Функциональное программирование на Haskell

Лекция 1: введение

Алексей Романов

18 февраля 2018 г.

Организация курса

- 8 лекций
- 4 лабораторных
- Итоговый проект

Парадигмы программирования

Что такое парадигма?

Парадигмы программирования

Что такое парадигма?

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Парадигмы программирования

Что такое парадигма?

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Основные парадигмы:

Парадигмы программирования

Что такое парадигма?

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Основные парадигмы:

- Структурное программирование
- Процедурное программирование
- **Функциональное программирование**
- Логическое программирование
- Объектно-ориентированное программирование

Парадигмы программирования

Что такое парадигма?

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Основные парадигмы:

- Структурное программирование
- Процедурное программирование
- **Функциональное программирование**
- Логическое программирование
- Объектно-ориентированное программирование

В парадигме важно не только то, что используется, но то, использование чего не допускается или минимизируется.

Парадигмы программирования

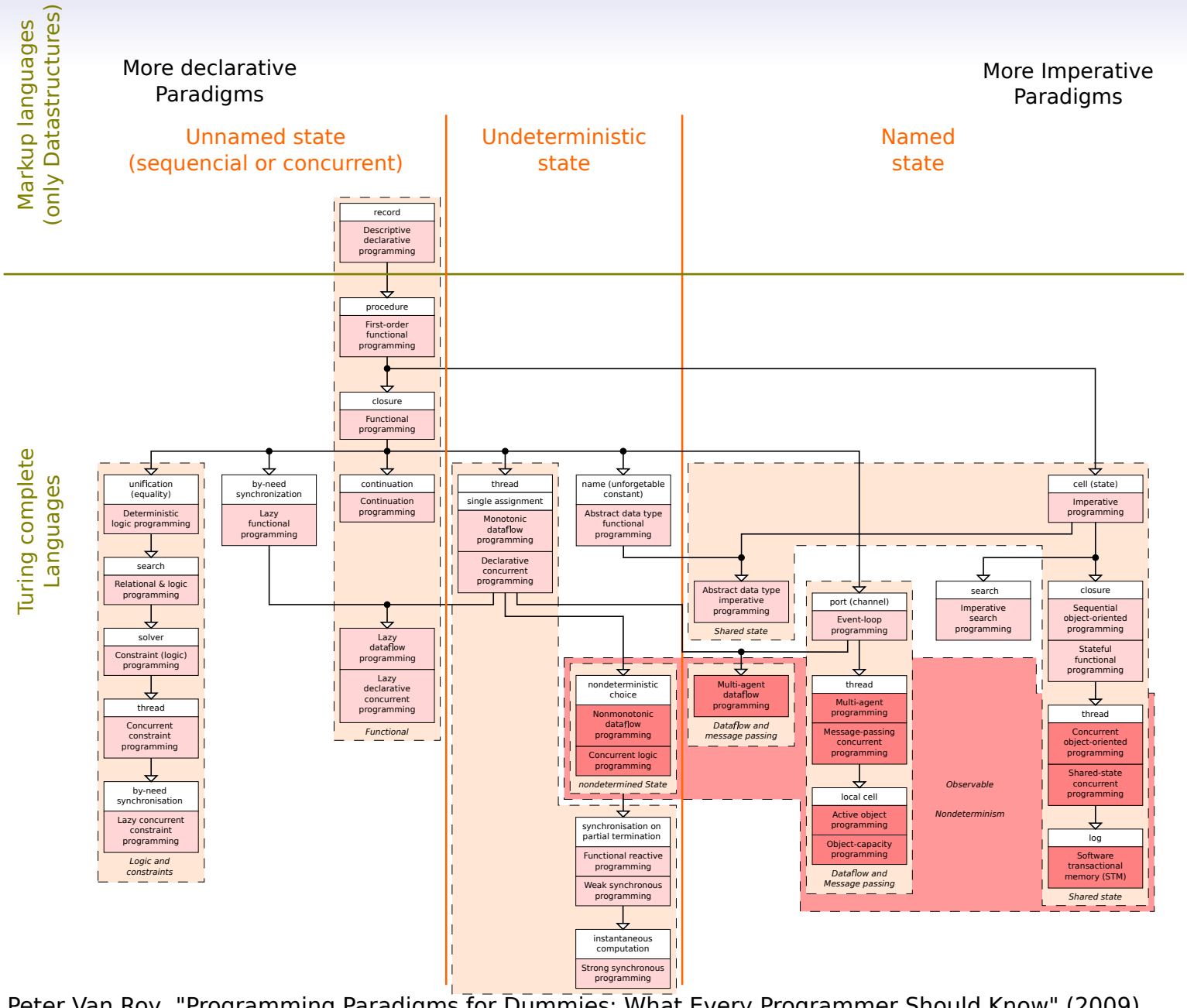
Что такое парадигма?

Совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Основные парадигмы:

- Структурное программирование
- Процедурное программирование
- **Функциональное программирование**
- Логическое программирование
- Объектно-ориентированное программирование

В парадигме важно не только то, что используется, но то, использование чего не допускается или минимизируется. Например, `goto` в структурном программировании, глобальные переменные в ООП.



Peter Van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know" (2009)

Функциональное программирование

- Значения лучше переменных.
  - Переменная даёт имя значению или функции, а не адресу в памяти.
  - Переменные неизменяемы.
  - Типы данных неизменяемы.
- Выражения лучше инструкций.
  - Аналоги `if`, `try-catch` и т.д. – выражения.
- Функции как в математике (следующий слайд)

Функциональное программирование

- Функции как в математике

Функциональное программирование

- Функции как в математике
  - Чистые функции: аргументу соответствует результат, а всё прочее от лукавого.
    - Нет побочных эффектов (ввода-вывода, обращения к внешней памяти, не связанной с аргументом, и т.д.)
    - При одинаковых аргументах результаты такой функции одинаковы
  - Функции являются значениями (функции первого класса)
  - Функции часто принимают и возвращают функции (функции высших порядков)

Функциональное программирование

- Функции как в математике
  - Чистые функции: аргументу соответствует результат, а всё прочее от лукавого.
    - Нет побочных эффектов (ввода-вывода, обращения к внешней памяти, не связанной с аргументом, и т.д.)
    - При одинаковых аргументах результаты такой функции одинаковы
  - Функции являются значениями (функции первого класса)
  - Функции часто принимают и возвращают функции (функции высших порядков)
- Опора на математические теории: лямбда-исчисление, теория типов, теория категорий

Языки ФП

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#

Языки ФП

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
  - Чисто функциональный

Языки ФП

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
  - Чисто функциональный
  - Строго статически типизированный (с очень мощной и выразительной системой типов)

Языки ФП

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
  - Чисто функциональный
  - Строго статически типизированный (с очень мощной и выразительной системой типов)
  - Ленивый

Язык Haskell: начало

- Установите Haskell Platform (https://www.haskell.org/platform/)
- Запустите WinGHCi (или просто GHCi)
- Это оболочка или REPL (Read-Eval-Print loop) для Haskell
  - Read: Вы вводите выражения (и команды GHCi)
  - Eval: GHCi вычисляет результат
  - Print: и выводит его на экран
- Пример:

```
GHCi, version 8.2.2:
   http://www.haskell.org/ghc/  :? for help
Prelude> 2 + 2
4
Prelude> :t True -- команда GHCi
True :: Bool
```

Язык Haskell: начало

- `2 + 2`, `True`: выражения
- `4`, `True`: значения
- `Bool`: тип

Язык Haskell: начало

- `2 + 2`, `True`: выражения
- `4`, `True`: значения
- `Bool`: тип
- Значение: «вычисленное до конца» выражение
- Тип (статический): множество значений и выражений, построенное по определённым законам таким образом, что компилятор может определить типы и проверить отсутствие ошибок в них без запуска программы.
- От типа зависит то, какие операции допустимы:

```
Prelude> True + False

<interactive>:12:1: error:
No instance for (Num Bool) arising from a use of '+'
In the expression: True + False
In an equation for 'it': it = True + False
```

Вызов функций

- Вызов (применение) функции пишется без скобок: `f x`, `foo x y`.
- Скобки используются, когда аргументы – сложные выражения, а не переменные: `f (g x)` (и внутри сложных выражений вообще).
- Бинарные операторы (как +) это просто функции.
  - Можно писать их префиксно, заключив в скобки: `(+) 2 2`
  - А любую функцию двух аргументов можно писать инфиксно, заключив в обратный апостроф: `4 div 2`
- Названия переменных и функций начинаются со строчной буквы
  - или состоят целиком из спец. символов.

