Душкин Роман Викторович
Автор нескольких книг по функциональному программированию.

Функциональный подход в программировании

В статье в сжатой форме рассказывается про функциональный подход к описанию вычислительных процессов (и в общем, к описанию произвольных процессов в реальном мире), а также про применение этого подхода в информатике в функциональной парадигме программирования. Примеры реализации функций даются на языке программирования Haskell.

Введение

Имеется несколько различающихся подходов к организации вычислительных процессов, наиболее известными и в какой-то степени «конкурирующими» являются императивный (процедурный) и функциональный стили. Эти стили вычислений были известны в далёком прошлом, и сейчас уже невозможно узнать, какой подход был разработан первым.

Последовательность шагов вычислений, как особенность процедурного стиля, можно рассматривать в качестве естественного способа выражения человеческой деятельности при её планировании. Это связано с тем, что человеку приходится жить в мире, где неумолимый бег времени и ограниченность ресурсов каждого отдельного индивидуума заставляли людей планировать по шагам свою дальнейшую жизнедеятельность.

Вместе с тем нельзя сказать, что функциональный стиль вычислений был неизвестен человеку, а появился только с возникновением теории вычислений в том или ином виде в конце XIX – начале XX века. Декомпозиция задачи на подзадачи и выражение ещё нерешённых проблем через уже решённые – эти методики также были известны с давних времён, а именно они составляют суть функционального подхода. Именно этот подход и является предметом рассмотрения настоящей статьи, а объясняться его положения будут при помощи функционального языка Haskell (для изучения языка можно воспользоваться книгой [4]).

Несмотря на то, что фактически функциональный подход к вычисле-

ниям был известен с давних времён, его теоретические основы начали разрабатываться вместе с началом работ над вычислительными машинами – сначала механическими, а потом уже и электронными. С развитием традиционной логики и обобщением множества сходных идей под сводом кибернетики появилось понимание того, что функция является прекрасным математическим формализмом для описания реализуемых в физическом мире устройств [2]. Но не всякая функция, а только такая, которая обладает рядом важных свойств, а именно: во-первых, она оперирует исключительно своей внутренней памятью, а во-вторых, она детерминирована (то есть её значение зависит только от входных параметров – это свойство будет подробно рассмотрено далее). Данные ограничения на реализуемость в реальности связаны с физическими законами сохранения, в первую очередь, энергии. Именно такие «чистые» процессы рассматриваются в кибернетике при помощи методологии чёрного ящика – результат работы такого ящика зависит только от значений входных параметров.

Классическая иллюстрация, демонстрирующая эту ситуацию, встречается в большинстве учебников по кибернетике и смежным дисциплинам:

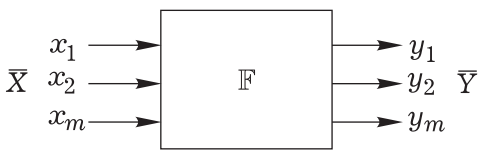

Рис. 1

Формальные основы теории вычислений были заложены несколькими учёными, одним из ведущих среди которых был Алонзо Чёрч, предложивший в качестве формализма для представления вычислимых функций и процессов λ-исчисление [1]. Само по себе λ-исчисление предлагает нотацию для простейшего языка программирования. Собственно, ядро языка программирования Haskell представляет собой типизированное λ-исчисление.

Функциональный подход имеет в своём основании серьёзную теоретическую проработку, а потому его изучение может стать серьёзным подспорьем как для начинающих разработчиков программного обеспечения, так и для профессионалов, желающих расширить свой кругозор и получить навыки владения новыми инструментами для решения практических задач.

1. Общие свойства функций в функциональных языках программирования

Перед изучением примеров необходимо рассмотреть общие свойства функций, рассматриваемые в функциональном программировании. К таким свойствам наиболее часто относят чистоту (то есть отсутствие по-

бочных эффектов и детерминированность), ленивость и каррированность (наличие у функции особого типа, что влечёт за собой возможность производить частичные вычисления и использовать функции в качестве объектов вычислений – получать их в качестве параметров и возвращать в качестве результатов).

