Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой

______ О.В. Непомнящий
подпись инициалы, фамилия
« _____» _______ 20 ___ г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 — Информатика и вычислительная техника
код – наименование направления

Динамический анализатор асимптотических временных сложностей
тема

Руководитель __________ ст. преподаватель И.В. Матковский
подпись, дата должность, ученая ступень инициалы, фамилия

Выпускник __________ М.В. Соколов
подпись, дата инициалы, фамилия

Нормоконтролер __________ И.В. Матковский
подпись, дата инициалы, фамилия

Красноярск 2021

РЕФЕРАТ

Выпускная квалификационная работа по теме «Динамический анализатор асимптотических временных сложностей» содержит 41 страницу текстового документа, 7 таблиц, 13 иллюстрации, 4 формулы, 16 использованных источников, 17 листингов, 2 приложения.

АЛГОРИТМЫ, АСИМПТОТИЧЕСКИЕ ВРЕМЕННЫЕ СЛОЖНОСТИ, RUST, АНАЛИЗАТОР

Проведен анализ существующих анализаторов асимптотических временных сложностей. На основе анализа были выделены основные требованию к динамическому анализатору асимптотических временных сложностей в виде программы.

Описаны этапы проектирования динамического анализатора — вызов пользовательской программы, замеры времени, сбор статистики, описание и генерация аргументов пользовательской программы.

Разработан динамический анализатор асимптотических временных сложностей в виде программы. Проведено тестирование анализатора. Для анализатора была написана документация и приведены примеры использования.

СОДЕРЖАНИЕ

Введение 3
1 Анализ задания 4
1.1 Выбор анализатора 4
1.2 Обзор готовых решений 5
1.2.1 Google benchmark 5
1.2.2 BigO Calculator 5
1.2.3 Сравнение готовых решений 6
1.3 Требования к динамическому анализатору 8
1.4 Выводы по главе 8
2 Этапы проектирования 9
2.1 Выбор языка программирования 9
2.2 Вызов сторонней программы 10
2.3 Замер времени работы пользовательской программы 11
2.4 Аргументы пользовательской программы 11
2.5 Генерация входных аргументов 12
2.6 Сериализация и десериализация аргументов 13
2.7 Разработка алгоритма 13
2.8 Выводы по главе 15
3 Разработка анализатора 16
3.1 Организация модулей программы 16
3.2 Вызов пользовательской программы 16
3.3 Пользовательские аргументы 21
3.4 Алгоритм нахождения асимптотических временных сложностей 27
3.5 Вывод в консоль 28
3.6 Выводы по главе 30
4 Тестирование и документация 31
4.1 Модульные тесты 31
4.2 Тестирование на программах 32
4.3 Документация 34
4.4 Выводы по главе 35
Заключение 37
Список использованных источников 38
Приложение А 39
Приложение Б 41

ВВЕДЕНИЕ

Одна из важных вещей при проектировании и разработке программного обеспечения является выбор правильного алгоритма. Неподходящий алгоритм может в разы замедлить работоспособность программы, что негативно сказывается на её использовании.

Выбор алгоритма зависит не только от обрабатываемых данных, но и от архитектуры всего приложения. Из-за этого не всегда понятна его асимптотическая сложность. Трудности с определением асимптотических сложностей можно часто увидеть у студентов начальных курсов. Данная ситуация возникает, когда студенты работают с практическими заданиями по определению асимптотических сложностей алгоритмов. Одним из решений данной проблемы может быть специальная программа или встраиваемая библиотека, которая измеряет время выполнения пользовательской программы/кода с некоторыми входными данными, и на основании полученных результатов выводит асимптотическую сложность.

2 Анализ задания

2.1 Выбор анализатора

Можно выделить два типа анализаторов:

– статический анализатор — программа, которая выводит асимптотическую сложность программы путем анализа её исходного кода. Данный метод сложный в реализации, привязан к конкретному языку программирования (C++/C# и т.п.) или промежуточному представлению (LLVM IR [2]). Но потенциально, может точнее всех выводить асимптотическую сложность за счет полного анализа исходного кода;

– динамический анализатор — программа или библиотека, которая выводит асимптотическую сложность программы путем анализа её времени выполнения с разными входными параметрами. Точность данного типа анализатора меньше, чем у статического анализатора, так как на время выполнения анализируемой программы могут влиять как другие запущенные программы и службы, так и операционная система. Сложность реализации также гораздо проще, так как нужно просто написать понятное API, без необходимости писать парсер или транслятор языка, что гораздо сложнее.

Динамический анализатор, распространяемый в виде библиотеки привязан к конкретным языкам программирования. То есть данный динамический анализатор должен уметь вызывать пользовательский код либо напрямую, либо через механизм FFI (Foreign function interface — Интерфейс внешних функций [3]). Данная особенность накладывает ограничения на использование анализатора, однако предоставляет более гибкий интерфейс взаимодействия с пользовательским кодом.