Определение функций и переменных

- Определение функции выглядит так же как вызов:

```
название параметр1 параметр2 = значение
название = значение -- переменная
```

- Тело функции это не блок, а одно выражение (но сколь угодно сложное).
- В GHCi перед определением нужен `let`, в отличие от кода в файлах:

```
Prelude> let x = sin pi
Prelude> x
```

Определение функций и переменных

- Определение функции выглядит так же как вызов:

```
название параметр1 параметр2 = значение
название = значение -- переменная
```

- Тело функции это не блок, а одно выражение (но сколь угодно сложное).
- В GHCi перед определением нужен `let`, в отличие от кода в файлах:

```
Prelude> let x = sin pi
Prelude> x
1.2246063538223773e-16
Prelude> let square x = x * x
Prelude> square 2
4
```

Базовые типы

- Названия типов всегда начинаются с заглавной буквы (или состоят целиком из спец. символов).
- `Bool`: логические значения `True` и `False`.
- Целые числа:
  - `Integer`: неограниченные (кроме размера памяти);
  - `Int`: машинные[1], `Word`: машинные без знака;
  - `Data.{Int.Int/Word.Word}{8/16/32/64}`: фиксированного размера в битах, со знаком и без.
- `Float` и `Double`: 32- и 64-битные числа с плавающей точкой, по стандарту IEEE-754.
- `Character`: символы Unicode.
- `()`: Единичный тип (unit) с единственным значением `()`.

[1] по стандарту минимум 30 бит, но в GHC именно 32 или 64 бита

Тип функций и сигнатуры

- Типы функций записываются через `->`. Например, `Int -> Char` это тип функции из `Int` в `Char`.
- Для нескольких аргументов это выглядит как

Тип функций и сигнатуры

- Типы функций записываются через `->`. Например, `Int -> Char` это тип функции из `Int` в `Char`.
- Для нескольких аргументов это выглядит как `Bool -> Bool -> Bool`.
- `::` читается как «имеет тип»; запись `выражение :: тип` называется «сигнатурой типа».
- При объявлении экспортируемой функции или переменной сигнатура обычно указывается явно:

```
foo :: Int -> Char
foo x = ...
```

- Компилятор обычно выведет типы и без этого, но явное указание защищает от *непреднамеренного* изменения.

Арифметика

- Это упрощённая версия, так как настоящее объяснение требует понятия класса типов, которое будет введено позже.
- Например, `:type 1` или `:type (+)` дадут тип, который понимать пока не требуется.
- То же относится к ошибкам вроде `No instance for (Num Bool) arising from a use of` на более раннем слайде.
- Пока достаточно понимать, что есть понятие «числового типа», которые делятся на целочисленные (`Int`, `Integer`) и дробные (`Float`, `Double`).

Числовые литералы

- Числовые литералы выглядят, как в других языках: `0`, `1.5`, `1.2E-1`, `0xDEADBEEF`.
- Целочисленные литералы могут иметь любой числовой тип, а дробные любой дробный.
- Но это относится *только* к литералам. Неявного приведения (например, `Int` в `Double`) в Haskell *нет*.

Приведение числовых типов

- Используйте `fromIntegral` для приведения из любого целочисленного типа в любой числовой. Тип-цель можно указать явно:

```
Prelude> let {x :: Integer; x = 2}
Prelude> x :: Double
<interactive>:22:1: error:
Couldn't match expected type 'Double' with
   actual type 'Integer' ...
Prelude> fromIntegral x :: Double
2.0
```

- Или не указывать, если компилятор может его вывести из контекста:

```
Prelude> :t fromIntegral 2 / (4 :: Double)
fromIntegral 2 / (4 :: Double) :: Double
```

Приведение числовых типов

- `toInteger` переводит любой целочисленный тип в `Integer`, `fromInteger` из `Integer` в любой целочисленный.
  - Если аргумент `fromInteger` слишком велик для типа цели, берётся его значение по модулю:

    ```
    Prelude> fromInteger (2^64) :: Int
    0
    ```

- `toRational` и `fromRational` – аналогично для `Rational` и дробных типов.
- `ceiling`, `floor`, `truncate` и `round` из дробных типов в целочисленные (по названиям должно быть понятно, как именно).

