

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

**ВЕРИФИКАЦИЯ АЛГОРИТМА ПОСТРОЕНИЯ БАЗИСА ГРЁБНЕРА И ЕГО
ПРИМЕНЕНИЙ В СИСТЕМАХ КОМПЬЮТЕРНОЙ АЛГЕБРЫ НА ЯЗЫКЕ
ИНТЕРАКТИВНОГО ДОКАЗАТЕЛЬСТВА ТЕОРЕМ LEAN**

Автор: Федоров Глеб Владимирович _____

Направление подготовки: 01.03.02 Прикладная
математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Трифанов А.И., канд. физ.-мат. наук _____

Санкт-Петербург, 2022 г.

Обучающийся Федоров Глеб Владимирович

Группа М34351 Факультет ИТиП

Направленность (профиль), специализация

Информатика и программирование

Консультанты:

а) Гилев П.А., без звания

ВКР принята «_____» _____ 20____ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «» июня 2022 г.

Секретарь ГЭК Павлова О.Н.

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

УТВЕРЖДАЮ

Руководитель ОП
проф., д.т.н. Парфенов В.Г. _____
« ____ » _____ 20 ____ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Федоров Глеб Владимирович

Группа М34351 **Факультет** ИТиП

Квалификация: Бакалавр

Направление подготовки: 01.03.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы: Информатика и программирование

Тема ВКР: Верификация алгоритма построения базиса Грёбнера и его применений в системах компьютерной алгебры на языке интерактивного доказательства теорем *lean*

Руководитель Трифанов А.И., канд. физ.-мат. наук, ординарный доцент Университета ИТМО

2 Срок сдачи студентом законченной работы до: «31» мая 2022 г.

3 Техническое задание и исходные данные к работе

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

5 Перечень графического материала (с указанием обязательного материала)

Графические материалы и чертежи работой не предусмотрены

6 Исходные материалы и пособия

а) -

7 Дата выдачи задания «22» октября 2022 г.

Руководитель ВКР _____

Задание принял к исполнению _____ «22» октября 2022 г.

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Федоров Глеб Владимирович

Наименование темы ВКР: Верификация алгоритма построения базиса Грёбнера и его применений в системах компьютерной алгебры на языке интерактивного доказательства теорем `lean`

Наименование организации, в которой выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработать программное обеспечение на языке `lean4`, вычисляющее базис Грёбнера. Код данного программного обеспечения должен быть верифицирован на том же языке.

2 Задачи, решаемые в ВКР:

- а) Реализация упорядочения `lex` и `grlex` для мономов. Доказательство, что реализованные доказательства являются линейными упорядочениями на множестве мономов;
- б) Реализация алгоритма деления. Доказательство корректности алгоритма;
- в) Реализация алгоритма построения базиса Грёбнера (алгоритм Бухбергера). Доказательство корректности алгоритма;
- г) Реализация возможности пользовательского взаимодействия с кодом.

3 Число источников, использованных при составлении обзора: 0

4 Полное число источников, использованных в работе: 4

5 В том числе источников по годам:

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	0	1	0	3	0

6 Использование информационных ресурсов Internet: да, число ресурсов: 1

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	Раздел работы
Пакет <code>tabularx</code> для чуть более продвинутых таблиц	??, Приложения А, ??
Пакет <code>biblatex</code> и программное средство <code>biber</code>	Список использованных источников

8 Краткая характеристика полученных результатов

9 Гранты, полученные при выполнении работы

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

-

Обучающийся Федоров Г.В. _____

Руководитель ВКР Трифанов А.И. _____

« ____ » _____ 20 ____ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1. Инструмент интерактивного доказательства теорем <code>lean4</code>	6
1.1. Введение	6
1.2. Основания	6
1.3. Процесс доказательства в <code>lean4</code>	7
1.4. Программирование в <code>lean4</code>	10
Выводы по главе 1	12
2. Теория базисов Грёбнера	13
2.1. Основные определения	13
2.2. Деление полиномов от одной переменной	14
2.3. Упорядочения мономов	15
2.4. Алгоритм деления полиномов от нескольких переменных	17
2.5. Базисы Грёбнера и алгоритм Бухбергера	19
2.6. Задача принадлежности идеалу	21
Выводы по главе 2	21
3. Реализация	22
3.1. Реализация полинома	22
3.2. Полиномиальные идеалы	24
3.3. Мономиальные упорядочения	25
3.4. Деление полиномов от нескольких переменных	28
3.5. Алгоритм Бухбергера	31
3.6. Консольная утилита	31
3.7. Тестирование	31
Выводы по главе 3	32
ЗАКЛЮЧЕНИЕ	33
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	34
ПРИЛОЖЕНИЕ А. Пример приложения	35

ВВЕДЕНИЕ

В данном разделе размещается введение.

ГЛАВА 1. ИНСТРУМЕНТ ИНТЕРАКТИВНОГО ДОКАЗАТЕЛЬСТВА

ТЕОРЕМ LEAN4

[1–4] В данной главе будет приведён обзор интерактивного помощника доказательства теорем lean4.

1.1. Введение

Язык lean – это функциональный язык программирования, разработанный Microsoft Research. До четвёртой версии его можно было использовать, в основном, как интерактивный помощник доказательства теорем, но сейчас стало возможным использовать его как язык программирования общего назначения. Доказательством этому служит тот факт, что текущая версия компилятора lean написана на lean.

Данный язык, помимо Microsoft, поддерживает довольно большое сообщество разработчиков и математиков. Так, например, на lean3 была разработана библиотека mathlib – проект по формализации математики, который позволяет переиспользовать различные математические факты в своей программе.

На момент написания данной работы lean4 находится в финальной стадии разработки, и уже пригоден для использования. Библиотека mathlib так же находится в финальной стадии переписывания на четвёртую версию.

1.2. Основания

Базисом для системы типов языка lean является исчисление конструкций – теории типов на основе λ -исчисления высшего порядка с зависимыми типами. Синтаксис термов в данном исчислении может быть записан в следующем виде:

$$e ::= T \mid P \mid x \mid e \mid \lambda x : e. e \mid \forall x : e. e,$$

где

- а) T – это типы.
- б) P – это тип, к которому относятся все утверждения.