Динамический анализатор, распространяемый в виде программы является более универсальным методом, так как не привязан к конкретному языку программирования. Такой анализатор должен запускать пользовательскую программу с некоторыми входными данными, которые должен предоставлять либо сам пользователь, либо анализатор должен генерировать их сам. Данный метод более универсальный, так как позволяет вызывает большинство программ. Однако, такой анализатор является менее гибкий, так как пользовательская программа для такого анализатора является «черным ящиком», к которому нет доступа. Все взаимодействие с которым сводится к запуску с некоторыми входными данными.

В качестве типа анализатора был выбран динамический анализатор, который будет распространяться как программа.

2.2 Обзор готовых решений

2.2.1 Google benchmark

Google benchmark — библиотека для тестирования производительности кусков кода компании Google. Написана на языке программирования C++. Использует Google Tests для тестирования. [4]

Имеет ряд возможностей:

– сбор статистики;

– вывод статистики в файл;

– многопоточное тестирование;

– тестирование шаблонов.

Помимо этого, позволяет выводит асимптотическую сложность тестируемого кода. Вывод асимптотической сложности происходит в нотации Большой «О». В листинге 1 приведен код вычисления асимптотической сложности функции сравнения двух строк.

Листинг 1 — Сравнение строк

```
static void BM_StringCompare(benchmark::State& state) {
  std::string s1(state.range(0), '-');
  std::string s2(state.range(0), '-');
  for (auto _ : state) {
    benchmark::DoNotOptimize(s1.compare(s2));
  }
  state.SetComplexityN(state.range(0));
}

BENCHMARK(BM_StringCompare)->RangeMultiplier(2)->Range(1<<10,      1<<18)-
>Complexity();
```

При запуске данного кода, в консоль будет выводиться информация о времени исполнения функции BM_StringCompare с разной длиной строк. Минимальная длина строк — 1024, максимальная — 262144, каждый последующий запуск будет увеличивать длину строки в 2 раза.

После выполнения данного тестирования в консоль выведется предполагаемая асимптотическая сложность данной функции и её нормализованное квадратичное отклонение.

2.2.2 BigO Calculator

BigO Calculator — библиотека для вывода асимптотической сложности функций сортировок на языке Python [5]. Также позволяет измерять и выводить время исполнения, сравнивать несколько алгоритмов сортировки.

Вывод асимптотической сложности происходит в нотации Большой «О».

В листинге 2 представлено использование данной библиотеки.

Листинг 2 — BigO Calculator

```
from bigO import BigO
from bigO import algorithm

lib = BigO()

lib.test(bubbleSort, "random")
lib.test_all(bubbleSort)
lib.runtime(bubbleSort, "random", 5000)
lib.runtime(algorithm.insertSort, "reversed", 32)
lib.compare(algorithm.insertSort, algorithm.bubbleSort, "all", 5000)
```

В листинге 3 представлен пример вывода.

Листинг 3 — Вывод BigO Calculator

```
''' Result
Running quickSort(random array)...
Completed quickSort(random array): O(nlog(n))

Running quickSort(sorted array)...
Completed quickSort(sorted array): O(nlog(n))

Running quickSort(reversed array)...
Completed quickSort(reversed array): O(nlog(n))

Running quickSort(partial array)...
Completed quickSort(partial array): O(nlog(n))

Running quickSort(Ksorted array)...
Completed quickSort(ksorted array): O(nlog(n))
'''
```

2.2.3 Сравнение готовых решений

В таблице 1 представлено сравнение описанных анализаторов.

Таблица 1 — Сравнение готовых решений

Название анализатора	Тип анализатора	Язык программиро вания	Многократны й запуск	Сравнение результатов	Подробный вывод	Тестируемые функции	Передача аргументов в тестируемые функции
Google Benchmark	Динамически й/библиотека	C++	+	С помощью сторонней программы	+	Любые	Только целые числа
BigO Calculator	Динамически й/библиотека	Python	Только в методах runtime и compare	+	-	Функции, принимающи е массив в качестве аргументов	-

2.3 Требования к динамическому анализатору

На основании выбора типа анализатора и обзора готовых решений ниже описаны необходимые требования к анализатору:

Общие:

– анализ основных (наиболее распространенных) асимптотических временных сложностей в нотации Большой «O»: O (1), O(Log N), O(N), (N Log N), O(N^2), O(N^3);

– подробный вывод;

– возможность многократного запуска;

– предоставлять «дружелюбный» формат данных или иные возможности для описания входных параметров пользовательской программы;

– генерация входных параметров пользовательской программы согласно конфигурационному файлу.

Дополнительные:

– распространяться как консольная утилита. По возможности, без требования установки дополнительного программного обеспечения для корректной работы;

– запускаться на операционных системах Windows и Linux.

2.4 Выводы по главе

– выполнен обзор предметной области;

– выполнен анализ существующих анализаторов;

– на основании предметной области и анализа существующих анализаторов были сформированы требования к разрабатываемому анализатора.

