Душкин Роман Викторович
Автор нескольких книг по функциональному программированию.

Кривая Дракона

В данной статье в очередной раз рассматриваются практические аспекты применения языка Haskell при решении расчётных задач. В качестве примера изучается один из видов фракталов, имеющий поэтическое наименование – «Кривая Дракона». Приводятся теоретические аспекты, рассматривается построение алгоритма на математическом уровне, после чего алгоритм реализуется на языке Haskell.

Введение

В математике имеется одна интереснейшая отрасль, которая занимается изучением самоподобных объектов, каждая мельчайшая часть которых подобна всему объекту. Под подобием здесь понимается похожесть до некоторой степени. Сама степень подобия в данном случае может являться фактором, при помощи которого можно классифицировать изучаемые объекты. Ну а сами такие объекты называются *фракталами*.

Одним из наиболее известных фракталов является *множество Мандельброта,* которое строится при помощи достаточно простой процедуры (подробности см. в [3]). Для произвольного комплексного числа z_0 строится отображение по формуле:

$$z_{i+1} = z_i^2 + c,$$

где c – некоторая константа. В этой процедуре индекс i последовательно пробегает все целые неотрицательные значения от 0 и далее. Так итеративно строится последовательность $z_0, z_1, z_2, \ldots$. Если зафиксировать $z_0 = 0$, то получаемые на очередном шаге значения будут полностью зависеть от выбора константы c, которая на любом шаге должна быть одинаковой.

Например, если $z_0 = 0$, то последовательность будет выглядеть так:

- $z_0 = 0$,
- $z_1 = c$;
- $z_2 = c^2 + c$;
- $z_3 = c^4 + 2c^3 + c^2 + c$;
- $z_4 = c^8 + 4c^7 + 6c^6 + 6c^5 + 5c^4 + 2c^3 + c^2 + c$;
- ...

И вот оказывается, что для некоторых значений c при фиксированном z_0 описанная процедура создаёт бесконечную последовательность, в которой любое число ограничено некоторой небольшой областью около

начала координат – комплексного числа $0+0i$. Например, тривиальным значением константы c является 0, поскольку в данном случае любой член последовательности также равен 0. Другой пример: $c=-1$.

Множество Мандельброта есть множество всех значений константы c, для которых получаемая по описанной процедуре последовательность является ограниченной. На рис. 1 показано множество Мандельброта для $z_0=0$. Как видно, множество Мандельброта впечатляет своей сложностью, особенно учитывая, как это часто бывает в математике, удивительную простоту его определения.

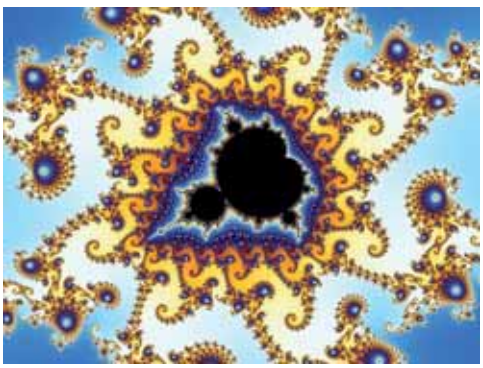
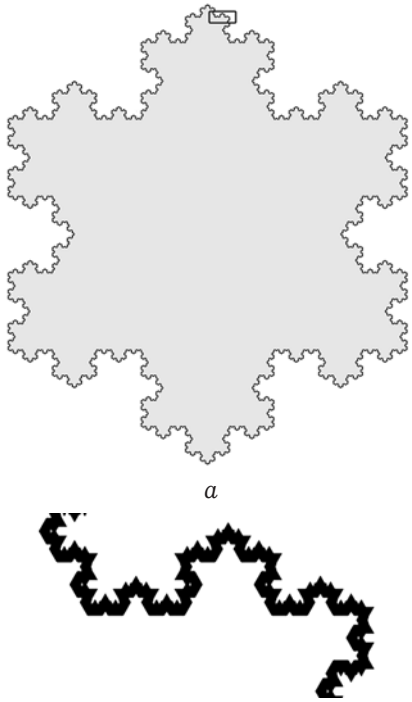


