Язык Haskell
и Алгебраические Типы Данных

ФОНД ПОДДЕРЖКИ
ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ
ФП(ФП)

ДУШКИН РОМАН ВИКТОРОВИЧ

Содержание

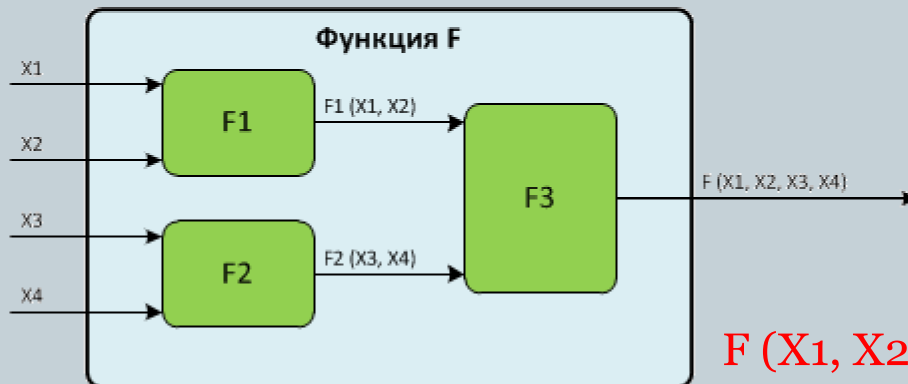
- **Быстрое введение в язык Haskell**
  - Функциональный стиль программирования
  - Свойства функциональных языков
  - Синтаксис языка Haskell
  - Пять программных сущностей
- **Что такое АТД?**
  - Кратко о системе типизации языка Haskell
  - Типы-суммы и типы-произведения
  - Алгебра над типами
- **Куда двигаться дальше?**

Быстрое введение в язык Haskell

- Функциональный стиль программирования
  - Декларативность
  - Функция — это значение
- Свойства функциональных языков
  - Чистота: детерминированность и отсутствие побочных эффектов
  - Каррированность и частичные применения
  - Статическая типизация
  - Полиморфизм
  - Ленивые вычисления
  - Функции высших порядков
  - Краткость и выразительность
  - «Математичность»
- Синтаксис языка Haskell
  - Сдвиг парадигмы
  - Типовая структура файла с исходными кодами
- Пять программных сущностей
  - Функции
  - Типы данных
  - Классы типов
  - Экземпляры классов
  - Модули

Функциональный стиль программирования

Любая программа представляет собой ничто иное, как функцию, которая принимает на вход некоторое количество аргументов и возвращает значение. Для вычисления возвращаемого значения функция использует аргументы, которые передаёт в другие функции, собирает и комбинирует возвращаемые ими результаты.



F (X1, X2, X3, X4) = F3 (F1 (X1, X2), F2 (X3, X4))

Декларативность

Декларативное программирование обладает рядом существенных свойств, которые иногда воспринимаются с некоторым удивлением теми, для кого первым языком программирования был язык наподобие Java или C#:

- Отсутствие присваивания (в принципе).
- Отсутствие глобальных состояний.
- Отсутствие переменных.
- Программа определяется через набор единообразных сущностей (функции, предикаты, продукции, правила, ограничения — в каждом декларативном языке используется своя сущность).

Декларативный стиль вполне может использоваться и в обычных языках программирования, типа C#. Все методы C# будут выглядеть примерно так:

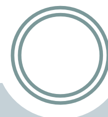
```
float expectationValue (int n, int sum)
{
  return (float)sum / (float)n;
}
```

Функции — это значения

В функциональном программировании функция является как той сущностью, которая занимается вычислениями, так и той, которая может быть объектом вычислений. Функцию можно передавать в другие функции в качестве аргумента, а также возвращать в качестве результата.

Конечно, в таких языках, как C и C++ функции тоже можно передавать в качестве аргументов при помощи указателей, равно как и возвращать функции. Однако именно в функциональных языках это сделано удивительно удобно и естественно.

Чистота функций



В функциональном программировании функции являются *чистыми*. Это означает, что любая функция обладает двумя важнейшими свойствами:

- *Детерминированность* — значение, возвращаемое функцией, зависит только от значений входных параметров. Для заданного набора значений входных параметров выходным значением функции всегда будет одним и тем же.
- *Отсутствие побочных эффектов* — функция работает только с выделенной для неё памятью, она не может менять значения каких-то внешних сущностей (объектов, переменных, любую внешнюю память).

