность `Maybe` . Значение `Nothing` имеет тип `Maybe a` где `a` может быть любым типом.

Эта идея обертывания типов, чтобы дать им дополнительную функциональность, очень полезна и применима к более чем просто `Maybe` . Другие примеры включают типы `Either` , `IO` и типы списков, каждый из которых обеспечивает различные функциональные возможности. Однако есть некоторые действия и способности, которые являются общими для всех этих типов обертки. Наиболее заметным из них является возможность изменения инкапсулированного значения.

Общепринято думать о таких типах ящиков, которые могут иметь значения, размещенные в них. Различные блоки содержат разные значения и делают разные вещи, но ни одна из них не полезна без возможности доступа к содержимому внутри.

Чтобы инкапсулировать эту идею, Haskell поставляется со стандартным классом типа `Functor` . Он определяется следующим образом.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Как видно, класс имеет одну функцию `fmap` из двух аргументов. Первый аргумент - это функция от одного типа, `a` , от другого, `b` . Второй аргумент - это функтор (тип-оболочка), содержащий значение типа `a` . Он возвращает функтор (тип-обертку), содержащий значение типа `b` .

Проще говоря, `fmap` принимает функцию и применяется к значению внутри функтора. Это единственная функция, необходимая для того, чтобы тип был членом класса `Functor` , но он чрезвычайно полезен. Функции, работающие на функторах с более конкретными приложениями, можно найти в стилях `Applicative` и `Monad` .

### Наследование типа класса: класс типа Ord

Haskell поддерживает понятие расширения класса. Например, класс `Ord` наследует все операции в `Eq` , но, кроме того, имеет функцию `compare` которая возвращает `Ordering` между значениями. `Ord` может также содержать общие операторы сравнения порядка, а также метод `min` и `max` метод.

Обозначение `=>` имеет то же значение, что и в сигнатуре функции, и требует, чтобы тип `a` реализовал `Eq` , чтобы реализовать `Ord` .

```
data Ordering = EQ | LT | GT

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

Все методы следующие `compare` можно извлечь из него в ряде способов:

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- Note the use of (==) inherited from Eq
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
               LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
               LT -> y
               GT -> x
```

Типовые классы, которые сами расширяют `Ord` должны реализовать по крайней мере либо метод `compare` либо метод `(<=)` , который создает направленную решетку наследования.

### уравнение

Все базовые типы данных (например, `Int` , `String` , `Eq a => [a]` ) из Prelude, за исключением функций и `IO` имеют экземпляры `Eq` . Если тип создает экземпляр `Eq` это означает, что мы знаем, как сравнивать два значения для *значения* или *структурного* равенства.

```
> 3 == 2
False
> 3 == 3
True
```

# Необходимые методы

- `(==) :: Eq a => a -> a -> Boolean` или `(/=) :: Eq a => a -> a -> Boolean` (если только один реализован, другой по умолчанию - отрицание определенный)

# Определяет

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

# Прямые суперклассы

Никто

# Известные подклассы

- `Ord`

### Ord

Типичные экземпляры `Ord` включают, например, `Int` , `String` и `[a]` (для типов `a` где есть экземпляр `Ord a` ). Если тип создает экземпляр `Ord` это означает, что мы знаем «естественный» порядок значений этого типа. Обратите внимание: часто существует множество возможных вариантов «естественного» упорядочения типа, и `Ord` заставляет нас это одобрить.

`Ord` предоставляет стандартные `(<=)` , `(<)` , `(>)` , `(>=)` операторы, но интересно их определяет, используя собственный тип алгебраических данных

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

# Необходимые методы

- `compare :: Ord a => a -> a -> Ordering` или `(<=) :: Ord a => a -> a -> Boolean` (метод `compare` по умолчанию используется `(<=)` в его реализации)

# Определяет

- `compare :: Ord a => a -> a -> Ordering`
- `(<=) :: Ord a => a -> a -> Boolean`
- `(<) :: Ord a => a -> a -> Boolean`
- `(>=) :: Ord a => a -> a -> Boolean`
- `(>) :: Ord a => a -> a -> Boolean`
- `min :: Ord a => a -> a -> a`
- `max :: Ord a => a -> a -> a`

# Прямые суперклассы

- `Eq`

## Monoid

Типы экземпляров `Monoid` включают списки, числа и функции с возвращаемыми значениями `Monoid` . Чтобы создать экземпляр `Monoid` тип должен поддерживать ассоциативную двоичную операцию ( `mappend` или `(<>)` ), которая объединяет его значения и имеет специальное «нулевое» значение ( `mempty` ), так что объединение значения с ним не изменяет это значение:

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

Интуитивно, типы `Monoid` являются «подобными спискам», поскольку они поддерживают добавление значений вместе. В качестве альтернативы, типы `Monoid` можно рассматривать как последовательности значений, для которых мы заботимся о порядке, но не о группировании. Например, двоичное дерево является `Monoid` , но, используя операции `Monoid` мы не можем наблюдать его ветвящуюся структуру, а только обход его значений (см. `Foldable` и `Traversable` ).

# Необходимые методы

- `mempty :: Monoid m => m`
- `mappend :: Monoid m => m -> m -> m`

# Прямые суперклассы

Никто

## Num

Самый общий класс для типов чисел, точнее для колец , то есть чисел, которые можно добавлять и вычитать и умножать в обычном смысле, но не обязательно разделять.

Этот класс содержит как интегральные типы ( `Int` , `Integer` , `Word32` и т. Д.), Так и дробные типы ( `Double` , `Rational` , также комплексные числа и т. Д.). В случае конечных типов семантика обычно понимается как *модульная арифметика* , т. Е. С избыточным и нижним потоком .

Обратите внимание, что правила для числовых классов гораздо менее строго соблюдаются, чем законы монады или моноида, или правила сравнения равенства . В частности, числа с плавающей запятой обычно подчиняются законам только в приблизительном смысле.

## Методы

- `fromInteger :: Num a => Integer -> a` . конвертировать целое число в общий тип номера (при необходимости обертывать диапазон). Литералы номера Haskell можно понимать как мономорфный литерал `Integer` с общим преобразованием вокруг него, поэтому вы можете использовать литерал `5` как в контексте `Int` и в параметре `Complex Double` .

