# Анонимные записи в Haskell

(Anonymous Records in Haskell)

Никита Волков

# Введение

# Введение

Выучи уже Haskell во имя добра!

http://learnyouahaskell.com/

# Терминология

# Терминология:
## Типы данных и конструкторы значений

```haskell
-- Enumeration
data Gender = Male | Female

-- Composite single-constructor
data Point = Point Int Int

-- Parametric single-constructor
data Identified a = Identified a Int

-- Parametric multi-constructor
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
```

# Терминология:
# Синоним типа

Декларация, которая позволяет присвоить альтернативное имя существующему типу или конкретизации полиморфного типа.

```
type AnAlternativeNameForPoint =
  Point

type FailableGender =
  Either String Gender
```

# Терминология: Классы типов

Конструкция в системе типов Haskell, являющая собой основной инструмент для работы с полиморфизмом.

# Терминология: Классы типов

Java-версия класса типов, который абстрагируется над чем-то, что может иметь пустое значение или соединяться с другим значением своего же типа.

```
interface Monoid<A> {
  A empty();
  A append(A a, A b);
}
```

# Терминология: Классы типов

Инстанционализируя данный интерфейс мы предоставляем поддержку для конкретных типов данных.

Вот, для примера, моноид для строки:

```
public final class Monoids {

  public static final Monoid<String> string =
    new Monoid<String>() {
      public String empty() {
        return "";
      }
      public String append(String a, String b) {
        return a + b;
      }
    };
```

# Терминология: Классы типов

Моноид для суммирования чисел:

```
public static final Monoid<Integer> intSum =
  new Monoid<Integer>() {
    public Integer empty() {
      return 0;
    }
    public Integer append(Integer a, Integer b) {
      return a + b;
    }
  };
```

# Терминология: Классы типов

В случае чисел есть и моноид для перемножения:

```
public static final Monoid<Integer> intProduct =
  new Monoid<Integer>() {
    public Integer empty() {
      return 1;
    }
    public Integer append(Integer a, Integer b) {
      return a * b;
    }
  };
```

# Терминология: Классы типов

Более неожиданный моноид: для вычислений.

```
public static final Monoid<Runnable> runnable =
  new Monoid<Runnable>() {
    public Runnable empty() {
      return new Runnable() {
        public void run() {}
      };
    }
    public Runnable append(final Runnable a, final Runnable b) {
      return new Runnable() {
        public void run() {
          a.run();
          b.run();
        }
      };
    }
  };
```

# Терминология: Классы типов

Ещё более неожиданный моноид: для вычислений с результатом. Сам использует моноид для результата.

```
public static <A> Monoid<Callable<A>> callable(final Monoid<A> resultMonoid) {
  return new Monoid<Callable<A>>() {
    public Callable<A> empty() {
      return new Callable<A>() {
        public A call() throws Exception {
          return resultMonoid.empty();
        }
      };
    }
    public Callable<A> append(final Callable<A> a, final Callable<A> b) {
      return new Callable<A>() {
        public A call() throws Exception {
          return resultMonoid.append(a.call(), b.call());
        }
      };
    }
  };
}
```

# Терминология: Классы типов

Вышеупомянутое нужно лишь для того чтобы иметь возможность писать полиморфные функции, в которых мы абстрагируемся от конкретного типа данных.

Пример: общая функция для объединения списка элементов в один элемент.

```
public static <A> A fold(Iterable<A> iterable,
                         Monoid<A> elementMonoid) {
  A result = elementMonoid.empty();
  for (A element : iterable) {
    result = elementMonoid.append(result, element);
  }
  return result;
}
```

# Терминология: Классы типов

Так как мы уже абстрагировались от всех логических проблем, применение объявленного функционала к конкретным типам – проблема тривиальной компоновки.

```
public static String mergedString(Iterable<String> iterable) {
  return Monoids.fold(iterable, Monoids.string);
}

public static Integer product(Iterable<Integer> iterable) {
  return Monoids.fold(iterable, Monoids.intProduct);
}

public static Callable<Integer>
                  callableProduct(Iterable<Callable<Integer>> iterable) {
  return Monoids.fold(iterable, Monoids.callable(Monoids.intProduct));
}
```

# Терминология: Классы типов

Вот так данный класс типов имплементирован в Haskell:

```haskell
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

instance Monoid String where
  mempty = ""
  mappend a b = a ++ b
```

# Терминология: Классы типов

Так выглядит полиморфная функция:

```
fold :: Monoid a => [a] -> a
fold list = case list of
  a : b -> mappend a (fold b)
  [] -> mempty
```

Обратите внимание, что в данном случае моноид является не параметром функции, а её условием.

