Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа компьютерных технологий и информационных систем

Отчёт по лабораторной работе № 3

Дисциплина: Компьютерная алгебра.

Выполнили студенты гр. 5130901/10101 _____ Д.Л. Симоновский
(подпись)
гр. 5130901/10101 _____ М.Т. Непомнящий
(подпись)
гр. 5130901/10202 _____ И.Н. Солодовник
(подпись)
гр. 5130901/10201 _____ Я.А. Балашов
(подпись)

Руководитель _____ И.А. Малышев
(подпись)

“06” декабря 2023 г.

Санкт-Петербург
2023

Оглавление

1. Постановка задачи:

Разработать программную библиотеку функций для обработки символьных представлений математических объектов, имеющих заданную алгебраическую структуру.
Библиотека должна обеспечивать:

1) Создание/удаление объекта.
2) (Пере-)определение свойств объекта.
3) Копирование объекта.
4) Построение выражений, содержащих объекты и их свойства.
5) Редукция (упрощение) выражений, содержащих объекты и их свойства.

Алгебраическая структура – моноид.

2. Теория:

Первым делом приведем необходимую теоретическую информацию.
Всё начинается с алгебраической структуры, это пара из множества и замкнутой бинарной операции.
Бинарная операция – операция, применяемая к двум элементам и порождающая новый элемент. Замкнутой операция становится, если порожденный элемент является частью исходного множества.
Примером алгебраической структуры может служить сложение целых чисел. В этом случае исходное множество является множество целых чисел, а бинарной операцией сложение. Очевидно, что при сложении целых чисел мы получаем целое число, что значит, что эта операция является замкнутой.
В качестве не алгебраической структуры служит бинарная операция деления на множестве целых чисел т.к. в результате деления может быть не целое число, что значит, что операция не является замкнутой, а значит это не алгебраическая структура.
После алгебраической структуры переходим к полугруппе – это таже самая алгебраическая структура, но помимо множества и замкнутой бинарной операции у нас появляется свойство ассоциативности. Т.е. для элементов (a, b, c) на множестве (S) и операции (°) должно быть справедливо следующее равенство (a ° b) ° c = a ° (b ° c).
Примерами может случить конкатенация строк. В качестве множества будет множество строк, в качестве бинарной операции будет конкатенация, очевидно, что эта операция является ассоциативной, а значит это полугруппа.
И наконец после полугруппы мы переходим к моноиду. Моноид это полугруппа с нейтральным элементом. Нейтральный элемент в контексте бинарных операций и моноидов — это элемент множества, который, когда применяется к бинарной операции, не изменяет другой элемент. Другими словами, если (e) - нейтральный элемент, и (a) - произвольный элемент множества, то выполняется: e ° a = a ° e = a.
Примером моноида является логическое И на множестве булевых значений. Очевидно, что нейтральном элементом в этом случае является элемент true, свойство ассоциативности выполняется, а также операция явно является замкнутой, из чего можно сделать вывод, что данный пример – моноид.

3. Программная реализация:

Перейдем к практической реализации.
Создадим класс моноида. На следующем скриншоте приведен код для создания моноида и get методы для полей класса:

```python
class Monoid:
    def __init__(self, elements, mult_function):
        self.__elements = elements
        self.__mult_function = mult_function

    def get_elements(self):
        return self.__elements

    def get_mult_function(self):
        return self.__mult_function

    def __copy__(self):
        return Monoid(self.__elements, self.__mult_function)
```

Рис. 1. Конструктор моноида.

Данная реализация принимает на вход множество элементов и бинарную операцию. Это означает что в этой реализации нет возможности передать бесконечное множество целых чисел, однако из этого появляются некоторые интересные возможности, которые мы рассмотрим позднее.

Основной функцией для нас будет «умножение» т.е. применение бинарной операции:

```python
    def multiplication(self, element_1, element_2):
        """
        Функция умножения, проверяет корректность введенных элементов,
        после чего выполняет умножение, проверяя результат
        :param element_1: Первый множитель
        :param element_2: Второй множитель
        :return: Результат умножения или строку ошибки
        """
        if element_1 not in self.__elements or element_2 not in self.__elements:
            raise Exception(f"[!] Incorrect input {element_1} or {element_2} not in {self.__elements}")
        result = self.__mult_function(element_1, element_2)
        if result not in self.__elements:
            raise Exception(
                f"[!] Incorrect monoid: {element_1} * {element_2} = {result}\n[!] But {result} not in {self.__elements}"
            )
        return result
```

Рис. 2. Функция применения бинарной операции.

Данная функция проверяет входные данные на принадлежность введенном множеству, а также после выполнения бинарной операции идет проверка на принадлежность результата множеству, чтоб проверить, что операция действительно является замкнутой.