Это позволяет трактовать функции, написанные для функциональных программ, как чистые математические объекты. Даже функции, работающие с вводом-выводом.

Каррированность и частичные применения

Каждая функция имеет определённый тип. Это не тип возвращаемого значения, а именно тип функции, как объекта:

- `one = 1                :: Int`
- `id x = x               :: α → α`
- `const x y = x          :: α → (β → α)`
- `s x y z = x z (y z)    :: (α → β → γ) → ((α → β) → (α → γ))`

Это позволяет осуществлять частичные применения функции к своим аргументам. В результате частичного применения получается функция, результат которой равен в точности результату исходной функции, как если бы все вхождения известных параметров были заменены на их значения:

`add x y = x + y` ⇨ `add′ = add 1 :: Int -> Int` ⇨ `add′ 5 = 1 + 5`

Статическая типизация

Тип любого объекта в функциональной программе (на таких языках программирования, как Haskell) известен или может быть выведен автоматически. Для этого используется такой формализм, как *Система типизации Хиндли-Милнера*.

- `s = y:show (x + 5)`
- `5     :: Int`
- `(+)   :: Int → Int → Int`
- **`x     :: Int`**
- `show  :: α → String`
- `(:)   :: Char → String → String`
- **`y     :: Char`**
- **`s     :: String`**

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} Var$$

$$\frac{\Gamma, x : \tau_p \vdash t : \tau_r}{\Gamma \vdash (\lambda x : \tau_p . t) : \tau_p \to \tau_r} Lam$$

$$\frac{\Gamma \vdash t_f : \tau_p \to \tau_r \qquad \Gamma \vdash t_p : \tau_p}{\Gamma \vdash t_f\, t_p : \tau_r} App$$

Полиморфизм

В функциональном программировании широко используется *параметрический полиморфизм*. Параметрический полиморфизм основан на передаче типов аргументов наряду с их значениями в виде параметров в функции и конструкторы (отсюда и атрибут «параметрический»). Реализация данного типа полиморфизма зачастую основана на типовых переменных, то есть таких переменных в сигнатурах определений функций и конструкторов типов, вместо которых можно подставлять произвольные типы (в том числе и с ограничениями).

```
reverse :: [α] → [α]
```

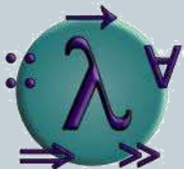
Эта функция обращает последовательность элементов заданного списка независимо от типа элементов списка. В таких языках, как C++, Java этот механизм больше всего напоминает «шаблоны».

- **Душкин Р. В.** *Полиморфизм в языке Haskell* // Научно-практический журнал «Практика функционального программирования», Выпуск 3, Декабрь 2009. — стр. 67-81. — ISSN 2075-8456. — http://fprog.ru/2009/issue3/roman-dushkin-haskell-polymorphism/

Ленивые вычисления

Обычно функциональные языки программирования предлагают программисту ленивую стратегию вычислений. В языке Haskell ленивые вычисления используются по умолчанию. Это позволяет, в частности, обрабатывать бесконечные структуры данных. Например:

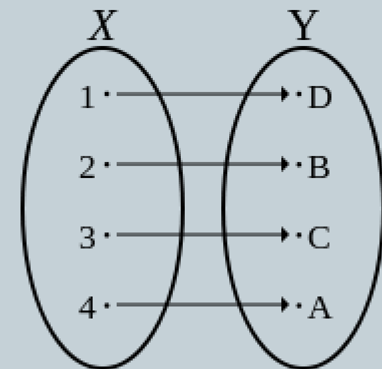
```
> let ones = 1:ones
> take 10 ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```



Кстати, в языке Haskell имеется ещё и иной вид полиморфизма — ограниченный полиморфизм, используемый для перегрузки имён функций. О нём мы поговорим позже

Функции высших порядков

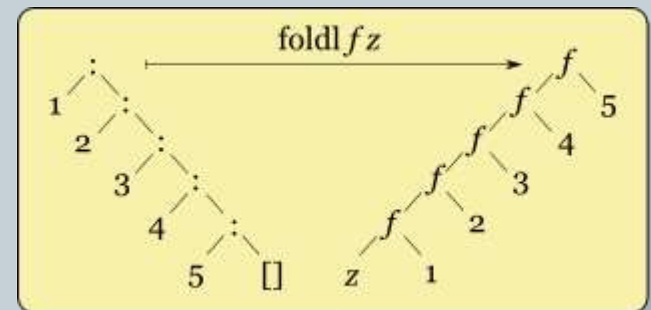
Поскольку функции являются полноценными вычислительными объектами, их можно передавать в другие функции в качестве аргументов. Функции, принимающие другие функции в качестве аргументов, называются *функциями высших порядков*. Это — мощное средство абстракции.



Например, вот пара важнейших ФВП:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z0 xs0 = lgo z0 xs0
  where
    lgo z [] = z
    lgo z (x:xs) = lgo (f z x) xs
```



