УДК: 004.021

Использование функциональных языков в промышленной разработке программного обеспечения

Левинков Г.С., Буняева Е.В.

Дальневосточный государственный университет путей сообщения, Хабаровск, email: gleb-hleb@mail.ru

В статье рассматриваются вопросы удобства и производительности разработчика написания программного кода, используя функциональные языки. В качестве примера для сравнения используется программа, написанная функционально на Haskell, и на функционально-императивном языке Kotlin.

Ключевые слова: программирование, функциональное программирование, Kotlin, Haskell, приложение, анализ.

Using functional languages in industrial software engineering

Levinkov G.S., Bunyaeva E.V.

Far East State Transport University, Khabarovsk, email: gleb-hleb@mail.ru

The article analyzes coding convenience and developer performance using functional languages. As an example, there is a program written in purely functional way using Haskell and Kotlin.

Keywords: programming, functional programming, Kotlin, Haskell, software, analysis.

На сегодняшний день существует множество стиль разработки, позволяющие программисту выбрать удобный ему способ написания программы. Например: императивный, когда программа описывается только последовательностью инструкций, объектно-ориентированный, где программа представляет собой совокупность классов и взаимодействия объектов, и функциональный, когда программа рассматривается как последовательность математических функций и предполагает отсутствие изменяемых данных и побочных эффектов, таких как запросы в базу данных или к серверу.

Самой популярной на сегодняшний день является объектно-ориентированная парадигма, так как программу легче и проще разбить на некоторое количество сущностей (классов) и наладить взаимодействие между ними.

Но некоторые компании начинают использовать функциональные языки программирования для разработки ПО. В этой статье дается сравнительный анализ эффективности и читабельности программного кода для алгоритма управления заметками, реализованного на полностью функциональном языке (Haskell), и новом языке Kotlin,

который поддерживает все вышеперечисленные парадигмы. При этом, в случае Kotlin, при написании программы будет использоваться только функциональный стиль, как и в программе на Haskell.

Среди особенностей функционального стиля написания кода выделим следующие:

- ясность – побочные эффекты (запись в файл и т.д.), должны быть изолированы, а поток данных и обработка ошибок — описаны явно;
- параллелизм – функциональный код по умолчанию проще выполнять многопоточно из-за концепции, известной как функциональная чистота;
- используются функции высшего порядка;
- неизменяемость – переменные, к не подлежат изменению после их инициализации.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы.

Начнем написание программы. Первый пункт программы, а именно стартовая точка, функция main. Написанная на Haskell:

```
main = do
  putStrLn "Commands:"
  putStrLn "+ <String> - Add a TODO entry"
  putStrLn "- <Int>    - Delete the numbered entry"
  putStrLn "q          - Quit"
  prompt []
```

И такая же, но на Kotlin:

```
fun main(args: Array<String>) {
  println("Commands:")
  println("+ <String> - Add a TODO entry")
  println("- <Int> - Delete the numbered entry")
  println("q - Quit")
  prompt(emptyList())
}
```

Функция main выводит в консоль меню выбора действия и в конце вызывает функцию prompt от пустого листа. В обоих случаях код можно понять, обойдясь простым знанием английского языка и не вдаваясь в синтаксис языка программирования.

Также можно было увидеть и подумать, что выше используется императивный стиль написания программы на Haskell (последовательность действий), хотя это не совсем так. Синтаксическая конструкция do неявно обеспечивает простую сокращённую форму записи

цепочки монадических операций. Понятие монад и монаидических операций подробно не рассматривается в этой статье, но их определение будет дано позже.

После вывода меню с выбором действия в консоль, нам необходимо получить ответ пользователя. За это отвечает функция prompt. Она выводит список заметок и получает введенные символы с клавиатуры. После, передает результат последнего действия в функцию interpret.

Haskell:

```
putTodo :: (Int, String) -> IO ()
putTodo (n, todo) = putStrLn (show n ++ ": " ++ todo)

prompt :: [String] -> IO ()
prompt todos = do
  putStrLn ""
  putStrLn "Current TODO list:"
  mapM_ putTodo (zip [0..] todos)
  command <- getLine
  interpret command todos
```

Kotlin:

```
fun putTodo(pair: Pair<String, Int>){
  val n = pair.second
  val todo = pair.first
  println("$n: $todo")
}

fun prompt(todos: List<String>){
  println()
  println("Current TODO list:")
  todos
     .zip((0..todos.size).toList()) //явно приводим к List, т.к. 1..n возвращает IntRange, а не List<Int>
     .map(::putTodo)
  val command = readLine()
  intepret(command.split(' '), todos)
}
```

Код можно понять поверхностно, зная что в языках программирования есть функции map, zip и упрощенное создание диапазона чисел с помощью a..b.

При чтении, вопрос, к примеру, может возникнуть взглянув на функцию mapm_. mapm не просто функция map, а map работающая с монадами. Имеет следующую сигнатуру:

Monad m => (a -> m b) -> t a -> m (t b),

где m это монада (набор правил, линейная цепочка связанных вычислений).

