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

Магические квадраты. Решение переборных задач на языке Haskell

В данной статье рассматриваются типовые способы решения задач на перебор, которые часто попадаются в рамках функционального программирования и программирования вообще. В качестве примера для рассмотрения предлагаемых методик предлагается задача получения списка магических квадратов для заданного размера квадрата. Рассмотрение, как обычно, производится на функциональном языке программирования Haskell.

Введение

Большая Советская Энциклопедия даёт сухое и математически чёткое определение магического квадрата: «Магический квадрат – квадрат, разделённый на равное число n столбцов и строк, со вписанными в полученные клетки первыми n^2 натуральными числами, которые дают в сумме по каждому столбцу, каждой строке и двум большим диагоналям одно и то же число (равное, как легко доказать, $\frac{n(n^2+1)}{2}$). Доказано, что магический квадрат можно построить для любого n, начиная с $n=3$. Существуют магические квадраты, удовлетворяющие ряду дополнительных условий, например магический квадрат с 64 клетками, который можно разбить на 4 меньших, содержащих по 16 клеток квадрата, причём в каждом из них сумма чисел любой строки, столбца или большой диагонали одна и та же (130). В Индии и некоторых других странах магические квадраты употребляли в качестве талисманов. Составление магических квадратов – классический образец математических развлечений и головоломок».

Но строгая наука, как обычно, суха. На самом же деле магические квадраты с древнейших времён используются для различных целей, связанных с волшебством и колдов-

ством – иначе откуда у них такое название? И, действительно, магические квадраты – это символы числовой гармонии, которые выражают космический принцип миропорядка. Каждый магический квадрат иллюстрирует законы онтологической симметрии, тем самым через него постигается присутствие рационального начала в мироздании.

В магической практике использовались квадраты, предназначенные для обретения сверхъестественных способностей – полётов в облике птиц, понимания языка животных, обретения невидимого для глаз состояния, нахождения кладов, раскрытия секретов прошлого, проникновения в будущее и т. п. Имелись квадраты-заклинания, направленные против врагов. Даже переписывание их в тетрадь представлялось весьма опасным, и маг в таких случаях воспроизводил их с умышленной ошибкой.

Разные магические квадраты приписывались разным силам. Так, к примеру, числовой квадрат с трёхклеточными сторонами соответствует планете Сатурн. Он демонстрирует симметрию между чётными и нечётными числами. Квадрат с четырёхклеточными сторонами относится к Юпитеру. Квадрат с пятиклеточными сторонами является архетипом Марса, считалось, что заклинания над ним развивают воинственность. Квадрат с шестиклеточными сторонами символизирует Солнце. Квадрат с семиклеточными сторонами соотносится с Венерой и так же, как трёхклеточный, демонстрирует симметрию чётных и нечётных цифр.

Как видно, магические квадраты в древние времена имели чрезвычайно серьёзное прикладное значение. А посему можно предположить, что знание о магических квадратах весьма важно, поэтому их изучение может способствовать развитию магической силы. Ну а людям, настроенным на абсолютное рациональное мышление, понравится решать задачу на поиск всех магических квадратов заданного размера в автоматизированном (или даже автоматическом) режиме, т. к. мы получили в своё распоряжение такие прекрасные инструменты для этих целей, как электронные вычислительные машины и функциональное программирование.

Принимая во внимание оба указанных аспекта, далее в этой статье рассматриваются переборные задачи сложного характера с большим объёмом перебора на примере поиска магических квадратов. Рассмотрение ведётся при помощи функционального языка программирования Haskell, поэтому для понимания читатель должен быть знаком с синтаксисом этого языка в объёме опубликованных в журнале «Потенциал» статей о нём.

1. Простейший вариант перебора

Для того, чтобы начать двигаться в каком-то направлении при решении поставленной задачи, необходимо рассмотреть какие-нибудь простейшие частные случаи, чтобы понять, что должно происходить при поиске магических квадратов. Для этих целей подойдёт магический квадрат размера 3×3, т. к. такой квадрат один (с точностью до поворотов и отражений), а потому осуществить его поиск будет легко. Кроме того, количество вариантов для перебора не так велико.