Итак, как уже упоминалось, физически реализуемыми являются такие кибернетические машины, выход которых зависит только от значений входных параметров. Это положение относится и к таким кибернетическим машинам, которые имеют внутренний накопитель – память. Данное положение нашло чёткое отражение в парадигме функционального программирования, поскольку в ней принято, что функции, являясь чистыми математическими объектами, должны обладать свойством чистоты. Это обозначает, что функция может управлять только выделенной для неё памятью, не модифицируя память во вне своей области. Любое изменение сторонней памяти называется побочным эффектом, и функциям в функциональных языках программирования обычно запрещено иметь побочные эффекты.

То же и с детерминированностью. Детерминированной называется функция, выходное значение которой зависит только от значений входных параметров. Если при одинаковых значениях входных параметров в различных вызовах функция может возвращать различные значения, то говорят, что такая функция является недетерминированной. Соответственно, обычно все функции в функциональной парадигме являются детерминированными.

Конечно, есть некоторые исключения, поскольку, к примеру, систему ввода-вывода невозможно сделать без побочных эффектов и в условиях полной детерминированности. Также и генерация псевдослучайных чисел осуществляется недетерминированной функцией. Само собой разумеется, что эти практические задачи должны решаться универсальным языком программирования, коим является язык Haskell. В языке имеются специальные механизмы для этого, но их рассмотрение выходит за рамки настоящей статьи.

Очень интересным свойством функций является ленивость. Не все функциональные языки предоставляют разработчику возможность определять ленивые функции, но язык Haskell изначально является ленивым, и разработчику необходимо делать специальные пометки для функций, которые должны осуществлять «энергичные» вычисления. Ленивая же стратегия вычислений заключается в том, что функция не производит вычислений до тех пор, пока их результат не будет необходим в работе программы. Так, значе-

ния входных параметров никогда не вычисляются, если они не требуются в теле функции. Это позволяет, в том числе, создавать потенциально бесконечные структуры данных (списки, деревья и т. д.), которые ограничены только физическим размером компьютерной памяти. Такие бесконечные структуры вполне можно обрабатывать ленивым способом, поскольку вычисляются в них только те элементы, которые необходимы для работы. И передача на вход какой-либо функции бесконечного списка не влечёт зацикливания программы, поскольку она не вычисляет весь этот список целиком (что было бы невозможным).

Наконец, уже упоминалось, что у функций есть тип. В функциональных языках программирования принято, что тип функций является каррированным, то есть таким, общий вид которого выглядит следующим образом:

$$A_1 \to (A_2 \to \dots (A_n \to B) \dots), \qquad (1)$$

где A_1, A_2, … A_n – типы входных параметров, а B – тип результата.

Каррированность функций означает, что такие функции принимают входные параметры по одиночке, а в результате такого одиночного применения получается новая функция. Так, если в функцию указанного выше типа подать первый параметр типа A_1, то в итоге получится новая функция с типом:

$$A_2 \to (A_3 \to \dots (A_n \to B) \dots). \qquad (2)$$

Когда на вход функции подаются все входные параметры, в результате получаются значения типа B.

В свою очередь, это означает не только возможность частичного применения (на вход функции передаётся только часть параметров, а не все сразу), но и то, что функции сами по себе могут быть объектами вычислений, то есть передаваться в качестве параметров другим функциям и быть возвращаемыми в качестве результатов. Ведь никто не огранивает указанные типы A_1, A_2, … A_n и B только атомарными типами, это всё могут быть также и функциональные типы.

Перечисленные свойства функций в функциональных языках программирования открывают дополнительные возможности по использованию функционального подхода, поэтому разработчикам программного обеспечения рекомендуется изучить этот вопрос более подробно.