- `(+) :: Num a => a -> a -> a` . Стандартное дополнение, обычно понимаемое как ассоциативное и коммутативное, т. Е.

  ```
  a + (b + c) ≡ (a + b) + c
  a + b ≡ b + a
  ```

- `(-) :: Num a => a -> a -> a` . Вычитание, которое является обратным добавлению:

  ```
  (a - b) + b ≡ (a + b) - b ≡ a
  ```

- `(*) :: Num a => a -> a -> a` . Умножение, ассоциативная операция, которая является дистрибутивной над добавлением:

  ```
  a * (b * c) ≡ (a * b) * c
  a * (b + c) ≡ a * b + a * c
  ```

  для наиболее распространенных экземпляров умножение также является коммутативным, но это определенно не является требованием.

- `negate :: Num a => a -> a` . Полное имя оператора унарного отрицания. `-1` - синтаксический сахар для `negate 1` .

  ```
  -a ≡ negate a ≡ 0 - a
  ```

- `abs :: Num a => a -> a` . Функция абсолютного значения всегда дает неотрицательный результат той же величины

  ```
  abs (-a) ≡ abs a
  abs (abs a) ≡ abs a
  ```

  `abs a ≡ 0` должно произойти только, если `a ≡ 0` .

  Для реальных типов ясно, какие неотрицательные средства: у вас всегда есть `abs a >= 0` . Сложные и т. Д. Типы не имеют четко определенного порядка, однако

результат `abs` всегда должен лежать в вещественном подмножестве ‡ (т. Е. Дать число, которое также может быть записано в виде единственного литерала без отрицания).

- `signum :: Num a => a -> a` . Функция знака, в соответствии с именем, дает только `-1` или `1` , в зависимости от знака аргумента. Собственно, это верно только для ненулевых действительных чисел; в общих `signum` лучше понимать как *нормализующая* функцию:

```
abs (signum a) ≡ 1   -- unless a≡0
signum a * abs a ≡ a -- This is required to be true for all Num instances
```

Обратите внимание, что в разделе 6.4.4 отчета Haskell 2010 явно требуется, чтобы это последнее равенство выполнялось для любого действительного экземпляра `Num` .

---

Некоторые библиотеки, в частности линейные и hmatrix , имеют гораздо более слабое понимание того, для чего предназначен класс `Num` : они рассматривают его как *способ перегрузить арифметические операторы* . Хотя это довольно просто для `+` и `-` , это уже становится проблематичным с `*` и тем более с другими методами. Например, *должно* `*` *означать матричное умножение или умножение по элементам?*
Вероятно, это плохая идея для определения таких не числовых экземпляров; рассмотрите специальные классы, такие как `VectorSpace` .

---

† В частности, «негативы» неподписанных типов обернуты вокруг большого положительного значения, например `(-4 :: Word32) == 4294967292` .

‡ Это широко *не* выполняется: типы векторов не имеют реального подмножества. Спорные `Num` состояния для таких типов обычно определяют `abs` и `signum` элементы, что математически не имеет смысла.

Прочитайте Типы классов онлайн: https://riptutorial.com/ru/haskell/topic/1879/типы-классов

# глава 73: Типы фантомов

## Examples

### Вариант использования для фантомных типов: Валюта

Типы фантомов полезны для работы с данными, которые имеют идентичные представления, но не являются логически одного и того же типа.

Хорошим примером является валюта. Если вы работаете с валютами, вы абсолютно не хотите, например, добавлять две суммы разных валют. Какова будет валюта результата `5.32€ + 2.94$` ? Это не определено, и для этого нет веских оснований.

Решение этого может выглядеть примерно так:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

data USD
data EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

Расширение `GeneralisedNewtypeDeriving` позволяет нам выводить `Num` для типа `Amount` . GHC повторно использует экземпляр `Double` 's `Num` .

Теперь, если вы представляете суммы в евро, например, `(5.0 :: Amount EUR)` вы решили проблему сохранения двойных сумм на уровне типа без введения накладных расходов. Такие вещи, как `(1.13 :: Amount EUR) + (5.30 :: Amount USD)` , приведут к ошибке типа и потребуют надлежащего обращения с конверсией валюты.

Более полную документацию можно найти в статье wiki haskell

Прочитайте Типы фантомов онлайн: https://riptutorial.com/ru/haskell/topic/5227/типы-фантомов

# глава 74: трубы

## замечания

Как описано в хакерской странице :

> pipe - это чистая и мощная библиотека обработки потоков, которая позволяет создавать и подключать многоразовые потоковые компоненты

Программы, реализованные посредством потоковой передачи, часто могут быть краткими и сложными, с простыми и короткими функциями, позволяющими легко «входить или выходить» с поддержкой системы типа Haskell.

`await :: Monad m => Consumer' ama`

Вытягивает значение из восходящего потока, где `a` - это наш тип ввода.

`yield :: Monad m => a -> Producer' am ()`

Произведите значение, где `a` - тип вывода.

Настоятельно рекомендуется прочитать встроенный пакет `Pipes.Tutorial` , который дает отличный обзор основных концепций Pipes и того, как взаимодействуют `Producer` , `Consumer` и `Effect` .

# Examples

## Производители

`Producer` - это одно монадическое действие, которое может `yield` значения для потребления в нисходящем потоке:

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

Например:

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each is a utility function exported by Pipes
```

Разумеется, у нас есть `Producer` s, которые также являются функциями других значений:

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

## Потребители

`Consumer` может только `await` значений от восходящего потока.

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

Например:

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
  liftIO $ putStrLn ("I received: " ++ numStr)
```

## трубы

Трубы могут `await` и `yield` .

```
type Pipe a b = Proxy () a () b
```

Эта труба ждет `Int` и преобразует ее в `String` :

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

## Запуск труб с runEffect

Мы используем `runEffect` для запуска нашей `Pipe` :

```
main :: IO ()
main = do
  runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Обратите внимание, что `runEffect` требует `Effect` , который является автономным `Proxy` без каких-либо входов или выходов:

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

(где `X` - пустой тип, также известный как `Void` ).

## Соединительные трубы

Используйте `>->` для подключения `Producer` , `Consumer` и `Pipe` для создания более крупных функций `Pipe` .

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Типы `Producer` , `Consumer` , `Pipe` и `Effect` определяются в терминах общего типа `Proxy` . Поэтому `>->` может использоваться для различных целей. Типы, определенные левым аргументом, должны соответствовать типу, потребляемому правильным аргументом:

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

## Трансформатор монумента Proxy

Основным типом данных `pipes` является преобразователь monad `Proxy` . `Pipe` , `Producer` , `Consumer` и т. Д. Определяются в терминах `Proxy` .

Поскольку `Proxy` является монадным трансформатором, определения `Pipe` s принимают форму монадических сценариев, которые `await` и `yield` значения, дополнительно выполняя эффекты от базовой монады `m` .

## Объединение труб и сетевых коммуникаций

Трубы поддерживают простую двоичную связь между клиентом и сервером

В этом примере:

1. клиент подключается и отправляет `FirstMessage`
2. сервер получает и отвечает на `DoSomething 0`
3. клиент получает и отвечает `DoNothing`
4. шаги 2 и 3 повторяются бесконечно

Тип данных команды обменивается по сети:

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
           | DoNothing
           | DoSomething Int
           deriving (Show,Generic)

instance Binary Command
```

