МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
им. И. И. Воровича

Направление подготовки
Прикладная математика и информатика

Кафедра информатики и вычислительного эксперимента

ПРИМЕНЕНИЕ МОНАД ЯЗЫКА HASKELL ДЛЯ МОДЕЛИРОВАНИЯ ИГР

Курсовая работа

Студентки 3 курса
О. Е. Филиппской

Научный руководитель:
асс. каф. ИВЭ А. М. Пеленицын

Ростов-на-Дону
2015

Содержание

Введение и постановка задачи

Постановка задачи

Данная работа посвящена развитию методов проектирования программного обеспечения средствами функциональных языков программирования и их идиоматических конструкций.

В работе разрабатываются обобщённые модули для реализации различных игр и приводится сравнение двух подходов к решению задач: чисто функциональный и с применением монад. Цель данной работы — увидеть преимущества второго подхода и продемонстрировать их на реализации различных игр.

1. Предварительные сведения

1.1. Функциональные языки и их свойства

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом их понимании [7]. Противопоставляется парадигме императивного программирования, в которой процесс вычислений описывается как последовательное изменение состояний. В императивных языках значение функции может зависеть не только от аргументов, но и от состояния внешних по отношению к функции переменных. Кроме того, императивные функции могут иметь побочные эффекты. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах алгоритма, можно получатьна выходе разные данные. В функциональном же языке вызов функции с одними и теми же аргументами всегда даёт одинаковый результат. Это даёт возможность средствам выполнения программ на функци-

ональных языках вызывать функции в порядке, не определяемом алгоритмом.

В основе функционального программирования лежит λ-исчисление — система, разработанная американским математиком Алонзо Чёрчем для представления понятия вычислимости в виде формальной системы [8]. λ-исчисление можно рассматривать как семейство прототипных языков программирования. Они являются языками высших порядков: этим обеспечивается то, что аргуметами операторов могут быть операторы, значением также может быть оператор.

Основные свойства некоторых функциональных языков программирования [9]:

Краткость и простота. Программы на функциональных языках обычно короче и проще, чем те же самые на императивных языках.

Строгая типизация. Позволяет компилятору оптимизировать программы. Кроме того, большая часть ошибок, как, например, несоответствующий тип входных данных, отслеживается уже на стадии компиляции, снижая затраты времени на отладку.

Модульность. Позволяет разделять программы на несколько относительно независимых частей — модулей. Между ними строго определены связи. Поддержка модульности — свойство не только функциональных языков.

Значения и объекты вычисления — функции. Функции сами по себе могут быть переданы другим функциям в качество аргумента и возвращены в качестве результата. Функции, принимающие функциональные аргументы называются функциями высших порядков.

Чистота. В чисто функциональных языках у функций отсутствуют побочные эффекты. Побочный эффект — это изменение функцией состояния программы помимо возвращения какого-либо значения. Чистота даёт такой преимущество, как параллелизм: неза-

висимые функции можно вычислять в произвольном порядке или параллельно, не опасаясь, что это повлияет на результат.

Отложенные (ленивые) вычисления. Это означает, что вычисления не проводятся до востребования: аргумент вычисляется только в том случае, если он нужен для результата. Функциональные языки, не поддерживающие отложенные вычисления, называются строгими.

1.2. Язык Haskell

Haskell — чисто функциональный язык программирования общего назначения [10]. Ядро этого языка полностью основано на механизме λ-исчисления. Одним из достоинств языка Haskell является его система типов. Эта система с автоматическим выводом типов: это означает, что если тип не описан явно, то система типов может вычислить его автоматически. Серьёзное отношение к типизации — отличительная черта языка Haskell. Можно отметить следующие характеристики языка Haskell в качестве основных:

- недопустимость побочных эффектов (чистота); возможность писать программы с побочными эффектами достигается за счёт использования монад (см. раздел *Монады в языке Haskell*);
- полная типизация с автоматическим выделением типов;
- ленивые вычисления;
- сопоставление с образцом (выполнение определённых действий при совпадении с тем или иным образцом);
- полиморфизм классов типов;
- алгебраические типы данных.