Краткость и выразительность

Предыдущие примеры должны были уже показать, что программы на функциональных языках и, в частности, на языке Haskell, являются краткими и весьма выразительными (конечно, для тех, кто уже проникся сутью и духом функциональной парадигмы).

Вот, например, как можно написать функцию для вычисления произведения элементов списка:

```
product [] = 1
product (x:xs) = x * product xs
```

Это определение с *явной рекурсией*, которая не рекомендуется к использованию, поскольку она загромождает код и затрудняет понимание. Попробуем выразить эту функцию через `foldl`:

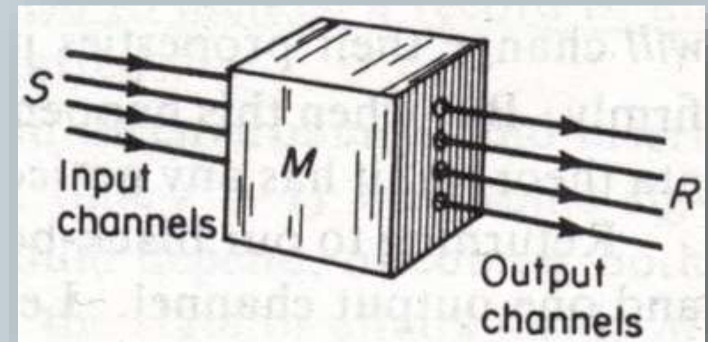
```
product xs = foldl (*) 1 xs
```

Поскольку в этом определении аргумент `xs` находится в самом конце выражения, то можно использовать *бесточечную нотацию*:

```
product = foldl (*) 1
```

«Математичность»

Ну и вспоминая о том, что функция в функциональном программировании имеет свойства детерминированности и отсутствия побочных эффектов, её можно воспринимать как чистый математический объект — функцию в математическом понимании. Это, в свою очередь, обозначает, что мы можем использовать всю мощь математического аппарата для исследования функций, доказательства их свойств, оптимизации, преобразования и для многих иных вещей.



Haskell: сдвиг парадигмы

Разработчик, который ранее использовал такие языки программирования, как C++, Java, C# и др. с подобным синтаксисом, необходимо осуществить определённого рода *сдвиг парадигмы* в своей голове, поскольку дело не только и не столько в синтаксисе языка (который хоть и кажется некоторым странным, но в целом очень простой).

Дело в сути функционального программирования, его внутренней логике. Разработчик должен научиться мыслить функционально, чтобы успешно использовать язык Haskell в своей работе.

Типовая структура файла с исходными кодами

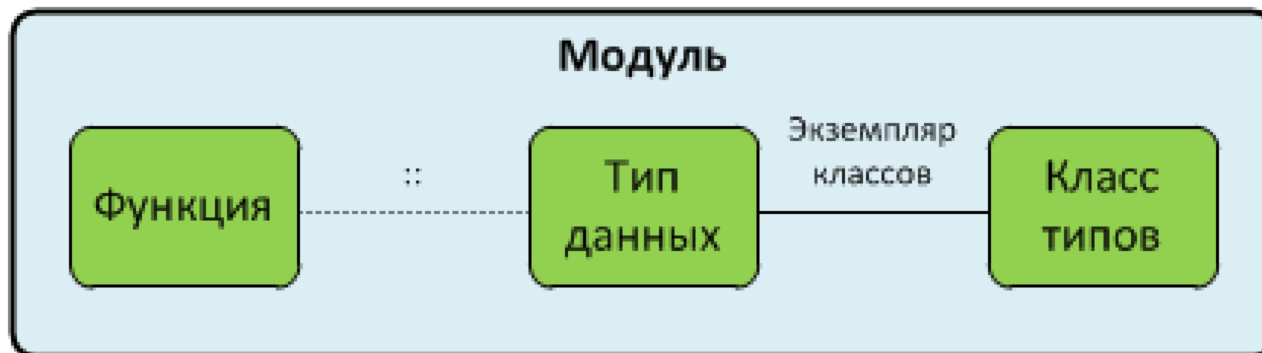
На диаграмме справа представлена типовая структура модуля на языке Haskell.