Здесь сервер ожидает подключения клиента:

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.map
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answers the same command that we have receveid

-- impure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "received message = " ++ (show c)
  return $ C.DoSomething 0
  -- whatever incoming command `c` from the client, answer DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                 putStrLn $ "Remote connection from ip = " ++ (show remoteAddress)
                 _ <- runEffect $ do
                   let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                   let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                   commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                   -- if we want to use the pureHandler
                   --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat
PipesBinary.Encode >-> PNT.toSocket connectionSocket
                 return ()
```

Клиент подключается таким образом:

```
module Client where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Pipes.Prelude as PipesPrelude
import qualified Pipes.Parse as PP
import qualified Command as C

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answer the same command received from the server

-- inpure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- whatever is received from server, answer DoNothing
```

```
main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "Connected to distant server ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
    return ()

sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

Прочитайте трубы онлайн: https://riptutorial.com/ru/haskell/topic/6768/трубы

# глава 75: Функтор

## Вступление

`Functor` - это класс типов `f :: * -> *` который может быть ковариантно *отображен* . Сопоставление функции над структурой данных применяет функцию ко всем элементам структуры без изменения самой структуры.

## замечания

Функтор можно рассматривать как контейнер для некоторого значения или контекст вычисления. Примерами могут `Maybe a` или `[a]` . В статье Typeclassopedia есть хорошая формулировка концепций, стоящих за Функциями.

Чтобы считаться реальным Функтором, экземпляр должен соблюдать два следующих закона:

### тождественность

```
fmap id == id
```

### Состав

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

## Examples

### Общие примеры Functor

# Может быть

`Maybe` , это `Functor` содержащий возможно отсутствующее значение:

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

`Maybe` , пример `Functor` применяет функцию к значению, заключенному в `Just` . Если вычисление ранее потерпело неудачу (так что значение `Maybe` - это `Nothing` ), тогда нет никакого значения для применения функции, поэтому `fmap` является no-op.

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
Just 9
> fmap sqrt Nothing
Nothing
```

Мы можем проверить законы функтора для этого примера, используя эквациональные рассуждения. Для закона идентичности,

```
fmap id Nothing
Nothing  -- definition of fmap
id Nothing  -- definition of id

fmap id (Just x)
Just (id x)  -- definition of fmap
Just x  -- definition of id
id (Just x)  -- definition of id
```

По закону о композиции,

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- definition of (.)
fmap f Nothing  -- definition of fmap
Nothing  -- definition of fmap
fmap (f . g) Nothing  -- because Nothing = fmap f Nothing, for all f

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- definition of (.)
fmap f (Just (g x))  -- definition of fmap
Just (f (g x))  -- definition of fmap
Just ((f . g) x)  -- definition of (.)
fmap (f . g) (Just x)  -- definition of fmap
```

# Списки

Экземпляр экземпляров `Functor` применяет функцию к каждому значению в списке.

```
instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

В качестве альтернативы это можно записать в виде понимания списка: `fmap f xs = [fx | x <- xs]` .

Этот пример показывает, что `fmap` обобщает `map` . `map` работает только в списках, тогда как `fmap` работает на произвольном `Functor` .

Можно доказать, что закон тождества сохраняется по индукции:

```
-- base case
```

```
fmap id []
[]  -- definition of fmap
id []  -- definition of id

-- inductive step
fmap id (x:xs)
id x : fmap id xs  -- definition of fmap
x : fmap id xs  -- definition of id
x : id xs  -- by the inductive hypothesis
x : xs  -- definition of id
id (x : xs)  -- definition of id
```

и аналогичным образом, составной закон:

```
-- base case
(fmap f . fmap g) []
fmap f (fmap g [])  -- definition of (.)
fmap f []  -- definition of fmap
[]  -- definition of fmap
fmap (f . g) []  -- because [] = fmap f [], for all f

-- inductive step
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- definition of (.)
fmap f (g x : fmap g xs)  -- definition of fmap
f (g x) : fmap f (fmap g xs)  -- definition of fmap
(f . g) x : fmap f (fmap g xs)  -- definition of (.)
(f . g) x : fmap (f . g) xs  -- by the inductive hypothesis
fmap (f . g) xs  -- definition of fmap
```

# функции

Не каждый `Functor` выглядит как контейнер. Экземпляр функций `Functor` применяет функцию к возвращаемому значению другой функции.

```
instance Functor ((->) r) where
    fmap f g = \x -> f (g x)
```

Заметим, что это определение эквивалентно `fmap = (.)` . Таким образом, `fmap` обобщает функцию композиции.

Еще раз проверьте закон о личности:

```
fmap id g
\x -> id (g x)  -- definition of fmap
\x -> g x  -- definition of id
g  -- eta-reduction
id g  -- definition of id
```

и закон состава:

```
(fmap f . fmap g) h
```

```
fmap f (fmap g h)  -- definition of (.)
fmap f (\x -> g (h x))  -- definition of fmap
\y -> f ((\x -> g (h x)) y)  -- definition of fmap
\y -> f (g (h y))  -- beta-reduction
\y -> (f . g) (h y)  -- definition of (.)
fmap (f . g) h  -- definition of fmap
```

## Определение класса функтора и законов

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Один из способов взглянуть на это - это то, что `fmap` *возвращает* функцию значений в функцию значений в контексте `f` .

Правильный экземпляр `Functor` должен удовлетворять *законам функтора* , хотя они не применяются компилятором:

```
fmap id = id                    -- identity
fmap f . fmap g = fmap (f . g)  -- composition
```

Существует широко используемый псевдоним infix для `fmap` называемый `<$>` .

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

## Замена всех элементов функтора на одно значение

Модуль `Data.Functor` содержит два комбинатора, `<$` и `$>` , которые игнорируют все значения, содержащиеся в функторе, заменяя их все одним постоянным значением.

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: Functor f => f a -> b -> f b
($>) = flip (<$)
```

`void` игнорирует возвращаемое значение вычисления.

```
void :: Functor f => f a -> f ()
void = (() <$)
```

## Полиномиальные функторы

Существует полезный набор комбинаторов типов для построения больших `Functor` s из

меньших. Они являются поучительными примерами `Functor` , и они также полезны в качестве метода для общего программирования, поскольку они могут использоваться для представления большого класса общих функторов.