Сделаем обзор основных возможностей языка Haskell, которые будут использованы при решении задачи, поставленной во введении.

1.3. Классы типов в языке Haskell

Классы типов — интерфейс, определяющий некоторое поведение для типа. Если тип является экземпляром класса типов, то он поддерживает и реализует поведение, описанное классом типов. Если говорить более конкретно, то класс типов определяет некоторый набор функций. Если необходимо сделать какой-либо тип экземпляром класса типов, то необходимо явно обозначить, что означают данные функции применительно к данному конкретному типу. Перечислим основные классы типов языка Haskell, определённые в стандартном модуле **Prelude**.

Класс Eq. Используется для типов, которые поддерживают проверку на равенство.

Класс Ord. Предназначен для типов, поддерживающих отношение порядка.

Класс Show. Экземпляры данного класса типов могут быть представлены в виде строки.

Класс Read. Экземпляр данного класса может быть преобразован из строки с помощью функции *read*.

Класс Num. Класс типов для действительных чисел. Его экземпляры могут вести себя как числа.

Класс Integral. Данный класс включает в себя только целые числа: типы *Int*, *Integer*.

Класс Floating. К данному классу типов относятся числа с плавающей точкой (типы *Floating* и *Double*)

Кроме того, в языке Haskell существует возможность создания собственных классов типов с помощью ключевого слова **class**.

```
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
```

Рисунок 1 — Определение класса Monad из модуля Control.Monad.

1.4. Монады в языке Haskell

Монады можно понимать как контейнерный тип данных (т. е. содержащий в себе значения иных типов), представляющий экземпляр класса *Monad*. Монады применяются в языке Haskell для написания программ с побочными эффектами. Из описания (рис. 1) видно, что экземпляр класса *Monad* должен поддерживать следующие операции. Первая, >>= (*связывание*), позволяет применить функцию, которая принимает обычное значение и возвращает монадические, к монадическому значению. Операция >>: применяет предыдущую операцию к двум монадическим значением. Функция *return* «вносит» свой аргумент в монаду, то есть принимает «чистое» значение и помещает его в минимальный контекст по умолчанию. Наконец, *fail* делает возможным неуспешное окончание действия; явно никогда не используется [1]. Стандартные монады, определённые в модуле **Prelude**:

- **IO**, применяется для совершения действий ввода/вывода;
- []: список, также является монадой;
- **Maybe**, монада с контекстом неудачи: для операций, которые могут закончиться с ошибкой.

2. Решение задачи без монад

Для того, чтобы оценить преимущество решения задачи с использованием монад и классов типов, опишем решение задачи в чи-

```
data Suit = C | D | H | S deriving (Show, Read, Eq)

data Value = C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | J | Q | K | A
        deriving (Ord, Eq, Read, Show, Enum)

data Card = Card {
                  getValue :: Value
                , getSuit  :: Suit
                } deriving (Show)
```

Рисунок 2 — Основные типы данных для игры покер.

```
instance Ord Card where
        (Card v1 _) `compare`  (Card v2 _) = v1 `compare` v2

instance Eq Card where
        (Card v1 _) ==  (Card v2 _) = v1 == v2
```

Рисунок 3 — Экземпляр класса Ord для типа Card.

сто функциональном стиле. Для решения поставленной задачи в модуле *Poker.hs* реализуем следующие пункты.

Новые типы данных. Введём новый тип данных *Card*. Для удобства будем использовать синтаксис записи с именованными полями (рис. 2). Экземпляр класса *Show* определён автоматически; определим экземпляры классов *Ord* и *Eq* самостоятельно (рис. 3), поскольку автоматически введённые операции сравнения будут давать неправильные результаты (для текущей задачи необходимо, чтобы сравнение происходило только по достоинству). Определим тип данных *HandType*, значения которого — названия ставок (рис. 4). Они перечислены в порядке возрастания, для того чтобы автоматические определение экземпляра класса *Ord* проходило корректно.

