Душкин Роман Викторович
Специалист по функциональному программированию и функциональным парсерам. Автор книги «Функциональное программирование на языке Haskell»

Объектно-ориентированное и функциональное программирование

Эта статья рассматривает вопросы слияния двух наиболее интересных парадигм программирования — объектно-ориентированной и функциональной, которые в настоящее время нашли наибольшее число применений в прикладной области. Статья продолжает цикл статей в журнале «Потенциал», посвящённых функциональному программированию и языку Haskell.

Введение

С момента рождения науки о вычислительных процессах в её прикладных аспектах было разработано достаточно широкое множество парадигм[1] программирования, которые использовались и до сих пор используются в различных областях компьютерной науки. При этом часто так получалось, что новая парадигма разрабатывалась для решения новых классов задач. Но всё разнообразие идей можно разделить на два больших класса — императивное и декларативное программирование. Внутри же

[1] Под **парадигмой** (от греческого слова παραδειγμα — пример, модель, образец) здесь и далее понимается общая концептуальная схема постановки проблем и методов их решения. В более узком смысле под парадигмой программирования будет пониматься не просто стиль написания программ, а способ мышления, который позволяет использовать тот или иной стиль при создании таких программ.

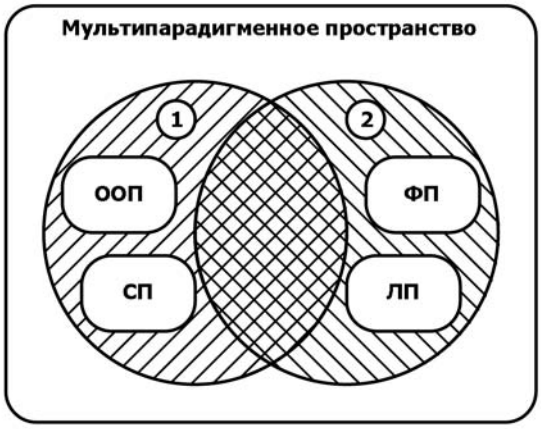

Рис. 1. Схематическое изображение мультипарадигменного пространства

этих классов рассматриваются отдельные парадигмы, которые могут вполне и пересекаться друг с другом. Да и сами эти направления также пересекаются — некоторые задачи можно легко решить как с помощью императивного, так и с помощью декларативного программирования.

Все разработанные парадигмы программирования можно свести к единой схеме областей в «мультипарадигменном пространстве», отвечающих за ту или иную парадигму. Такие области будут пересекаться, и все их можно разделить в соответствии с упомянутым ранее принципом императивности/декларативности. Схематически такое пространство показано на рис. 1.

На изображённой схеме цифрой 1 указано императивное программирование, в котором программы как бы предписывают ЭВМ выполнять некоторые алгоритмические действия (от лат. «imperativus» — «повелительный»). С другой стороны (под цифрой 2) находится декларативное программирование, характеризующееся описательным подходом к построению программ (от лат. «declarativus» — «описательный»). Декларативные программы описывают способ решения задачи в терминах проблемной области, а ЭВМ сама выбирает способ нахождения решения (последовательность действий получения решения).

Во всём множестве парадигм выделяются две, которые используются особенно часто. Первая — объектно-ориентированная парадигма программирования (ООП), которая нашла самое широкое применение в прикладных областях. Вторая — функциональное программирование, ставшее основным инструментом в научных исследованиях по компьютерной науке. Первая парадигма оперирует объектами и их классами, при помощи которых описывается проблемная область и все взаимосвязи между сущностями в ней. Второй стиль использует функции для представления вычислительных процессов, так как функции являются естественным способом описания преобразования входных параметров в выходной результат.

В этой статье описывается слияние объектно-ориентированной и функциональной парадигм на примере функционального языка Haskell, в котором имеются достаточные средства для выражения всех принятых в объектно-ориентированном стиле конструкций.

Именованные поля и структуры

Разработчики языка Haskell предусмотрели возможность определять свои структуры данных — *алгебраические типы данных*. И хотя их определение выглядит весьма традиционным, их внутренняя реализация и использование достаточно сильно отличаются от структур в императивных языках программирования. Ничего не поделаешь — функциональный стиль накладывает свои ограничения на все аспекты деятельности программиста.