Рис. 1

Другой пример фрактала приводит в своей книге [1] замечательный американский популяризатор науки Мартин Гарднер. Он строит снежинку, начиная с простого равностороннего треугольника, заканчивая фигурой с бесконечным периметром, но ограниченной площадью. На каждой итерации построения периметр фигуры становится всё сложнее и замысловатее, повышение точности его измерения влечёт неограниченный рост его длины, а площадь остаётся ограниченной очень небольшой областью. Как пишет сам М. Гарднер, такая фигура может быть изображена на обычной почтовой марке, а сумма длин её сторон будет сравнима с диаметром нашей галактики (естественно, при должной точности нанесения изображения и последующего измерения). Примерный вид фрактала «Снежинка» для небольшого числа итераций и отдельная его увеличенная часть изображены на рис. 2 *а, б*.



а

б

Рис. 2

Ещё один фрактал в своё время был разработан автором по мотивам прочтённого описания фрактала «Снежинка». Для его построения используются уже не равносторонние треугольники, а квадраты. Его периметр также стремится к бесконечности, хотя площадь всегда ограничена. Примерный вид этого фрактала для небольшого числа итераций показан на рис. 3.

Все эти примеры сложных фигур объединяет одно свойство – они самоподобны. Если рассмотреть какую-либо часть фигуры произвольного

размера на границе, то при должном приближении и достаточной точности будет видно, что малая часть очень похожа на всю фигуру – подобна ей.

Фракталы имеют множество областей применения как в чистой науке, так и на практике. Например, некоторые современные техники шифрования информации основаны на результатах, полученных при изучении фракталов. Но кроме научного и практического применения у этих фигур имеется ещё одно – эстетическое. Они необычайно красивы.

Далее в настоящей статье рассматривается ещё один вид фрактала, имеющий название «Кривая Дракона».

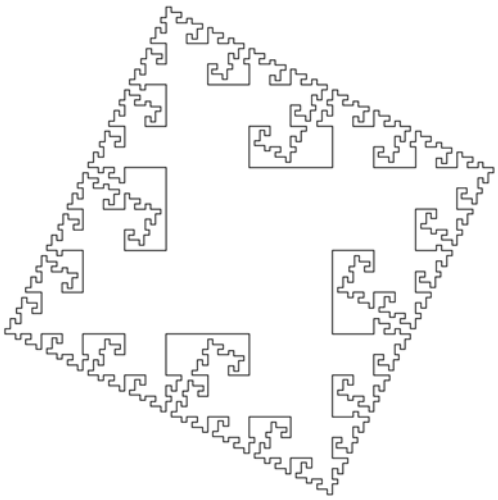


Рис. 3

1. Что такое Кривая Дракона?

Как и рассмотренные во введении фракталы, Кривая Дракона строится весьма простым способом. Любой может построить начальные итерации при помощи бумажной ленты. Любознательному читателю можно посоветовать следующий способ построения Кривой Дракона.

Необходимо взять бумажную ленту, скажем, размером 100 × 1 см. Чем тоньше будет бумага, тем лучше, поскольку такую ленту придётся многократно сгибать, а при сгибании толщина бумаги будет многократно складываться, внося погрешности в создаваемую фигуру. Лучше всего подойдёт тонкая калька – достаточно тонкая, но в то же время прочная бумага.

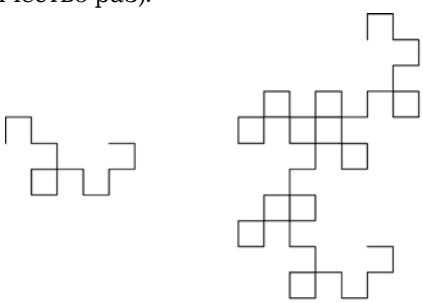
Итак, имеется лента длиной в 100 сантиметров. Один конец этой ленты необходимо закрепить (или держать в уме, что этот конец закреплённый; в действительности закреплять клеем или чем-то подобным не следует, поскольку впоследствии ленту придётся разворачивать). Второй конец ленты является свободным. Каждая итерация по построению Кривой Дракона заключается в простой процедуре. Лента складывается пополам – свободный конец совмещается с закреплённым. На первом шаге получается сложенная вдвое лента размером 50×1 см. Первоначальный свободный конец оказался совмещённым с закреплённым концом. Этот совмещённый конец ленты остаётся закреплённым, а свободным теперь считается место сгиба ленты.