в) x – переменная.

Исчисление конструкций позволяет строить суждения о типах. Так, например, выражение

$$x_1 : \alpha_1 \dots x_n : \alpha_n \vdash y : \beta$$

можно прочесть как импликацию: если переменные $x_1 \dots x_n$ имеют типы $\alpha_1 \dots \alpha_n$, то терм t имеет тип β .

Суждение Ψ является допустимым, если из контекста, аксиом и утверждений, полученных при помощи правил вывода, можно вывести Ψ . Подробнее о правилах вывода в исчислении конструкций можно прочесть в книге Бенжамина Пирса ”Типы в языках программирования. 2009” на странице 357.

Важным свойством исчисления конструкций является наличие зависимых типов – типов, которые зависят от некоторого значения. Примером такого типа в `lean` является `Vector`, который зависит от натурального числа. То есть `Vector n` – это вектор размерности n . Наличие зависимых типов позволяет писать программы, которые корректны по построению. Например, можно реализовать функцию `sort`, которая имеет тип $Vector\ n \rightarrow Vector\ n$, то есть функция возвращает вектор той же длины, что и на входе. Более того, можно написать функцию `sort` таким образом, чтобы её тип говорил, что возвращаемый вектор отсортирован.

К сожалению, написание подобных функций – это довольно трудоёмкий процесс. Ведь, помимо реализации самого алгоритма необходимо предоставить полный вывод того, что суждение о возвращаемом типе является допустимым.

1.3. Процесс доказательства в `lean4`

Доказательство в `lean` начинается с ключевого слова `by`, которое переводит язык в режим тактик. В данном режиме в среде разработки появляется окно, в котором описано текущее состояние доказательства, разделённое на две части – контекст доказательства (называемую так же `Tactic state`) и текущую

цель доказательства. В листинге 1 приведены две функции – f , принимающую два натуральных числа и доказательство того, что их сумма больше пяти, и g , вызывающую функцию f с аргументами 6, x и доказательством того, что $6 + x > 5$.

Листинг 1 – Пример доказательства

```
def f (x y: Nat) (h: x + y > 5) := x
def g (x: Nat): Nat := f 6 x (
    by
      induction x
      simp
      rename_i n h
      simp at h
      simp [Nat.add_succ]
      rw [Nat.succ_eq_add_one
        ]
      have six_add_n_lt_y : 6
        + n < 6 + n + 1
      simp
      exact Nat.lt_trans h
        six_add_n_lt_y
    )
-----
Tactic state
x: ℕ
⊢ 6 + x > 5
```

Для того, чтобы переиспользовать доказательства, используется ключевое слово `theorem`. Пример такого переиспользования доказательства – теорема `Nat.add_succ` из стандартной библиотеки `lean`, определение которой можно увидеть в листинге 2.

Правилами вывода служат тактики. Опишем некоторые из них на примере доказательства из листинга 1.

Наиболее частым типом доказательства является доказательство того, что объекты A и B находятся в некотором отношении. Если отношение ре-

Листинг 2 – Примеры теорем из стандартной библиотеки lean

```
theorem succ_add : ∀ (n m : Nat), (succ n) + m = succ (
  n + m)
| _, 0    => rfl
| n, m+1 => congrArg succ (succ_add n m)

theorem add_succ (n m : Nat) : n + succ m = succ (n + m)
:=
rfl
```

флексивно, и объект A возможно тривиальным образом перестроить в объект B , то тактика `rfl` закроет цель.

Тактика `induction` использует метод математической индукции по переданному ей аргументу. Результат применения тактики можно увидеть в листинге 2.

Листинг 3 – induction

```
Tactic state
case zero
⊢ 6 + Nat.zero > 5
case succ
n+: ℕ
n_ih+: 6 + n+ > 5
⊢ 6 + Nat.succ n+ > 5
```

Если в результате применения тактики появляются новые переменные, `lean` даст им имена автоматически и отобразит их в `Tactic state`. Обычно, полученные имена – это имена переменных из применённой теоремы или функции. Для того, чтобы доказательство не ломалось при каких-либо изменениях имён вне текущей теоремы, автоматически сгенерированные имена использовать запрещено, что символизирует крест в конце имени. Чтобы явно использовать переменную в доказательстве её нужно переименовать, для чего используется тактика `rename _i`.

Тактика `simp` преобразует текущую цель при помощи всех теорем, помеченных тегом `@[simp]`, и закрывает её, если возможно. Так, цель $6 + \text{Nat.zero} > 5$ упрощается в $6 > 5$ что, очевидно, верно. Данная тактика может принимать аргументы, которые она так же использует в процессе упрощения.

Тактика `rw` переписывает текущую цель при помощи переданных ей аргументов, которыми могут быть, например, теоремы и функции. Так, цель $\vdash 5 < \text{Nat.succ}(6 + n)$ при помощи `rw[Nat.succ_eq_add_one]` будет переписана в $\vdash 5 < 6 + n + 1$.

Тактика `have` связывает тип и/или значение с именем. Так, мы ввели утверждение `have six_add_n_lt_y : 6 + n < 6 + n + 1`, после чего доказали его тактикой `simp`. После чего данное утверждение стало возможно использовать в доказательстве основной теоремы.

Тактика `exact` пытается закрыть цель при помощи переданной ей теоремы. В листинге 1 мы закрыли цель по транзитивности отношения `<` на натуральных числах – `exact Nat.lt_trans h six_add_n_lt_y`.

В данном параграфе были описаны далеко не все тактики, использованные в данной работе. Подробное описание основных тактик в языке `lean` можно найти на сайте URL: https://leanprover.github.io/theorem_proving_in_lean4/tactics.html.

1.4. Программирование в `lean4`

Процесс программирования на `lean` очень похож на программирование на многих других функциональных языках программирования, таких как `Haskell` и `OCaml`. В данном параграфе будет дано описание основных конструкций, использованных в этой работе.