Что есть магический квадрат размера 3×3 с математической точки зрения? Это матрица указанного размера, в которой размещены числа от 1 до 9 таким образом, что суммы чисел в горизонтальных рядах, вертикальных столбцах и двух диагоналях равны между собой. Это наталкивает на мысль о том, как должна выглядеть функция для перебора в первом приближении:

```
(//) :: Eq a => [a] -> a -> [a]
[] // y      = []
(x:xs) // y = if (x == y)
               then xs
               else x:(xs // y)
(///) :: Eq a => [a] -> [a] ->
[a]
s /// [] = s
s /// (x:xs) = (s // x) /// xs

ms_3 :: ms [[Integer]]
ms_3 = [[x1, x2, x3,
         y1, y2, y3,
         z1, z2, z3] |
    x1 <- [1..9],
    x2 <- [1..9] /// [x1],
    x3 <- [1..9] /// [x1, x2],
    y1 <- [1..9] /// [x1, x2,
x3],
    y2 <- [1..9] /// [x1, x2, x3,
                      y1],
    y3 <- [1..9] /// [x1, x2, x3,
                      y1, y2],
    z1 <- [1..9] /// [x1, x2, x3,
                      y1, y2, y3],
    z2 <- [1..9] /// [x1, x2, x3,
                      y1, y2, y3,
                      z1],
    z3 <- [1..9] /// [x1, x2, x3,
                      y1, y2, y3,
                      z1, z2],
    x1 + x2 + x3 == y1 + y2 + y3,
    x1 + x2 + x3 == z1 + z2 + z3,
    x1 + x2 + x3 == x1 + y1 + z1,
    x1 + x2 + x3 == x2 + y2 + z2,
    x1 + x2 + x3 == x3 + y3 + z3,
    x1 + x2 + x3 == x1 + y2 + z3,
    x1 + x2 + x3 == x3 + y2 + z1]
```

Вспомогательные операции (**//**) и (**///**) используются для получения списка с исключённым из него элементом и подсписком соответственно.

Как видно, эта функция достаточно громоздка. Однако она полностью воспроизводит математическое определение, данное магическому квадрату размера 3×3. Она использует представление матрицы в виде списка из девяти элементов (что не так принципиально), а сами элементы выбираются из набора чисел от 1 до 9, при этом на каждом следующем шаге из этого набора удаляются предыдущие выбранные элементы. После чего производится проверка.

Запуск этой функции на исполнение в интерпретаторе HUGS 98 даст неутешительные результаты. Весь поиск займёт около минуты времени, но при этом будет произведено 150 миллионов редукций, будет задействовано 267 миллионов ячеек памяти, а сборщик мусора запустится 295 раз. И это всё только ради того, чтобы получить один единственный магический квадрат (в результате, конечно, будет выведено 8 изоморфных друг другу квадратов) следующего вида:

2	7	6
9	5	1
4	3	8

Это совершенно неприемлемый результат, т. к. это не позволит вычислить ни одного магического квадрата размером 4×4 и выше за доступное время. Поэтому необходима какая-то оптимизация. Первое, что приходит в голову после изучения функции генерации – это использование магической суммы для сравнения в выражениях охраны. Ведь формула её вычисления известна. Ну а второе – использование иного порядка перебора и поиска, который позволит отсечь заведомо тупиковые ветви дерева перебора. Например, новая, более оптимизированная функция может выглядеть следующим образом:

```
magicSum :: Fractional a => a -> a
magicSum n = n * (n^2 + 1) / 2

ms_3' :: [[Double]]
ms_3' = [[x1, x2, x3
          y1, y2, y3,
          z1, z2, z3] |
    x1 <- [1..9]
    x2 <- [1..9] /// [x1],
    x3 <- [1..9] /// [x1, x2],
    x1 + x2 + x3 == ms,
    y1 <- [1..9] /// [x1, x2, x3],
    z1 <- [1..9] /// [x1, x2, x3,
                      y1],
    x1 + y1 + z1 == ms,
    y2 <- [1..9] /// [x1, x2, x3,
                      y1, z1],
    x3 + y2 + z1 == ms,
    y3 <- [1..9] /// [x1, x2, x3,
                      y1, z1, y2],
    y1 + y2 + y3 == ms,
    z2 <- [1..9] /// [x1, x2, x3,
                      y1, z1, y2,
                      y3],
    x2 + y2 + z2 == ms,
    z3 <- [1..9] /// [x1, x2, x3,
                      y1, z1, y2,
                      y3, z2],
    z1 + z2 + z3 == ms,
    x3 + y3 + z3 == ms,
    x1 + y2 + z3 == ms]
  where ms = magicSum 3
```

