

УДК. 004.432.42

## **Представление подходов и сравнение с императивными языками основных парадигм функционального программирования на примере языка Haskell**

*Лифарь А.С., студент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,  
кафедра «Системы обработки информации и управления»*

*Научный руководитель: Гапанюк Ю.Е., к.т.н, доцент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана*

[\*gapyu@bmstu.ru\*](mailto:gapyu@bmstu.ru)

### Введение:

Функциональное программирование – раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). В императивных языках на результат выполнения функции влияют не только аргументы, но и внешние переменные, описанные вне тела функции. Это не соответствует математическому пониманию функции. Парадигма функционального программирования основана на фундаментальном понимании функции. Рядом с этим подходом процедурное программирование отходит на задний план. Появляются новые возможности при решении задач.

Для изучения и анализа подходов функционального программирования был выбран язык Haskell, так как он является чисто функциональным языком. Он наиболее ярко и широко способен продемонстрировать все отличия от привычных подходов и реализаций решения задач. В основной части статьи продемонстрированы важнейшие парадигмы функционального программирования.

### Основная часть:

1. У функций отсутствуют побочные эффекты

Так же это называют «ссылочной прозрачностью». Если функция дважды вызывается несколько раз с одними и теми же аргументами, она непременно вернет одинаковые результаты. Это позволяет строить на основе простых функций более сложные, не опасаясь неожиданных результатов. Все функции являются детерминированными. Это полностью отвечает фундаментальному математическому пониманию функции.

При проектировании информационных систем сегодня применяются методы декомпозиции функциональности и связывания отдельных функций в цепь вычислений. Функциональное программирование предлагает практические методы реализации этих идей. Результаты выполнения функции зависят только от входных параметров (принцип «черного ящика» в кибернетике).

## 2. Функции проявляют «ленивое» поведение.

В языке Haskell функции необходимо явно указать, что функции необходимо производить вычисления обязательно. Ленивость заключается в том, что компилятор не будет выполнять функции, пока это действительно не потребуется для вывода результата. Это позволяет создавать бесконечные структуры данных (списки, деревья, интервалы и др.), потому что реально вычислять требуется только та часть данных, которую необходимо отобразить.

Продemonстрируем ленивую стратегию на примере простейшей функции в синтаксисе языка Haskell:

```
takeSomeNumbers n = take n [5 ...]
```

Здесь стандартная функция `take` принимает на вход два аргумента: число и список. Как видно синтаксис похож на синтаксис функции в математике. Функция `take` возвращает `n` первых чисел из списка `[5 ...]`.

## 3. Haskell – статически типизированный язык.

Если необходимо объявить переменную – целое число или строку – то прописывать тип явно не нужно, компилятор способен сам распознать, какому типу соответствует переменная. У функций тоже есть тип, тип возвращаемого значения. Однако, в

функциональном языке тип возвращаемого значения должен быть каррированным, то есть иметь следующий вид:

$$A1 \rightarrow (A2 \rightarrow \dots \rightarrow (A_n \rightarrow B) \dots),$$

где  $A_1, A_2, \dots, A_n$  – типы аргументов,  $B$  – тип возвращаемого значения.

Каррирование – это преобразование функции от многих аргументов в функцию, берущую свои аргументы по одному. Реализация такого подхода, к примеру, в C++ требует наличие дополнительной библиотеки Boost и реализация вовсе не выглядит наглядной и простой. В стандарте C++11 в библиотеку STL был добавлен модуль `functional`, предоставляющий набор шаблонов классов для работы с функциональными объектами, а также набор вспомогательных классов для их использования в алгоритмах стандартной библиотеки. Реализацию каррированной функции в C++11 нельзя назвать наглядной и очевидной, и выглядит она следующим образом:

```
#include<functional>

auto curry = ([](int x)->std::function<int(int)>{
    return [x](int y)->int {
        return x+y;
    };
});

int a = curry(4)(5); // 9
auto curry_4 = curry(4);
int b = curry_4(5); // 9
```

В языке Haskell код такой же функции занимает две строчки (первая строчка не является обязательной) и выглядит не в пример читабельнее:

`curry :: Int -> Int -> Int` — явное указание типа функции (каррированной)

`curry x = x + y`

`curry 4 5 -- вернет 9`

4. Функциональное программирование позволяет реализовать подход сопоставления с образцом.