А теперь перейдем к тому моменту, для чего была выбрана именно такая реализация с замкнутым множеством, а именно проверка свойства замкнутости у операции умножения, а также свойство ассоциативности от полугруппы:

```python
def check_monoid_multiplication(self):
    """
    Проверяет все возможные перемножения в множестве, дабы убедиться
    что они не выходят за само множество сложность O(n^3)
    :return: Ошибку или True
    """
    # Умножение двух элементов остается частью множества
    for i in self.__elements:
        for j in self.__elements:
            try:
                self.multiplication(i, j)
            except Exception as err:
                return err
    # При умножении трех элементов можно менять порядок скобок
    for i in self.__elements:
        for j in self.__elements:
            for k in self.__elements:
                result_1 = self.__mult_function(self.__mult_function(i, j), k)  # (i * j) * k
                result_2 = self.__mult_function(i, self.__mult_function(j, k))  # i * (j * k)
                if result_1 != result_2:
                    raise Exception('[!] Бинарная операция не является ассоциативной!')
    return True
```

Рис. 3. Проверка свойств моноида.

Эта функция перебирает все возможные комбинации в множестве, применяя к ним бинарную операцию, чтоб удостовериться, что всё действительно работает корректно.

Также была написана функция для поиска нейтрального элемента:

```python
def find_zero_element(self):
    """
    Выполняет поиск нейтрального элемента, сложность O(n^2)
    :return: Нейтральный элемент или None
    """
    for zero in self.__elements:
        for i in self.__elements:
            if self.multiplication(zero, i) != i or self.multiplication(i, zero) != i:
                break
        else:
            return zero
```

Рис. 4. Поиск нейтрального элемента.

Далее перейдем к примерам реализации этого класса:

В качестве первого примера рассмотрим булево множество и операцию ИЛИ:

```python
monoid = Monoid(
    {True, False},
    lambda a, b: a or b
)
```

Рис. 5. Создание моноида.

Выполним ряд операция над данным моноидом:

```
print_result(lambda: monoid.multiplication(True, False))
print_result(lambda: monoid.multiplication(False, False))
print_result(lambda: monoid.multiplication(True, True))
print_result(lambda: monoid.multiplication(False, "Hello"))
print_result(lambda: monoid.find_zero_element())
print_result(lambda: monoid.check_monoid_multiplication())
print_result(lambda: monoid.get_elements())
print_result(lambda: monoid.get_mult_function())
print_result(lambda: copy(monoid))
```

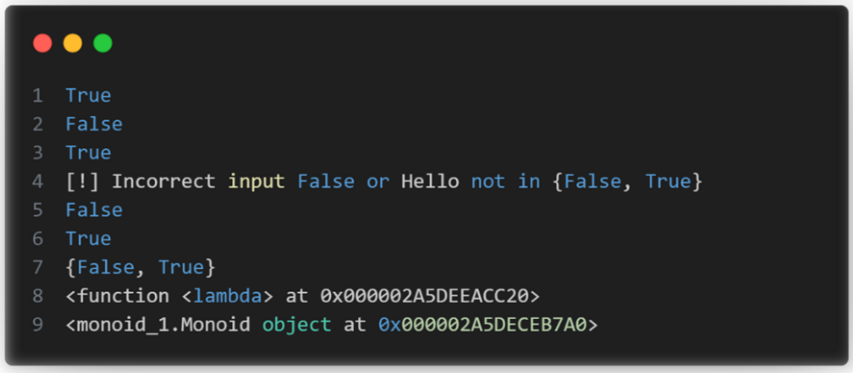
Рис. 6. Пример использования функций моноида.

Здесь первые 3 строки – корректные операции на множестве.

В 4 строке проверяется, что будет, если передать не часть множества.

В остальных строках проверка остальных функций.

Результат приведен ниже:

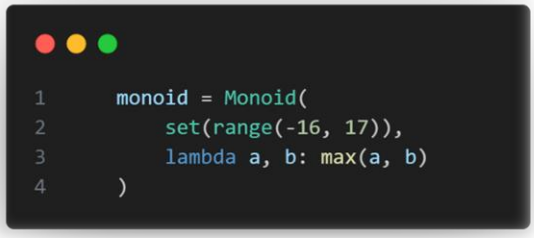


```
True
False
True
[!] Incorrect input False or Hello not in {False, True}
False
True
{False, True}
<function <lambda> at 0x000002A5DEEACC20>
<monoid_1.Monoid object at 0x000002A5DECEB7A0>
```

Рис. 7. Результат выполнения функций.

Как видим, результат выполнения функций соответствуют ожиданию.