Выполнение функции **ms_3'** происходит уже практически мгновенно. При этом требуется 208 тысяч редукций и 365 тысяч ячеек памяти. Это выигрыш в эффективности почти в 723 раза по сравнению с предыдущим определением. Можно ещё поиграться с дальнейшей оптимизацией алгоритма перебора, например, удаляя из набора **[1..9]** не подсписки, а одиночные элементы, полученные на предыдущем шаге перебора, определяя локальные переменные для этих целей, но смысла в этом особенного нет, т. к. в подобных определениях функций имеются две проблемы – одна не очень серьёзная, а другая – весьма серьёзная.

Первая заключается в том, что при использовании функции **magicSum**, которая вычисляет сумму магического квадрата, происходит выход за пределы множества натуральных чисел. Это нехорошо, ибо использование операций над числами с плавающей точкой, когда такие числа даже не используются, необоснованно. Эта проблема решается просто заменой определения функции (самостоятельно можно доказать, что это определение тождественно по результату предыдущему, т. е. имеет место экстенсиональное тождество определений функций):

```
magicSum :: Integral a => a -> a
magicSum n = sum [1..(n^2)] `div` n
```

Использование такого определения ещё больше снижает затраты вычислительных ресурсов. Вместо вычисления суммы элементов списка можно также пользоваться оператором целочисленного деления **div** в

предыдущем определении. Эффективность примерно одинаковая. Однако новое определение функции **magicSum** не устраняет другую проблему, более серьёзную. Проблема сия связана с весьма частным видом функций **ms_3** и **ms_3'** – они предназначены только для вычисления магических квадратов 3×3. А что, если потребуется вычислить магические квадраты произвольного размера $n\times n$? Эти функции, естественно, не подойдут. Само собой, что необходимо создать универсальное определение функции для любого заданного n. Но подойти к этой задаче не так просто...

2. Перебор с использованием перестановок

При рассмотрении задачи для произвольного размера магического квадрата $n\times n$ возникает проблема представления такого квадрата в памяти. Можно отметить, что предложенный в предыдущем разделе способ представления в виде списка имеет свою привлекательность как наиболее простой способ. Им можно воспользоваться и в общем случае.

Однако всё ещё стоит вопрос о способах получения различных комбинаций, которые будут рассматриваться в качестве магических квадратов. В этом деле поможет такая отрасль математики, как комбинаторика, т. к. она имеет достаточные механизмы для описания различных способов перебора. К ней и необходимо обратиться.

Для решения задачи о построении списка магических квадратов размерности $n\times n$ необходимо получать всевозможные комбинации чисел от 1 до n^2, выстроенные в список, после чего каждую комбинацию проверять на «магичность». Это – простейшее понимание обобщённого переборного алгоритма. А для получения списка всех возможных комбинаций заданных чисел как раз и можно воспользоваться одним из понятий комбинаторики. Это понятие – *перестановка.*

Перестановками из n элементов называются комбинации таких элементов, каждая из которых содержит все n элементов, отличающихся поэтому друг от друга только порядком расположения элементов. Например, из 3-ёх элементов (a, b, c) можно образовать следующие перестановки: *abc, bac, cab, acb, bca, cba.* Число всех возможных перестановок, которые можно образовать из n элементов, обозначается символом P_n и вычисляется по формуле:

$$P_n = n! \qquad (1)$$

Таким образом, необходимо создать функции для генерации всех возможных перестановок для элементов списка от 1 до n^2 и последующей проверки сгенерированных комбинаций. Первую функцию написать достаточно просто:

```
permutations :: Eq a => [a] -> [[a]]
permutations [] = [[]]
permutations l = [x:ps |
  x <- l,
  ps <- (permutations (l // x))]
```

Список всех перестановок для заданного списка элементов вычисляется. Первый *клоз* в этом определении является выходом из рекурсии, определяя результат функции на пустом списке. Второй клоз определения – самый главный. Именно он определяет способ вычисления списка перестановок элементов от 1 до n^2. Смысл можно понять, прочитав определение этого клоза примерно так:

Перестановки элементов списка l вычисляются как список списков, где голова каждого списка выбирается из списка l, а хвост – из списка перестановок, полученных для исходного списка l без выбранного на предыдущем шаге элемента x.