Вторая итерация повторяет первую – свободный конец совмещается с закреплённым. Получается сложенная вчетверо лента размером 25× 1 см. Места сложения необходимо тщательно проглаживать, чтобы они были как можно более плоскими (именно тут нарастает неточность при дальнейших сложениях, поскольку на сгиб необходим материал, и на самом деле сложенная вчетверо лента имеет меньший размер, но пока эта неточность незаметна).

Описанную итерацию необходимо повторять столько, сколько это возможно. В идеале для достаточно тонкой бумаги можно получить сложен-

ную в шестьдесят четыре раза ленту размером 1,5625×1 см. Но до этого вряд ли дойдёт – помешает именно толщина бумаги. Пусть получится сложенная в шестнадцать раз лента размером 6,25×1 см. После этих сложений лента будет уже напоминать брусок. Но необходимо помнить, что это всего лишь грубая физическая модель математического процесса. Математическая лента не имеет толщины, её можно складывать до бесконечности.

Наконец, можно приступить к самому построению Кривой Дракона. Для этого необходимо развернуть получившийся брусок бумаги, озаботившись тем, чтобы каждый угол на сгибе равнялся 90°. Разворачивать, естественно, необходимо, поставив согнутую ленту «на попа», а не просто положив её. Это сделать не так просто, но при должном терпении лента развернётся в Кривую Дракона для определённого шага построения (в идеале сгибание ленты нулевой толщины производится бесконечное количество раз).



а *б*

Рис. 4

На рис. 4 показана Кривая Дракона при сгибании ленты в шестнадцать (рис. 4 *а*) и в шестьдесят четыре раза (рис. 4 *б*) соответственно.

Интерес представляет тот факт, что сколько бы ни сгибали ленту, при разворачивании её указанным образом она никогда не пересечётся сама с собой. Лента поразительным образом выстраивается в замысловатую кривую, несколько напоминающую рассмотренное ранее множество Мандельброта. Если рассмотреть математическое описание этой кривой, то её внешняя граница опять получится самоподобной – каждая мельчайшая часть напоминает всю кривую. Общий вид математической Кривой Дракона представлен на рис. 5.

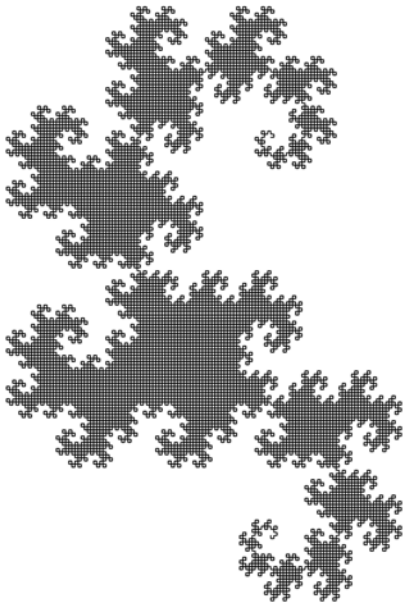


Рис. 5

Другим интересным фактом относительно этой кривой является то, что саму кривую можно построить из отрезка любой длины. Длина отрезка полностью определяет площадь кривой. Собственно, длина отрезка это и есть длина кривой, уложенной замысловатым образом. Чем длиннее кривая, тем большую площадь она захватит. Однако идеальный математический объект этой замысловатой формы может быть построен из отрезка любой длины. Обычно используется единичный отрезок.