- В заголовке модуля приводится его наименование и описание интерфейса, а также декларации импорта.
- Остальные разделы являются необязательными (все и каждый — модуль может быть вообще без определений, используется для реэкспорта).
- В каждом разделе приводятся определения только указанных программных сущностей.
- Конечно, определения можно приводить вразнобой, однако такая структура делает модули унифицированными.

Заголовок модуля
Описания классов
Описания типов данных
Описания экземпляров классов
Описания приоритетов бинарных операций
Описания функций

Пять программных сущностей языка Haskell

ФУНКЦИИ
ТИПЫ ДАННЫХ
КЛАССЫ ТИПОВ
ЭКЗЕМПЛЯРЫ КЛАССОВ
МОДУЛИ



Функции

Функция является краеугольным элементом программы на языке Haskell. Вся программа состоит из набора функций, при этом выделяется одна функция в качестве точки входа (функция `main` из модуля `Main`).

Функции определяется непосредственно на самом верхнем уровне модуля. Определение функции состоит из следующих элементов:

- Необязательной декларации типа функции.
- Уникального имени функции.
- Перечня аргументов функции.
- Тела функции.

Принимая во внимание наличие механизма *сопоставления с образцами*, в одной декларации функции может быть несколько перечней аргументов и соответствующих им тел.

Функции. Примеры

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _    []     = []
filter pred (x:xs)
  | pred x         = x : filter pred xs
  | otherwise      = filter pred xs
```

```haskell
take :: Int -> [a] -> [a]
take n _  | n <= 0 =  []
take _ []          =  []
take n (x:xs)      =  x : take (n-1) xs
```

```haskell
reverse :: [a] -> [a]
reverse l = rev l []
  where
    rev []     a = a
    rev (x:xs) a = rev xs (x:a)
```

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

```haskell
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

```haskell
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
(.*.) `on` f = \x y -> f x .*. f y
```

```haskell
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

```haskell
permutations :: [a] -> [[a]]
permutations xs0 = xs0 : perms xs0 []
  where
    perms []     _  = []
    perms (t:ts) is = foldr interleave
                        (perms ts (t:is))
                        (permutations is)
      where
        interleave xs r
          = let (_, zs) = interleave' id xs r
            in  zs

        interleave' _ []     r = (ts, r)
        interleave' f (y:ys) r
          = let (us,zs)
                  = interleave' (f . (y:)) ys r
            in  (y:us, f (t:y:us) : zs)
```

```haskell
lines :: String -> [String]
lines s = let (l, s') = break (== '\n') s
          in  l : case s' of
                    []      -> []
                    (_:s'') -> lines s''
```

Типы данных

Язык Haskell имеет три вида деклараций типов:

- Синонимы типов (`type`).
- **Алгебраические типы данных** (`data`).
- Изоморфные типы (`newtype`).

Синонимы типов предназначены исключительно для повышения читаемости программ. Изоморфные типы решают пару или даже тройку непростых задач, которые обычно рассматриваются в более глубоких справочниках и учебниках.

Алгебраические типы данных — суть и соль системы типизации языка Haskell. Им будет посвящён отдельный раздел.

АТД. Примеры

```haskell
data WeekDay = Monday
             | Tuesday
             | Wednesday
             | Thursday
             | Friday
             | Saturday
             | Sunday
```

```haskell
data Colour = Red
            | Orange
            | Yellow
            | Green
            | Blue
            | Indigo
            | Violet
            | RGB Int Int Int
```

```haskell
data Bool = True
          | False
```

```haskell
data Maybe a = Nothing
             | Just a
```

```haskell
data Either a b = Right a
                | Left b
```

```haskell
data List a = Nil
            | Cons a (List a)
```

```haskell
data Timestamp = Timestamp
                 {
                   year    :: Int,
                   month   :: Month,
                   day     :: Int,
                   hour    :: Int,
                   minute  :: Int,
                   second  :: Int,
                   weekday :: Weekday
                 }
```

```haskell
data Ratio a = a :% a
```

```haskell
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

```haskell
data Tree a = Nil
            | Node a (Tree a) (Tree a)
```

```haskell
data Rose a = Nil
            | Rose a [Rose a]
```

```haskell
data Rope a b = Nil
              | Twisted b (Rope b a)
```

```haskell
data Function a b = F (a -> b)
                  | G (b -> a)
```

Классы типов

Классы в языке Haskell скорее соответствуют интерфейсам в таких языках, как C# и Java. Они определяют наборы методов (функций), которые можно использовать с теми типами данных, для которых они реализованы (определён экземпляр).