Имея в своём арсенале эту функцию, можно получить список всех возможных перестановок чисел от 1 до n^2. Остаётся проверить такие комбинации на «магичность». А для этого необходима функция (или набор функций), которая возвратит булевское значение в зависимости от того, является ли переданная ей на вход комбинация магическим квадратом. При представлении магических квадратов в виде списка такая функция может выглядеть так:

```
isMagic :: Int -> [Int] -> Bool
isMagic n ms = if (n^2 /= length ms)
                 then False
                 else (testH s n ms) &&
                      (testV s n ms) &&
                      (testD s n ms)
  Where s = magicSum n
```

Эта функция для заданного размера n и заданной комбинации **ms** возвращает значение **True**, если комбинация ms является магическим квадратом $n \times n$, и значение **False** в противном случае. В её определении используются вспомогательные функции **testH**, **testV** и **testD**, которые используются для проверки суммы чисел горизонталей, вертикалей и диагоналей магического квадрата **ms** на равенство магической сумме. Эти функции определить не так сложно, принимая во внимание способ представления магического квадрата.

```
testH :: Int -> [Int] -> Bool
testH _ _ [] = True
testH s n ms = (sum (take n ms) == s) &&
               (testH s n (drop n ms))

testV :: Int -> Int -> [Int] -> Bool
testV s n ms = testV' s n n ms
  where testV' _ 0 _ _  = True
```

```
        testV' s a n ms =
          (sum [ms !! ((a - 1) + (j * n)) |
            j <- [0..(n - 1)]] == s) &&
          (testV' s (a - 1) n ms)

testD :: Int -> Int -> [Int] -> Bool
testD s n ms =
  (sum [ms !! (i * (n + 1)) |
    i <- [0..(n - 1)]] == s) &&
  (sum [ms !! ((i + 1) * (n - 1)) |
    i <- [0..(n - 1)]] == s)
```

Функция **testH** устроена просто. Она последовательно сверяет сумму каждых n элементов переданного ей для проверки списка с магическим числом, которое также передаётся в качестве входного параметра (это сделано в качестве оптимизации). Это делается выборкой первых n элементов (стандартная функция **take**), проверкой их суммы и передачей оставшихся элементов списка (стандартная функция **drop**) в эту же самую функцию **testH**.

Функция **testV** немного сложнее. Главный смысл её работы заключается в получении списка элементов исходного списка, расположенных в одном столбце. Для этого используется вспомогательная функция **testV'**, у которой во втором клозе первым операндом конъюнкции (**&&**) как раз и находится формула для вычисления позиций элементов в списке, которые в квадрате находятся в одном столбце (для нумерации столбца используется параметр **a**). Читателю рекомендуется самостоятельно проверить эту формулу.

Наконец, функция **testD** просто проверяет две диагонали квадрата. Опять же, вдумчивый читатель найдёт проверку формул получения элементов из диагоналей квадрата, который представляется списком, интересной задачей.

Остаётся реализовать главную функцию для получения списка магических квадратов размера $n \times n$. После проведённых подготовительных работ это не представляется сложным делом. Её определение выглядит просто:

```
getMagicSquares :: Int -> [[Int]]
getMagicSquares n = [ms |
  ms <- permutations [1..(n^2)],
  isMagic n ms]
```

Как видно, здесь устроен перебор всех возможных перестановок с последующей проверкой их на «магичность». В очередной раз можно удивиться выразительности языка Haskell. Единственное, что необходимо объяснить, – это использование ограничения на тип функции: **Int -> [[Int]]**. Использование типа **Int** (ограниченные целые числа) вместо **Integer** (целые числа неограниченного размера) позволяет весьма серьёзно снизить ресурсные затраты на проведение вычислений.

Замеры производительности работы этой функции имеет смысл делать для $n = 3$, т. к. для больших n функция работает слишком медленно. Её запуск с параметром 3 даёт такие результаты: 140 миллионов редукций, 205 миллионов ячеек памяти и 230 запусков процесса сборки мусора. Как видно, эти значения не сильно отличаются от значений функции **ms_3**, поэтому можно предположить, что для $n > 3$ функция будет работать очень медленно. Поэтому имеет

смысл задуматься об оптимизации, ибо несмотря на то, что функция **getMagicSquares** стала универсальной с точки зрения размера магических квадратов, но её производительность оставляет желать лучшего.