Иными словами, данная функция требует наличия нужного нам моноида. Т.е., в отличие от наших примеров в Java, инстансы классов типов передаются неявно (подразумеваются).

# Терминология:
# Запись (Record)

Композитный тип, позволяющий обращаться к его составляющим по имени. Только и всего.

В случае Java это был бы POJO:

```
class Point {
  int x;
  int y;
}
```

В случае Haskell это синтаксическое расширение для декларации типов данных:

```
data Point = Point {
  x :: Int,
  y :: Int
}
```

# Как записи работают в Haskell сейчас

Декларация типа данных с использованием синтаксиса для записей просто “рассахаривается” компилятором в объявления одноимённых функций.

Таким образом, следующий код:

```haskell
data Person = Person { name :: String, age :: Int }
```

Превращается в

```haskell
data Person = Person String Int

name :: Person -> String
name (Person theName theAge) = theName

age :: Person -> Int
age (Person theName theAge) = theAge
```

# Как записи работают в Haskell сейчас

Изменение значений записей производится с использованием специального синтаксиса:

```haskell
data Person = Person { name :: String, age :: Int }

incrementPerson'sAge :: Person -> Person
incrementPerson'sAge person =
  person {
    age = age person + 1
  }
```

# В чём проблема?

# В чём проблема?
# 1. Конфликты имён

В силу того, что поля преобразуются в функции и перегрузки функций в Haskell нет, нет и возможности сосуществовать двум записям в одном модуле, если они используют одни и те же имена.

Данный код не будет компилироваться:

```
data A = A { field :: String }
data B = B { field :: String }
```

# В чём проблема?
# 2. Частичность

Генерируемые функции определены не для всех значений. Из-за этого, несмотря на то, что следующий код пройдёт компиляцию без ошибок, он выдаст ошибку только в рантайме:

```
data A = A1 { field1 :: String } |
       | | | | A2 { field2 :: String }

main = print $ field1 $ A2 "abc"
```

# В чём проблема?
# 3. Конфликты типов

Одно имя может быть использовано только для одного типа данных.

Следующий код пройдёт компиляцию:

```
data A = A1 { field :: Int } |
       | A2 { field :: Int }
```

Этот - нет:

```
data A = A1 { field :: String } |
       | A2 { field :: Int }
```

# В чём проблема?
# 4. Избыточность конструктора

На практике записи крайне редко используются для типов данных со множеством конструкторов.

Набирание “Person” дважды в следующем коде явно избыточно:

```
data Person = Person { name :: String, birthday :: Date }
```

# В чём проблема?
# 5. Изменение полей

Муторность изменения полей записей внутри других записей растёт экспоненциально по отношению к количеству уровней такого вложения. Чем глубже мы пытаемся обратиться – тем больше нам приходится повторяться.

```
data Person = Person { name :: String, birthday :: Date }
data Date = Date { year :: Int, month :: Int, day :: Int }

incrementPerson'sBirthdayYear :: Person -> Person
incrementPerson'sBirthdayYear person =
  person {
    birthday = (birthday person) {
      year = (year (birthday person)) + 1
    }
  }
```

# Линза (Lens)

Комбинируемая абстракция, которая решает проблему изменения полей вложенных записей.

```
incrementPerson'sBirthdayYear :: Person -> Person
incrementPerson'sBirthdayYear person =
  over (birthday . year) succ person
```

- “over” - функция-комбинатор, позволяющая применять функцию к значению поля, на которое ссылается линза
- “birthday” и “year” - линзы, вместе образующие единую линзу при помощи комбинатора композиции, обозначаемого точкой.
- “succ” - функция, прибавляющая единицу к числу

# Линза (Lens)

Нуждается в шаблонном коде.

```
data Person = Person { _name :: String, _birthday :: Date }
data Date = Date { _year :: Int, _month :: Int, _day :: Int }

mkLenses ''Person
mkLenses ''Date
```

# Линза (Lens)

Не решает остальных проблем системы записей:

1. Конфликты имён
2. Частичность
3. Конфликты типов
4. Избыточность конструктора
5. ~~Изменение полей~~

# Решение:
# Анонимные записи

В данный момент имплементированы как препроцессор компилятора. Подробнее здесь:

http://hackage.haskell.org/package/record

# Анонимные записи: Декларации типов

```
type Person = {~
  name :: String,
  birthday :: {~ year :: Int, month :: Int, day :: Int },
  country :: Country
}

type Country = {~
  name :: String,
  language :: String
}
```

- Отсутствие конфликтов имён. И “Person”, и “Country” используют поле “name”.
- Нет нужды декларировать записи, что используется в случае с полем “birthday”.