Однако стоит рассмотреть пример, чтобы понять принципы и способы использования в языке Haskell алгебраических типов данных. В качестве такого примера возьмём различные геометрические фигуры на плоскости и операции над ними.

Для начала необходимо определить тип, который будет представлять собой точку на плоскости. Это базовый тип для любых геометрических фигур. Точка характеризуется наличием двух координат — **x** и **y**, причём координаты имеют одинаковый тип, который позволяет производить над координатами арифметические операции и вычислять тригонометрические функции. Тип данных **Point** (Точка) на языке Haskell определяется достаточно просто:

```
data Floating a => Point a = Point
{
 x :: a,  y :: a
}
```

Правильнее сказать, что это не тип, а шаблон типа. У него есть входной параметр **a** — тип, который имеют координаты. Объявление начинается с ключевого слова **data**. Затем указано ограничение **Floating a** на входной тип **a**, которое означает, что тип **a** должен иметь возможность участвовать в арифметических операциях и тригонометрических функциях.

Далее идёт разделитель «**=>**», который можно интерпретировать как «если выполнено условие на тип слева, тогда ...».

В результате определяется алгебраический тип данных с одним конструктором **Point**, который параметризуется типом **a** с ограничением **Floating**.

Может показаться, что отличие от объявления структур данных (записей) в императивных языках Pascal или Си только в синтаксисе. Но в действительности есть несколько тонких моментов.

Вместо записи

```
Point { x :: a, y :: a }
```

можно было бы написать **Point a a**. Грубо можно считать, что **Point a a** — это тип списка трёх элементов. Первый элемент — специальный маркер **Point**, а затем два элемента типа **a**.

Запись **Point { x :: a, y :: a}** заставляет транслятор языка Haskell соз-

дать две функции для доступа к определённым полям (второму и третьему элементам списка). Такое определение неявно подразумевает наличие двух функций:

```
x :: Fractional a => Point a -> a
x (Point v1 v2) = v1
y :: Fractional a => Point a -> a
y (Point v1 v2) = v2
```

Но с помощью имён **x** и **y** можно не только читать содержимое полей, но и менять их. Для этого нужно использовать эти имена слева от знака равно:

```
shiftByX :: Fractional a =>
   Point a -> a -> Point a
shiftByX (Point v _) dX =
   Point { x = v + dX }
shiftByY :: Fractional a =>
   Point a -> a -> Point a
shiftByY (Point _ v) dY =
   Point { y = v + dY }
```

Это две функции параллельного переноса (**shift**) вдоль осей координат.

Важно отметить, что при «выполнении» данных функций будет создаваться новый объект с изменённым значением поля, а не меняться значение поля данного. Вторая координата у созданного объекта останется неопределённой — **undefined** («неопределено» — ⊥), поэтому вызов **x (shiftByY (Point 0 0) 1)** выдаст ошибку. Чтобы этого не было, следует писать:

```
shiftByX (Point v1 v2) dX =
   Point { x = v + dX, y = v2}
```

Для того, чтобы избежать таких ошибок, достаточно использовать *именованные образцы*, для которых уже изменять значения определённых полей. В этом случае значения других полей останутся неизменными, а не получат значение **undefined**:

```
shiftByX :: Fractional a =>
   Point a -> a -> Point a
shiftByX p@(Point v _) dX =
   p { x = v + dX }
shiftByY :: Fractional a =>
   Point a -> a -> Point a
shiftByY p@(Point _ v) dY =
   p { y = v + dY }
```

Можно пользоваться и обычным способом записи новых значений в поля структуры:

```
shift :: Fractional a =>
   Point a -> a -> a -> Point a
shift (Point v1 v2) dX dY =
   Point (v1 + dX) (v2 + dY)
```

Эта функция осуществляет перемещение данной точки на **dX** вдоль оси **X** и на **dY** вдоль оси **Y** (параллельный сдвиг на вектор **(dX, dY)**).

Данные функции являются примером того, как можно работать с вновь созданным типом данных, представляющим собой точку.

