Расширения, библиотеки, подходы и дополнительные материалы

Михаил Беляев

December 7, 2016

Functional programmer:
(noun) One who names variables «x», names functions «f», and names code patterns «zygohistomorphic prepromorphism»

©Some twitter guy

- Как вы уже поняли, Haskell — не самый простой язык
- Особенно для процедурных программистов
- Сегодня мы поговорим немного о том, что он дал миру

Расширения языка

- Haskell поддерживает расширения, как на уровне всей программы, так и отдельных файлов

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype MyNumber = MyNumber Int
    deriving(Eq, Ord, Integral)
newtype MyMonad a = MyMonad (List a)
    deriving(Monoid, Functor, Monad)
```

Расширения языка

Интересные расширения:

- CPP — поддержка препроцессора из C
- GeneralizedNewtypeDeriving — deriving любых классов через newtype
- TupleSections — секции для кортежей
- OverloadedStrings — создание своих типов для строк
- Различные расширения для классов и инстансов

ViewPatterns

Позволяет использовать функции внутри паттернов

```
data MyType = C1 Int | C2 String | C3 Product
cost :: Product -> Int

fun (C1 i) = ...
fun (C2 s) = ...
fun (C3 (cost -> 2)) = ...
fun (C3 (cost -> (> 5) -> True)) = ...
```

PatternSynonyms

- Позволяет описывать паттерны в отрыве от структур данных
- В Scala похожий механизм через метод `unapply`
- Определяет одновременно и паттерн, и функцию
- Позволяет, в том числе, сделать удобные паттерны для чужих типов

```
data Term = ... | BinaryTerm Char Term Term
pattern Plus a b = BinaryTerm '+' a b
pattern Minus a b = BinaryTerm '-' a b
```

PatternSynonyms

- Позволяет описывать паттерны в отрыве от структур данных
- В Scala похожий механизм через метод `unapply`
- Определяет одновременно и паттерн, и функцию
- Позволяет, в том числе, сделать удобные паттерны для чужих типов

```
xplusy = Plus x y

eval :: Term -> Integer
...
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
```

Data/Typeable/Generic

- Любой тип в Haskell, кроме функции, можно рассматривать как дерево
- В листьях этого дерева примитивные типы

Data/Typeable/Generic

- Любой тип в Haskell, кроме функции, можно рассматривать как дерево
- В листьях этого дерева примитивные типы

- Идея за Data/Typeable/Generic — обобщать функции над этими деревьями

Идея:

- Вы описываете свой алгоритм над классами Data/Typeable
- После этого он работает на **любых** типах данных, для которых есть инстансы
- Generic это абсолютно аналогичный механизм, только придуманный авторами GHC

DeriveDataTypeable/DeriveGeneric

- Генерирует инстансы Data/Typeable/Generic для любых ваших типов данных
- Где это используется?

DeriveDataTypeable/DeriveGeneric

```haskell
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Aeson

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

instance ToJSON Person
instance FromJSON Person
```

DeriveDataTypeable/DeriveGeneric

```
> encode $ Person "Vasya" 23
"{\"name\":\"Vasya\", \"age\":23}"
```

Аналогичные реализации есть для XML и многих других форматов

Где это ещё используется?

- Система преобразования документов Pandoc
- Скрипты сборки документации

```
main :: IO ()
main = toJSONFilter guillemots
  where guillemots (Just (Format fmt)) (Quoted DoubleQuote str)
                        | fmt == "latex" || fmt == "beamer" =
            [RawInline latex "{<<}"]
                ++ str
                ++ [RawInline latex "{>>}"] where latex = Format "latex"
        guillemots (Just html@(Format "html")) (Quoted DoubleQuote str) =
            [RawInline html "&laquo;"]
                ++ str
                ++ [RawInline html "&raquo;"]
        guillemots _ x = [x]
```

Template Haskell

- Система макросов, которая позволяет на Haskell генерировать программы на Haskell
- Встроенная система шаблонов (quasiquoting)

```
dtype = [t| String |]
func = [t| Integer -> $dtype |]
```

- Каждый шаблон возвращает значение типа `Q <something>`
- `Q` это монада, которая в зависимости от контекста реализована по-разному

Template Haskell

Написав генератор кода в одном файле, вы можете написать в другом файле вашей программы

```
$( codeGenerator ''Type 'function )
```

Сгенерированный код автоматически подцепляется компилятором в процессе сборки

Language.Haskell.Exts

- Полноценный парсер языка Haskell с поддержкой всей грамматики и всех расширений
- Если вам нужно динамически разбирать файлы на Haskell из Haskell
- Существуют также интерпретаторы Haskell на Haskell

Парсер-комбинаторы

На данный момент самой известной является библиотека `Parsec`, самой быстрой — библиотека `Attoparsec`

Существует большое количество подражаетелей в других языках:

- `JParsec` и `ParsecJ` для Java
- `Bennu` для Javascript
- `Parsy`, `Picoparse`, `FuncparserLib` для Python
- `Boost::Spirit` и другие для C++
- Стандартные библиотеки Scala, F# и некоторых других языков
- и т.д.

Основная идея

- Есть базовый набор простых парсеров, которые умеют разбирать примитивный ввод
- Есть набор простых комбинаторов, позволяющих на основе одних парсеров строить другие
- Можно мешать логику с парсингом естественным образом
- Класс грамматик, которые так можно разбирать, называется PEG и плохо укладывается в классификацию из трансляторов

Простые парсеры