Для описания объектов предметной области удобно группировать объекты разных типов в один. Данная функциональность реализована при помощи

структур, которые объявляются при помощи ключевого слова `structure`. Пример объявления структуры можно увидеть в листинге ...

Листинг 4 – Point

```
structure Point where
  x: Nat
  y: Nat
```

В `lean` структура может не содержать ни единого поля. При этом структуры без полей `A` и `B` будут разными типами. Такие структуры удобно использовать в качестве меток для типов. Например, в данной работе структуры без полей `Lex` и `GrLex` используются для выбора способа упорядочения мономов в полиноме.

Наконец, тип поля может быть сколь угодно сложным. Например, в листинге ... представлена структура, содержащая натуральное число x и доказательство, что $x > 5$. Благодаря такой возможности, становится более удобно формулировать спецификацию типа функции.

Листинг 5 – Difficult type example

```
structure T where
  x: Nat
  h: x > 5
```

В языке `lean` присутствует способ наложения ограничений на тип, известный как класс типов (далее `typeclass`). Рассмотрим, например, функцию `print`, которая выводит строку на экран. Тогда, чтобы вывести значение своего типа `MyType` на экран, необходимо сначала преобразовать его в строку, а потом вызвать функцию `print`. Либо создать свою функцию `print`, которая принимает не строку, а `MyType`, переводит его в строку, а потом выводит. В обеих реализациях есть минус – нужно писать много дополнительного кода. Этого можно избежать, если сделать функцию `print` полиморфной, а на тип принимаемого

аргумента наложить ограничение, что для него существует экземпляр класса типа `ToString`.

Описание класса типа `ToString` и пример создания его экземпляра можно увидеть в листинге ...

Листинг 6 – Print

```
class ToString ( $\alpha$ : Type) where
  toString x :  $\alpha \rightarrow String$ 

structure MyType

instance: ToString MyType where
  toString x := "Hello, MyType"

def print [ToString  $\alpha$ ] (x:  $\alpha$ ) := ...
```

Свойства, которые могут быть наложены при помощи `typeclass` могут быть самыми разными. Например, в данной работе используется класс типов `LinearOrder α` , который говорит, что для типа α существует линейное упорядочение.

Выводы по главе 1

Вывод: в данной главе был проведён обзор теории, которая стоит за системой типов, а так же были описаны основные аспекты доказательства и программирования на язык `lean4`.

ГЛАВА 2. ТЕОРИЯ БАЗИСОВ ГРЁБНЕРА

В данной главе будет проведён обзор основных понятий теории колец многочленов. А именно, будет приведено понятие идеала, рассмотрена задача принадлежности многочлена идеалу, будет описан алгоритм деления и алгоритм Бухбергера.

2.1. Основные определения

Будем называть вектором степеней конструкцию следующего вида

$$\alpha = (\alpha_1 \dots \alpha_n), \alpha_i \in \mathbb{N}. \quad (1)$$

Назовём вектором переменных следующий вектор

$$x = (x_1 \dots x_n). \quad (2)$$

Мономом от переменных $x_1 \dots x_n$ называется конструкция следующего вида

$$x^\alpha = (x_1^{\alpha_1} \dots x_n^{\alpha_n}). \quad (3)$$

Полиномом f с коэффициентами из поля K , называется конечная линейная комбинация мономов, которая записывается следующим образом

$$f = \sum_{\alpha} c_{\alpha} * x^{\alpha}, c_{\alpha} \in K. \quad (4)$$

Множество R с операциями $+$ и $*$ называется коммутативным кольцом с единицей, если R замкнуто относительно этих операций, и $\forall a, b, c \in R$ выполнены следующие аксиомы:

- а) Коммутативность сложения — $a + b = b + a$.
- б) Ассоциативность сложения — $a + (b + c) = (a + b) + c$.
- в) Существует нейтральный элемент относительно сложения — $0 + a = a + 0 = a$.

г) Существует обратный элемент относительно сложения —

$$\exists d \in R : d + x = x + d = 0.$$

д) Коммутативность умножения — $a * b = b * a$.

е) Ассоциативность умножения — $a * (b * c) = (a * b) * c$.

ж) Существует нейтральный элемент относительно умножения —

$$1 * a = a * 1 = a.$$

и) Левая дистрибутивность — $a * (b + c) = a * b + a * c$.

к) Правая дистрибутивность — $(a + b) * c = a * c + b * c$.

Множество всех полиномов от переменных $x_1 \dots x_n$ над полем K будем обозначать как $K[x_1 \dots x_n]$. Заметим, что данное множество удовлетворяет аксиомам кольца.

Подмножество $I \subset K[x_1 \dots x_n]$ называется идеалом кольца, если выполнены следующие условия:

а) $0 \in I$;

б) если $f, g \in I$, то $f + g \in I$;

в) если $f \in I$ и $h \in K[x_1 \dots x_n]$, то $hf \in I$.

Пусть $f_1 \dots f_s$ — набор полиномов от нескольких переменных, тогда множество $\langle f_1 \dots f_s \rangle = \{\sum_i^s h_i * f_i | h_1 \dots h_s \in K[x_1 \dots x_n]\}$ является идеалом в $K[x_1 \dots x_n]$, а полиномы $\langle f_1 \dots f_s \rangle$ называются образующими идеала.

2.2. Деление полиномов от одной переменной

Теория базисов Грёбнера опирается на операцию деления многочленов. Перед тем, как определить алгоритм деления в кольце полиномов от нескольких переменных, рассмотрим алгоритм в кольце полиномов от одной переменной.

Определим функцию, возвращающую старший член полинома. Пусть $f = \alpha_0 x^m + \alpha_1 x^{m-1} \dots + a_m$, где $a_i \in K$, $a_0 \neq 0$. Тогда

$$LT(f) = \alpha_0 x^m$$

называется старшим членом полинома f . Для $K[x]$ данная функция кажется более, чем естественной, но для $K[x_1 \dots x_n]$ будет дано несколько иное определение.

Опишем алгоритм деления в $K[x]$. Пусть $g \in K[x]$ – ненулевой полином. Тогда любой полином $f \in K[x]$ может быть записан в виде

$$f = qg + r,$$

где $q, r \in K[x]$ и либо $r = 0$, либо $\deg(r) < \deg(g)$, причём q и r определены однозначно. Многочлены q и r могут быть найдены следующим алгоритмом.