# Тождественный функтор

Идентичный функтор просто завершает свой аргумент. Это реализация уровня `I` комбинатора из исчисления SKI.

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

`I` могу найти под именем `Identity` в модуле `Data.Functor.Identity` .

# Постоянный функтор

Постоянный функтор игнорирует свой второй аргумент, содержащий только постоянное значение. Это аналоговый аналог типа `const` , комбинатор `K` из исчисления SKI.

```
newtype K c a = K c
```

Заметим, что `K ca` не содержит никаких `a` -значений; `K ()` изоморфно `Proxy` . Это означает, что реализация `K fmap` не делает никакого отображения вообще!

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

`K` иначе известен как `Const` , из `Data.Functor.Const` .

Остальные функторы в этом примере объединяют меньшие функторы в более крупные.

# Продукты Functor

Произведение функтора принимает пару функторов и упаковывает их. Это аналогично кортежу, за исключением того, что while `(,) :: * -> * -> *` работает на `types *` , `(:*:) :: (* -> *) -> (* -> *) -> (* -> *)` действует на `functors * -> *` .

```
infixl 7 :*:
data (f :*: g) a = f a :*: g a

instance (Functor f, Functor g) => Functor (f :*: g) where
```

```
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

Этот тип можно найти под названием `Product` в модуле `Data.Functor.Product` .

# Сопутствующие товары

Точно так же `:*:` аналогично `(,)` `:+:` является аналогом уровня функтора `Either` .

```
infixl 6 :+:
data (f :+: g) a = InL (f a) | InR (g a)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

`:+:` можно найти под именем `Sum` , в модуле `Data.Functor.Sum` .

# Состав functor

Наконец, `:.:` Работает как тип уровня `(.)` , Беря вывод одного функтора и вставляя его во вход другого.

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

Тип `Compose` можно найти в `Data.Functor.Compose`

# Полиномиальные функторы для общего программирования

`I` , `K` `:*:` , `:+:` и `:.:` Можно рассматривать как набор строительных блоков для определенного класса простых типов данных. Набор становится особенно мощным, когда вы объединяете его с фиксированными точками, потому что типы данных, построенные с помощью этих комбинаторов, автоматически являются экземплярами `Functor` . Вы используете комплект для создания типа шаблона, маркировки рекурсивных точек с помощью `I` , а затем подключите его к `Fix` чтобы получить тип, который можно использовать со стандартным зоопарком рекурсивных схем.

| название | Как тип данных | Использование набора функций |
|---|---|---|
| Пары ценностей | `data Pair a = Pair aa` | `type Pair = I :*: I` |
| Сети «два-два» | `type Grid a = Pair (Pair a)` | `type Grid = Pair :.: Pair` |
| Естественные числа | `data Nat = Zero | Succ Nat` | `type Nat = Fix (K () :+: I)` |
| Списки | `data List a = Nil | Cons a (List a)` | `type List a = Fix (K () :+: K a :*: I)` |
| Двоичные деревья | `data Tree a = Leaf | Node (Tree a) a (Tree a)` | `type Tree a = Fix (K () :+: I :*: K a :*: I)` |
| Розовые деревья | `data Rose a = Rose a (List (Rose a))` | `type Rose a = Fix (K a :*: List :.: I)` |

Этот подход «набора» к разработке типов данных является идеей создания *общих* библиотек *программирования,* таких как `generics-sop` . Идея состоит в том, чтобы писать общие операции, используя набор, подобный представленному выше, а затем использовать класс типа для преобразования произвольных типов данных в их общее представление:

```
class Generic a where
    type Rep a  -- a generic representation built using a kit
    to :: a -> Rep a
    from :: Rep a -> a
```

## Функторы в теории категорий

Функтор определен в теории категорий как сохраняющее структуру отображение (а 'гомоморфизм') между категориями. В частности, (все) объекты отображаются на объекты, а (все) стрелки отображаются на стрелки, так что законы категории сохраняются.

Категорией, в которой объекты являются типами и морфизмами Haskell, являются функции Haskell, называемые **Hask** . Таким образом, функтор от **Hask** до **Hask** будет состоять из отображения типов в типы и отображения из функций в функции.

Взаимоотношения, которые теоретическая концепция этой категории относится к программированию программирования Haskell, `Functor` довольно прямая. Отображение от типов к типам принимает вид типа `f :: * -> *` , а отображение функций в функции принимает вид функции `fmap :: (a -> b) -> (fa -> fb)` , Объединяя их в классе,

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` - операция, которая принимает функцию (тип морфизма), `:: a -> b` и сопоставляет ее с

другой функцией `:: fa -> fb` . Предполагается (но оставленный программисту для обеспечения), что экземпляры `Functor` действительно являются математическими функторами, сохраняя категориальную структуру **Хаска** :

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

`fmap` поднимает функцию `:: a -> b` в подкатегорию **Хаска** таким образом, который сохраняет как существование любых точечных стрелок, так и ассоциативность композиции.

---

Класс `Functor` только кодирует *эндофункторы* на **Hask** . Но в математике функторы могут отображать между произвольными категориями. Более точное кодирование этой концепции будет выглядеть так:

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

Стандартный класс Functor - это особый случай этого класса, в котором исходная и целевая категории являются как **Hask** . Например,

```
instance Category (->) where        -- Hask
    id    = \x -> x
    f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

## Получение эффекта

`DeriveFunctor` языка `DeriveFunctor` позволяет GHC автоматически генерировать экземпляры `Functor` .