Арифметические операции

- Операции +, -, * – как в других языках (с учётом отсутствия приведения). Унарный минус – единственный унарный оператор.
- / – деление *дробных* чисел
  - Использование / для целых чисел даст ошибку `No instance... arising from a use of '/'`. Нужно сначала использовать `fromIntegral`.
- `div` – частное целых чисел, а `mod` – остаток
  - `quot` и `rem` – тоже, отличаются от них поведением на отрицательных числах.
- ^ – возведение любого числа в неотрицательную целую степень.
- ^^ – дробного в любую целую.
- ** – дробного в степень того же типа.

Операции сравнения и логические операции

- Большинство операций сравнения выглядят как обычно: `==`, `>`, `<`, `>=`, `<=`.
- Но $\neq$ обозначается как `/=`.
- Функция `compare` возвращает `Ordering`: тип с тремя значениями `LT`, `EQ` и `GT`.
- Есть функции `min` и `max`.

- «и» это `&&`, а «или» – `||`, как обычно.
- «не» – `not`.

Существенные отступы

- Вспомним виденное раньше

  ```
  let {x :: Integer; x = 2}
  ```

  По правилам Haskell можно написать это же без фигурных скобок:

  ```
  let x :: Integer
      x = 2
  ```

- Если два выражения (или других фрагмента кода) относятся к одному уровню одного блока, то они должны иметь одинаковый отступ (начинаться в одном и том же столбце).
- Фрагмент кода, являющийся частью другого, должен иметь отступ больше.

Правило преобразования отступов

- Если после ключевых слов `where`, `let`, `do`, `of` нет открывающей фигурной скобки {[2], то такая скобка вставляется и начинается новый блок.
- Его базовый отступ – столбец, содержащий следующий непробельный символ.
- После этого каждая строка, которая начинается:
  - в том же столбце: относится к этому блоку и перед ней ставится `;`
  - правее: продолжает предыдущую, перед ней ничего не ставится.
  - левее: блок закончился, перед ней ставится `}` (одновременно может закончиться несколько блоков).

[2] Если она есть, то правило перестаёт действовать до соответствующей ей закрывающей.

Условные выражения

- В любом языке нужна возможность выдать результат в зависимости от какого-то условия.
- В Haskell это выражения `if` и `case`.
- Синтаксис `if`:

  ```
  if условие then выражение1 else выражение2
  ```

  Варианта без `else` нет.
- Многострочно пишется так:

  ```
  if условие
      then выражение1
      else выражение2
  ```

- Тип условия обязательно `Bool`.
- Типы выражения1 и выражения2 должны совпадать.
- Это же и тип всего `if ... then ... else ...`
- Ближе к `?:`, чем к `if` в C-подобных языках.

Сопоставление с образцом

- Синтаксис case:

```
case выражение of
    образец1 -> выражение1
    образец2 -> выражение2
```

- Образец это «форма» для значений типа, которая может содержать несвязанные переменные. Для конкретного значения он либо подходит (и связывает эти переменные), либо не подходит.
- Процедура вычисления:
  - Вычислить значение выражения.
  - Сопоставить его с каждым образцом по очереди.
  - Если первым подошёл образецN, вычислить выражениеN и вернуть его значение.
  - Если ни один не подошёл, выкинуть ошибку.

Образцы для известных нам типов

- `True` и `False` – образцы для `Bool`. Они подходят, только если значение совпадает с ними.
- Аналогично `LT`, `EQ` и `GT` для `Ordering`, а числовые литералы для числовых типов.
- Переменная – образец для любого типа, который подходит для любого значения.
  - В Haskell все переменные в образцах «свежие» и перекрывают видимые снаружи переменные с тем же названием.
- `_` тоже подходит для любого значения любого типа и означает, что это значение не важно.
- Образцы похожи на выражения, но ими не являются: это новая синтаксическая категория!