2. Алгоритм построения

Рассмотренный в разделе 1 способ построения Кривой Дракона позволяет сформулировать алгоритм её построения для некоторого языка программирования. Собственно, при разворачивании ленты этот алгоритм должен был появиться в голове вдумчивого экспериментатора, поскольку каждый очередной разворот показывает, как именно строится кривая на произвольном шаге.

Алгоритм достаточно прост. Однако в целях упрощения далее будет предполагаться, что на каждом очередном шаге длина кривой увеличивается в два раза, а не остаётся постоянной, как в случае сгибания ленты. Это непринципиально, поскольку на общность рассуждений никак не влияет. Построенная для какого-то шага кривая будет иметь определённую длину, и эту длину можно считать длиной первоначальной ленты.

Итак, алгоритм построения Кривой Дракона идёт не от сгибания ленты, а от разворачивания единичного отрезка в кривую. Он состоит всего из двух шагов.

1. На нулевом шаге строится единичный отрезок с концами в координатах $(0;0)$ и $(1;0)$ соответственно. Этот отрезок называется Кривой Дракона нулевой итерации.

2. Полученная на предыдущем шаге Кривая Дракона n-ой итерации дублируется, и дубликат поворачивается вокруг начала координат $(0;0)$ против часовой стрелки на 90°, после чего сдвигается к концу оригинала кривой своим свободным концом. Свободный конец – это тот конец, который лежит вне начала координат. Он свободен потому, что свободно вращается. Первая точка Кривой Дракона лежит в начале координат, а потому при вращении не изменяет своего положения, поэтому она и называется закреплённой. Оригинал и дубликат сшиваются в одну кривую. Получается Кривая Дракона $(n+1)$-ой итерации. Данный процесс повторяется до тех пор, пока не достигнута требуемая точность (в идеале – до бесконечности).

Последовательное построение Кривой Дракона при помощи описанного алгоритма для четырёх итераций показано на рис. 6.

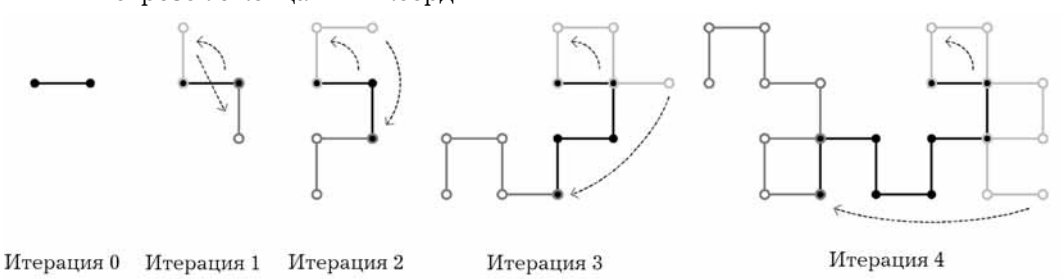


Рис. 6

3. Реализация на языке Haskell

Описанный в разделе 2 алгоритм на первый взгляд выглядит достаточно императивно. Шаг за шагом строится кривая, последовательно наращивая свою сложность. Однако на самом деле этот алгоритм достаточно легко преобразуется в рекурсивный декларативный алгоритм,

причём практически без каких-либо дополнений и изменений. Ниже в последующих подразделах будет написана небольшая программа на функциональном языке Haskell, которая вычисляет координаты последовательных точек Кривой Дракона.

3.1. Подготовительные описания геометрических образов

Прежде чем начать реализацию функции, возвращающей последовательность координат точек Кривой Дракона, необходимо провести некоторые подготовительные работы. Для построения кривой нужно определить специальные типы данных, которые будут использоваться для описания примитивных элементов, из которых будет состоять кривая, а также для описания самой кривой. Резонно все такие описания вынести в отдельный модуль, ответственный за геометрические описания:

```
module Geometry where
```

Первым делом необходимо подумать, что имеется общего во всех тех объектах, которые будут использоваться в алгоритме. Сам алгоритм подсказывает – все объекты можно поворачивать на 90° против часовой стрелки относительно начала координат и сдвигать на заданное расстояние. Сам алгоритм ничего не говорит о типе координат – целочисленные они или произвольные. Для описания таких ситуаций в языке Haskell имеется отличный формализм – классы с параметризацией типов. Поэтому для описания интерфейса к объектам, которые можно вращать и двигать, можно определить следующий класс:

```
class Figure f where
  shift    :: Num a => a -> a -> f a -> f a
  rotate90 :: Num a => f a -> f a
```