Возможность ветвления в функциональном языке поддерживается, существуют такие конструкции, как `if, then .. else`. Например, как в следующей функции:

```
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
```

Функция увеличивает число в два раза, только если это число не превосходит 100. Разница между условной конструкцией `if` в Haskell и операторами `if` в императивных языках заключается в том, что ветвь `else` в языке Haskell обязательна. В императивных языках существует возможность просто пропустить несколько шагов, если условие не выполняется, в функциональном программировании каждое выражение или функция должны что-то возвращать. Поэтому использование конструкций ветвления в функциональном программировании часто бывает избыточно и громоздко. Но существует решение для этой проблемы. Можно использовать сопоставление с образцом. То есть прописать несколько реализаций функции, что в императивных языках реализуется посредством `switch ... case` и конструкциями `if ... else`. Во время выполнения программы данные проверяются на соответствие образцу. Если они подходят под образец, то будут разобраны в соответствии с ним:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Когда определяется функция, ее определение можно разбить на несколько частей (клозов). Можно создавать образцы любого типа данных – чисел, символов, списков, кортежей и т.д.

## 5. Параметрический полиморфизм

В функциональном программировании очень широко применяется параметрический полиморфизм. Он является одним из наиболее важных понятий в парадигме функционального программирования. Функции можно описывать не для конкретных типов, но и для обобщенных, то есть задавать тип параметром. Это аналогов шаблонов и generics в императивных языках. Для примера приведем следующую функцию:

```
list_pair = [a]->[b]->[(a,b)]
```

Оператор [...] обозначает в языке Haskell список, а в круглых скобках заключается пара значений. Списки в функциональном программировании являются гомогенными структурами. Это означает, что элементы внутри списка должны быть строго одного типа. Функция, приведенная в примере, принимает на вход два списка с элементами переменного типа. А в результате получается список пар из значений списков.

Другими словами, для параметрического полиморфизма вводится формализация, которая позволяет связывать не только простые переменные, но и типовые переменные. При использовании в дальнейшем полиморфных типов типовые переменные конкретизируются определенными типами.

В зависимости от того, какие значения могут принимать типовые переменные, различают два подвида параметрического полиморфизма:

- Предикативный параметрический полиморфизм проявляется тогда, когда в качестве значений типовых переменных могут быть подставлены только мономорфные типы («монотипы»), то есть такие типы, в определениях которых не используются типовые переменные.

- Непредикативный параметрический полиморфизм позволяет конкретизировать типовые переменные произвольными типами (мономорфными и полиморфными), в том числе и рекурсивно: если в определении типа  $T$  используется типовая переменная  $a$ , то при конкретизации вместо нее может быть подставлен сам тип  $T$ . При определении сущности возможно использование ссылки на саму определяемую сущность.

В языке Haskell используется непредикативный параметрический полиморфизм первого ранга. Хорошим примером для определения полиморфных типов данных в языке Haskell можно рассмотреть определения структур данных, таких как списки, например:

```
data List a = Nil
           | Cons a (List a)
```

Это простое описание односвязного списка на языке Haskell. Типовая переменная  $a$ , как и в других примерах, может принимать любое значение. Однако, предполагается, что при ее инстанцировании, она будет замещена любимым мономорфным типом. В функциональном программировании можно обрабатывать, к примеру, списки списков, списки деревьев; при этом уровень вложенности может быть неограничен. В Haskell есть удобный набор функций, позволяющий работать с подобными полиморфными типами данных.