3. Перебор с использованием размещений

При рассмотрении предыдущего алгоритма можно было увидеть одну довольно интересную его особенность, которая и сводит на нет все возможности построения за реальное время магических квадратов размера 4×4. Особенность сия заключается в том, что во время осуществления перебора совершается полная выборка комбинации, которая только после этого проверяется на «магичность», хотя эту проверку можно сделать до получения полного набора чисел (как это сделано в функции **ms_3'**).

Например, в случае размера 4×4 нет никакой надобности рассматривать всю комбинацию, если сумма первых четырёх чисел не равна 34. Поэтому для магического квадрата размера $n\times n$ необходимо ограничивать перебор при помощи проверки суммы каждых n чисел в получаемой комбинации. Это поможет сделать понятие *размещения*, которое также определяется в комбинаторике.

Размещениями из n элементов по k называются комбинации, которые можно образовать из заданных n элементов, собирая в каждую комбинацию по k элементов, при этом сами комбинации могут отличаться друг от друга как самими элементами, так и порядком их взаимного расположения. Например, из 3 элементов (a, b, c) по 2 можно образовать следующие размещения: ab, ac, ba, bc, ca, cb. Число всех возможных размещений, которые можно образовать из n элементов по k, обозначается символом A_n^k и вычисляется по формуле:

$$A_n^k = \frac{n!}{(n-k)!}. \qquad (2)$$

Итак, в случае магического квадрата размера $n\times n$ необходимо генерировать n размещений из n^2 по n и каждое из них проверять на соответствие магической сумме. Для этого нужна функция генерации всех размещений элементов из заданного списка. Определить её несложно:

```
arrangements :: (Num a, Eq b) =>
                  a -> [b] -> [[b]]
arrangements 0 _ = [[]]
arrangements k l = [x:as |
  x <- l,
  as <- (arrangements (k - 1) (l // x))]
```

Смысл её вполне понятен – первый аргумент определяет количество элементов в размещении, а второй – список элементов. В результате будет выдан список списков, каждым элементов которого является список, являющийся одним из размещений по k элементов из списка **l**. Однако, как видно, эта функция не генерирует магические квадраты заданного размера, т. к. она должна лишь использоваться для генерации таких квадратов. Поэтому необходима дополнительная функция, которая возвращала бы список списков, каждым элементом которого было бы представление одного варианта «кандидата» в магические-квадраты, как это делает функция

permutations. Реализовать такую функцию можно по аналогии:

```
constructSquares :: Integral a =>
                    a -> [a] -> [[a]]
constructSquares _ [] = [[]]
constructSquares n l  = [a ++ as |
  a <- (arrangements n l),
  sum a == s,
  as <- (constructSquares n (l /// a))]
    where s = magicSum n
```

Как видно, определение этой функции практически полностью повторяет определение функции **permutations** (конечно, в рамках используемых понятий), за исключением нового условия охраны, которое отсекает перебор в случаях, если очередной набор из n чисел не проходит первоначальный тест на «магичность». В этом выражении охраны использовано локальное определение **s** для того, чтобы вычислить магическую сумму для заданного n только один раз, а не для каждого **a** в отдельности (один из способов оптимизации вычислений – читатель самостоятельно может сравнить показатели эффективности в случаях использования *локальной переменной* и в случае её отсутствия).

Остаётся создать новое определение функции для получения списка магических квадратов заданного размера. Это уже не представляет и вовсе никакой сложности:

```
getMagicSquares' :: Int -> [[Int]]
getMagicSquares' n = [ms |
  ms <- constructSquares n [1..(n^2)],
  isMagic n ms
```

Однако как оценить возможности нового определения? Сможет ли оно помочь получить список всех магических квадратов с размером, хотя бы, 4×4? Для получения ответа на этот вопрос необходимо провести небольшое исследование, которое поможет оценить время выполнения функции перед её непосредственным запуском. Для этого необходимо собрать фактические данные, которые обозримы за малое время. Это – замеры вычислительных параметров для функций **getMagicSquares** и **getMagicSquares'** на аргументах 1, 2, и 3. Все эти замеры были предварительно сделаны и сведены в одну таблицу:

Таблица 1

n	getMagicSquares		getMagicSquares'	
1	801	1 352	975	1 488
2	13 742	19 850	8 983	13 152
3	$2{,}42\cdot10^{8}$	$3{,}44\cdot10^{8}$	$4{,}78\cdot10^{6}$	$6{,}69\cdot10^{6}$
4	$4{,}36\cdot10^{15}$	$7{,}02\cdot10^{15}$	$1{,}47\cdot10^{11}$	$1{,}96\cdot10^{11}$

В первом столбце для каждой функции показано количество проведённых во время вычислений редукций, вторая – количество использованных ячеек памяти соответственно. Четвёртая строка таблицы показывает вычисленные (прогнозируемые) значения. Как были получены эти значения? При помощи несложных вычислений.

Первоначально было замечено, что порядок (степень числа 10) перечисленных значений изменяется в некоторой последовательности. Это можно видеть в представленной таблице – для первой функции порядок изменяется как 2, 4, 8 (очень похоже на степени числа 2); для второй функции последовательность такая: 2, 4, 6 (вроде просто чётные числа, но это может быть и неверным предположением). Соответственно, из предположения о том, что отношение порядков значений не изменяется при увеличении n, сделано вычисление этих параметров для $n=4$. Например, значение параметра в первой ячейке четвёртой строки вычислено по следующей формуле:

$$R_4 = R_3 \cdot \frac{\left(\frac{R_3}{R_2}\right)^2}{\left(\frac{R_2}{R_1}\right)} = \frac{R_3^3 \cdot R_1}{R_2^3}. \qquad (3)$$

Аналогично вычисляются и значения других ячеек четвёртой строки (при вычислении по таблице значения могут немного отличаться от тех, что представлены, т. к. при вычислении во время эксперимента использовались более точные значения, чем те, которые внесены в таблицу).

Для проверки полученных результатов можно построить график (см. рис. 1), при помощи которого можно оценить правильность спрогнозированных результатов. Конечно, из-за гигантских значений необходимо использовать *логарифмическую шкалу* по оси Y, которая характеризуется тем, что равные расстояния по ней соответствуют равным отношениям величин. Использование такой шкалы следует ещё и из того, что именно порядок величин используется при прогнозировании.

Рис. 1. График изменения количества редукций для функций **getMagicSquares** и **getMagicSquares'**

Как видно на графике, кривые достаточно плавные, а потому новые точки (спрогнозированные) вполне вписываются в изучаемую канву. Следовательно, на основании этих значений можно спрогнозировать время, которое будет затрачено на перебор и поиск всех магических квадратов размера 4×4. Время можно вычислить по следующей формуле (учитывая, что время вычислений зависит линейно от количества редукций):

$$T_4 = T_3 \cdot \frac{R_4}{R_3}.$$

Для вычисления всех магических квадратов размера 3×3 при помощи первой функции было затрачено 28,98 секунд, при помощи второй – 1,13 секунд (эти времена замерены уже после компиляции функций в компиляторе GHC в целях ускорения процесса, поэтому они отличаются от временных значений, упомянутых ранее). Соответственно, для функции **getMagicSquares** время выполнения для $n=4$ будет равно 522 миллиона секунд (16 с половиной лет), а для функции **getMagicSquares'** – 34,7 тысяч секунд (9 с половиной часов).

После получения такого оптимистического результата было решено проверить теоретические выкладки на практике. Для этого на ночь была запущена функция **getMagicSquares'** с параметром 4. И результат был получен: через 10 часов было получено 7 040 магических квадратов размера 4×4, первый и последний из которых представлены ниже:

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

16	15	2	1
5	3	14	12
4	10	7	13
9	6	11	8

Как видно, эти два квадрата являются зеркальным отражением друг друга. Действительно, алгоритм получает все магические квадраты, в том числе отражённые и повёрнутые. Количество неизоморфных друг другу магических квадратов, таким образом, необходимо получать, деля общее их количество на 8. Для размера 4×4 это количество будет равняться 880.