Также протестируем реализацию на множестве целых чисел от -16 до 16 и бинарной операции – функции максимума:



```
monoid = Monoid(
    set(range(-16, 17)),
    lambda a, b: max(a, b)
)
```

Рис. 8. Создание моноида.

Этот моноид представляет интерес т.к. на первый взгляд им не является, однако при ближайшем рассмотрении становится понятно, что все свойства действительно выполняются. Проверим это, используя следующие функции:

```
    print_result(lambda: monoid.multiplication(5, 10))
    print_result(lambda: monoid.multiplication(12, 0))
    print_result(lambda: monoid.multiplication(6, 3))
    print_result(lambda: monoid.multiplication(1, "Hello"))
    print_result(lambda: monoid.find_zero_element())
    print_result(lambda: monoid.check_monoid_multiplication())
    print_result(lambda: monoid.get_elements())
    print_result(lambda: monoid.get_mult_function())
    print_result(lambda: copy(monoid))
```

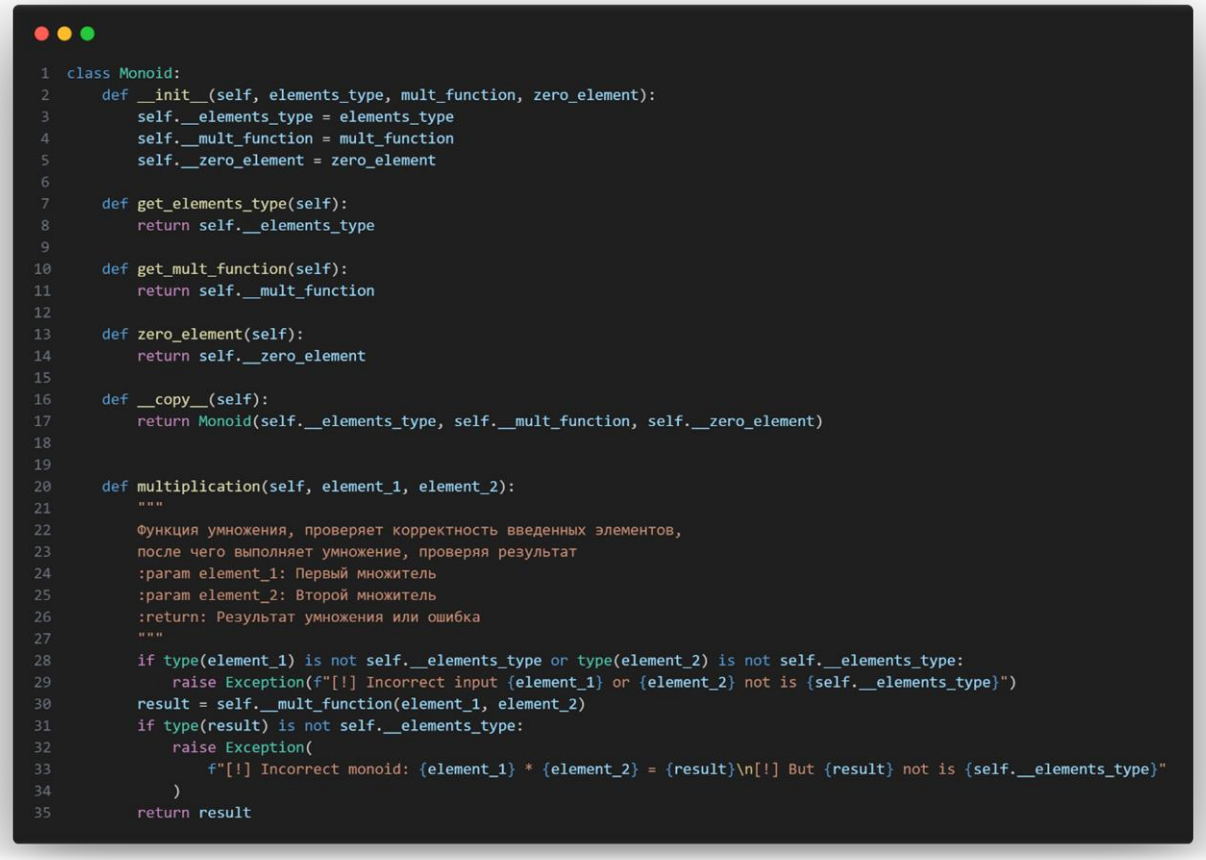
Рис. 9. Тестирующие функции.

Результат запуска выглядит следующим образом:

```
10
12
6
[!] Incorrect input 1 or Hello not in {-16, -15, ... , 16}
-16
True
{-16, -15, ... , 16}
<function <lambda> at 0x000001C6B48AC540>
<monoid_1.Monoid object at 0x000001C6B489EED0>
```

Рис. 10. Результат запуска тестов.