Листинг 7 – Деление в $K[x]$

```
function Divide( $q, f$ )
   $q = 0$ ;
   $r = f$ ;
  while  $r \neq 0$  &  $LT(g) | LT(r)$  do
     $q = q + LT(r)/LT(g)$ 
     $r = r - (LT(r)/LT(g))g$ 
  end while
  return  $q, r$ 
end function
```

Доказательство данного алгоритма можно найти в книге. Формального доказательства данного алгоритма не проводилось, так как он является частным случаем алгоритма деления в $K[x_1 \dots x_n]$.

2.3. Упорядочения мономов

После прочтения предыдущего параграфа может сложиться впечатление, что алгоритм деления полиномов из $K[x]$ будет работать и в $K[x_1 \dots x_n]$. К сожалению, это не совсем так.

Заметим, что в кольце $K[x]$ у мономов есть естественный порядок – по степеням, которые являются натуральными числами. Но в $K[x_1 \dots x_n]$ степень монома – не число, а вектор натуральных чисел. Поэтому, в данном параграфе

будет определено упорядочение мономов в $K[x_1 \dots x_n]$, и будут рассмотрены основные упорядочения.

Мономиальным упорядочением на $K[x_1 \dots x_n]$ называется любое бинарное отношение \leq на $\mathbb{Z}_{\geq 0}^n$, обладающее следующими свойствами:

- а) \leq является линейным упорядочением на $\mathbb{Z}_{\geq 0}^n$;
- б) если $\alpha \leq \beta$ и $\gamma \in \mathbb{Z}_{\geq 0}^n$, то $\alpha + \gamma \leq \beta + \gamma$;
- в) Отношение \leq вполне упорядочивает $\mathbb{Z}_{\geq 0}^n$. То есть, в каждом подмножестве множества $\mathbb{Z}_{\geq 0}^n$ есть минимальный элемент.

Условие а нужно, чтобы мы могли для любого полинома расположить мономы в порядке \leq . То есть, для любой пары мономов x^α, y^β должно выполняться одно из следующих соотношений

$$x^\alpha < y^\beta, x^\alpha = y^\beta, x^\alpha > y^\beta.$$

Условие б нужно, чтобы упорядоченность мономов была согласована с аксиомами кольца. Заметим, что задача умножения полинома на полином естественным образом сводится к задаче умножения полинома на моном. Но, если упорядочение не удовлетворяет свойству б, то умножение монома на старший член полинома не будет являться старшим членом. Примером упорядочения, не удовлетворяющего свойству б, может служить *max* упорядочение: $\leq_{\max}: \alpha \leq_{\max} \beta \Leftrightarrow \max(\alpha) \leq \max(\beta) \vee \max(\alpha) = \max(\beta) \wedge \alpha \leq_{\text{lex}} \beta$. Тогда $x^2y^3 \leq_{\max} xyz^5$, но $x^2y^3 * y^8 \geq_{\max} xyz^5 * y^8$.

Условие в необходимо для доказательства корректности алгоритмов в следующих параграфах. А именно, критерий остановки алгоритма будет основан на том, что старший член полинома убывает на каждом шаге алгоритма.

В данной работе будут рассмотрены два упорядочения:

- а) Лексикографическое упорядочение — $a \leq_{\text{lex}} b \Leftrightarrow$ первая ненулевая координата вектора $b - a$ положительна;

- б) Градуированное лексикографическое упорядочение – $a \leq_{grlex} b \Leftrightarrow |a| < |b| \vee (|a| = |b| \wedge a \leq_{lex} b)$.

Доказательства того, что эти упорядочения удовлетворяют условиям, определённым выше, будут предоставлены в следующей главе.

Функция ЛТ для полиномов из $K[x_1 \dots x_n]$ определяется аналогично с функцией ЛТ для $K[x]$, с той лишь разницей, что является согласованной с некоторым мономиальным упорядочением. Например, рассмотрим полином $f = y^3z^4 + xyz$. Тогда $LT(f, lex) = xyz$, а $LT(f, grlex) = y^3z^4$.

2.4. Алгоритм деления полиномов от нескольких переменных

Алгоритм деления в $k[x_1 \dots x_n]$ во многом похож на алгоритм деления в $k[x]$. Основная разница, помимо упорядочения мономов, состоит в том, что мы делим не на один полином, а сразу на несколько.

Для полиномов из $K[x_1 \dots x_n]$ верно следующее утверждение. Зафиксируем некоторое мономиальное упорядочение \leq . Пусть $F = (f_1 \dots f_s)$ – упорядоченный набор полиномов из $k[x_1 \dots x_n]$. Тогда любой полином $f \in k[x_1 \dots x_n]$ может быть представлен в виде

$$f = \alpha_1 f_1 + \dots + \alpha_s f_s + r, \quad (5)$$

где $\alpha_i, r \in k[x_1 \dots x_n]$, причём либо $r = 0$, либо r есть линейная комбинация мономов, ни один из которых не делится ни на один из старших членов $LT(f_1) \dots LT(f_s)$. Причём, существует алгоритм, который строит данное представление, состоящий из двух шагов:

- а) Шаг деления – если некоторый $LT(f_i)$ делит $LT(p)$, то алгоритм деления продолжает свою работу, как в случае с одной переменной.
- б) Шаг вычисления остатка – если никакой из $LT(f_i)$ не делит $LT(p)$, то $LT(p)$ прибавляется к остатку. Алгоритм продолжает свою работу для $p - LT(p)$.

Псевдокод алгоритма можно увидеть в листинге ...