```
{-# LANGUAGE DeriveFunctor #-}

data List a = Nil | Cons a (List a) deriving Functor

-- instance Functor List where            -- automatically defined
--   fmap f Nil = Nil
--   fmap f (Cons x xs) = Cons (f x) (fmap f xs)

map :: (a -> b) -> List a -> List b
map = fmap
```

Прочитайте Функтор онлайн: https://riptutorial.com/ru/haskell/topic/3800/функтор

# глава 76: Функции более высокого порядка

## замечания

Функции более высокого порядка - это функции, которые выполняют функции в качестве параметров и / или возвращают функции в качестве возвращаемых значений.

## Examples

### Основы функций более высокого порядка

Просмотрите частичное приложение перед продолжением.

В Haskell функция, которая может принимать другие функции в качестве аргументов или возвращаемых функций, называется функцией более *высокого порядка* .

Ниже перечислены все функции *более высокого порядка* :

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

Они особенно полезны в том, что они позволяют нам создавать новые функции поверх тех, которые у нас уже есть, передавая функции в качестве аргументов другим функциям. Отсюда и название, функции *более высокого порядка* .

Рассматривать:

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

Эта способность легко создавать функции (например, частичное приложение, как используется здесь) является одной из функций, которая делает функциональное программирование особенно мощным и позволяет нам выводить короткие, элегантные

решения, которые в противном случае занимали бы десятки строк на других языках. Например, следующая функция дает нам количество выровненных элементов в двух списках.

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

## Лямбда-выражения

*Лямбда-выражения* аналогичны *анонимным функциям* на других языках.

Лямбда-выражения - это открытые формулы, которые также определяют переменные, которые должны быть связаны. Затем оценка (поиск значения вызова функции) достигается путем подстановки связанных переменных в тело лямбда-выражения с предоставленными пользователем аргументами. Проще говоря, лямбда-выражения позволяют нам выражать функции посредством привязки переменных и их замены .

Лямбда-выражения выглядят

```
\x -> let {y = ...x...} in y
```

В пределах лямбда-выражения переменные в левой части стрелки считаются связанными в правой части, т. Е. Телом функции.

Рассмотрим математическую функцию

```
f(x) = x^2
```

В качестве определения Haskell это

```
f    x =  x^2

f = \x -> x^2
```

что означает, что функция `f` эквивалентна лямбда-выражению `\x -> x^2` .

Рассмотрим параметр `map` функции высшего порядка, являющийся функцией типа `a -> b` . Если он используется только один раз в вызове для `map` и нигде в программе, его удобно указывать как выражение лямбда вместо того, чтобы называть такую функцию throwaway. Написанное в виде лямбда-выражения,

```
\x -> let {y = ...x...} in y
```

`x` имеет значение типа `a` , `...x...` является выражением Haskell, которое ссылается на переменную `x` , а `y` имеет значение типа `b` . Так, например, мы могли бы написать следующее

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

## Карринг

В Haskell все функции считаются curried: то есть все функции в Haskell принимают только *один* аргумент.

Возьмем функцию `div` :

```
div :: Int -> Int -> Int
```

Если мы будем называть эту функцию 6 и 2, то неудивительно получим 3:

```
Prelude> div 6 2
3
```

Однако это не совсем так, как мы думаем. Первый `div 6` оценивается и **возвращает функцию** типа `Int -> Int` . Затем эта полученная функция применяется к значению 2, которое дает 3.

Когда мы смотрим на подпись типа функции, мы можем перенести наше мышление с «принимает два аргумента типа `Int` » на «принимает один `Int` и возвращает функцию, которая принимает `Int` ». Это подтверждается, если учесть, что стрелки в обозначении типа связаны *справа* , поэтому `div` фактически можно читать следующим образом:

```
div :: Int -> (Int -> Int)
```

В общем, большинство программистов могут игнорировать это поведение, по крайней мере, пока они изучают язык. С теоретической точки зрения , «формальные доказательства легче, когда все функции обрабатываются равномерно (один аргумент в, один результат)».

Прочитайте Функции более высокого порядка онлайн:
https://riptutorial.com/ru/haskell/topic/4539/функции-более-высокого-порядка

# глава 77: Частичное применение

## замечания

Давайте проясним некоторые заблуждения, которые могут сделать новички.

Возможно, вы столкнулись с такими функциями, как:

```
max :: (Ord a) => a -> a -> a
max m n
  | m >= n = m
  | otherwise = n
```

Начинающие обычно рассматривают `max :: (Ord a) => a -> a -> a` as, которая принимает два аргумента (значения) типа `a` и возвращает значение типа `a` . Однако то, что действительно происходит, заключается в том, что `max` **принимает один аргумент** типа `a` и **возвращает функцию** типа `a -> a` . Затем эта функция принимает аргумент типа `a` и возвращает окончательное значение типа `a` .

Действительно, `max` можно записать как `max :: (Ord a) => a -> (a -> a)`

Рассмотрим сигнатуру типа `max` :

```
Prelude> :t max
max :: Ord a => a -> a -> a

Prelude> :t (max 75)
(max 75) :: (Num a, Ord a) => a -> a

Prelude> :t (max "Fury Road")
(max "Fury Road") :: [Char] -> [Char]

Prelude> :t (max "Fury Road" "Furiosa")
(max "Fury Road" "Furiosa") :: [Char]
```

`max 75` и `max "Fury Road"` могут не *выглядеть* как функции, но на самом деле они есть.

Путаница проистекает из того факта, что в математике и многих других обычных языках программирования нам разрешено иметь функции, которые принимают несколько аргументов. Однако в Haskell **функции могут принимать только один аргумент,** и они могут возвращать либо значения, такие как `a` , либо такие функции, как `a -> a` .

# Examples

### Частично примененная функция добавления

Мы можем использовать *частичное приложение* для «блокировки» первого аргумента.

После применения одного аргумента мы остаемся с функцией, которая ожидает еще один аргумент перед возвратом результата.

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

Затем мы можем использовать `addOne` , чтобы добавить его в `Int` .

```
> addOne 5
6
> map addOne [1,2,3]
[2,3,4]
```

### Возврат частично прикладной функции

Возвращение частично применяемых функций - это один из способов записи сжатого кода.

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

В этом примере (+ x) - частично применяемая функция. Обратите внимание, что второй параметр функции добавления необязательно указывать в определении функции.

Результатом вызова `add 5 2` составляет семь.

### Разделы

Разделение является кратким способом частичного применения аргументов к инфиксным операторам.

Например, если мы хотим написать функцию, которая добавляет «ing» в конец слова, мы можем использовать раздел, чтобы кратко определить функцию.

```
> (++ "ing") "laugh"
"laughing"
```

Обратите внимание, как мы частично применили второй аргумент. Обычно мы можем лишь частично применять аргументы в указанном порядке.

Мы также можем использовать левую секцию для частичного применения первого аргумента.