И действительно, в 1930 году немецкий математик Ф. Фиттинг теоретически рассчитал количество магических квадратов размером 4×4, которое по его расчётам равно как раз 880. А ещё ранее, в XVII веке французский математик: В. Френикль построил (вручную) все возможные магические квадраты размера 4×4.

4. Дальнейшая универсализация алгоритма

Полученные функцией **getMagicSquares'** результаты впечатляют. Однако у неё всё ещё остаются недоработки, которые желательно устранить. Недоработки эти касаются уже больше интерфейсного плана, нежели оптимизации переборного процесса. Например, вывод магических квадратов на экран оставляет желать лучшего – они представляются в виде списка, а потому так и выводятся. Интерпретировать результаты очень сложно, читать полученные записи неудобно. С другой стороны, представленный алгоритм работает только с числовыми магическими квадратами, хотя в целях развлечения можно придумать и нечи-

словые квадраты, а также прочие магические фигуры. Из нечисловых магических квадратов, к примеру, можно привести такой, который до сих пор используется разными людьми в «магических» целях:

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

Вот и хотелось бы сделать некоторый интерфейсный класс, который описывал бы методы для генерации различных комбинаций и дальнейшей проверки их на «магичность». Это позволит определять дополнительные типы данных и реализовывать для них методы генерации магических фигур.

Описать сам интерфейсный класс достаточно просто (необходимость введения изоморфного списка типа **MagicList** для представления списка магических фигур будет объяснена позже):

```
newtype MagicList f = ML [f]

class Magic f where
  getMagicFigures :: Int -> MagicList f
  isMagic         :: f -> Bool
```

Как видно, этот класс описывает два метода: для генерации списка магических фигур (**getMagicFigures**) и для проверки некой комбинации на «магичность» (**isMagic**). Теперь остаётся определить новые *алгебраические типы данных* (АТД), при помощи которых представлять в памяти сами магические фигуры. Список в этом деле также может помочь, но можно сделать и более интересные способы представления, которые помогут, в том числе, оптимизировать вычислительные процессы.

Собственно, изученные уже числовые магические квадраты можно представлять списком, но добавить к нему дополнительное поле в виде размера квадрата, чтобы не заниматься постоянной передачей этого параметра из функции в функцию. Это делается при помощи такого определения:

```
data Square a =
  Square
  {
    dimension :: Int,
    values    :: [a]
  }
```

Тут видно, что данный АТД можно использовать для представления любых квадратов. Чтобы его использовать, необходимо описать экземпляр класса **Magic** для этого АТД. Т. к. пока исследуются только числовые магические квадраты, да и невозможно сделать общее правило для проверки «магичности» числовых и нечисловых квадратов, экземпляр класса будет несколько суженным:

```
instance Magic (Square Int) where
  getMagicFigures d =
    ML [sq | sq <- constructSquares d [1..(d^2)],
             isMagic sq]

  isMagic (Square d v) =
```

```
    if (d^2 /= length v)
      then False
      else (testH ms d v) &&
           (testV ms d v) &&
           (testD ms d v)
    where ms = magicSum d
```

Как видно, здесь использованы старые вспомогательные функции для генерации комбинаций, а также для проверки «магичности» по горизонталям, вертикалям и диагоналям. В этом нет ничего зазорного.

Но больше всего вызывает интерес определение для этого АТД экземпляра стандартного класса **Show**, методы которого используются для вывода значений типов на экран (и в файлы). Хотелось бы, чтобы на экране квадраты выглядели именно квадратами, а не невзрачными списками чисел. Да и список квадратов тоже надо бы приукрасить, а именно выводить квадраты один за другим с отделением друг от друга пустой строкой. Для этого и необходим изоморфный тип **MagicList** – его также необходимо сделать экземпляром класса **Show** (в случае, если бы не было определения такого типа, невозможно было бы переопределить способ вывода списка магических фигур на экран). Такие экземпляры определяются следующим образом:

```
instance Show a =>
         Show (Square a) where
  show (Square d v) =
    if (null v)
      then "+" ++ (showLine d)
      else "+" ++ (showLine d) ++
           "\n|" ++ showRow (take d v) ++
           "\n" ++ show (Square d (drop d v))
    where showRow []     = ""
          showRow (x:xs) = (showCell x) ++
                           (showRow xs)

          showCell x = " " ++ (replicate
                               (nLength d2 -
                                nLength x)
                               ' ') ++
                       (show x) ++ " |"

          showLine 0 = ""
          showLine i = "-" ++ (replicate
                               (nLength d2)
                               '-') ++
                       "-+" ++
                       (showLine (i - 1))

          nLength x = length $ show x

          d2 = d^2

instance Show a =>
         Show (MagicList (Square a)) where
```

```
show (ML [])     = ""
show (ML (x:xs)) = show x ++
                   "\n\n" ++ show (ML xs)
```