А сейчас на его основе сделаем тип данных, представляющий собой любую геометрическую фигуру. Можно сделать что-то типа следующего:

```
data Floating a => Figure a =
   Circle {  center :: Point a,
   radius :: a }
| Rectangle {  anchor :: Point a,
   width :: a,   height :: a }
| Polygon {  points :: [Point a]}
```

Однако такая запись не очень хороша с точки зрения масштабируемости решения. Если потребуется добавить новое описание какой-либо геометрической фигуры, то придётся дописывать

строчки к определению этого типа, а это неудобно с точки зрения *модульности* проекта. Поэтому лучше сделать несколько типов, набор которых можно дополнять по мере необходимости в том числе и из внешних модулей, описывая в них новые геометрические фигуры. Таким образом, первоначальный набор фигур таков:

```
data Floating a => Circle a =
   Circle {  center :: Point a,
   radius :: a
}
data Floating a => Rectangle a =
   Rectangle {
   anchor :: Point a,
   width :: a,
   height :: a
}
 data Floating a => Polygon a =
      Polygon {
   points :: [Point a]
}
```

Но теперь возникает желание применить к объектам этих типов такую же операцию для сдвига **shift**. Просто так это не получится, так как сигнатура функции **shift** позволяет ей работать только с объектами типа **Point a**. Придётся писать отдельные методы с уникальными именами для каждой геометрической фигуры. Но ведь этого так не хочется делать! На сцену выходят такие сущности языка Haskell, как классы.

Классы

Необходимо сразу отметить, что в языке Haskell под понятием «класс» понимается нечто иное, нежели в объектно-ориентированных языках подобных C++, Java и др. Класс в языке Haskell — это более абстрактное понятие, которое относится к системе типов языка программирования. Класс — это не тип данных, это «тип типов». Экземплярами класса могут быть типы данных.

Другая точка зрения на класс заключается в том, что под классом в языке Haskell понимается интерфейс, который специфицирует определённый набор методов для работы с некоторой структурой данных. Такой подход имеет право на существование, так как в действительности при определении классов происходит описание функций, которые необходимо определить для тех типов, которые впоследствии станут экземплярами создаваемого класса.

В этом и заключается ключ к решению тех проблем, которые поставлены в предыдущем разделе. Класс определяет имена функций, которые будут оперировать с объектами определённых типов, причём достаточно все эти типы определить экземплярами класса. Прежде чем перейти к созданию класса для описания действий, которые могут быть предприняты над геометрическими фигурами, стоит рассмотреть пример из стандартного модуля **Prelude** языка Haskell, чтобы более чётко понимать, что такое классы.

В качестве примера можно рассмотреть базовые арифметические операции. В математике для их обозначения традиционно используются символы (+), (-) и (×) (операция деления (:) не рассматривается, так как она не является замкнутой относительно множества целых чисел, а сейчас важно рассмотреть именно целые

числа в качестве обучающего примера). Но применять арифметические операции можно над операндами разных типов — натуральные числа, целые числа, действительные числа и так далее. В «чистой» математике учёный редко задумывается над тем, что эти объекты разных типов, но программист об этом помнит всегда. Естественно, в силу традиции хочется обозначать арифметические операции одинаково для всех типов чисел.

В разных языках программирования для этих целей используются разные механизмы. В большинстве случаев базовые арифметические операции и перегрузка их имён «вшита» в транслятор языка. Но создатели языка Haskell пошли иным путём. Для этих целей в стандартном модуле **Prelude** описан класс **Num**, представляющий собой класс типов, объекты которых «похожи» на целые числа. Над объектами этих чисел можно производить базовые арифметические операции, целочисленное деление, взятие остатка от деления, а также несколько других функций. Определение этого класса выглядит так:

```
class (Eq a, Show a) => Num a
where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
  fromInteger   :: Integer -> a
  fromInt       :: Int -> a
  x - y    = x + negate y
  negate x = 0 - x
  fromInt  = fromIntegral
```

Как видно, определение класса начинается с ключевого слова **class**, после которого идёт секция со спецификацией имени класса (о ней позже) до ключевого слова **where**. После этого ключевого слова идёт перечисление методов класса, а также описание взаимосвязей методов друг с другом (при необходимости). Методы описываются просто — после имени стоит символ (**::**) «имеет тип», после которого описывается сигнатура соответствующего метода. Методы с одинаковыми сигнатурами можно перечислять через запятую.