2. Примеры определения функций

В одной из компаний, где в своё время довелось работать автору, при отборе кандидатов на вакантные должности инженеров-программистов пришедшим задавалась простая задача – необходимо написать функцию, которая получает на вход некоторое целое число, а возвращает строку с представлением данного числа в шестнадцатеричном виде. Задача очень простая, но вместе с тем она очень легко может проявить методы решения задач у кандидатов, поэтому основной упор на собеседовании делался не на правильность написания кода, а на подход и канву рассуждений при написании этой функции. Более того, если кандидат затруднялся с алгоритмом, ему он полностью разъяснялся, поскольку интересен был именно ход рассуждений и окончательный способ реализации алгоритма. Соответственно, для решения задачи можно было использовать произвольный язык программирования, на выбор кандидата. Можно было даже использовать псев-

доязык описания алгоритмов, блок-схемы и прочие подобные вещи.

Сам алгоритм прост. Необходимо делить заданное число на основание (в задаваемой задаче, стало быть, на 16), собирать остатки и продолжать этот процесс до тех пор, пока в результате деления не получится 0. Полученные остатки необходимо перевести в строковый вид посимвольно (учитывая шестнадцатеричные цифры), после чего конкатенировать все эти символы в результирующую строку в правильном направлении (первый остаток должен быть последним символом в результирующей строке, второй – предпоследним и т. д.).

Каковы были типовые рассуждения большинства приходящих на собеседования? «Получаем входное число – организуем цикл **while** до тех пор, пока параметр цикла не станет равен 0 – в цикле собираем остатки от деления параметра на основание, тут же переводим их в символы и конкатенируем с переменной, которая потом будет возвращена в качестве результата – перед возвращением переменную обращаем». К сожалению, за всё время работы автора в этой компании ни один из кандидатов не предложил решения задачи в функциональном стиле.

Вот как выглядит типовая функция для описанной цели на языке C++:

```
std::string int2hex (int i)
{
  std::string result = "";
  while (i)
  {
    result = hexDigit (i % 16) + result;
    i /= 16;
  }
  return result;
}
```

Здесь функция **hexDigit** возвращает символ, соответствующий шестнадцатеричной цифре.

Как же решить эту задачу при помощи функционального подхода? При размышлении становится ясно, что взяв первый остаток от деления на 16 и после этого целочисленно разделив само число на 16, задача сводится к той же самой. И такое сведение будет происходить до тех пор, пока число, которое необходимо делить, не станет равным 0. Налицо рекурсия, которая является одним из широко используемых методов

функционального программирования. На языке Haskell эта задача может быть решена следующим образом:

```
int2hex :: Integer -> String
int2hex 0 = ""
int2hex i = int2hex (div i 16) ++ hexDigit (mod i 16)
```

Здесь функции **div** и **mod** возвращают соответственно результат целочисленного деления и остаток от такого деления. Функция (**++**) конкатенирует две строки. Все эти функции определены в стандартном модуле **Prelude**. Первая строка определения функции – так называемая сигнатура, которая определяет тип функции (необходимо обратить внимание на эту запись – она полностью соответствует тому, что описано в предыдущем разделе относительно функциональных типов).

Вторая строка определяет результат функции **int2hex** в случае, если значением её единственного входного параметра будет 0. Третья строка, соответственно, определяет результат функции в оставшихся случаях (когда значение входного параметра ненулевое). Здесь применён механизм сопоставления с образцами, когда для определения функции записывается несколько выражений (иногда в литературе по функциональному программированию используется термин «клоз» от англ. «clause» для обозначения одного такого выражения в определении функции), каждый из которых определяет значение функции в определённых условиях.