Не стоит пугаться столь непростых, на первый взгляд, определений. Все эти функции направлены лишь на то, чтобы выводить числовые магические квадраты в читабельном виде. Например, один из магических квадратов размера 5×5, полученный в процессе работы над статьёй, был выведен этими функциями в таком виде:

```
+----+----+----+----+----+
|  3 | 16 |  9 | 22 | 15 |
+----+----+----+----+----+
| 20 |  8 | 21 | 14 |  2 |
+----+----+----+----+----+
|  7 | 25 | 13 |  1 | 19 |
+----+----+----+----+----+
| 24 | 12 |  5 | 18 |  6 |
+----+----+----+----+----+
| 11 |  4 | 17 | 10 | 23 |
+----+----+----+----+----+
```

Остаётся отметить, что вызывать новую функцию **getMagicFigures** так просто уже не получится, т. к. транслятор языка Haskell не сможет автоматически определить тип возвращаемого результата, чтобы выбрать требуемый экземпляр класса **Magic**. Поэтому тип необходимо в данном случае указывать явно:

```
main :: Int -> IO ()
main d =
  print (getMagicFigures d
         ::  MagicList  (Square
Int))
```

Это – небольшая плата за большую универсальность в представлении.

Заключение

Дальнейшая работа над усовершенствованием переборного алгоритма может принести ощутимые результаты. Ведь в этой небольшой статье намечен только путь, по которому можно двигаться. Здесь показаны некоторые из принципов оптимизации программ на функциональных языках программирования. Но, в случае необходимости, далее надо двигаться самостоятельно.

Например, можно ещё больше поработать над представлением магических квадратов, чтобы сделать функции **test*** более простыми. Этого можно добиться при помощи явного хранения позиции числа в самом магическом квадрате. Это увеличит количество занимаемой памяти, но достаточно сильно ускорит процессы проверки на «магичность» горизонталей, вертикалей и диагоналей, т. к. отпадёт необходимость перебирать списки при помощи оператора (**!!**).

Также необходимо поработать над самим алгоритмом перебора, т. к. он ещё весьма далёк от совершенства (например, оценка времени, необходимого для вычисления всех магических квадратов размера 5×5, показало, что для вычисления их при помощи функции **getMagicSquares** потребуется почти 17 тысячелетий, а при помощи функции **getMagicSquares'** – 24 дня). В этом деле поможет также внедрение в определения функций возможности осуществлять поиск магических квадратов, начиная с определённой комбинации. Это позволит распределить усилия по поиску магических фигур как во времени, так и в пространстве

(задействовав, к примеру, несколько компьютеров, которые будут работать в ночное время).

Так что возможностей для самостоятельного творчества в этом направлении – тьма. Дерзайте!

Автор благодарит Антонюка Д. А., вместе с которым все описанные в этой статье программные сущности были обсуждены, разработаны и протестированы, а также произведены замеры временны́х и прочих параметров.

Все перечисленные в этой статье определения классов, 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/** (внимание, доступна новая версия интерпретатора, датированная сентябрём 2006 года), а также в компиляторе GHC.

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.

Юмор Юмор Юмор Юмор Юмор Юмор

- Известно, что в году приблизительно пи на десять в седьмой секунд. И это просто объяснить с физической точки зрения.

  В самом деле: пи – потому что орбита у Земли круглая, в седьмой – потому что в неделе семь дней. Ну а приблизительно, потому что орбита всё-таки не совсем круглая, а эллиптическая.

- Проекция на плоскость общественного мнения вектора достижений.

  Результат зависит от склонности общественного мнения к вашим достижениям.

- Однажды Ньютону гости пожаловались, что калитка в его сад туго открывается, и попросили сделать другую, получше.

  – Я не знаю, куда лучше, – ответил физик. – И так каждый входящий наливает в бак для дома не меньше галлона воды.