Листинг 8 – Деление в $K[x]$

```

function DivideMany( $f, f_1 \dots f_s$ )
   $a_1 = 0 \dots a_s = 0$ ;
   $r = 0$ ;
   $p = f$ ;
  while  $p \neq 0$  do
     $i = 1$ ;
     $hasDiv = false$ ;
    while  $i \leq s \wedge !hasDiv$  do
      if  $LT(f_i) \mid LT(p)$  then
         $a_i = a_i + LT(p)/LT(f_i)$ 
         $p = p - (LT(p)/LT(f_i))f_i$ 
      else
         $i = i + 1$ 
      end if
    end while
    if  $!hasDiv$  then
       $r = r + LT(p)$ 
       $p = p - LT(p)$ 
    end if
  end while
  return  $a_1 \dots a_s, r$ 
end function

```

Для доказательства корректности алгоритма необходимо:

- а) На каждом шаге выполняется равенство: $f = \sum_i^s a_i f_i + p + r$;
- б) Ни один из мономов из r не делится ни на один из старших членов $LT(f_1) \dots LT(f_s)$;
- в) Алгоритм завершает свою работу.

Доказательства данных пунктов и реализация алгоритма будут приведены в главе ”Реализация”.

Заметим, что результат деления в кольце $K[x_1 \dots x_n]$ не определён однозначно. Например, пусть $f_1 = xy + 1$ и $f_2 = y^2 - 1$. Разделив $f = xy^2 - x$ на (f_1, f_2) мы получим $xy^2 - x = y(xy - 1) + 0(y^2 - 1) + (-x - y)$. Но если мы поделим f на (f_2, f_1) , то результат будет уже другим –

$xy^2 - x = x(y^2 - 1) + 0(xy + 1) + 0$. О том, когда результат деления, определён, в некотором смысле, однозначно, будет сказано в следующих параграфах.

2.5. Базисы Грёбнера и алгоритм Бухбергера

В данном параграфе будет дано определение базиса Грёбнера.

Пусть некоторое зафиксированно мономиальное упорядочение. Тогда множество $\{g_1 \dots g_m\} \subset I$ называется базисом Грёбнера идеала I в том и только в том случае, когда старший член любого полинома $p \in I$ делится хотя бы на один старший член $LT(g_i)$. Иначе говоря, справедлива следующая формула:

$$\forall p \in I \exists g \in \{g_1 \dots g_m\} : LT(g) | LT(p).$$

К сожалению, данное определение ничего не говорит о том, как построить базис Грёбнера. Поэтому, введём конструкцию, которая используется при построении базиса Грёбнера, и при проверке, что некоторый базис G является базисом Грёбнера. Пусть $f, g \in K[x_1 \dots x_n]$ – ненулевые полиномы. Тогда, назовём S -полиномом следующую конструкцию:

$$S(f, g) = \frac{x^\gamma}{LT(f)} * f - \frac{x^\gamma}{LT(g)} * g, \quad (6)$$

где x^γ – наименьшее общее кратное мономов $LT(f)$ и $LT(g)$. S -полином специально сконструирован для сокращения старших членов полиномов, что показано в лемме 5 в книге...

Для S -полиномов справедлива следующая теорема: Критерий Бухбергера. Пусть I – полиномиальный идеал. Тогда базис $G = \{g_1 \dots g_s\}$ идеала I является базисом Грёбнера в том и только в том случае, когда для всех пар $i \neq j$ остаток от деления $S(g_i, g_j)$ равен нулю. Доказательство данного критерия можно найти в теореме 6...

Из данного критерия следует алгоритм построения базиса Грёбнера, псевдокод которого приведён в листинге 6.

Листинг 9 – Алгоритм Бухбургера

```

function BuildGroebner( $F = f_1 \dots f_s$ )
   $G = F$ ;
  while True do
     $G' = G$ ;
    for  $\forall \langle p, q \rangle, p \neq q \in G'$  do
       $S = S(p, q) \bmod G'$ ;
      if  $S \neq 0$  then
         $G = G \cup S$ 
      end if
    end for
    if  $G == G'$  then break;
    end if
  end while
  return  $G$ 
end function

```

Во-первых, покажем, что идеал, порождённый множеством $F = f_1 \dots f_s$, и идеал, порождённый множеством, которое возвращает функция BuildGroebner совпадают. Заметим, что остаток от деления S-полинома на базис идеала I лежит в I . Действительно, в формуле 5 $f \in I$, а множитель, назовём его α , при f лежит в кольце $K[x_1 \dots x_n]$. Тогда, по определению идеала, $\alpha * f \in I$. Для g справедливы те же самые рассуждения. Тогда, из замкнутости идеала относительно операции сложения, S-полином от f и g лежит в I . Далее при помощи алгоритма деления в $K[x_1 \dots x_n]$, перепишем S-полином в виде формулы 5:

$$S(f, g) = a_1 g_1 + \dots + a_s g_s + r.$$

Так как $S(f, g) \in I$ и комбинация $a_1 g_1 + \dots + a_s g_s \in I$, то и $r \in I$, что и требовалось доказать. Формальное доказательство данного факта будет приведено в главе "Реализация".

Во-вторых, нужно показать, что данный алгоритм завершает свою работу. В данной работе это не будет формально доказано, о чём будет подроб-

нее сказано в главе "Реализация". С исходным доказательством можно ознакомиться в ...

2.6. Задача принадлежности идеалу

В данной работе реализовано одно из применений базисов Грёбнера, а именно, задача принадлежности полинома идеалу. В данном параграфе будут описаны теоретические аспекты решения этой задачи.

Пусть $I = \langle g_1 \dots g_m \rangle$. Тогда, если остаток от деления многочлена p на $\{g_1 \dots g_m\}$ равен нулю, то $p \in I$. Данный факт довольно очевиден, ведь, если мы запишем p в виде формулы 5, то будет видно, что p - это линейная комбинация $\{g_1 \dots g_m\}$. Но, если остаток от деления не равен нулю, то это не значит, что многочлен не принадлежит идеалу. Ведь, как обсуждалось в параграфе 2.4, остаток от деления зависит от порядка делителей. Получается, что для произвольного базиса задачу о принадлежности идеалу не решить.

Пусть $G = \{g_1 \dots g_m\}$ - базис Грёбнера. Тогда, для любого полинома $p \in K[x_1 \dots x_n]$ остаток от деления p на G определён однозначно, доказательство чего можно найти в ... Однако, если остаток r не нулевой, то разложение по формуле 5 для полинома $p - r$ не является однозначным.