```
> ("re" ++) "do"
"redo"
```

Мы могли бы эквивалентно написать это, используя обычное префиксное частное приложение:

```
> ((++) "re") "do"
"redo"
```

# Примечание по вычитанию

Начинающие часто неправильно разделяют отрицание.

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

Это не работает, поскольку `-1` анализируется как литерал `-1` а не секционированный оператор `-` применяется к `1` . Функция `subtract` существует, чтобы обойти эту проблему.

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

Прочитайте Частичное применение онлайн: https://riptutorial.com/ru/haskell/topic/1954/частичное-применение

# глава 78: Шаблон Haskell & QuasiQuotes

## замечания

## Что такое шаблон Haskell?

Шаблон Haskell ссылается на объекты мета-программирования шаблонов, встроенные в GHC Haskell. Статья, описывающая первоначальную реализацию, можно найти здесь .

## Что такое этапы? (Или, что такое ограничение на сцену?)

Этапы относятся к тому, *когда* выполняется код. Обычно код вызывается только во время выполнения, но с шаблоном Haskell код может быть выполнен во время компиляции. «Нормальный» код - это этап 0, а код компиляции - этап 1.

Сценарное ограничение относится к тому, что на этапе 1 программа этапа 0 не может быть выполнена - это было бы эквивалентно возможности запуска любой *обычной* программы (а не только метапрограммы) во время компиляции.

По соглашению (и ради простоты реализации) код в текущем модуле всегда является этапом 0, а код, импортированный из всех других модулей, является этапом 1. По этой причине могут быть объединены только выражения из других модулей.

Обратите внимание, что программа 1-го этапа представляет собой выражение этапа 0 типа `Q Exp` , `Q Type` и т. Д .; но обратное неверно - не каждое значение (программа этапа 0) типа `Q Exp` является программой этапа 1,

Более того, поскольку сращивания могут быть вложенными, идентификаторы могут иметь этапы, превышающие 1. Тогда может быть обобщено ограничение на сцену - программа этапа *n* может не выполняться ни на одной стадии *m*> *n* . Например, в некоторых сообщениях об ошибках можно увидеть ссылки на такие этапы, которые превышают 1:

```
>:t [| \x -> $x |]

<interactive>:1:10: error:
    * Stage error: `x' is bound at stage 2 but used at stage 1
    * In the untyped splice: $x
      In the Template Haskell quotation [| \ x -> $x |]
```

# Использование шаблона Haskell приводит к ошибкам не в области видимости от несвязанных идентификаторов?

Обычно все объявления в одном модуле Haskell можно рассматривать как все взаимно-рекурсивные. Другими словами, каждое объявление верхнего уровня входит в объем каждого другого в одном модуле. Когда Template Haskell включен, правила определения области действия изменяются - модуль вместо этого разбивается на группы кода, разделенные с помощью сплайнов TH, и каждая группа является взаимно рекурсивной, и каждая группа охватывает все остальные группы.

## Examples

### Тип Q

Конструктор типа `Q :: * -> *` определенный в `Language.Haskell.TH.Syntax` - абстрактный тип, представляющий вычисления, которые имеют доступ к среде времени компиляции модуля, в котором выполняется вычисление. Тип `Q` также обрабатывает переменную подстановку, называемую *захватом имени* TH (и обсуждаемой здесь ). Все сращивания имеют тип `QX` для некоторого `X`

Среда компиляции включает в себя:

- идентификаторы в области видимости и информацию об упомянутых идентификаторах,
  - типы функций
  - типы и исходные типы данных конструкторов
  - полная спецификация деклараций типов (классы, типы семейств)
- расположение в исходном коде (строка, столбец, модуль, пакет), где происходит сращивание
- исправления функций (GHC 7.10)
- включенные расширения GHC (GHC 8.0)

Тип `Q` также имеет возможность генерировать новые имена, с функцией `newName :: String -> Q Name` . Обратите внимание, что имя не привязано нигде неявно, поэтому пользователь должен привязать его самостоятельно, и поэтому убедитесь, что полученное использование имени хорошо охвачено пользователем.

`Q` имеет экземпляры для `Functor,Monad,Applicative` и это основной интерфейс для манипулирования значениями `Q` вместе с комбинаторами, предоставляемыми в `Language.Haskell.TH.Lib` , которые определяют вспомогательную функцию для каждого

конструктора TH ast формы:

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

Обратите внимание, что `ExpQ` , `TypeQ` , `DecsQ` и `PatQ` являются синонимами для типов AST, которые обычно хранятся внутри `Q` типа.

Библиотека TH предоставляет функцию `runQ :: Quasi m => Q a -> ma` , и есть экземпляр `Quasi IO` , поэтому кажется, что тип `Q` - просто причудливый `IO` . Однако использование `runQ :: Q a -> IO a` вызывает действие `IO` которое *не* имеет доступа к какой-либо среде компиляции - это доступно только в реальном `Q` типе. Такие действия `IO` будут работать во время выполнения, если вы попытаетесь получить доступ к указанной среде.

## Карьера n-arity

Знакомый

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

функция может быть обобщена на кортежи произвольной арности, например:

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

Однако писать такие функции для кортежей arity 2 (например, 20) вручную было бы утомительно (и игнорируя тот факт, что наличие 20 кортежей в вашей программе почти наверняка сигнализирует о проблемах дизайна, которые должны быть исправлены с помощью записей).

Мы можем использовать Template Haskell для создания таких функций `curryN` для произвольного `n` :

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

Функция `curryN` принимает натуральное число и производит функцию карри этой арности, как Haskell AST.

```
curryN n = do
```

```
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
```

Сначала мы производим *новые* переменные типа для каждого из аргументов функции - один для входной функции и по одному для каждого из аргументов указанной функции.

```
  let args = map VarP (f:xs)
```

Выражение `args` представляет собой шаблон `f x1 x2 .. xn` . Обратите внимание, что шаблон представляет собой отдельный синтаксический объект - мы могли бы взять этот же шаблон и поместить его в лямбда или привязку функции или даже LHS привязки let (что было бы ошибкой).

```
      ntup = TupE (map VarE xs)
```

Функция должна строить кортеж аргумента из последовательности аргументов, что мы и сделали здесь. Обратите внимание на различие между переменными шаблона ( `VarP` ) и переменными выражения ( `VarE` ).