Секция с описанием взаимосвязей записывается при необходимости и возможности выразить один метод класса через другой. Так, в приведённом примере вычитание одного числа из другого можно выразить через сложение и функцию **negate** (отрицание). Естественно, что такие взаимосвязи можно описать не всегда.

Спецификация имени класса состоит из ограничения на переменные типов (необязательная часть), собственно имени класса и переменной, которая обозначает будущие экземпляры описываемого класса (в рассматриваемом примере — **a**). Ограничение на тип **a** задаётся при помощи перечисления классов, экземпляром которых должен быть тип **a**, написанных до символа (**=>**). Если таких ограничений несколько, то их надо перечислить через запятую и заключить в круглые скобки.

Таким образом, приведённое определение класса **Num** можно читать следующим образом: «Класс **Num** специфицирует для некоторого типа **a**, который обязан являться экземпляром классов **Eq** и **Show**, методы (**+**), (**-**), (*****), **negate**, **abs**, **signum**, **fromInteger**, **fromInt**, которые имеют заданные сигна-

туры. Кроме того, для методов (**-**), **negate** и **fromInt** определяются выражения через иные методы этого класса и прочие функции». Такой класс можно использовать для определения методов над элементами определённого типа, которые предназначены для сложения, вычитания и т. д. Это могут быть любые элементы — всё зависит только от фантазии разработчика.

Теперь можно возвратиться к примеру с геометрическими фигурами. Любая фигура, независимо от её фактического типа, может перемещаться, масштабироваться и вертеться относительно какой-то точки. Поэтому можно определить три метода: **shift**, **scale** и **rotate** в классе **Figure**:

```
class Figure f where
  shift  :: Floating a => f a ->
   a -> a -> f a
  scale  :: Floating a => f a ->
   Point a -> a -> f a
  rotate :: Floating a => f a ->
   Point a -> a -> f a
```

В классе описаны только *сигнатуры* этих методов. Сигнатуры содержат информацию об имени функции, о входных и выходных данных (количество элементов, их типы, ограничения на типы). Они могут помочь понять, что должны делать сами методы. Так, к примеру, метод **shift** получает на вход описание фигуры и два коэффициента сдвига, а возвращает новую фигуру. Методы **scale** и **rotate** получают на вход фигуру, точку, относительно которой производится преобразование, и коэффициент преобразования (масштаб и угол соответственно), а возвращают новую преобразованную фигуру.

Теперь остаётся объявить все определённые ранее алгебраические типы данных для представления геометрических фигур экземплярами нового класса.

Экземпляры классов

Создание класса не влечёт ничего, что влияет на исполнение программы. Для создания функций, сигнатуры которых описаны в классе и которые бы работали с определёнными типами данных, необходимо определить такие данные в качестве экземпляров соответствующего класса.

Для определения экземпляра необходимо воспользоваться ключевым словом **instance** с указанием имени класса. Затем нужно указать тип, который мы собираемся объявить экземпляром этого класса. После ключевого слова **where** следует дать определения методов класса. Например, для типа **Point** это делается следующим образом:

```
instance Figure Point where
  shift (Point x y) dX dY =
    Point (x + dX)
          (y + dY)
  scale (Point x y) (Point xa ya)
k =
    Point (xa + k * (x - xa))
          (ya + k * (y - ya))
  rotate (Point x y) (Point xa ya)
k =
    Point (xa + r * cos (alpha + k))
          (ya + r * sin (alpha + k))
        where r     = sqrt ((x -
xa)^2 + (y - ya)^2)
              alpha = acos ((x - xa) / r)
```

Как видно, ничего сложного нет. Это определение гласит, что для типа **Point** определены методы класса **Figure**: **shift**, **scale** и **rotate**. После этого можно использовать данные методы на операндах типа **Point**.

Надо отметить, что сами по себе алгебраические типы данных и классы — совершенно не связанные друг с другом единицы исходного кода на языке Haskell. Экземпляры классов — это та сущность, при помощи которой происходит связывание классов и типов. Но и экземпляр — тоже независимая программная сущность: она может существовать, а может и не существовать. Существование экземпляров позволяет говорить транслятору языка Haskell, что функции класса могут работать с объектами определённых типов (если, конечно, они соответствуют ограничениям, прописанным в сигнатурах функций).

