Функциональное программирование на Haskell

Лекция 1: введение

Алексей Романов

17 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

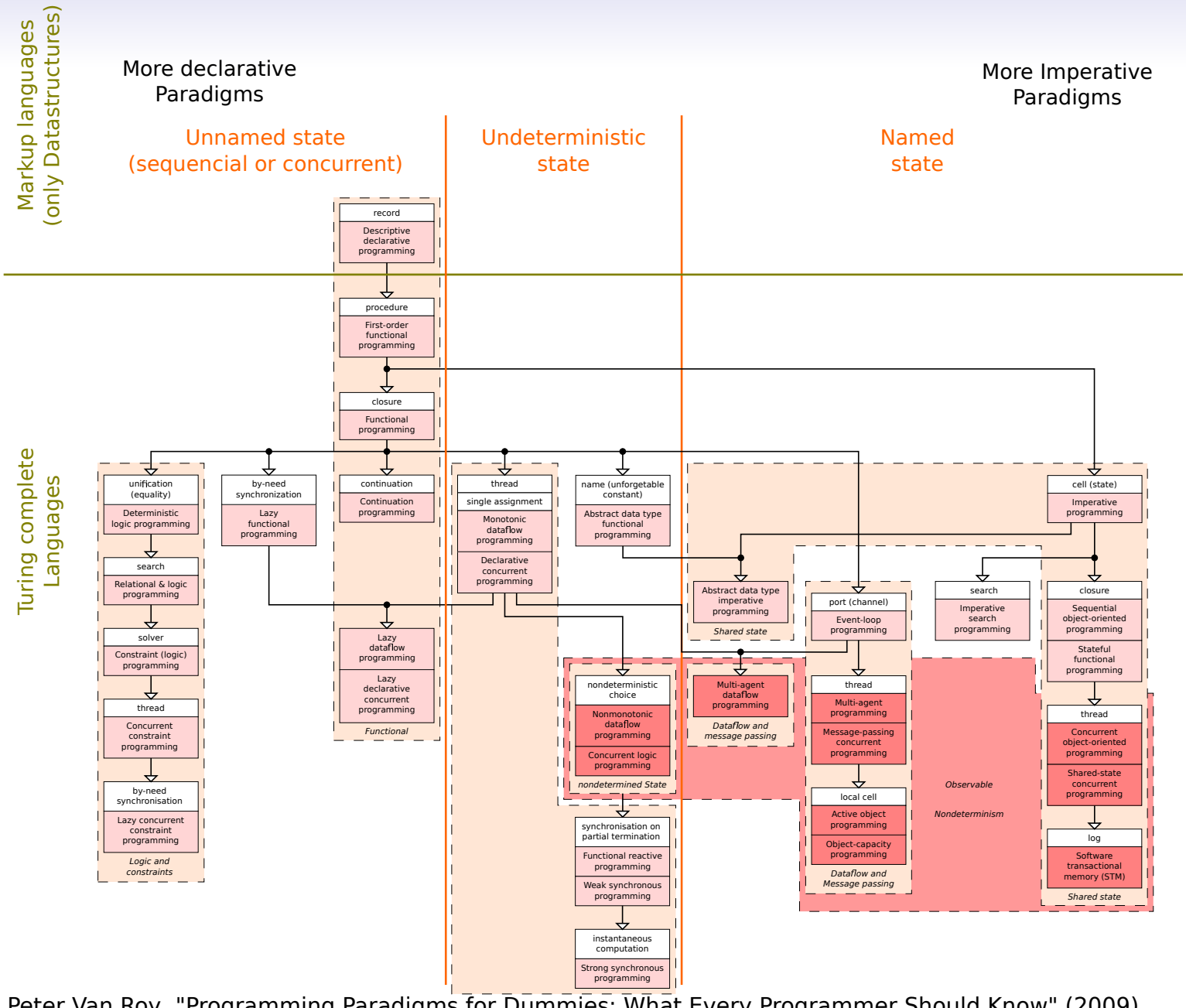
Что такое парадигма?

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.
- `Character`: символы Unicode.
- `()`: Unit (единичный тип), единственное значение `()`.

[1] по стандарту минимум 30 бит, но в GHC это 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, только аргументы обязательно имеют одинаковый тип.
- Унарный минус – единственный унарный оператор.
- `/` – деление *дробных* чисел, `div` – целочисленное, `mod` – остаток (есть ещё `quot` и `rem`, они отличаются поведением на отрицательных числах).
  - Использование `/` для целых чисел даст ошибку `No instance... arising from a use of '/'`. Нужно сначала использовать `fromIntegral`.
- `^` – возведение любого числа в целую неотрицательную степень, `^^` – дробного в любую целую, `**` – дробного в степень того же типа.

Операции сравнения и логические операции

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

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

Существенные отступы (двухмерный синтаксис)

- TODO

Условные выражения

- В любом языке должна быть возможность выдать результат в зависимости от какого-то условия.
- Первый способ сделать это в Haskell: выражения `if` и `case`.
- Синтаксис `if`:

  `if условие then выражение1 else выражение2`.

  Поскольку это выражение, то `else` обязательно! `условие` имеет тип `Bool`, `выражение1` и `выражение2` – любой одинаковый тип, который будет и типом всего `if ... then ... else ...`. Это ближе к `?:`, чем к `if` в C-подобных языках.
- Многострочно пишется так:

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

Сопоставление с образцом

- Синтаксис `case`:

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

- Его смысл: вычислить значение выражение и сопоставить с каждым образцом по очереди. Если первым подошёл образецN, вернуть значение выражениеN.
- Если ни один образец не подошёл, то вычисление `case` выкидывает ошибку.
- В первом приближении, образец это «форма для значения», которая может содержать неопределённые переменные (они получат значение при удачном сопоставлении).
- Образцы похожи на выражения, но ими не являются: это новая синтаксическая категория!

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. Она подходит для любого значения, и получает это значение при сопоставлении.
- _ тоже образец для любого типа, который подходит для всех значений, но ничего не связывает. Можно читать как «значение не важно».

Связь case и if

- Пример:

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

это ровно то же самое, что

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

На самом деле, при компиляции Haskell это преобразование действительно делается, чтобы не дублировать правила оптимизации.

Охраняющие условия

- У каждого образца могут быть дополнительные условия, которые могут содержать переменные из образца:

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

  В этом случае при удачном сопоставлении проверяется по очереди каждое условие. Если ни одно из них не выполнено, сопоставление переходит к следующему образцу.

Определение функций по случаям

- TODO

Локальные переменные

- TODO