Действительно, класс типов представляет собой интерфейс, то есть набор сигнатур функций без определений.
В определениях АТД и сигнатурах функций можно указывать ограничения на используемые типовые переменные. Такие ограничения означают, что соответствующая переменная типов должна инстанциироваться только такими типами, для которых реализованы функции класса.

Однако имеются и серьёзные отличия от интерфейсов из ООП.
В частности, механизм классов более широк и гибок, чем интерфейсы.

Классы. Примеры

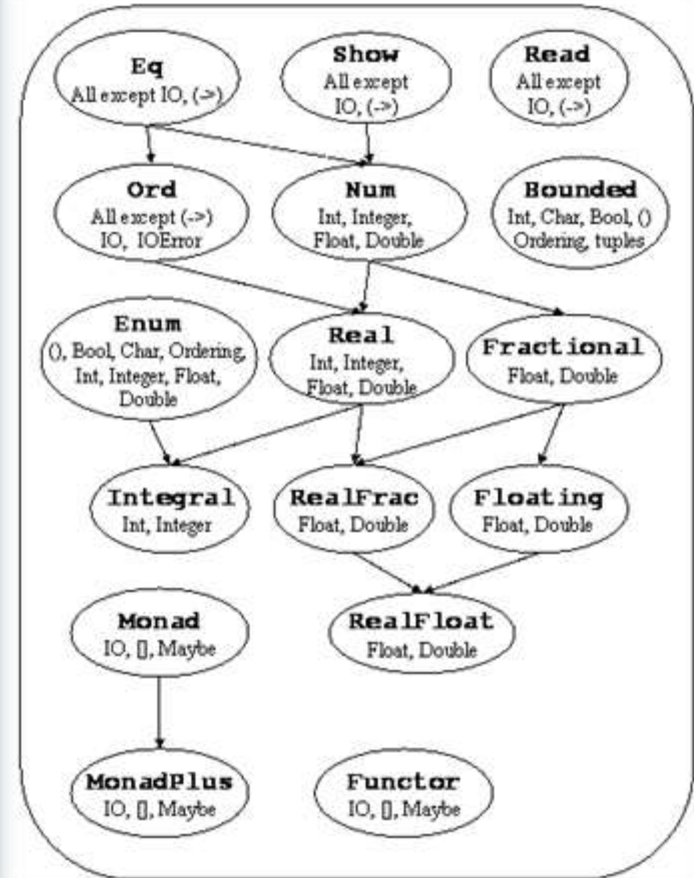
```
class  (Eq α, Show α) => Num α  where
  (+)         :: α -> α -> α
  (-)         :: α -> α -> α
  (*)         :: α -> α -> α
  negate      :: α -> α
  abs         :: α -> α
  signum      :: α -> α
  fromInteger :: Integer -> α

  x - y    = x + negate y
  negate x = 0 - x
```

```
class Logic α where
  not  :: α -> α
  (&&) :: α -> α -> α
  (||) :: α -> α -> α
```

```
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a

  (<$) = fmap . const
```



Экземпляры классов

Экземпляр класса — это связь между классом и типом. Говорят: «Экземпляр такого-то класса для такого-то типа» — это обозначает, что для такого-то типа определены функции (по крайней мере, минимальный набор), сигнатуры которых заданы в определении класса.

- Экземпляр — это отдельная сущность языка, слабо связанная с другими.
- Тип может быть определён в одном модуле, класс в другом, а экземпляр в третьем.
- Наличие экземпляра позволяет работать со значениями типа всем функциям, в сигнатурах которых есть ограничение в виде класса.
- Классы и их экземпляры — это как бы расширение объектов в ООП. По сути, методы больше не зависят об объектов, и любой программист может создавать новые методы.