Предложенный пример уже достаточно показывает отличие двух подходов к представлению вычислений. Тем не менее уже сейчас видно, что есть широкий простор для усовершенствования кода. В первую очередь это касается основания преобразования, ведь часто при программировании необходимы числа в двоичной и восьмеричной записи. Более того, почему бы не сделать универсальную функцию для преобразования числа в произвольную систему счисления? Эта задача легко решается преобразованием уже написанной функции:

```
convert :: Int -> Int -> String
convert _ 0 = ""
convert r i = convert r (div i r) ++ digit r (mod i r)
```

Здесь в сигнатуру внесены два изменения. Во-первых, тип **Integer** изменён на тип **Int**, что связано с необходимостью ограничения (тип **Integer** представляет неограниченные целые числа, тип **Int** – ограниченные интервалом $[-2^{29};\ 2^{29}-1]$ для оптимизации вычислений). Во-вторых, теперь функция **convert** принимает два параметра – первым параметром она принимает значения основания, в систему счисления по которому необходимо преобразовать второй параметр. Как видно, определение функции стало не намного сложнее. Ну и в-третьих, в первом клозе определения на месте первого параметра стоит так называемая маска подстановки (_), которая обозначает, что данный параметр не используется в теле функции.

Соответственно, функция **digit**, возвращающая цифру в заданном основании, теперь тоже должна получать и само основание. Но её вид, в отличие от функции **hexDigit**, которая являлась простейшим отображением первых шестнадцати чисел на соответствующие символы шестнадцатеричной системы счисления, теперь должен стать совершенно иным. Например, вот таким:

```
digit r i | r < 37    = if  (i < 10)
                          then show  i
                          else [(toEnum (i + 55))::Char]
         | otherwise = " (" ++ (show i) ++ ")"
```

Здесь, в определении функции **digit** имеется несколько интересных особенностей языка Haskell. Во-первых, вместо механизма сопоставления с образцами определение произведено через механизм охраны (охранных выражений), которые также позволяют сравнивать входные параметры с некоторыми значениями и осуществлять ветвление вычислений. Вторая особенность – использование выражений **if-then-else** для тех же самых целей в первом варианте. Особой разницы между этими подходами нет, вдумчивому читателю предлагается поэкспериментировать с охранными и условными выражениями (подробности синтаксиса – в специализированной литературе, рекомендуется использовать справочник [5]).

Функции **show** и **toEnum** опять же описаны в стандартном модуле **Prelude**, который подгружается всегда. Первая функция преобразует любое значение в строку (её тип **a -> String)**, вторая – преобразует целое число в заданный тип (её тип **Int -> a**, причём конкретно в данном случае она преобразует целое в код символа **Char**). Таким образом, алгоритм работы функции **digit** прост: если основание системы счисления меньше 37 (это число – сумма количества десятеричных цифр и букв латинского алфавита), то результирующая строка собирается из символов цифр и латинских букв. Если же основание больше или равно 37, то каждая цифра в таких системах счисления записывается как соответствующее число в десятеричной системе, взятое в круглые скобки.

Теперь можно определить несколько дополнительных функций, которые наиболее часто могут использоваться на практике:

```
int2bin = convert 2
int2oct = convert 8
int2hex = convert 16
```

Вот здесь и применяется подход, который называется частичным применением. В данных определениях производится частичный вызов уже определённой ранее функции **convert**, которая, как видно, ожидает на вход два параметра. Но здесь ей на вход передаётся всего один параметр, в результате чего получаются новые функции, ожидающие на вход один параметр. Этот подход проще всего понять, представив, что первый параметр функции **convert** просто подставлен во все места, где он встречается в теле функции. Так, частичная подстановка **convert 2** превращает определение в:

```
convert :: Int -> Int -> String
convert 2 0 = ""
convert 2 i = convert 2 (i `div` 2) ++ digit 2 (i `mod` 2)
```

Поскольку данное определение можно легко преобразовать в функцию одного параметра (первый же теперь зафиксирован и является константой), современные трансляторы языка Haskell проводят именно такую оптимизацию, создавая дополнительное определение новой функции для частичных применений.

Осталось немного оптимизировать определение функции **convert**, чтобы изучить некоторые элементы функционального программирования, связанные с оптимизацией, улучшением внешнего вида исходного кода и т. д. Вот новое определение функции преобразования числа:

```
convert' :: Int-> Int-> String
convert' r i = convert_a r i ""
  where
  convert_a _ 0 result = result
  convert_a r i result = convert_a r (i `div` r)
  (digit r (i `mod` r) ++ result)
```

Данное определение необходимо разобрать подробно.