# Анонимные записи: Декларации типов

Несмотря на то, что анонимные записи являют собой тип, а не декларацию, они по-прежнему могут быть использованы для объявления новых типов без каких-либо накладных расходов.

Для этого нужно использовать конструкцию “newtype”:

```
newtype Event = Event {~ name :: String, time :: UTCTime }
```

# Анонимные записи: Декларации типов

Если нужно, анонимные записи можно использовать и для объявления типов со множеством конструкторов:

```
data Event = Event1 {~ name :: String, time :: UTCTime } |
           | Event2 {~ name :: String, time :: LocalTime }
```

# Анонимные записи: Расход памяти

Следующие типы занимают абсолютно одинаковое количество памяти, несмотря на то, что второй основан на анонимной записи:

```
data Event = Event { name :: String, time :: UTCTime }

newtype Event = Event {~ name :: String, time :: UTCTime }
```

# Анонимные записи: Расход памяти

То же относится и к строгим записям:

```
data Event = Event { name :: !String, time :: !UTCTime }

newtype Event = Event {! name :: String, time :: UTCTime }
```

# Анонимные записи: Расход памяти

Следующий тип займёт меньше памяти благодаря использованию оптимизации по устранению промежуточных конструкторов во вложенных типах.

```
data Event = Event { name :: {-# UNPACK #-} !String,
                     time :: {-# UNPACK #-} !UTCTime }
```

К сожалению, к анонимным записям данная оптимизация не применима. Стоит отметить, однако, что “UNPACK” может уменьшать производительность.

# Анонимные записи: Производительность

В точности такое же, как и у стандартных типов данных и лучше, чем у типов, использующих преждеупомянутую прагму “UNPACK”.

# Анонимные записи: Изменение значений полей

Никакого велосипеда. Просто используем линзы.

Препроцессор предоставляет специальный сахар для объявления линз, используя символ “@”. Никаких издержек.

```
type Person = {~
  name :: String,
  birthday :: {~ year :: Int, month :: Int, day :: Int }
}

incrementPerson'sBirthdayYear :: Person -> Person
incrementPerson'sBirthdayYear person =
  over (@birthday . @year) succ person

getPerson'sBirthdayYear :: Person -> Int
getPerson'sBirthdayYear person =
  view (@birthday . @year) person
```

# Анонимные записи: Объявление значений

Всё прямолинейно:

```
person :: Person
person =
  {~
    name = "Yuri Alekseyevich Gagarin",
    birthday = {~ year = 1934, month = 3, day = 9 }
  }
```

# Анонимные записи: Объявление значений

Для удобства есть ещё и синтаксис частичного объявления, который генерирует функцию, возвращающую значение.

```
personByName :: String -> Person
personByName =
  {~
    name,
    birthday = {~ year = 1934, month = 3, day = 9 }
  }
```

Что, как понимают знающие, конечно же, особенно полезно когда речь заходит об аппликативных функторах:

```
getPerson :: Applicative f => f Person
getPerson =
  {~ name, birthday } <$> getName <*> getBirthday
```

# Анонимные записи: Все проблемы решены!

~~1. Конфликты имён~~
~~2. Частичность~~
~~3. Конфликты типов~~
~~4. Избыточность конструктора~~
~~5. Изменение полей~~

# Анонимные записи: Синтаксис

Никаких конфликтов с существующим синтаксисом Haskell!

# Анонимные записи: Как они устроены?

Библиотека предоставляет набор полиморфных типов данных, представляющих собой строгие и ленивые записи с арностью до 24.

```
data LazyRecord2 (n1 :: Symbol) v1
                 (n2 :: Symbol) v2 = LazyRecord2 v1 v2

data LazyRecord3 (n1 :: Symbol) v1
                 (n2 :: Symbol) v2
                 (n3 :: Symbol) v3 = LazyRecord3 v1 v2 v3
...

data StrictRecord2 (n1 :: Symbol) v1
                   (n2 :: Symbol) v2 = StrictRecord2 !v1 !v2
...
```

# Анонимные записи: Как они устроены?

Строчные значения на уровне типов используются для обозначения имён полей.