## 6. Моноиды в функциональном программировании

Из линейной алгебры: моноид – это множество элементов  $S$ , над которым задана ассоциативная бинарная операция  $(*)$ , называемая умножением), и элемент  $e$  – такой, что

для любого  $x$ , принадлежащего  $S$  верно:  $x * e = e * x = x$ . Этот элемент называется нейтральным относительно операции умножения. Также, моноид является алгеброй, поэтому результат любых операций над элементами множества  $S$  также входит в  $S$ .

Моноиды часто можно встретить в порой совершенно различных областях:

а) Множество целых чисел с заданными операциями  $(+,* )$  и нейтральными элементами  $(0, 1)$  для операций соответственно

- $X * (Y * Z) = (X * Y) * Z$ ,  $(X + Y) + Z = (Y + X) + Z$  для любых  $X, Y, Z$ ;
- $X + 0 = 0 + X = X$ ,  $X * 1 = 1 * X = X$  для любых  $X$ ;

Таким образом, класс целых чисел является одновременно моноидом с нейтральным элементом 0 относительно операции  $+$ , и нейтральным элементом 1 относительно операции умножения  $*$ .

- b) Множество списков с операцией соединения ( $++$ ) и нейтральным элементом  $[]$ :
- $[x, y] ++ [z] = [x] + [y, z]$  для любых  $x, y, z$ ;
  - $[x, y, z] ++ [] = [] ++ [x, y, z] = [x, y, z]$ ;
- c) Множество строк с операцией конкатенации ( $.$ ) нейтральным элементом  $""$ :
- $"foo"."barbaz" = "foobar"."baz"$ ;
  - $"foobar"."" = ""."foobar" = "foobar"$ .

Как же перейти от математического определения моноида к терминам программирования?

Зная определение моноида, мы можем выделить общие свойства функций, благодаря которым их можно назвать умножением для какого-либо моноидного класса:

- Функция принимает два параметра ( умножение моноида бинарно );
- Принимаемые и возвращаемое значение имеет одинаковый тип ( свойство моноида как алгебры);
- Существует значение, которое не изменяет другие значения, когда является одним из параметров функции (существует нейтральный элемент);
- Вызовы функции должны быть ассоциативны (то есть  $f( f(a, b), c) \Leftrightarrow f( a, f(b, c))$  ).

Моноидные классы могут быть представлены во многих языках, и разработчики часто делают это, не задумываясь об общей природе таких классов. Haskell представляет более математический строгий и, одновременно с этим, более элегантный и гибкий способ описания моноидных классов, предоставляя класс типов `Monoid`:

Class Monoid w where

    mempty :: m

    mappend :: m -> m -> m

    mconcat :: [m] -> m

    mconcat = foldr mappend mempty

Пользователь может вносить свои типы в класс Monoid, предоставляя собственные mempty, mappend и mconcat, где

**mempty** – константа, задающая нейтральный элемент будущего моноида

**mappend** – бинарная функция, представляющая умножение в моноиде и удовлетворяющая условиям выше

**mconcat** – функция, принимающая список значений и применяющая **mappend** между элементами списка, получая в результате единственное значение (это возможно благодаря ассоциативности операции **mconcat**).

## 7. Монады в функциональном программировании

В понимании функционального программирования монада - это абстракция линейной цепочки связанных вычислений. Ее основное назначение инкапсуляция функций с побочным эффектом от чистых функций, а точнее их выполнений от вычислений. Монады применяются в языке Haskell, так как он повсеместно использует ленивые вычисления, которые вместе побочным эффектом, как правило, образуют плохо прогнозируемый результат. Монада описывается полиморфным контейнерным типом. Монады это типы, представляющие собой экземпляры одного из следующих монадических классов: Functor, Monad, MonadPlus. Ни один из этих классов не может быть предком для другого класса, то есть монадические классы не наследуемы. Математически монада определяется через набор правил, которые связывают операции, производимые над монадой. Эти правила дают интуитивное понимание, как должна использоваться монада и какова ее внутренняя структура. Приведем в качестве примера класс Monad:

```

m :: * -> *

class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a

class Functor f where
  fmap :: (a -> b) -> f a -> f b