Связь case и if

- Пример:

```
if условие
    then выражение1
    else выражение2
```

это ровно то же самое, что

```
case условие of
    True -> выражение1
    False -> выражение2
```

или

```
case условие of
    True -> выражение1
    _ -> выражение2
```

Охраняющие условия

- У каждого образца могут быть дополнительные условия (зависящие от его переменных):

```
образец
    | условие1 -> выражение1
    | условие2 -> выражение2
```

  При удачном сопоставлении они проверяются по порядку. Если все оказались ложны, сопоставление переходит к следующему образцу.
- Последнее условие часто `otherwise` (синоним `True`), тогда хотя бы одно условие точно истинно.
- Чтобы сравнить переменную в образце с внешней, нужно использовать == в условии:

```
case foo x of
    y | y == x -> ... -- не то же, что x -> ...
    _ -> ...
```

Многоветвенные `if`

- Цепочка `if ... then ... else if ...`, соблюдающая правила отступа, всё время уходила бы направо:

```
if условие1
    then результат1
    else if условие2
        then результат2
        else результат3
```

  Вместо этого можно включить расширение `MultiWayIf` [3] и писать

```
if | условие1 -> результат1
   | условие2 -> результат2
   | otherwise -> результат3
```

- Это можно сделать и с case без расширений: как?

[3] Например, прагмой `{-# LANGUAGE -XMultiWayIf #-}` в начале файла

Многоветвенные `if`

- Цепочка `if ... then ... else if ...`, соблюдающая правила отступа, всё время уходила бы направо:

```
if условие1
    then результат1
    else if условие2
        then результат2
        else результат3
```

  Вместо этого можно включить расширение `MultiWayIf` [3] и писать

```
if | условие1 -> результат1
   | условие2 -> результат2
   | otherwise -> результат3
```

- Это можно сделать и с `case` без расширений: как?

```
case () of _ | условие1 -> ...
```

[3] Например, прагмой `{-# LANGUAGE -XMultiWayIf #-}` в начале файла

Определение функций по случаям

- Часто тело функции – case по аргументу, например.

```
not x = case x of
    True -> False
    False -> True
```

- Такие функции можно записать несколькими равенствами, по одному на ветвь case:

```
not True = False
not False = True
```

- Это работает и с охранными условиями (без повторения названия функции):

```
not x | x = False
      | otherwise = True
```

- и в случае нескольких параметров:

```
nand True True = False
nand _    _ = True
```

Локальные определения: `let`

- Локальные определения дают две выгоды:
  - Более читаемый код.
  - Избавление от повторяющихся вычислений.
- В Haskell два способа их задать: `let` и `where`.
- Синтаксис `let`:

```
let переменная1 = выражение1
    функция2 x = выражение2
    ...
in выражение3
```

- Всё `let ... in ...` – выражение.
- Первое проявление ленивости Haskell: будут вычислены только те переменные, которые понадобятся для вычисления выражения3.

Локальные определения: where

- Синтаксис where:

```
функция образец1 | условие1 = выражение3
                 | условие2 = выражение4
    where переменная1 = выражение1
          функция2 x = выражение2
```

- Видимы в условиях и в правых частях (но не для других образцов).
- Можно применить только к определениям (в том числе локальным).

Модули

- Программа на Haskell состоит из модулей.
- Модуль Модуль (названия с заглавной буквы) определяется в файле Модуль.hs такого вида:

```
module Модуль(функция1) where

import ...

функция1 :: ...
функция1 x = ...
```

- функция1 экспортирована, она доступна другим модулям и GHCi. Все остальные – нет.
- Можно экспортировать всё, опустив список экспортов (включая скобки).
- Модуль может иметь название вида A.B.C и лежать в A/B/C.hs.

Импорт из другого модуля

- У директивы `import` есть много вариантов.
  - По адресу `https://wiki.haskell.org/Import` есть полный перечень.
- Нам пока достаточно простейшего из них

  `import Модуль`

  который позволяет использовать в нашем модуле всё, что экспортирует `Модуль` (как `функция` и как `Модуль.функция`).

Загрузка модуля

- В GHCi можно скомпилировать и загрузить модуль (вместе с зависимостями) командой `:load Модуль`. Подробности в документации:
  - The effect of :load on what is in scope
- Под Windows можно просто дважды щёлкнуть на файл или нажать Enter в Проводнике.
- Потом при изменениях файла повторить `:load` или сделать `:reload`.
- Многие редакторы позволяют это автоматизировать.