Так как это значения, а понятие “пространство имён” в принципе не применимо к значениям, проблема конфликтов имён отпадает автоматически.

# Анонимные записи: Как они устроены?

Декларация типа:

```
type Person = {~
  name :: String,
  birthday :: {~ year :: Int, month :: Int, day :: Int }
}
```

Преобразуется препроцессором в:

```
type Person =
  LazyRecord2
    "birthday" (LazyRecord3 "day" Int "month" Int "year" Int)
    "name" String
```

Обратите внимание на пересортировку полей по алфавиту...

# Анонимные записи: Как они устроены?

Пересортировка полей позволяет добиться следующего свойства:

```
{~ year = 1958, month = 1, day = 18 }
  ==
{~ month = 1, day = 18, year = 1958 }
```

Иными словами, имена полей предопределяют запись, а позиции – нет.

# Анонимные записи: Как они устроены?

Класс типов используется для работы с полями. Все предобъявленные в библиотеке типы записей имеют инстансы этого класса.

Ниже представлена упрощённая версия реализации.

```
class Field (n :: Symbol) r v where
  fieldLens :: FieldName n -> Lens r v

data FieldName (n :: Symbol)

instance Field n1 (Record2 n1 v1 n2 v2) v1 where ...
instance Field n2 (Record2 n1 v1 n2 v2) v2 where ...
```

# Анонимные записи: Как они устроены?

Выражение ссылки на поле, как следующее:

```
@birthday
```

Преобразовывается в:

```
(fieldLens (undefined :: FieldName "birthday"))
```

# Применение: Преобразование в структурированные данные

Следующего выражения:

```
Aeson.encode {~
  name = "Yuri Alekseyevich Gagarin",
  birthday = {~ year = 1934, month = 3, day = 9 }
}
```

Достаточно чтобы сгенерировать такой-вот JSON:

```
{
  "name": "Yuri Alekseyevich Gagarin",
  "birthday": { "year": 1934, "month": 3, "day": 9 }
}
```

# Применение: Именованные параметры функций

```
connect :: {~
             host :: ByteString,
             port :: Int,
             user :: ByteString,
             password :: ByteString
           } ->
           IO Connection
```

Вместо

```
connect :: ByteString ->
           Int ->
           ByteString ->
           ByteString ->
           IO Connection
```

# Применение: Шаблонизаторы: Laika

Шаблон:

```
<h1>{title}</h1>
<div>
  <p>{info/date}</p>
  <p>{info/venue}</p>
</div>
```

Можно использовать так:

```
render :: {~
            title :: TextBuilder,
            info :: {~
              date :: TextBuilder,
              venue :: TextBuilder
            }
          } ->
          TextBuilder
render = $(Laika.file "a/path/to/the/template.html")
```

# Как пользоваться

Ниже приведён пример того, как Вы можете настроить проект использовать препроцессор во всех модулях.

```
library
  build-depends:
    -- A required dependency on the preprocessor:
    record-preprocessor == 0.1.*,
    -- A required dependency on the library of record-types:
    record == 0.4.*,
    -- An optional dependency on the basic subset of the "lens" library:
    basic-lens == 0.0.*
  ghc-options:
    -- The following options enable the compiler-preprocessor
    -- for the whole project.
    --
    -- For this to work you need to manually execute "cabal install record-preprocessor" and
    -- make sure that your Cabal "bin" installation folder is on "PATH".
    -F -pgmF record-preprocessor
```

# Как пользоваться

Также возможно включать препроцессор индивидуально для модуля. Для этого нужно добавить следующую прагму в шапку модуля:

```
{-# OPTIONS_GHC -F -pgmF record-preprocessor #-}
```

Конечно же, подразумевается, что Ваш проект имеет те же зависимости, как и на предыдущем слайде.

# Как пользоваться

Оба способа использования подразумевают, что директория бинарников, устанавливаемых Cabal, (e.g., “.cabal/bin/”) упомянута в PATH.

# Ссылки

- Библиотека типов анонимных записей: http://hackage.haskell.org/package/record
- Препроцессор: http://hackage.haskell.org/package/record-preprocessor
- Мой блог с моими контактами: http://nikita-volkov.github.io/