Таким образом, задача о принадлежности полинома p идеалу I решается следующим образом:

- а) Находим базис Грёбнера G идеала I .
- б) Считаем остаток от деления p на G .
- в) Если остаток равен нулю, то $p \in I$, иначе - $p \notin I$.

Выводы по главе 2

В данной главе были рассмотрены основные понятия теории колец многочленов, такие как мономиальные упорядочения, алгоритм деления многочленов, базис Грёбнера и алгоритм Бухбергера. Так же было дано описание решения задачи о принадлежности многочлена идеалу.

ГЛАВА 3. РЕАЛИЗАЦИЯ

В данной главе будет описана реализация описанной в предыдущей главе теории на языке интерактивного доказательства теорем `lean`.

3.1. Реализация полинома

В библиотеке `mathlib` уже есть реализация полинома от нескольких переменных под названием `MvPolynomial`, которая удовлетворяет всем аксиомам кольца. Но, к сожалению, это реализация явно использует аксиому выбора, что делает её неконструктивной. Иначе говоря, её возможно использовать только для доказательства теорем, но сгенерировать исполняемый код при использовании данной реализации использованию не выйдет. Поэтому в данной работе была написана собственная реализация.

Введём основные определения. Вектором степеней назовём вектор натуральных чисел длины n , упорядоченный согласно мономиальному упорядочению ord . Произведением двух векторов одинаковой длины с одинаковым упорядочением назовём их покомпонентную сумму. Возможная путаница в терминологии возникает, потому что, что данная операция будет использоваться далее при определении умножения монома на моном, а именно:

$$(\alpha_1 \dots \alpha_n) + (\beta_1 \dots \beta_n) \Leftrightarrow x_1^{\alpha_1} \dots x_n^{\alpha_n} * x_1^{\beta_1} \dots x_n^{\beta_n} = x_1^{\alpha_1 + \beta_1} \dots x_n^{\alpha_n + \beta_n}.$$

Листинг 10 – Вектор степеней

```
def Variables (n: Nat) (ord: Type) := Vector Nat n

def Variables.mul (v1 v2: Variables n ord): Variables n
  ord :=
  map2 (fun x y => x + y) v1 v2
```


В качестве поля K будет взято поле рациональных чисел. Тогда моном реализован как пара из рационального числа и вектора длины n с упорядочением ord .

Листинг 11 – Моном

```
def Monomial (n: Nat) (ord: Type) := Rat × (Variables n
  ord)

def Monomial.mul (m1 m2: Monomial n ord) : Monomial n
  ord :=
  (m1.fst * m2.fst, Variables.mul m1.snd m2.snd)
```

Полином был реализован на красно-чёрном дереве, элементами которого являются мономы с числом переменных n , упорядоченные согласно ord , причём для ord обязан существовать экземпляр класс типов `MonomialOrder (Variablesn ord n)`, иначе говоря, ord обязан быть мономиальным упорядочением. Функция `m_cmp`, определённая в файле `MonomialOrderInterface.lean`, позволяет использовать упорядочение ord как функцию сравнения для мономов. Подробнее об `MonomialOrder` будет написано в следующей главе.

Листинг 12 – Полином

```
def Polynomial (n: Nat) (ord: Type)
  [MonomialOrder $ Variables n ord] :=
  Std.RBSet (Monomial n ord) ordering.m_cmp
```

Все функции и экземпляры классов типов для монома и полинома определены в файле `Polynomial.lean`.

В процессе работы не получилось доказать, что данное определение полинома с операциями $+$ и $*$ удовлетворяют аксиомам кольца из-за довольно сложного изменения структуры красно-чёрного дерева при применении данных операций к полиномам. Чтобы было возможно проводить какие-либо доказательства, аксиомы кольца были постулированы при помощи команды `axiom`, что можно увидеть в файле `PolynomialRing.lean`.

3.2. Полиномиальные идеалы

В данной работе была взята реализация идеала из библиотеки `mathlib`. Но, так как во всех алгоритмах идёт работат со списками, нужно было дополнительно доказать, что список полиномов можно использовать в качестве образующих для идеала.

Во-первых, нужно было показать, что список полиномов в некотором смысле эквивалентен множеству.

Листинг 13 – Преобразование списка в множество

```
def asSet [MonomialOrder (Variables n ord)]
  (ps: List (Polynomial n ord)): Set (
    Polynomial n ord) := {x | x ∈ ps }

theorem listInSet [MonomialOrder (Variables n ord)]
  (l: List (Polynomial n ord)): ∀ x ∈ l
    , x ∈ asSet l := by
  intros y h
  simp [asSet]
  exact h
```

Во-вторых, нужно было реализовать функцию, которая по списку полиномов строит идеал.

Листинг 14 – Построение идеала по списку

```
def asIdeal [MonomialOrder (Variables n ord)]
  (l: List (Polynomial n ord)):
  Ideal (Polynomial n ord) := Ideal.span (
    asSet l)
```

В-третьих, нужно было показать, что исходный список содержится в построенном идеале.

Возможно, в процессе прочтения данного параграфа, возник вопрос: почему бы сразу не использовать множества, а не списки. Ответ простой – множества в `lean`, это функция, которая возвращает по элементу x значение типа

Листинг 15 – Доказательство, что список, по которому построен идеал, содержится в идеале

```
theorem listInIdeal [MonomialOrder (Variables n ord)]
  (l: List (Polynomial n ord)):  $\forall x \in l, x \in \text{asIdeal } l := \text{by}$ 
  intros y h
  rw [asIdeal, asSet]
  have in_set := listInSet l y h
  rw [asSet] at in_set
  apply Ideal.subset_span
  exact in_set
```

Prop. Иначе говоря, множество – это некоторый предикат. А раз так, то пришлось бы отдельно доказывать вычислимость функции принадлежности, которая необходима для перебора полиномов в алгоритмах. Подробнее о вычислимости будет сказано в следующем параграфе.