Экземпляры. Примеры

```
class Logic α where
  not  :: α -> α
  (&&) :: α -> α -> α
  (||) :: α -> α -> α
```

```
instance Logic Bool where
  not  = Prelude.not
  (&&) = Prelude.(&&)
  (}}) = Prelude.(||)
```

```
instance Logic Float where
  not  = (1 -)
  (&&) = min
  (||) = max
```

```
data Ternary = TTrue
             | TUndefined
             | TFalse

instance Logic Ternary where
  not TTrue      = TFalse
  not TUndefined = TUndefined
  not TFalse     = TTrue

  TTrue      && TTrue      = TTrue
  TTrue      && TUndefined = TUndefined
  TUndefined && TTrue      = TUndefined
  TUndefined && TUndefined = TUndefined
  _          && _          = TFalse

  TFalse     || TFalse     = TFalse
  TFalse     || TUndefined = TUndefined
  TUndefined || TFalse     = TUndefined
  TUndefined || TUndefined = TUndefined
  _          || _          = TTrue
```

Модули

Модули в языке Haskell используются не только для группировки программных сущностей, объединённых единой задачей, но и для инкапсуляции данных. Система экспорта/импорта функций и конструкторов типов/данных очень гибкая.

- По умолчанию из модуля экспортируется всё его содержимое.
- Можно явно указать перечень экспортируемых программных сущностей. Для АТД и изоморфных типов можно указать, какие конструкторы данных импортируются, и импортируются ли в принципе (инкапсуляция).
- При импорте по умолчанию импортируются все программные сущности, указанные в интерфейсе модуля.
- Можно явно перечислить только те программные сущности, которые необходимо импортировать из модуля.
- Экземпляры классов экспортируются и импортируются всегда.
- Импорт можно производить квалифицированно. Тогда при вызове функций или конструкторов перед их именами необходимо указывать имя или синоним модуля.

Что такое Алгебраические Типы Данных?

- Кратко о системе типизации языка Haskell
  - Синонимы, изоморфные типы и АТД — сравнение
  - Когда что использовать?
- Типы-суммы и типы-произведения
  - Размеченное объединение
  - Декартово произведение
  - АТД: собираем всё вместе
  - Аксиома тектоничности
- Алгебра над типами

Типы языка Haskell: сравнение деклараций

Синонимы типов предназначены только для задания нового имени произвольному типу:

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

Изоморфные типы всегда имеют только один конструктор данных, который «обёртывает» один тип. Предназначены для создания нового типа, тождественного имеющемуся:

```
newtype List a = List [a]
```

Алгебраические типы данных позволяют создать тип с произвольной структурой:

- Тип-сумма (перечисление).
- Тип-произведение.
- Структура с именованными полями.
- Произвольное выражение в алгебре типов.

Пример:

```
data Bool = True
          | False
```

Типы языка Haskell: когда что использовать

- Используйте синонимы типов, если у вас есть часто используемая комбинация типов, собранная из имеющихся конструкторов.
- Используйте изоморфный тип, если вам необходимо:
  - Сделать тип-произведение, в котором только одно поле данных.
  - Определить для некоторого типа экземпляр, отличный от уже определённого для этого типа (для каждой пары «тип — класс» можно определять только один экземпляр). В этом случае достаточно обернуть требуемый тип в новый конструктор.
  - Есть тонкое различие в семантике ИТ и АТД, заключающееся в различной обработке ленивости. Все заинтересованные могут найти дополнительную информацию в официальном описании языка или многочисленной литературе.
- Используйте алгебраические типы данных во всех остальных случаях.

АТД: определение

Алгебраический тип данных неформально можно определить как множество значений, представляющих собой некоторые контейнеры, внутри которых могут находиться значения каких-либо иных типов (в том числе и значения того же самого типа — в этом случае имеет место рекурсивный АТД). Так что множество таких контейнеров и составляет сам тип данных, множество его значений.

АТД — размеченное объединение декартовых произведений множеств или, другими словами, размеченная сумма прямых произведений множеств.

Размеченное объединение

Пусть есть набор множеств $A_i, i \in I$, из которых создаётся их размеченное объединение. В этом случае под размеченным объединение понимается полное объединение пар:

$$\coprod_{i \in I} A_i = \bigcup_{i \in I} \{(x, i) | x \in A_i\}$$

Здесь (x, i) — упорядоченная пара, в которой элементу x приписан индекс множества, из которого элемент попал в размеченное объединение. В свою очередь множества A_i *канонически* вложены в размеченное объединение, то есть пересечения всех таких вложений пусты.

Декартово произведение

Каждое множество A_i^* (каноническое вложение множества A_i) является декартовым произведением n множеств ($n \in [0, \infty)$), то есть:

$$A_i = A_{i_1} \times A_{i_2} \times \cdots \times A_{i_n},$$

где множества $A_{i_k}, k = \overline{1, n}$ являются произвольными (в том числе нет ограничений на рекурсивную вложенность).