Как видим, наш пример действительно моноид, что подтверждают разработанные функции. Однако ограничивать примеры замкнутым множеством не очень интересно, поэтому была сделана вторая реализация:



```python
class Monoid:
    def __init__(self, elements_type, mult_function, zero_element):
        self.__elements_type = elements_type
        self.__mult_function = mult_function
        self.__zero_element = zero_element

    def get_elements_type(self):
        return self.__elements_type

    def get_mult_function(self):
        return self.__mult_function

    def zero_element(self):
        return self.__zero_element

    def __copy__(self):
        return Monoid(self.__elements_type, self.__mult_function, self.__zero_element)


    def multiplication(self, element_1, element_2):
        """
        Функция умножения, проверяет корректность введенных элементов,
        после чего выполняет умножение, проверяя результат
        :param element_1: Первый множитель
        :param element_2: Второй множитель
        :return: Результат умножения или ошибка
        """
        if type(element_1) is not self.__elements_type or type(element_2) is not self.__elements_type:
            raise Exception(f"[!] Incorrect input {element_1} or {element_2} not is {self.__elements_type}")
        result = self.__mult_function(element_1, element_2)
        if type(result) is not self.__elements_type:
            raise Exception(
                f"[!] Incorrect monoid: {element_1} * {element_2} = {result}\n[!] But {result} not is {self.__elements_type}"
            )
        return result
```

Рис. 11. Вторая реализация моноида.

Здесь уже нет дополнительных функций проверки, а соответствие исходному множеству проверяется путем сохранения исходного типа данных.

Рассмотрим несколько примеров.

Первым примером будет объединение массивов. Эта операция, очевидно, является моноидом, на языке python её реализация будет выглядеть следующим образом:

```python
monoid = Monoid(
    list,
    lambda a, b: a + b,
    []
)
```

Рис. 12. Создание моноида.

Запустим несколько функций, демонстрирующий функционал данной реализации:

```
    print_result(lambda: monoid.multiplication([1, 2, 3], [4, 5, 6]))
    print_result(lambda: monoid.multiplication([1, 2, 3, 4, 5, 6], []))
    print_result(lambda: monoid.multiplication([], []))
    print_result(lambda: monoid.multiplication([], 255))
    print_result(monoid.get_elements_type)
    print_result(monoid.get_mult_function)
    print_result(monoid.zero_element)
    print_result(lambda: copy(monoid))
```

Рис. 13. Тестирование моноида.

Запустим эти тесты:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
[]
[!] Incorrect input [] or 255 not is <class 'list'>
<class 'list'>
<function <lambda> at 0x000001CE765ACC20>
[]
<monoid_2.Monoid object at 0x000001CE76566E70>
```

Рис. 14. Результат тестирования.

Как видим всё работает корректно и были получены ожидаемые результаты выполнения.

И в качестве последнего, классического примера моноида рассмотрим умножение на множестве целых чисел.

Реализация выглядит следующим образом:

```
    monoid = Monoid(
        int,
        lambda a, b: a * b,
        1
    )
```

Рис. 15. Реализация моноида.

Выполним проверку реализации на следующем примере:

```
print_result(lambda: monoid.multiplication(0, 1))
print_result(lambda: monoid.multiplication(1, 0))
print_result(lambda: monoid.multiplication(1, 1))
print_result(lambda: monoid.multiplication(6, 2))
print_result(monoid.get_elements_type)
print_result(monoid.get_mult_function)
print_result(monoid.zero_element)
print_result(lambda: copy(monoid))
```

Рис. 16. Тестирование моноида.
Результат запуска выглядит следующим образом:

```
0
0
1
12
<class 'int'>
<function <lambda> at 0x000002C02028A0C0>
1
<monoid_2.Monoid object at 0x000002C020266390>
```

Рис. 17. Результат запуска тестов.
Как видим результат запуска соответствует ожиданиям.

4. Вывод:

В ходе лабораторной работы было рассмотрено, что такое моноид и какое он имеет отношение к нескольким другим алгебраическим структурам. Был рассмотрен ряд интересных примеров моноидов, которые сильно упростили понимание темы.
В теории, при реализации моноида нейтральный элемент а также проверку, что введенная структура действительно является моноидом, можно было проверять не перебором, а используя индукцию, однако это бы заметно усложнило реализацию а также могло остаться не очевидным читателю, поэтому было принято решение отказаться от этого, однако взамен была предложена реализация на конечном множестве, где наглядно и понятно проверяется корректность введенных данных а также производится поиск нейтрального элемента.

5. Листинг:

В силу большого количества файлов код был загружен на github: github.com/comp_alg_monoid