Для того, чтобы работать с описанными ранее геометрическими фигурами, необходимо создать соответствующие экземпляры класса **Figure**. При этом следует учесть, что, имея в наличии экземпляр типа **Point**, остальные экземпляры определяются уже достаточно легко. Надо обратить внимание, что для элементов типа **Point** можно использовать определённые методы **shift**, **scale** и **rotate**. Собственно, дело за малым:

```
instance Figure Circle where
  shift (Circle c r) dX dY =
    Circle (shift c dX dY) r
  scale (Circle c r) pa k =
    Circle (scale c pa k) (k * r)
  rotate (Circle c r) pa k =
    Circle (rotate c pa k) r

instance Figure Rectangle where
  shift (Rectangle a w h) dX dY =
    Rectangle (shift a dX dY) w h
  scale (Rectangle a w h) pa k =
    Rectangle (scale a pa k) (w * k)
(h * k)
  rotate (Rectangle a w h) pa k =
    Rectangle (rotate a pa k) w h

instance Figure Polygon where
  shift (Polygon pts) dX dY =
    Polygon (modify pts dX dY)
      where modify [] _ _       = []
            modify (p:ps) dX dY =
              (shift p dX dY):(mo-
dify ps dX dY)
  scale (Polygon pts) pa k  =
    Polygon (modify pts pa k)
      where modify [] _ _       = []
            modify (p:ps) pa k  =
              (scale p pa k):(mo-
dify ps pa k)
  rotate (Polygon pts) pa k =
    Polygon (modify pts pa k)
      where modify [] _ _       = []
            modify (p:ps) pa k  =
              (rotate p pa k):(mo-
dify ps pa k)
```

Всё просто, но остаётся несколько проблем больше эстетического характера. Первое, что бросается в глаза — много практически одинаковых строк кода в определении экземпляра **Figure Polygon**. Конечно, хотелось бы все подобные повторения свести в единую функцию **modify**, которая была бы функцией высшего порядка и принимала бы на вход тот метод, который необходимо применить к набору точек. Но тут не всё так просто. К сожалению, класс **Figure** спроектирован так, что у метода **shift** тип совершенно иной, нежели у остальных методов.

Поэтому простая функция не поможет. Решение данной задачи оставляется вдумчивому читателю.

Вторая проблема кроется в методе **rotate** для прямоугольника **Rectangle**. Этот тип описан так, что вращать можно только опорную точку, а его стороны всегда остаются параллельны осям координат (хотя это явно и не описано; при таком описании можно предположить, что стороны вообще направлены так, как заблагорассудится — здесь вопрос в интерпретации типа). Поэтому, если есть необходимость во вращении прямоугольника в виде целостной фигуры, необходимо менять сам тип данных. Эта задача также остаётся для самостоятельного решения.

Отличительные черты ООП на Haskell

Здесь приведены важные замечания относительно сравнения понятия «класс» в языке Haskell и в прочих объектно-ориентированных языках программирования.

1. Язык Haskell разделяет определения классов и их методов, в то время как такие языки, как C++ и Java вместе определяют структуру данных и методы для её обработки. В определении класса в языке Haskell даётся только сигнатура методов, которые должны быть реализованы в экземплярах класса. Вместе с этим, возможно написание определений таких методов, используемых по умолчанию, когда в типах-экземплярах класса реализации метода нет.

2. Можно сказать, что определения методов классов в языке Haskell больше всего соответствуют виртуальным функциям языка C++. Каждый конкретный экземпляр класса должен переопределять методы класса для использования их над своей собственной структурой. В случае необходимости и возможности можно воспользоваться определениями методов, определённых по умолчанию.

3. С точки зрения языка Java, более всего классы в языке Haskell похожи на интерфейсы. Как и определение интерфейса, классы в языке Haskell предоставляют только протокол использования объекта, вместо определения самих объектов. Так и в языке Java можно реализовать в каком-либо классе столько интерфейсов, сколько потребуется, но наследовать можно только один класс. Типы в языке Haskell могут быть экземплярами стольких классов,

сколько потребуется — равно как и в языке Java.