```

Функция **return** описывает «возвращение» (втягивание) типа *a* в монаду *m*, то есть обррамление его контейнером. Функция **fail** не имеет отношения к теоретической сущности монад, однако используется в случае ошибки сопоставления с образцом внутри монадического кода - останавливает процесс последовательных действий и выводит сообщение о причине ошибки. Оператор **>>=** описывает, что в монаде действия происходят последовательно, то есть после применения функции её результат передаётся далее (*.. -> a -> b -> ..*), примером которой может быть передача текста в буфер: типы данные облачаются в монаду (конструктором), а затем с ними функция производит действия, в данном случае добавление. Оператор **>>** — частный случай оператора **>>=**, когда предыдущие данные просто заменяются следующими, которые не формируются на основании предыдущих.

В частности, к монадам относятся:

- **IO** (монада строго последовательных вычислений): стратегия связывания — «сначала первое вычисление, затем второе»;
- **Maybe** (монада вычислений с отсутствующими значениями): стратегия связывания — «если первое вычисление дало результат, то второе; иначе — отсутствие результата»;
- **List** (монада вычислений с несколькими результатами): стратегия связывания — «все возможные результаты второго вычисления, примененного к каждому из вычисленных первым значений параметра»;
- **State** (монада вычислений с переменной состояния): стратегия связывания — «начать второе вычисление с состоянием, измененным в результате первого»;

и некоторые другие типы.

Монады также применяются для синтаксического анализа, продолжений (continuations), вероятностных вычислений и в других случаях.

Заключение:

Функциональное программирование имеет ряд преимуществ перед императивными языками:

1. Благодаря вычислению без состояний повышается надежность кода. Любая функция работает только с локальными данными и работает с ними всегда одинаково. Это помогает избежать трудно отслеживаемых ошибок в коде.

2. Функциональное программирование позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

3. Широкие возможности распараллеливания вычислений. В любом вызове функции всегда допустимо параллельное вычисление двух различных параметров – порядок их вычисления не может оказать влияния на результат вызова.

Функциональные языки используются во многих системах:

- Системы управления телефонными станциями в компании Ericsson написана на языке Erlang (разработанным самой компанией Ericsson).
- Haskell используют для прототипирования приложений обработки цифровых сигналов.

Создатели императивных языков стараются создать универсальное средство разработки, поэтому на практически на всех языках программирования можно использовать функциональный стиль программирования. Но часто универсальность языка программирования не всегда является положительным фактором. Часто оно влечет неоправданное усложнение языковых конструкций. Данная статья демонстрирует, что не следует всегда отдавать предпочтение универсальным средствам разработки, такому как C++.

Кроме перечисленных достоинств существуют и недостатки функционального программирования. Программы, написанные на функциональных языках программирования, работают медленнее, чем реализованные на императивных языках. Это вполне объяснимо, так как перевод в ассемблерный код осуществляется дольше. Тем не менее, программы на функциональном программировании пишутся гораздо быстрее. И часто это может быть тем аспектом, который повлияет на решение выбора именно функционального языка для решения той или иной задачи. Кроме всего прочего, активно развиваются алгоритмы трансляции функциональных языков, и по эффективности ассемблерного кода они постепенно начинают догонять императивные языки. Так что в скором времени функциональное программирование сможет предоставить не только красивое, изящное и простое решение программируемой задачи, но и на столько же эффективное.

#### **Список литературы**

1. Миран Липовача. Изучай Haskell во имя добра! М.: ДМК Пресс, 2014. 490с.
2. Душкин Р.В. Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2007. 608 с.
4. Пипони Д. Моноиды в Haskell и их использование // Практика функционального программирования. 2009. №1(1). С. 77–86.
5. Душкин Р. В. Полиморфизм в функциональном программировании // Практика функционального программирования. 2009. №2(2). С. 86–105,.