```
  return $ LamE args (AppE (VarE f) ntup)
```

Наконец, значение, которое мы производим, это AST `\f x1 x2 .. xn -> f (x1, x2, .. , xn)` .

Мы могли бы также написать эту функцию, используя цитаты и «поднятые» конструкторы:

```
...
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
        [| $(varE f) $(tupE (map varE xs)) |]
```

Обратите внимание, что цитаты должны быть синтаксически действительными, поэтому `[| \ $(map varP (f:xs)) -> .. |]` недействителен, потому что в регулярном Haskell нет способа объявить «список» шаблонов - вышеупомянутое интерпретируется как `\ var -> ..` и ожидается, что сплайсированное выражение будет иметь тип `PatQ` , т. е. один шаблон, а не список шаблонов.

Наконец, мы можем загрузить эту функцию TH в GHCi:

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

Этот пример адаптирован в основном отсюда .

### Синтаксис шаблона Haskell и Quasiquotes

Шаблон Haskell включен расширением `-XTemplateHaskell` GHC. Это расширение позволяет использовать все синтаксические функции в этом разделе. Подробные сведения о шаблоне Haskell приведены в руководстве пользователя .

# Сращивания

- Сплайсинг - это новый синтаксический объект, разрешенный Template Haskell, написанный как `$(...)` , где `(...)` - некоторое выражение.
- Между `$` и первым символом выражения не должно быть пробела; и Template Haskell переопределяет синтаксический анализ оператора `$` - например, `f$g` обычно анализируется как `($) fg` тогда как с включенным Template Haskell он анализируется как сплайсинг.
- Когда сращивание появляется на верхнем уровне, значение `$` может быть опущено. В этом случае сплайсированное выражение представляет собой всю строку.
- Сплайсинг представляет собой код, который запускается во время компиляции, чтобы получить Haskell AST, и что AST составлен как код Haskell и вставлен в программу
- Сплавы могут появляться вместо выражений, шаблонов, типов и объявлений верхнего уровня. Тип сращиваемого выражения в каждом случае, соответственно, `Q Exp` , `Q Pat` , `Q Type` , `Q [Decl]` . Обратите внимание, что сплайты объявления могут отображаться *только* на верхнем уровне, тогда как другие могут быть внутри других выражений, шаблонов или типов, соответственно.

# Котировки выражения (примечание: *не* QuasiQuotation)

- Котировка выражения - это новый синтаксический объект, написанный как один из следующих:
  - `[e|..|]` или `[|..|]` - `..` является выражением, а котировка имеет тип `Q Exp` ;
  - `[p|..|]` - `..` является шаблоном, а котировка имеет тип `Q Pat` ;
  - `[t|..|]` - `..` является типом, а предложение имеет тип `Q Type` ;
  - `[d|..|]` - `..` является списком деклараций, а цитата имеет тип `Q [Dec]` .
- Котировка выражения принимает программу времени компиляции и выдает АСТ,

представленную этой программой.

- Использование значения в цитате (например, `\x -> [| x |]` ) без сращивания соответствует синтаксическому сахару для `\x -> [| $(lift x) |]` , где `lift :: Lift t => t -> Q Exp` исходит из класса