Монады неотъемлемая часть функциональных языков, хотя для начинающих изучение – эта тема с первого взгляда является сложной. Требуется долгое время чтобы её понять, но знание о них считается необходимым для написания серьезных программ, а времени при написании промышленного ПО очень часто нет.

Следующий вопрос, почему в названии функции стоит нижнее подчеркивание. «Все функции, которые были до этого – без нижнего подчеркивания, а в чем тут загвоздка?».

Функция mapM_ - разновидность функции mapM, игнорирующая в итоге результат выполнения. Имеет следующую сигнатуру, из которой видно, что результат выполнения функции пустой (о чём говорят две пустые скобки в конце):

Monad m => (a -> m b) -> [a] -> m ()

При написании кода на Kotlin, мы используем функцию zip для создания списка пар (1, “Имя заметки”) и каждый элемент этого списка передаем в функцию putTodo. Количество строк кода заметно больше, но код от этого стал понятнее. При написании кода на Haskell не всегда такое возможно.

Функция interpret в зависимости от аргументов отвечает за добавление и удаление заметки из списка заметок.

Снова код на Haskell:

```
interpret :: String -> [String] -> IO ()
interpret ('+':' ':todo) todos = prompt (todo:todos)
interpret ('-':' ':num ) todos =
  case delete (read num) todos of
    Nothing -> do
      putStrLn "No TODO entry matches the given number"
      prompt todos
    Just todos' -> prompt todos'
interpret  "q"          todos = return ()
interpret  command      todos = do
  putStrLn ("Invalid command: `" ++ command ++ "`")
  prompt todos
```

```
delete :: Int -> [a] -> Maybe [a]
delete 0 (_:as) = Just as
delete n (a:as) = do
  let n' = n - 1
  as' <- n' `seq` delete n' as
  return (a:as')
delete _ [] = Nothing
```

И Kotlin:

```
fun intepret(promptResult: List<String>, todos: List<String>){
  val operator = promptResult.first()
  when(operator){
    "+" -> prompt(todos + promptResult.last())
    "-" -> {
      val indexOfElement = promptResult.last().toInt()
      prompt(todos - todos[indexOfElement])
    }
    "q" -> return
    else -> println("Invalid command: '$operator'")
  }
}
```

В Haskell данный функционал реализован более удобно и кратко: с помощью pattern matching (парсинг аргументов исходя из сигнатуры) вместе с перегрузкой функций (вызов функций в зависимости от переданных ей аргументов). Но при этом пришлось написать собственную функцию delete для проверки наличия и удаления элемента из списка.

Здесь вопросы может вызвать использование функции read, и оператора ‘:’ при работе со списком.

Функция read приводит значение типа String в значение типа Int. Оператор ‘:’ в 2-ой строке кода на Haskell используется для добавления элемента todo в список todos. A Nothing и Just – типы, связанные с опциональным типом Maybe, которые в свою очередь является монадой.

В Kotlin для добавления и удаления элементов используется перегруженные операторы + и -. В коде на Kotlin не реализована проверка наличия элемента в списке, но это легко делается одним выражением if (вышел ли индекс за границы списка). Оно не было добавлено для емкости кода.

Как можно видеть из написанного кода, сами функциональные языки далеки от моделей реального мира, от моделей задач, что появляются в головах программистов, и от того, как работает компьютер. Изменяемое состояние — это то, что близко реальному миру, человеку, компьютеру.

Также одним из аргументов непопулярности функциональных языков является TIOBE Index. В первой двадцатке нет ни одного функционального языка.

В итоге, на примере простого консольного приложения для работы с заметками можно увидеть следующие результаты:

- написание серьезных программ на Haskell и других функциональных языках требует глубокого понимания принципов и особенностей построения программ именно в стиле функционального программирования;
- монады, функторы и остальные конструкции функционального программирования можно использовать в Kotlin, но вникать в их работу нужно только по необходимости глубоко разобраться в теме. Все такие конструкции явно и понятно описаны в документации. Хотя в Haskell знание конструкций выше и их работы обязательно;
- функциональный код на Kotlin может работать также, как и код на Haskell, но изучение Kotlin, зная другие императивные языки (Java, C++) быстрее и проще. Изучение Haskell требует больше усилий и времени.

Большинство разработчиков считает, что причина недостаточной популярности функциональных языков намного проще: функциональное программирование слишком неестественное, а именно программирование в функциональном стиле часто происходит «задом наперед» и выглядит больше, как решение головоломок, чем объяснение задачи компьютеру.

Список литературы:

1. Душкин Р., 14 занимательных эссе о языке Haskell и функциональном программировании / Душкин Р. – 2016. – 222 с.
2. Мена А.С. Изучаем Haskell / Мена А.С. – Питер: Изд-во Питер, 2015. – 315 с.
3. Жемеров Д. Исакова С., Kotlin в действии / Жемеров Д. Исакова С. – Питер: Изд-во Питер, 2017. – 402 с.