Здесь, в этом определении, сделано небольшое допущение. Предполагается, что произвольная фигура типа **f** параметризует собой некоторый специальный тип **a**, со значениями которого можно совершать арифметические операции (тем читателям, которым сложно понять вышеприведённые определения, необходимо обратиться к предыдущим статьям автора в журнале «Потенциал»). Параметризация типа **a** сделана именно для того, чтобы не ограничивать координаты объектов каким-либо специализированным типом, поскольку в изначальном алгоритме о типе ничего не сказано. То, что над координатами необходимо совершать арифметические операции, следует из того, что их необходимо сдвигать, то есть к координатам необходимо прибавлять значения сдвигов по двум осям.

Следующий шаг – определение типа данных для представления точки на плоскости. Для этих целей можно было бы воспользоваться обычной парой, однако этот путь не совсем правильный, поскольку не использует возможности языка Haskell по абстракции данных. И несмотря на то, что в описываемом достаточно простом примере возможности по абстракции будут использованы слабо, хороший тон в программировании на языке Haskell требует применения грамот-

ных идиом всегда, когда это возможно. Итак, алгебраический тип данных для представления одной точки можно определить так:

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

Это определение означает, что тип для представления точки **Point** параметризует некоторый тип **a**, который используется для представления типов двух координат – **x** и **y**.

Для того чтобы значениями только что созданного типа **Point** можно было оперировать в соответствии с описанным ранее алгоритмом, для этого типа необходимо определить экземпляр класса **Figure**. Это делается достаточно просто:

```
instance Figure Point where
  shift dx dy (Point x y) = Point (x + dx) (y + dy)
  rotate90 (Point x y)    = Point (-y) x
```

Теперь в этом определении видно, что метод **shift** используется для сдвига объекта на заданные значения по осям абсцисс и ординат, а метод **rotate90** всего лишь поворачивает заданный объект на 90° против часовой стрелки. Для реализации описанного алгоритма нет надобности создавать метод для вращения на произвольное количество градусов, поскольку это не потребуется. Если бы это было необходимо, то было бы невозможно использовать для представления координат произвольные типы, над значениями которых можно производить только арифметические операции, поскольку для вращения на произвольное число градусов необходимо использовать функции sin и cos, которые в языке Haskell определены над действительными значениями.

Дополнительно желательно определить для типа **Point** экземпляр класса **Show**, чтобы было возможно выводить результаты работы с точками на координатной плоскости на экран. Реализация экземпляра по умолчанию преобразует значения не очень приятным способом, поэтому в нижеследующем определении используется обычная математическая нотация для записи точек на плоскости.

```
instance Show a => Show (Point a) where
  show (Point x y) = "(" ++ show x ++ "," show y ++ ")"
```

Дополнительно необходимо определить утилитарную функцию, которая возвращала бы расстояние по двум осям между двумя точками. Эта функция потребуется для определения того, на какое расстояние сдвигать один из экземпляров кривой на очередном шаге построения. Функция **distance** выглядит следующим образом:

```
distance :: Num a => Point a -> Point a -> (a, a)
distance (Point x1 y1) (Point x2 y2) = (x1 - x2, y1 - y2)
```

Эта функция возвращает пару расстояний. Первое значение в паре символизирует расстояние между точками по оси абсцисс, второе – по оси ординат соответственно. Такое представление наиболее просто, а так как это будет использоваться только в одном месте, создавать новый тип для представления такой пары не требуется.