```haskell
-- Принимаем 1 символ, возвращаем его же
char :: Char -> Parser Char
-- Принимаем символы из диапазона
range :: Char -> Char -> Parser Char
-- Принимаем символы из списка
oneOf :: [Char] -> Parser Char
```

Простые парсеры

```haskell
-- Принимаем 1 символ, возвращаем его же
char :: Char -> Parser Char
-- Принимаем символы из диапазона
range :: Char -> Char -> Parser Char
-- Принимаем символы из списка
oneOf :: [Char] -> Parser Char
```

Или, более общий вариант

```haskell
conforms :: (Char -> Bool) -> Parser Char
char c = conforms (== c)
range a b = conforms (\c -> c >= a && c <= b)
oneOf cs = conforms (\c -> c `elem` cs)
```

Простые парсеры: квазипарсеры

```haskell
-- Не пытаться ничего читать,
-- сразу вернуть ошибку
parserError :: Error -> Parser a
-- Не пытаться ничего читать,
-- сразу вернуть значение
parserSuccess :: a -> Parser a
```

Парсер-комбинаторы: простые

```haskell
-- Всё, кроме заданного
parserNot :: (Parser Char) -> Parser Char
-- Попробовать первый, если не сработал, то второй
parserOr :: Parser a ->
            Parser b ->
            Parser (Either a b)
```

Парсер-комбинаторы: монадные операции

```haskell
-- Парсер является функтором!
parserMap :: (a -> b) -> Parser a -> Parser b
-- Парсер является монадой!
parserReturn = parserSuccess
parserFail = parserError
parserFlatten :: (Parser (Parser a)) -> Parser a
```

Пример: разбор номера группы

```
data GroupNumber = GroupNumber Integer Integer
parseNumber = read <$> parserMany1 digit
parseGroupNumber = do{ dept <- parseNumber;
                       char '/';
                       gp <- parseNumber;
                       return $ GroupNumber dept gp  }
parseManyGNs = do{ gn <- parseGroupNumber;
                   gns <- parserMany parseGroupNumber';
                   return (gn: gns) }
                 where
                  parseGroupNumber' = do { char ',';
                                           parseGroupNumber }
```

Парсер-комбинаторы: Packrat

- Подход к построению парсер-комбинаторов, который гарантирует линейное время работы на любом входе для любой PEG-грамматике
- Работает за счёт ленивости
- Общая идея: в каждой точке парсим **все** нетерминалы сразу, но из-за ленивости мы всегда разбёрем только те, которые нужны
- К сожалению, в другие языки перенести сложно

Quickcheck

- Подход к тестированию, зародившийся в Haskell, который сейчас медленно захватывает мир
- Blackbox random testing с финтифлюшками

Quickcheck

Общая идея:

- Есть класс Arbitrary для генерации случайных данных
- Есть инстансы для примитивных типов
- Можно генерировать новые через Generic/Data/Typeable
- PROFIT

Quickcheck: как это работает

- Генерируем тонну случайных данных
- Если на каком-то наборе тест упал, делаем шринкинг
    - Шринкинг это процесс «уменьшения» данных до тех пор, пока тест продолжает падать
- Выдаём уменьшенный тест пользователю

Как ни странно, данный подход позволяет **очень** быстро и почти без усилий находить широкие классы ошибок

Quickcheck: библиотеки

Собственно, библиотека Quickcheck

Существует большое количество подражаетелей в других языках:

- `quickcheck` для Erlang
- `java-quickcheck` для Java
- `scalacheck` для Scala
- Есть реализации для Python
- Скоро, возможно, начнёт входить в штатные тестовые фреймворки

Что ещё есть в Haskell? (О чем я вам не расскажу)

- Легковесные потоки через `forkIO`
  - Миллионы потоков прекрасно работают в одном приложении почти без оверхеда
- «Честный» STM (Software Transactional Memory)
- Очень быстрая система ввода-вывода на основе `Bytestring`
- Интерфейсы взаимодействия с другими языками через FFI
- Continuations and asynchronous programming

Что ещё есть в Haskell? (О чем я вам не расскажу)

- Lenses and optics (удобные абстракции для работы с immutable объектами)
- Тонны матана
  - Классы почти для всего, что придумано математиками
  - Стрелки как **ещё** более мощная абстрация над функторами-монадами-комонадами и т.д.
- **Ещё** более мощная система типов:
  - Universal and existential types
  - Quasi-dynamic types
  - GADTs, type families, kinds, dependent types, etc.
- И много чего ещё

Что ещё есть в ФП? (Но нет в Haskell)

- Сверхэффективные оптимизации (см. OCaml)
- TypeProviders и более дружественные системы макросов
  - LISP (в том числе, всё в динамике)
  - F#
  - OCaml (camlp4/camlp5)
- **Ещё** более мощные системы типов:
  - Доказатели теорем: F*, Coq, HOL, Gwelf, Agda, etc.
- Модули с параметрами (OCaml)
- Зигохисторические препроморфизмы)
- И много чего ещё

Выводы

- В процессе данного курса мы только едва коснулись вершины айсберга
- Широта кругозора никогда не бывает лишней
- До встречи на зачёте