Функции чтения карт из текстового файла. Определим следующие функции:

- чтения одной карты: функция *readCard*;

```
data HandType = HighCard | OnePair | TwoPairs |
                ThreeOfAKind | Straight | Flush |
                FullHouse | FourOfAKind | StraightFlush | RoyalFlush
                deriving (Ord, Eq, Show)
```

Рисунок 4 — Тип с названиями покерных ставок.

```
type Hand = (HandType, [Card])
```

Рисунок 5 — Синоним с информацией о типе ставки и списком карт.

- разбиение списка карт из файла на списки по десять карт: функция *makeTens*;
- разбиение каждой "десятки"на пары списокв по пять карт: функция *readCardList*.

Функции, определяющие тип ставки игрока. Это функции, принимающие список карт и возвращающие *True* или *False* в зависимость от того, представляет собой данный список карт некоторую ставку или нет.

Другие вспомогательные функции и типы данных. Для удобства введём синоним (рис. 5). Он содержит информацию о типе ставки, а так же список карт игрока, на случай, если у игроков окажутся одинаковые ставки, и необходимо будет проводить дальнейшее сравнение. Кроме того, определим функцию *identHand*, преобразующую список карт одного игрока в список карт с соответствующей ему ставкой.

Функции, определяющие победителя. Опишем функцию *compareHand*, которая производит сравнение по названию ставки и возвращает *True* или *False* в зависимости от того, выиграл первый игрок или нет. Если же ставки у игроков оказались одинаковыми, то она вызывает функцию *compareEqualHandType*, которая, в свою очередь анализирует карты на руках у игроков. Наконец, функ-

```
class (Functor g, Monad g) ⇒ PlayGame g where
        type Payer g
        type Payer g            = Int
        type GameInfo g
        checks          :: [GameInfo g → g (GameInfo g)]
        getPlayer               :: g (GameInfo g) → Payer g
        getGameInfo             :: g (GameInfo g) → (GameInfo g)
```

Рисунок 6 — Задание класса PlayGame.

ция *decideWinner* применяет композицию функций *compareHand* и *makeHand* к сформированному списку игр.

Более подробно см. исходные коды в репозитории [11]. Решение в чисто функциональном стиле приводит нас к правильному результату. Однако написанный код применим только к данной конкретной задаче.

3. Решение задачи с применением монад и классов типов

Выделим типы данных *Card*, *Value*, *Suit*, а так же функции чтения карт в отдельный модуль *Card.hs*.

3.1. Задание класса типов

Приведём реализацию класса типов *PlayGame* (рис. 6). Синонимы типов *Player g* и *GameInfo g* ассоциированны с данным классом [6]. Таким образом, в каждом экземпляре необходимо будет определить *Player g* (по умолчанию — *Int*) и *GameInfo g*. Это позволяет использовать данный класс и для решения задачи об определении победителя и в других играх, не обязательно карточных (см. *Примеры других игр*). Подробнее реализацию можно увидеть в модуле *GameClass.hs* (см. [11])

```
data Game g a = Player Int | Scoring a
                                deriving (Show, Eq, Ord, Read)

instance Functor (Game g) where
        fmap f (Scoring a)      = Scoring (f a)
        fmap f (Player n)       = Player n

instance Monad (Game g) where
        return                          = Scoring
        Scoring a       >>= f           = f a
        Player n        >>= _   = Player n
```

Рисунок 7 — Задание монады Game.

```
 data Poker

instance PlayGame (Game Poker) where
        type GameInfo (Game Poker)      = PokerHand
        getPlayer (Player n)            = n
        getGameInfo (Scoring a)         = a
        checks                          = [ compareHand, firstCheck
                                          , secondCheck, thirdCheck]
```

Рисунок 8 — Экземпляр класса PlayGame для монады Game в модуле Poker.

3.2. Написание монады

Далее реализуем монаду *Game* в модуле *GameMonad.hs* (рис. 7). Данный тип данных содержит два значения. *Player Int* — номер победившего игрока; монада принимает данное значение в том случае, если при применении какой-либо функции удалось определить победителя. *Scoring a* — информация об игре; данное значение принимается, если определить победителя пока не удалось.