```
class Lift t where
  lift :: t -> Q Exp
  default lift :: Data t => t -> Q Exp
```

# Типизированные сращивания и котировки

- Типизированные сращивания аналогичны ранее упомянутым (нетипизированным) сращиваниям и записываются как `$$(..)` где `(..)` - выражение.
- Если `e` имеет тип `Q (TExp a)` то `$$e` имеет тип `a` .
- Типизированные цитаты принимают вид `[||..||]` где `..` - выражение типа `a` ; результирующая котировка имеет тип `Q (TExp a)` .
- Типированное выражение может быть преобразовано в нетипизированные: `unType :: TExp a -> Exp` .

# QuasiQuotes

- QuasiQuotes обобщает цитаты с выражениями - ранее, синтаксический анализатор, используемый котировкой выражения, является одним из фиксированных множеств ( `e,p,t,d` ), но QuasiQuotes позволяет определить пользовательский парсер и использовать его для создания кода во время компиляции. Квази-котировки могут появляться во всех тех же контекстах, что и обычные котировки.
- `[iden|...|]` как `[iden|...|]` , где `iden` является идентификатором типа `Language.Haskell.TH.Quote.QuasiQuoter` .
- `QuasiQuoter` просто состоит из четырех парсеров, по одному для каждого из разных контекстов, в которых могут появляться цитаты:

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

# имена

- Идентификаторы Haskell представлены типом `Language.Haskell.TH.Syntax.Name` . Имена образуют листья абстрактных деревьев синтаксиса, представляющие программы Haskell в Template Haskell.
- Идентификатор, который в настоящее время находится в области видимости, может быть превращен в имя с именем: `'e` или `'T` В первом случае `e` интерпретируется в области выражения, тогда как во втором случае `T` находится в области типов (напомним, что типы конструкторов типов и значений могут совместно использовать имя без каких-либо ошибок в Haskell).

Прочитайте Шаблон Haskell & QuasiQuotes онлайн: https://riptutorial.com/ru/haskell/topic/5216/шаблон-haskell--amp--quasiquotes

## кредиты

| S. No | Главы | Contributors |
|---|---|---|
| 1 | Начало работы с языком Haskell | 4444, Adam Wagner, alejosocorro, Amitay Stern, arseniiv, baxbaxwalanuksiwe, Benjamin Hodgson, Benjamin Kovach, Burkhard, Carsten, colinro, Community, Daniel Jour, Echo Nolan, erisco, gdziadkiewicz, HBu, J Atkin, Jan Hrcek, Jules, Kwarrtz, leftaroundabout, M. Barbieri, mb21, mnoronha, Mr Tsjolder, ocharles, pouya, R B, Ryan Hilbert, Sebastian Graf, Shoe, Stephen Leppik, Steve Trout, Tim E. Lord, Turion, user239558, Will Ness, zbw, λlεx |
| 2 | Attoparsec | λlεx |
| 3 | Data.Aeson - JSON в Хаскелле | Chris Stryczynski, Janos Potecki, Matthew Pickering, rob, xuh |
| 4 | data.text | Benjamin Hodgson, dkasak, Janos Potecki, jkeuhlen, mnoronha , unhammer |
| 5 | GHCJS | Mikkel |
| 6 | GTK3 | bleakgadfly |
| 7 | IO | 3442, Benjamin Hodgson, David Grayson, J. Abrahamson, Jan Hrcek, John F. Miller, leftaroundabout, mnoronha, Sarah, user2407038, Will Ness, λlεx |
| 8 | Monad Transformers | Damian Nadales |
| 9 | Monoid | Benjamin Hodgson, Kwarrtz, mnoronha, Will Ness |
| 10 | Profunctor | zbw |
| 11 | Reader / ReaderT | Chris Stryczynski |
| 12 | XML | Mikkel |
| 13 | zipWithM | zbw |
| 14 | Алгоритмы сортировки | Brian Min, pouya, Romildo, Shoe, Vektorweg, Will Ness |
| 15 | Анализ HTML с объективом и | Kostiantyn Rybnikov, λlεx |

|  |  |  |
|---|---|---|
|  | объективом |  |
| 16 | Аппликативный метод | Benjamin Hodgson, Kritzefitz, mnoronha, Will Ness, λlεx |
| 17 | арифметика | ocramz |
| 18 | Базы данных | λlεx |
| 19 | Бесплатные монады | Benjamin Hodgson, Cactus, J. Abrahamson, pyon, sid-kap |
| 20 | бифунктора | Benjamin Hodgson, liminalisht |
| 21 | Буферы протокола Google | Janos Potecki, λlεx |
| 22 | Быстрая проверка | Benjamin Hodgson, gdziadkiewicz, Matthew Pickering, Steve Trout |
| 23 | Веб-разработка | arrowd, λlεx |
| 24 | векторы | Benjamin Hodgson, λlεx |
| 25 | взыскательность | Benjamin Hodgson, Benjamin Kovach, Cactus, user2407038, Will Ness |
| 26 | Государственная Монада | Apoorv Ingle, Kritzefitz, Luis Casillas |
| 27 | Графика с блеском | Wysaard, Zoey Hewll |
| 28 | Дата и время | mnoronha, Will Ness, λlεx |
| 29 | Доверенные | Benjamin Hodgson |
| 30 | Интерфейс внешней функции | arrowd, bleakgadfly, crockeea |
| 31 | интрига | tlo |
| 32 | Использование GHCi | Benjamin Hodgson, James, Janos Potecki, mnoronha, RamenChef, wizzup, λlεx |
| 33 | Контейнеры - Data.Map | Cactus, Itbot, Will Ness, λlεx |
| 34 | Кортежи (пары, тройки, ...) | Cactus, mnoronha, Toxaris, λlεx |

| 35 | логирование | λlεx |
|---|---|---|
| 36 | Модули | Benjamin Hodgson, Kapol, Will Ness, λlεx |
| 37 | Монады | Alec, Benjamin Hodgson, Cactus, fgb, Kapol, Kwarrtz, Lynn, Mario Román, Matthew Pickering, Will Ness, λlεx |
| 38 | Обобщенные типы алгебраических данных | mniip |
| 39 | Общие монады как свободные монады | leftaroundabout |
| 40 | Общие расширения языка GHC | Antal Spector-Zabusky, Bartek Banachewicz, Benjamin Hodgson, Benjamin Kovach, Cactus, charlag, Doomjunky, Janos Potecki, John F. Miller, K48, Kwarrtz, leftaroundabout, Mads Buch, Matthew Pickering, mkovacs, mniip, phadej, Yosh, λlεx |
| 41 | Общие функторы как основа cofree comonads | leftaroundabout |
| 42 | объектив | Bartek Banachewicz, bennofs, chamini2, dfordivam, dsign, Hjulle, J. Abrahamson, John F. Miller, Kwarrtz, Matthew Pickering, λlεx |
| 43 | Объявления о фиксации | Alec, Will Ness |
| 44 | Операторы Infix | leftaroundabout, mnoronha |
| 45 | оптимизация | λlεx |
| 46 | оспоримый | Benjamin Hodgson, ErikR, J. Abrahamson |
| 47 | параллелизм | λlεx |
| 48 | Перегруженные литералы | Adam Wagner, Carsten, Kapol, leftaroundabout, pdexter |
| 49 | Переписать правила (GHC) | Cactus |
| 50 | Полиморфизм произвольного ранга с RankNTypes | ocharles |

| 51 | Потоковое IO | λlεx |
|---|---|---|
| 52 | Реактивный-банан | arrowd, Dair, Undreren |
| 53 | Рекурсивные схемы | arseniiv, Benjamin Hodgson |
| 54 | Роль | xuh |
| 55 | Семейство типов | leftaroundabout, mniip, xuh |
| 56 | Синтаксис в функциях | Delapouite, James, Janos Potecki, Will Ness, λlεx |
| 57 | Синтаксис вызова функции | Zoey Hewll |
| 58 | Синтаксис записи | Cactus, Janos Potecki, John F. Miller, Mario, Matthew Pickering, Will Ness, λlεx |
| 59 | складываемый | Benjamin Hodgson, Cactus, Dair, David Grayson, J. Abrahamson, Jan Hrcek, mnoronha |
| 60 | совпадение | Janos Potecki, λlεx |
| 61 | Создание пользовательских типов данных | Janos Potecki, Kapol |
| 62 | Состав функции | arseniiv, Will Ness |
| 63 | Списки | Benjamin Hodgson, erisco, Firas Moalla, Janos Potecki, Kwarrtz, Lynn, Matthew Pickering, Matthew Walton, Mirzhan Irkegulov, mnoronha, Tim E. Lord, user2407038, Will Ness, Yosh, λlεx |
| 64 | Список рекомендаций | Cactus, Kwarrtz, mnoronha, Will Ness |
| 65 | стек | 4444, curbyourdogma, Dair, Janos Potecki |
| 66 | Стрелы | artcorpse, leftaroundabout |
| 67 | Теория категорий | arrowd, Benjamin Hodgson, Benjamin Kovach, J. Abrahamson, Mario Román, mmlab, user2407038 |
| 68 | Тестирование с вкусной | tlo |
| 69 | Тип приложения | Luka Horvat, λlεx |

| | | |
|---|---|---|
| 70 | Типированные отверстия | Cactus, leftaroundabout, user2407038, Will Ness |
| 71 | Типная алгебра | Mario Román |
| 72 | Типы классов | arjanen, Benjamin Hodgson, Billy Brown, Cactus, Dair, gdziadkiewicz, J. Abrahamson, Kwarrtz, leftaroundabout, mnoronha, RamenChef, Will Ness, zeronone, λlεx |
| 73 | Типы фантомов | Benjamin Hodgson, Christof Schramm |
| 74 | трубы | 4444, Benjamin Hodgson, Stephane Rolland, λlεx |
| 75 | Функтор | Benjamin Hodgson, Delapouite, Janos Potecki, liminalisht, mathk, mnoronha, Will Ness, λlεx |
| 76 | Функции более высокого порядка | Community, Doruk, Matthew Pickering, mnoronha, Will Ness, λlεx |
| 77 | Частичное применение | arseniiv, Benjamin Hodgson, CiscoIPPhone, Dair, Matthew Pickering, Will Ness |
| 78 | Шаблон Haskell & QuasiQuotes | user2407038 |