Функция **convert'** выполняет абсолютно то же вычисление, что и функция **convert**, однако оно основано на подходе, который называется «накапливающий параметр» (или «аккумулятор»). Дело в том, что в изначальном определении функции **convert** используется рекурсия, которая в некоторых случаях может приводить к неоптимальным вычислительным цепочкам. Ну и, собственно, для некоторых рекурсивных функций можно провести преобразование так, что они принимают вид хвостовой рекурсии, которая может выполняться в постоянном объёме памяти.

В функциональном программировании такое преобразование делают при помощи накапливающего параметра. Определение начальной функции заменяют на вызов новой функции с накапливающим параметром, а в данном вызове передают начальное значение этого параметра. Дополнительная же функция производит вычисления как раз в накапливающем параметре, делая рекурсивный вызов самой себя в конце всех вычислений (в этом и заключается смысл хвостовой рекурсии). Соответственно, здесь видно, что функция **convert_a** вызывает саму себя в самом конце вычислений, а приращение цифр в новой системе счисления производится как раз в третьем параметре, который и является накапливающим.

Особо надо обратить внимание на вид функции **convert_a**. Её определение записано непосредственно в теле функции **convert'** после ключевого слова **where**. Это ещё один из элементов программирования, который заключается в создании локальных определений функций или «замыканий». Замыкание находится в области имён основной функции, поэтому из его тела видны все параметры. Кроме того, замыкания используются для оптимизации вычислений – если в теле основной функции несколько раз вы-

звать локальную функцию с одним и тем же набором параметров, то результат будет вычислен один раз.

Дополнительные приёмы программирования, описание ключевых слов, а также описание метода преобразования функции к хвостовой рекурсии можно детально изучить при помощи книги [5]. Здесь же осталось упомянуть то, что полученные функции **`convert`** и **`convert'`** можно использовать так, как любые иные.

Заключение

Обсуждение преимуществ и недостатков тех или иных подходов к программированию – удел идеалистов. Вместе с тем знание обоих методов описания вычислительных процессов позволяет более полноценно взглянуть на проектирование и разработку программных средств. Но, к сожалению, в учебных заведениях редко изучают оба подхода на уроках информатики, а потому у начинающих специалистов и интересующихся имеется известный перекос в сторону процедурного стиля.

Вместе с тем владение функциональным стилем и его основными методиками (декомпозицией и выражением ещё нерешённых задач через уже решённые) позволяет более эффективно решать не только повседневные, но и управленческие задачи, поскольку эти приёмы также повсеместно встречаются в области регулирования и управления. Ввиду вышеизложенного автор надеется, что распространение и популяризация парадигмы функционального программирования позволит не только взращивать более серьёзных и вдумчивых специалистов в области информационных и автоматизированных систем, но и решит некоторые проблемы подготовки управленческих кадров.

Автор будет рад любому конструктивному комментарию, направленному на адрес электронной почты roman.dushkin@mail.com.

Список литературы

1. *Барендрегт Х.* Ламбда-исчисление. Его синтаксис и семантика: Пер. с анг. – М.: Мир, 1985. – 606 стр.

2. *Винер Н.* Кибернетика, или Управление и связь в животном и машине: Пер. с анг. – М.: Советское радио, 1985. – 216 стр.

3. *Вольфенгаген В.Э.* Комбинаторная логика в программировании. Вычисление с объектами в примерах и задачах. – М.: МИФИ, 1994. –204 стр. – ISBN 5-89158-101-9.

4. *Душкин Р.В.* Функциональное программирование на языке Haskell. – М.: ДМК Пресс, 2007. – 608 стр., ил. – ISBN 5-94074-335-8.

5. *Душкин Р.В.* Справочник по языку Haskell. – М.: ДМК Пресс, 2008. – 544 стр., ил. –ISBN 5-94074-410-9.