3.3. Определение экземпляра класса типа

Рассмотрим теперь определение экземпляра в модуле *Poker.hs* (рис. 8). Используем определение пустого типа (англ. *empty data declaration*) [2]. Фиктивный тип (англ. *phantom type* [3]) *Poker* позволяет

избежать конфликта модулей при попытке их использования в одной программе.

3.4. Примеры других игр

Чтобы продемонстрировать универсальность данного решения, продемонстрируем использование разработанных модулей на примере других игр.

3.4.1. Червы

Постановка задачи Требуется, используя модули *GameClass* и *GameMonad* из решения основной задачи, написать программу, определяющую, сколько игр выиграл первый игрок.

Решение

1. Поключим модули из предыдущего решения *GameClass*, *GameMonad* и *Card.hs*.
2. В модуле *Hearts.hs* определим экземпляр класса *Game* (рис. 9). Здесь *Hearts* — фиктивный тип, позволяющий избежать конфликта модулей при одновременном их подключении в программе. Определение экземпляра опишем в виде таблицы (табл. 1).
3. Функции *shootingTheMoon* и *getWinner* определяют победителя, причём если победитель уже известен, то никаких вычислений производиться уже не будет: это достигается за счёт использования *связывания*.

3.4.2. Квиддич

Продемонстрируем универсальность решения основной задачи на примере не карточной игры квиддич.

Таблица 1 — Подробное определение экземпляра класса PlayGame в модуле Hearts.

Поле класса	**Значение, придаваемое ему в модуле Hearts**	**Расшифровка**
GameInfo	Scores	Массив целых чисел: список очков, заработанных каждым из четырёх игроков
getPlayer (Player n)	n	Номер игрока
getGameinfo (Scoring a)	a	Информация об игре
checks	[shootingTheMoon, getWinner]	Список проверок на выигрыш

```
data Hearts

instance PlayGame (Game Hearts) where
        type GameInfo (Game Hearts)     = Scores
        getPlayer (Player n)            = n
        getGameInfo (Scoring a)         = a
        checks                          = [shootingTheMoon, getWinner]
```

Рисунок 9 — Экземпляр класса PlayGame для монады Game в модуле Hearts.

```
data Ball = Snitch | Quaffle deriving (Eq, Show, Read)
```

Рисунок 10 — Тип данных Ball.

```
newtype EndOfGame = EndOfGame {getListOfBalls :: ([Ball],[Ball])}
```

Рисунок 11 — Новый тип EndOfGame.

Постановка задачи Требуется, используя модули *GameClass* и *GameMonad* из решения основной задачи, написать программу, определяющую, сколько игр выиграла первая команда.

Решение

1. Подключим модули из предыдущего решения *GameClass*, *GameMonad*.
2. В модуле *Quidditch.hs* опишем новый тип данных *Ball* (рис. 10).
3. Обернём пару списков очков каждой команды в новый тип (рис. 11).
4. В модуле *Quidditch.hs* определим экземпляр класса *Game* (рис. 12). Более подробное определение приводится в таблице (табл. 2).
5. Снова вводим фиктивный тип *Quidditch*.
6. Функции, входящие в массив *checks*, работают по тому же принципу, что и в предыдущих случаях.

3.5. Тестирование

Тестирование будет проводиться при помощи библиотеки модульного тестирования *HUnit* [5]. Данная библиотека позволяет легко создавать, изменять и запускать тесты. При запуске программа выведет количество тестовых случаев, ошибок, а так же неудач, связанных,

Таблица 2 — Подробное определение экземпляра класса PlayGame в модуле Quidditch.

Поле класса	**Значение, придаваемое ему в модуле Quidditch**	**Расшифровка**
GameInfo	EndOfGame	Пара списков мячей, которые приносили очки командам
getPlayer (Player n)	n	Номер игрока
getGameinfo (Scoring a)	a	Информация об игре
checks	[fastEndOfGame, standartEndOfGame, rareCase]	Список проверок на выигрыш

```
data Quidditch

instance PlayGame (Game Quidditch) where
        type GameInfo (Game Quidditch)      = EndOfGame
        getPlayer (Player n)                = n
        getGameInfo (Scoring a)             = a
        checks                              = [ fastEndOfGame
                                              , standartEndOfGame
                                              , rareCase ]
```

Рисунок 12 — Экземпляр класса PlayGame для монады Game в модуле Quidditch.