4. Язык Haskell не поддерживает стиль перегрузки функции, используемый в C++, когда функции с одним и тем же именем получают данные различных типов для обработки. Вместо этого применяется технология использования синонимов имён функций, когда разные функции с различными наименованиями для обработки разных данных в определённых ситуациях могут вызываться по одинаковому идентификатору. Это как раз и реализуется при помощи классов в языке Haskell и указания ограничений на типы аргументов.

5. В классах языка Haskell не существует понятия контроля за доступом — нет публичных и защищённых методов, т.е. нет аналогов служебных слов **public**, **private** и др. Все методы в классах языка Haskell являются открытыми, более того, они определены в контексте модуля, т.е. являются декларациями самого высокого уровня.

Всё, что было рассмотрено в предыдущих разделах, а также в предыдущих статьях на эту тему, подводит к мысли о том, что при программировании на языке Haskell используются всего пять видов деклараций, пять сущностей: функции, типы, классы, экземпляры классов и модули. Все эти сущности являются самостоятельными, только опосредованно связанными друг с другом. Это особенно касается классов и их экземпляров — эти сущности не связаны ни друг с другом, ни с типами. Классы и алгебраические типы данных могут быть описаны независимо друг от друга, а экземпляр класса — это сущность, связывающая класс и тип.

То, что записывается в контексте (**Class a =>**), является всего лишь ограничением вида «должен существовать экземпляр указанного класса **Class** для заданного типа **a**». Это значит, что типы и их реализации для классов существуют раздельно.

В качестве примера можно рассмотреть такую ситуацию. Пусть кто-то давно создал некий тип данных **D** и некоторую очень полезную функцию **f**, которая в своей сигнатуре имеет ограничение (**C a**), где **a** — тип одного из аргументов. Пусть эти сущности определены в разных модулях, которые никак не связаны друг с другом. Разработчики, создававшие эти исходные сущности, даже не догадывались о взаимном существовании. И вдруг третий разработчик понял, что ему необходимо использовать функцию **f** над объектами типа **D**. Что делать?

Нет никакой необходимости искать программиста, создавшего класс **C**, чтобы он включил в его определение возможность работать с типом **D**. Достаточно в своём модуле реализовать экземпляр этого класса для типа **D**, что позволит использовать значения этого типа в функции **f**. Исходные модули остаются нетронутыми. И это есть серьёзное преимущество системы модулей, классов и их экземпляров языка Haskell.

Заключение

Таким образом, в результате получился некоторый модуль с описанием базовых методов и типов для проведения геометрических преобразований. Этот модуль можно использовать в качестве «затравки» для полноценной библиотеки, решающей различные геометрические задачи. При этом надо учесть, что сами определения из этого модуля являются достаточно абстрактными — они не привязаны к какой-либо предметной области. Функциям и методам классов безразлично, что обсчитывать — графические образы на экране монитора (в пикселях) либо аналитические геометрические задачи с координатами произвольной точности — везде созданные методы будут применимы. При особом желании модуль действительно можно развить в нечто большее.

Интересующемуся читателю рекомендуется подумать и сделать подобный модуль для работы с геометрическими фигурами, узловые точки которых задаются в полярных координатах. Для этого можно использовать созданный модуль, а также определение типа:

```
data Floating a => PolarPoint a =
   PolarPoint {
      angle    :: a,
      distance :: a
}
```

В стандартном модуле **Prelude** описана большая иерархия классов, которые подходят для определения типов данных различной природы, — от перечислимых множеств до комплексных чисел и более сложных объектов. Можно обратиться к этому модулю для дополнительного изучения того, что представляют собой классы в языке Haskell.

Автор благодарит Антонюка Д.А., вместе с которым все описанные в этой статье программные сущности были обсуждены, разработаны и проверены.

Все перечисленные в этой статье определения типов и функций сведены в отдельный модуль, который каждый желающий может получить, послав электронное письмо с соответствующим запросом (пожалуйста, указывайте в запросе наименование статьи, для которой необходимо прислать определения) на электронный адрес автора статьи — darkus.14@gmail.com. Надо отметить, что сам модуль тестировался в интерпретаторе HUGS 98, бесплатную версию которого можно найти по адресу в интернете: **http://www.haskell.org/hugs/.**

Кроме того, автор будет признателен любым отзывам и комментариям к статье, которые помогут сделать дальнейшие статьи более интересными и полезными для читателей. Всё это также можно посылать по указанному выше адресу электронной почты.