Так что *общий вид АТД* задаётся формулой:

$$\coprod_{i \in I} A_i = \bigcup_{i \in I} \left\{ \left((x_1, x_2, \dots, x_{n_i}), i \right) | x \in A_i \right\}$$

Общий вид АТД

АТД A

Part A^*_1

1 x_1 x_2 … x_n

Part A^*_2

2 y_1 y_2 … y_m

Part A^*_N

N z_1 z_2 … z_l

Метка размеченного объединения

Декартово произведение

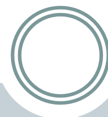
Каноническое вложение множества A_N

Синтаксически-ориентированное конструирование

Чарльз Энтони Хоар предложил более «человеческую» нотацию для описания типов на абстрактном уровне (вне синтаксиса какого-либо конкретного языка программирования) — *синтаксически-ориентированное конструирование*.

- Типы именуются словами в латинском алфавите, начинающимися с заглавной буквы. Конкретные типы имеют конкретные названия. Типовые переменные обозначаются одиночными заглавными буквами.
- **constructors** — конструкторы типа, описывают декартовы произведения в составе типа.
- **selectors** — специальные функции-геттеры, позволяющие получать отдельные значения из декартовых произведений.
- **parts** — наименования отдельных канонических множеств из состава размеченного объединения в составе типа.
- **predicates** — функции, позволяющие идентифицировать принадлежность значения заданному каноническому множеству.

Нотация Ч. Э. Хоара. Пример



List(A) = NIL + (A × List(A))

nil, prefix = **constructors** List(A)

head, tail = **selectors** List(A)

NIL, nonNIL = **parts** List(A)

null, nonNull = **predicates** List(A)

Здесь *A* — типовая переменная, произвольный тип данных, значения которого содержатся в списке. При конкретизации списка вместо *A* подставляется необходимый тип.

Аксиома тектоничности

Каждый селектор имеет тип вида $A \rightarrow A_{ik}$, и такой селектор принимает на вход значение своего АТД, а возвращает значение из декартова произведения, которому он соответствует. Для каждой части АТД имеется столько селекторов, сколько типов упаковывается в декартовом произведении. Соответственно, для каждой части типа выполняется следующее тождество:

$$\forall x \in A_i\colon \mathbf{constructor}\ A_i\left((s_{i1}(x)), (s_{i2}(x)), \ldots \left(s_{in_i}(x)\right)\right) = x,$$

где $s_{i1}, s_{i2}, \ldots, s_{in_i}$ — селекторы соответствующих компонентов декартова произведения.

Алгебра над типами

Описанная система типов позволяет мгновенно создавать новые типы на основе имеющихся при помощи двух базовых операций — сложения и произведения. Пусть есть два типа *A* и *B*. Тогда:

- *Сложение*. Следующий тип является суммой типов *A* и *B*:

Sum(A, B) = A + B

partA, partB = **parts** Sum(A, B)

isA, isB = **predicates** Sum(A, B)

- *Произведение*. Следующий тип является произведением типов *A* и *B*:

Product(A, B) = A × B

product = **constructors** Product(A, B)

selectA, selectB = **selectors** Product(A, B)

Алгебра над типами в нотации языка Haskell

То же самое в нотации языка Haskell будет выглядеть следующим образом. Пусть есть два типа α и β. Тогда:

- *Сложение*:

```
Sum α β = PartA α
        | PartB β

isA :: Sum α β → Bool
isA (PartA _) = True
isA _         = False

isB :: Sum α β → Bool
isB (PartB _) = True
isB _         = False
```

- *Произведение*:

```
Product α β = Product α β

selectA :: Product α β → α
selectA (Product a _) = a

selectB :: Product α β → β
selectB (Product _ b) = b
```

Куда двигаться дальше?

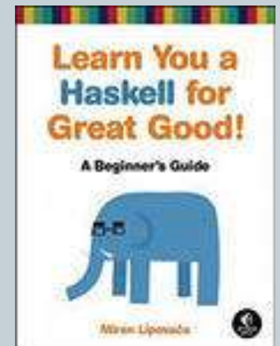
- Книги
  - Книги Душкина Р. В. о языке Haskell
  - Другие книги о языке Haskell и функциональном программировании
  - Обзор литературы по функциональному программированию от Алекса Отта
- Некоторые ресурсы в сети Интернет
- Haskell Platform
- Конкурсы по ФП