к примеру, с ошибками входных данных для тестов либо с появлением непредусмотренного программой случая. Для начала составим текстовые файлы, содержащие входные данные для тестирования и вручную определим результаты. Файл *poker.txt* был взят из интернет-сборника задач *Project Euler* [4]. Фалы *hearts.txt* и *quidditch.txt* были составлены вручную.

В файле *HUtest.hs* опишем тестовые случаи (рис. 13). Каждому случаю в качестве параметров передадим входные данные, ожидаемый результат, а также сообщение, выводимое в случае ошибки. Сгруппируем тесты в список и передадим их в в функцию *main* (рис. 14).

Результат работы программы:

```
$ ./HUtest
Cases: 3 Tried: 3 Errors: 0 Faillures: 0
```

Заключение

В ходе работы над задачей было получено два различных решения: в чисто функциональном стиле и с применением монад. Оба решения дают ответ на вопрос задачи. Проанализировав результаты,

```
msgPoker = "Should be 376 for the first player"
pokerTest = TestCase $ (words <$> readFile "poker.txt") >>=
                assertEqual msgPoker 376 ∘ extractPokerWinner

msgHearts = "Should be 1 for the first player"
heartsTest = TestCase $ (lines <$> readFile "hearts.txt") >>=
                assertEqual msgHearts 1 ∘ extractHeartsWinner

msgQuidditch = "Should be 4 for the first player"
quidditchTest = TestCase $ (lines <$> readFile "quidditch.txt") >>=
                assertEqual msgQuidditch 6 ∘ extractQudditchWinner
```

Рисунок 13 — Тестовые случаи.

```
testAll = TestList [TestLabel "PokerTest" pokerTest
                        , TestLabel "QuidditchTest" quidditchTest
                        , TestLabel "HeartsTest" heartsTest]

main = runTestTT testAll
```

Рисунок 14 — Передача тестов в основную функцию.

приходим к выводу о преимуществе второго варианта решения задачи:

1) монады дают возможность получения данных с контекстом выигрыша, не выходя при этом за рамки функционального стиля;
2) классы типов увеличивают повторную используемость кода в рамках параметрического полиморфизма.

Список литературы

1. A Gentle Introduction to Haskell, Version 98. — URL: https://www.haskell.org/tutorial/monads.html (дата обр. 22.05.2015).
2. Haskell Wiki: Empty Data Declaration. — URL: https://ghc.haskell.org/trac/haskell-prime/wiki/EmptyDataDecls (дата обр. 30.05.2015).
3. Haskell.org. 7.4. Extensions to data types and type synonyms. Chapter 7. GHC Language Features. — URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/data-type-extensions.html#nullary-types (дата обр. 30.05.2015).
4. HUnit: A unit testing framework for Haskell. — URL: https://hackage.haskell.org/package/HUnit (дата обр. 30.05.2015).
5. Project Euler. Poker Hands. — URL: https://projecteuler.net/problem=54 (дата обр. 30.05.2015).
6. The Haskell Programming Language. — URL: https://wiki.haskell.org/GHC/Type_families (дата обр. 22.05.2015).
7. Wikipedia. The Free Encyclopedia. Functional Programming. — URL: https://en.wikipedia.org/wiki/Functional_programming (дата обр. 22.05.2015).
8. Wikipedia. The Free Encyclopedia. Lambda Calculus. — URL: https://en.wikipedia.org/wiki/Lambda_calculus (дата обр. 22.05.2015).
9. *Душкин Р. В.* Функциональное программирование на языке Haskell. — М. : ДМК Пресс, 2007.
10. *Липовача М.* Изучай Haskell во имя добра! — М. : ДМК Пресс, 2012.

11. Репозиторий с исходными кодами. — URL: https://github.com/Valoisa/Coursework (дата обр. 30.05.2015).