Сама кривая состоит из последовательного набора точек. Можно было бы использовать обычный список значений типа **Point**, однако не для этого вначале был определён класс **Figure**. Простой список использовать нельзя, поскольку по правилам языка Haskell его нельзя будет сделать экземпляром класса. Необходимо определить новый тип, который имел бы абсолютно такое же поведение, как и имеющийся тип [] . Для этих целей используется ключевое слово **newtype**:

```
newtype Curve a = Curve [Point a]
  deriving Show
```

Данное определение говорит транслятору языка Haskell, что необходимо создать тип, изоморфный списку значений типа **Point a**. Изоморфность обозначает в данном случае полную тождественность. Разница заключается лишь в том, что для нового типа **Curve** можно определить экземпляр класса **Figure**, что позволит оперировать значениями этого типа абсолютно так же, как простыми точками – вращать и сдвигать их. Делается это намного проще, чем для точек типа **Point**:

```
instance Figure Curve where
  shift dx dy (Curve ps) = Curve (map (shift dx dy) ps)
  rotate90 (Curve ps)    = Curve( map rotate90 ps)
```

Наконец, было бы очень неплохо определить ещё одну утилитарную функцию, которая возвращала бы список точек из значения **Curve**. Эта функция просто «разворачивает» тип **Curve**, возвращая его содержимое:

```
points :: Curve a -> [Point a]
points (Curve ps) = ps
```

Таким образом, весь модуль **Geometry** выглядит следующим образом:

```
module Geometry where

class Figure f where
  shift    :: Num a => a -> a-> f a -> f a
  rotate90 :: Num a => f a ->f a

data Point a
```

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

instance Figure Point where
  shift dx dy (Point x y) = Point (x + dx) (y + dy)
  rotate90 (Point x y)    = Point (-y) x

instance Show a => Show (Point a) where
  show (Point x y) = "(" ++ show x ++ "," ++ show y ++ ")"

distance :: Num a => Point a -> Point a -> (a, a)
distance (Point x1 y1) (Point x2 y2) = (x1 - x2, y1 - y2)

newtype Curve a = Curve [Point a]
  deriving Show

instance Figure Curve where
  shift dx dy (Curve ps) = Curve (map (shift dx dy) ps)
  rotate90 (Curve ps)    = Curve (map rotate90 ps)

points :: Curve a -> [Point a]
points (Curve ps) = ps
```

3.2. Построение Кривой Дракона

Теперь всё готово, чтобы реализовать описанный в разделе 2 алгоритм построения Кривой Дракона. Для её построения будет использоваться одна функция, полностью реализующая алгоритм так, как он написан:

```
makeDragonCurve :: Int -> Curve Int
makeDragonCurve 0 = Curve [Point 0 0, Point 1 0]
makeDragonCurve n = Curve (points c ++ (reverse $ init $ points
(shift dx dy rc)))
  where
    c        = makeDragonCurve (n - 1)
    rc       = rotate90 c
    (dx, dy) = distance (last $ points c) (last $ points rc)
```

Первая строка определения описывает тип значения, возвращаемого функцией **makeDragonCurve**. В этом описании имеется ограничение на тип координат, используемых для представления точек на координатной плоскости. В предложенной реализации функции этот тип – **Int**, то есть ограниченные целые числа. Однако, как говорилось ранее, в качестве типа координат может использоваться произвольный тип, для которого имеется экземпляр класса **Num.**

Вторая строка один в один соответствует пункту 1 алгоритма построения Кривой Дракона. Для нуле-

вой итерации в построении просто возвращается единичный отрезок $(0;0) - (1;0)$.

Соответственно, третья строка определения функции **makeDragonCurve** соответствует пункту 2 алгоритма. Как видно, эта строка более сложная, равно как и второй пункт алгоритма занимает намного больше строк в описании, чем первый пункт. Эта строка гласит, что Кривая Дракона на n-ом шаге построения состоит из точек (функция **points**) некоторой кривой **c**, к которым присоединены обращённое задом наперёд (функция **reverse** из стандартного модуля **Prelude**) начало (функция **init** из стандартного модуля **Prelude**) списка точек смещённой (функция **shift**) на значения **dx** и **dy** по осям абсцисс и ординат соответственно некоторой кривой **rc**.

Это определение необходимо рассмотреть подробнее. Некоторая кривая **c** является просто-напросто Кривой Дракона, построенной на предыдущем, $(n-1)$-ом шаге. Кривая **rc** является дубликатом кривой **c**, который повёрнут на 90° против часовой стрелки вокруг начала координат. Значения для сдвига **dx** и **dy** являются расстоянием между последними точками кривых **c** и **rc**. Последние точки в описании кривой при помощи типа **Curve** являются как раз «свободными» концами, поскольку именно они вращаются вокруг начала координат. Пока всё идёт в точности по алгоритму. Все эти дополнительные объекты получены в локальных определениях, перечисленных после ключевого слова **where**.

Стандартная функция **init** используется для того, чтобы отсечь последнюю точку в дубликате кривой, полученной на предыдущем шаге. Эта точка при помощи сдвига подсоединяется к последней точке оригинала кривой. Поскольку точки совмещаются, в описании кривой на n-ом шаге нет надобности дублировать запись о точках. Именно для этого производится отсечение.

Наконец, обращение повёрнутой и сдвинутой кривой при помощи стандартной функции **reverse** производится для того, чтобы перевернуть последовательность точек. Ведь именно переворот происходит в процедуре построения, описанной в разделе 1, когда свободный конец ленты совмещается с закреплённым. А из этой процедуры непосредственно получен алгоритм построения.

Для тех читателей, которые подзабыли, остаётся отметить, что стандартный оператор **($)** определён как обычное применение функции, но с самым низким приоритетом исполнения:

```
Infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Такое определение позволяет записывать последовательное применение функций к результатам выполнения других функций без лишних скобок.

В конечном итоге получается достаточно миловидный модуль:

```
module Dragon where

import Geometry
```

```
makeDragonCurve :: Int -> Curve Int
makeDragonCurve 0 = Curve [Point 0 0, Point 1 0]
makeDragonCurve n = Curve (points c ++ (reverse $ init $ points
(shift dx dy rc)))
  where
    c         = makeDragonCurve (n - 1)
    rc        = rotate90 c
    (dx, dy) = distance (last $ points c) (last $ points rc)
```

Запуск функции **makeDragonCurve** с некоторым числом, определяющим номер итерации в построении Кривой Дракона, вернёт список точек этой кривой на координатной плоскости для заданного шага. Эта функция является ленивой, поэтому в интерпретаторе языка Haskell она начнёт выводить промежуточные результаты сразу же после запуска.

Заключение

На этом примере в очередной раз показана мощь языка Haskell, в частности, функционального программирования, а именно, в общем, в решении расчётных математических задач. К сожалению, рамки простой научно-популярной статьи не позволяют провести дальнейшие исследования и показать возможности по графической визуализации полученных результатов. Для этого было бы необходимо предварительно описать графические библиотеки, что полностью заняло бы несколько номеров журнала подряд. Однако вдумчивому читателю можно порекомендовать создать для типа **Curve** экземпляр класса **Show**, который выводил бы заданную Кривую Дракона в виде псевдографики. Это не такая уж сложная задача.

Автор будет признателен любым отзывам и комментариям по этой статье, которые можно присылать по адресу электронной почты

darkus.14@gmail.com.

Также автор хотел бы дополнительно указать, что в 2007 году в издательстве «ДМК-Пресс» вышла книга «Функциональное программирование на языке Haskell», являющаяся первым рассмотрением этого языка программирования на русском языке. Выходные данные книги указаны в списке литературы под номером [2].

Список литературы

1. *Гарднер М.* А ну-ка, догадайся! Пер. с англ. Ю.А. Данилова. – М.: Мир, 1984. – 212 с., ил.

2. *Душкин Р.В.* Функциональное программирование на языке Haskell. – М.: ДМК-Пресс, 2007. – 608 с.

3. *Пенроуз Р.* Новый ум короля: О компьютерах, мышлении и законах физики. Пер. с англ./ Общ. ред. В. О. Малышенко. Предисловие Г.Г. Малинецкого. Изд. 2-е, испр. – М.: Едиториал УРСС, 2005. – 400 с. (Синергетика: от прошлого к будущему.)