Отдельно хочется сказать, что идеал в `mathlib` реализован как подмодуль полукольца R по R . Поэтому нётеревость колец(условие обрыва возрастающих цепочек идеалов, далее УОВЦ) формулируется иначе. А именно, что каждый подмодуль конечно порождён. Так как УОВЦ, по-сути, отсутствует, то адаптировать оригинальное доказательство того, что алгоритм Бухбергера завершает свою работу, не вышло.

3.3. Мономиальные упорядочения

Для доказательства того, что реализованные упорядочения удовлетворяют аксиомам мономиального упорядочения, был реализован класс типов `MonomialOrder`, наследованный от классов `LinearOrder`(линейное упорядочение) и `WellFoundedRelation`(вполне упорядочивание).

В данной работе были определены два мономиальных упорядочения: лексикографическое(далее `Lex`) и градуированное лексикографическое(далее `GrLex`).

```

theorem lex_le_refl :  $\forall$  (a : Variables n order.Lex),
  Order.lex a a := by
  intro a
  let rec aux (m: Nat) (v: Variables m order.Lex) :
    Order.lex_impl v v := by
    match v with
    |  $\langle [], p \rangle$  => rw [Order.lex_impl]
      split
      simp at *
      simp at p
      simp at *
    |  $\langle x :: xs, \_ \rangle$  => rw [Order.lex_impl]
      split
      simp
      simp at *
      simp at *
      simp [Nat.le_refl]
      rename_i x1 _ x2 _ h1 h2
      have h3 := Eq.symm h1.left
      have h4 := Eq.symm h2.left
      rw [h3, h4]
      simp [Nat.le_refl]
      apply aux (m-1) (tail  $\langle x :: xs, \_ \rangle$ )

```

Листинг 18 – GrLex упорядочение

```

def Order.grlex (vs1 vs2: Variables n order.GrLex): Prop
:=
  let sum1 := elem_sum vs1
  let sum2 := elem_sum vs2
  if sum1 < sum2 then True
  else if sum1 = sum2 then if Order.lex vs1 vs2 then True
                             else False
  else False
where
  elem_sum (vs: Variables n order.GrLex): Nat :=
    List.foldl (fun x y => x + y) 0 vs.toList

```

Доказательства остальных свойств для Lex и GrLex упорядочений находятся в файле MonomialOrder.lean.

Помимо основных свойств линейного упорядочения, а именно рефлексивности, транзитивности, антисимметричности и требования, чтобы любые два элемента были сравнимы, lean требует ещё два. А именно:

- а) Отношение \leq должно быть вычислимым(decidable);
- б) Lean автоматически строит отношение $<$. Поэтому, нужна проверка согласованности отношений $<$ и \leq . А именно – $a < b \Leftrightarrow (a \leq b \wedge b \not\leq a)$.

Остановимся подробнее на первом. Как было в первой главе, в lean есть две основных разновидности типов – Type и Prop. Заметим, что свойства линейного упорядочения – это математические утверждения, то есть они принадлежат типу Prop. Следовательно, утверждение, что $\alpha \leq \beta$ также принадлежит типу Prop. Но математическое утверждение не обязано быть разрешимым. Примером тому может служить проблема останова. Но было бы довольно не практично, если бы некоторое линейное упорядочение было неразрешимым, ведь сразу перестает работать множество теорем и тактик для доказательства свойств каких-либо объектов, использующих данное упорядочение. Поэтому lean дополнительно требует доказательство того, что упорядочение разрешимо.

Чтобы доказать разрешимость можно воспользоваться следующим трюком: по пропозициональной формуле ϕ построим выражение ψ , возвращающее значение, имеющее тип из вселенной `Type`, после чего докажем, что если ψ возвращает x , то x удовлетворяет формуле ψ . В случае с упорядочениями, ϕ – это утверждения, приведённые в листингах 6 и 7, а ψ – это аналогичные функции, возвращающие `Bool` вместо `Prop`. Доказательства данного свойства для `lex` и `grlex` упорядочений можно найти в файле `MonomialOrder.lean`.

3.4. Деление полиномов от нескольких переменных

Для того, чтобы показать, что результат деления обладает свойствами, определёнными в предыдущей главе, был реализован тип `DivisionResult`, с шаблонными параметрами `divisible` – делимое, и `dividers` – делители. В данном параграфе идеал, образованный делителями, будем называть просто идеалом, без каких-либо уточнений. В структуре `DivisionResult` имеются следующие поля:

- а) `p` – частное от деления. Так как для построения базиса Грёбнера нужно уметь получать только остатки, то для простоты доказательства, будем считать частным упрощённый полином $a_1g_1 + \dots + a_sg_s$;
- б) `r` – остаток от деления;
- в) `r_as_list` – представление мономов остатка, как списка мономов;
- г) `correct_r` – доказательство того, что либо `r_as_list` – пустой список, либо каждый его элемент не делится на старшие члены делителей;
- д) `sum_eq` – утверждение, что сумма частного, промежуточного частного и остатка равна делимому;
- е) `p_in_ideal` – утверждение, что частное всегда лежит в идеале;
- ж) `r_in_ideal` – утверждение, что если делимое лежит в идеале, то и остаток лежит в идеале.

К сожалению, алгоритм деления слишком сложный, чтобы `lean` мог вычислить результат тактики `gw`, которая необходима, чтобы перейти к рассмот-

рению структуры функции в процессе доказательства. Поэтому, свойства делимости записаны в `DivisionResult`, а не вынесены в отдельные теоремы, и доказываются в самой функции. А именно, аргументами подфункции `impl` функции деления, являются доказательства того, что все свойства деления были верны на предыдущей итерации алгоритма, и на основании этих аргументов строится доказательство того, что после текущей итерации эти свойства останутся корректными. Подобный процесс доказательства, по-сути, является доказательством по индукции.

Опишем доказательство некоторых свойств результата деления.