Мои книги

- Душкин Р. В. *Функциональное программирование на языке Haskell*. — М.: ДМК-Пресс, 2007. — 608 стр., ил. — ISBN 5-94074-335-8.
- Душкин Р. В. *Справочник по языку Haskell*. — М.: ДМК-Пресс, 2008. — 544 стр., ил. — ISBN 5-94074-410-9.
- Душкин Р. В. *Практика работы на языке Haskell*. — М.: ДМК-Пресс, 2010. — 288 стр., ил. — ISBN 978-5-94074-588-4.
- Душкин Р. В. *14 занимательных эссе о языке Haskell и функциональном программировании*. — 2-ое изд., исп. — М.: 2011. — 282 стр., ил. — http://www.twirpx.com/file/643353/
- Душкин Р. В. *Другие 14 эссе о языке Haskell и функциональном программировании — серьёзные*. — М.: 2012. — 365 стр., ил. — http://www.twirpx.com/file/999451/
- Душкин Р. В. *Альманах «Конкурсы по Функциональному Программированию» за 2011 год*. — М.: 2012. — 59 стр., ил. — http://www.twirpx.com/file/774800/
- Душкин Р. В. *Альманах «Конкурсы по Функциональному Программированию» за 2012 год*. — М.: 2013. — 210 стр., ил. — http://www.twirpx.com/file/1108038/

Книги других авторов на русском языке

- Роганова Н. В. *Функциональное программирование*. — М.: Институт ИНФО, 2002.
- Липовача М. *Изучай Haskell во имя добра*. — М.: ДМК-Пресс, 2012. — 490 стр., ил. — ISBN 978-5-94074-749-9.
- Холомьёв А. *Учебник по Haskell*. — http://anton-k.github.com/ru-haskell-book/book/home.html
- *Мягкое введение в Haskell*:
  - http://rsdn.ru/article/haskell/haskell_part1.xml
  - http://rsdn.ru/article/haskell/haskell_part2.xml



Обзор литературы по ФП

Полный, постоянно дополняемый обзор литературы о функциональном программировании от Алекса Отта:

http://alexott.net/ru/fp/books/

Некоторые ресурсы в сети Интернет

- Центр мирового сообщества языка Haskell: http://www.haskell.org/ — здесь есть абсолютно всё о языке, его библиотеках, применении, приложениях и всём остальном.
- Полный перевод описания стандарта Haskell-98 и стандартных библиотек на русский язык: http://www.haskell.ru/
- Интерактивная книга «Learn you a Haskell for Great Good»: http://learnyouahaskell.com/
- Книга «Real World Haskell»: http://book.realworldhaskell.org/
- Журнал «The Monad.Reader»: http://themonadreader.wordpress.com/
- Интерактивный интерпретатор языка Haskell: http://tryhaskell.org/

Журнал «ПФП»

Научно-практический журнал
«**Практика Функционального Программирования**»

- Адрес: http://fprog.ru/
- Предлагается разнообразное количество вариантов для самых разных устройств.
- Также имеется возможность заказать бумажные экземпляры.
- Лента Russian Lambda Planet: http://fprog.ru/planet/

Haskell Platform

Все, заинтересовавшиеся языком Haskell и предоставляемыми им преимуществами по сравнению с императивными («мейнстимовыми») языками программированию, могут начать непосредственную работу с ним при помощи пакета прикладных программ Haskell Platform. На текущий момент выпущена версия 2012.4.0.0, которая работает под ОС Windows, Mac и Linux. Этот пакет содержит компилятор GHC, кучу дополнительных утилит к нему (всё, что необходимо для разработки), полный набор библиотек для начала разработки. Остальные библиотеки ставятся при помощи утилиты Cabal из архива Hackage. Всё очень просто.

Адрес для скачивания: http://www.haskell.org/platform/

Конкурсы по ФП

В августе 2011 года был запущен процесс под названием «Конкурсы по Функциональному Программированию». Конечно, для участия в конкурсе приглашаются все желающие, независимо от используемого языка программирования. Сегодня конкурс проводится один раз в 2 месяца (по чётным месяцам года), на него предлагаются интересные и занимательные задачи, победителям выдаются разнообразные призы.

Следующий конкурс будет объявлен на первой неделе июля 2013 года. Задача будет посвящена Универсиаде, которая будет проводиться в Казани этим летом. Участвуйте!

Адрес: **http://haskell98.blogspot.ru/**

Благодарю за внимание

ДУШКИН Р. В.

ТЕЛ.: +7 (909) 695-41-38

SKYPE: ZIL-ROMAN

E-MAIL: ROMAN.DUSHKIN@GMAIL.COM

БЛОГ ФП(ФП): HASKELL98.BLOGSPOT.RU