Свойство `sum_eq` утверждает, что на каждом шаге алгоритма, верно равенство $divisible = p + quotient + remainder$, где p — это многочлен, над которым происходит операция деления в текущей итерации, $quotient$ — это промежуточное частное, а $remainder$ — остаток. На первом шаге алгоритма инициализируем $p = divisible$, а $quotient = remainder = 0$. Равенство, очевидно, верное, и `lean` может его доказать при помощи тактики `simp`. Далее, согласно описанию алгоритма деления в главе 2, в параграфе 2.3, есть два варианта изменения членов данного равенства. Во-первых, если p делится на какой-либо f_i , то происходит вычитание из p определённого полинома, назовём его $reducer$, и добавление его в частное $quotient$. Тогда необходимо доказать, что следующее равенство $divisible = p - reducer + (quotient + reducer) + remainder$ корректно. Доказательство корректности было вынесено в теорему `erase_reducer`. Во-вторых, если p не делится ни на один из f_i , то старший член p выносится в остаток. Тогда, нужно показать, что равенство $divisible = p - Polynomial.Ltp + quotient + (remainder + Polynomial.Ltp)$ справедливо. Доказательство корректности данного равенства было вынесено в теорему `erase_lt`, представленной в листинге 19.

Следующее свойство, которое будет рассмотрено в пояснительной записке, будет `p_in_ideal`. Для доказательства того, что промежуточное частное p

Листинг 19 – `erase_lt`

```

theorem erase_lt [MonomialOrder (Variables n ord)]
  (divisible p quotient remainder:
    Polynomial n ord)
  (sum_eq: divisible = p + quotient +
    remainder)
  : divisible = p - Polynomial.Lt p +
    quotient + (remainder + Polynomial
      .Lt p) :=
by
  have h: remainder + Polynomial.Lt p = Polynomial.Lt p
    + remainder :=
    add_comm remainder (Polynomial.Lt p)
  have h2: -Polynomial.Lt p + quotient = quotient + -
    Polynomial.Lt p :=
    add_comm (-Polynomial.Lt p) quotient
  rw [h, sub_eq_add_neg, add_assoc, add_assoc, add_comm
    , ←add_assoc, ←
      add_assoc, add_comm, h2, add_comm, add_assoc,
      add_comm,
      add_assoc, add_assoc, add_left_neg, add_zero,
      add_comm]
  exact sum_eq

```

, находится в идеале необходимо, во-первых, нужно доказать, что после предыдущей итерации промежуточное частное находится в идеале, и, во-вторых, что полином *reducer*, который мы будем прибавлять к p , так же лежит в идеале. Если оба свойства соблюдены, то, по свойству замкнутости идеала относительно сложения, полином $p + reducer$ будет лежать в идеале. Приступим к доказательству. Так как на первом шаге алгоритма $p = 0, p \in I$ по определению идеала. Далее нужно показать, что $reducer \in I$. В коде *reducer* - это конструкция $LT(p)/LT(f_i) * f_i$ из листинга 8. Так как $f_i \in I$, то, по свойству домножения элемента на идеал, $reducer \in I$.

Сам алгоритм деления, и доказательство всех его свойств можно посмотреть в.

3.5. Алгоритм Бухбергера

3.6. Консольная утилита

Одной из целей данной работы была реализация взаимодействия пользователя с программой. Для выполнения данной цели была написана простая консольная утилита.

Утилита работает следующим образом: до тех пор, пока не была введена команда `exit`, программа будет ожидать пользовательский ввод. После того, как пользователь ввёл строку, заканчивающуюся символом перевода строки, происходит парсинг команды и её аргументом. Если команда и её аргументы были успешно распаршены, начинается выполнение команды, иначе – пользователь получит сообщение об ошибке.

Поддержаны следующие команды для работы с полиномами:

- а) `set_n` – устанавливает число переменных в полиномах. По умолчанию работа происходит с полиномами от трёх переменных;
- б) `simp` – принимает название мономиального упорядочения *ord* и набор полиномов. Возвращает набор упрощённых полиномов, упорядоченных согласно *ord*;
- в) `is_in` – принимает многочлен *p* и набор полиномов *ps*. Проверяет, принадлежит ли полином *p* идеалу $\langle ps \rangle$;
- г) `groebner` – принимает название мономиального упорядочения *ord* и набор полиномов *ps*. Возвращает базис Грёбнера идеала $\langle ps \rangle$ для *ord* упорядочения.

Реализация и парсинг команд находится в файле `Interactive.lean`. Пользовательский ввод-вывод реализован в файле `Main.lean`. Парсинг многочленов реализован в файле `Parser.lean`.

3.7. Тестирование

В языке `lean` нет стандартных средств для тестирования кода. Поэтому были написаны стандартные функции, используемые при тестировании –

AssertEq, проверяющая, возвращающая строку Ok, если переданные ей аргументы равно, AssertNEq, возвращающая Ok, если аргументы не равны и AssertTrue, возвращающая Ok, если переданный ей булевый аргумент имеет значение true.

В работе тестировались: мономиальные упорядочения, парсинг полиномов, операции сложения, умножения, деления и алгоритм построения базиса Грёбнера. Тестирование проводилось на полиномах от трёх переменных.

В параграфе Консольная утилита была упомянута возможность указывать число переменных. Она так же была протестирована.

Все тесты и вспомогательные функции находятся в директории tests.

Выводы по главе 3

ЗАКЛЮЧЕНИЕ

В данном разделе размещается заключение.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Буздалов М. В.* Генерация тестов для олимпиадных задач по программированию с использованием генетических алгоритмов // Научно-технический вестник СПбГУ ИТМО. — 2011. — 2(72). — С. 72–77.
- 2 *Buzdalov M., Doerr B., Kever M.* The Unrestricted Black-Box Complexity of Jump Functions // Evolutionary Computation. — 2016. — Accepted for publication.
- 3 *Buzdalov M., Shalyto A.* Hard Test Generation for Augmenting Path Maximum Flow Algorithms using Genetic Algorithms: Revisited // Proceedings of IEEE Congress on Evolutionary Computation. — 2015. — P. 2121–2128.
- 4 *Doerr B., Doerr C.* Optimal Parameter Choices Through Self-Adjustment: Applying the 1/5-th Rule in Discrete Settings [Электронный ресурс]. — 2015. — URL: <http://arxiv.org/abs/1504.03212>.

ПРИЛОЖЕНИЕ А. ПРИМЕР ПРИЛОЖЕНИЯ