3 Этапы проектирования

3.1 Выбор языка программирования

Анализатор будет написан на языке программирования Rust. Rust — мультипарадигмальный императивная компилируемый язык программирования общего назначения. Сочетает процедурный, структурный и функциональные парадигмы. Первая стабильная версия вышла 15 мая 2015 года [6]. Изначально разрабатывался совместно с компанией Mozilla. 8 февраля 2021 года из-за сокращений в Mozilla была основана независимая некоммерческая организация Rust Foundation, которая будет развивать язык программирования Rust и его экосистему [7].

За счет этого можно выделить особенности, присущие анализатору:

– высокая скорость работы — Rust компилируется в нативный код платформы, используя LLVM в качестве бэкенда своего компилятора. Это позволяет достигнуть высокой производительности программ, на уровне языков C и C++;

– кроссплатформенность — анализатор можно портировать на все платформы, которые поддерживает LLVM (Windows, Linux, macOS и т.д.) практически без изменения исходного кода за счет богатой стандартной библиотеки;

– безопасность — по сравнению с упомянутыми выше C и C++ модель памяти языка Rust позволяет без сборщика мусора на этапе компиляции выявить большинство проблем работы с памятью;

– Rust является высокоуровневым языком программирования, что позволяет писать эффективные абстракции «нулевой стоимости».

Rust обладает набором стандартных инструментов, которые позволяют тестировать, документировать и форматировать исходный код.

В Rust имеются средства для обобщенного программирования. Обобщения в Rust схожи с таковыми в современных языках программирования: Haskell, C# и других. Отличие обобщений от шаблонов в C++ являются: раскрытие обобщений на уровне синтаксического дерева программы (шаблоны в C++ раскрываются как строковые литералы), а также возможность наложения контракта на обобщенные типы с помощью типажей.

Типаж — схож с интерфейсами в объектно-ориентированных языках программирования. Однако, имеет ряд отличий:

– к типажам применяется статическая диспетчеризация — при компиляции полученная абстракция преобразуется в обычный вызов метода, без необходимости использования таблицы виртуальных функций (абстракции нулевой стоимости);

– при необходимости, можно использовать динамическую диспетчеризацию как в объектно-ориентированных языках программирования,

но это нужно явно указывать;

– в отличие от интерфейсов, типажи можно применять к сторонним типам без конфликтов;

– внутри типажей можно определить синоним для некоторого типа, который будет использоваться в одном из методов. Программист, при реализации типажа для некоторого типа, может присвоить синониму значение любого типа. Таким образом, разные реализации одного и тоже типажа, могут отличаться в зависимости от определения синонима типа.

За счет вышеописанных особенностей типажей, они повсеместно используются в сторонних библиотеках. К примеру, преобразование одного типа в другой через типажи, операции между типами (сложение, вычитание) производятся через типажи.

Использование Rust в качестве языка разработки анализатора позволяет:

– уменьшить объем кода за счет высокоуровневых абстракций без потери производительности;

– за счет развитой экосистемы языка использовать большое количество библиотек без необходимости пользования сторонними инструментами для их работы;

– стандартными средствами тестировать и документировать проект;

– получить итоговый бинарный файл для целевой платформы без необходимости установки дополнительных программ и служб.

3.2 Вызов сторонней программы

Динамический анализатор должен вызывать пользовательскую программу, указанную в конфигурационном файле.

В стандартной библиотеке Rust есть специальные структуры и типажи для данной задачи. Они предоставляют единый программный интерфейс для разных платформ.

Для запуска сторонней программы используется структура Command [8]. Command предоставляя методы для запуска программы по пути с одним или несколькими аргументами, захват потокового ввода и вывода пользовательской программы, а также возвращения результата пользовательской программы.

Из-за того, что на разных операционных системах отличаются кодировки символов и форматы путей, то стандартная библиотека Rust предоставляет единый интерфейс над родными строками операционных систем — OsString(владеющая данными на куче) и OsStr(«толстый» указатель на данные на куче). Данные структуры могут использоваться при передаче аргументов в программу. Rust гарантирует, что данные, хранящиеся в OsString и OsStr всегда корректны, и не содержат недопустимых символов, также предоставляя дешевые преобразования из UTF-8 строки (структура String в Rust) в родную строку операционной системы.

Поверх OsString и OsStr в стандартной библиотеке Rust имеются обертки PathBuf и Path соответственно. Данные структуры предоставляют единый интерфейс для работы с путями в операционных системах. Именно данные структуры будут использоваться, при работе с путями.

3.3 Замер времени работы пользовательской программы

Динамический анализатор должен измерят время исполнения пользовательской программы.

Для замера времени в Rust используется специальный счетчик Instant. Счетчик Instant является монотонным предоставляя гарантии, что при создании Instant всегда будет не меньше, чем предыдущая попытка замера времени. Instant предоставляет единый интерфейс для замера времени используя системные счетчики высокой точности [9] путем вызова соответствующих системных вызовов. На Windows — это QueryPerformanceCounter. На Unix — это clock_gettime (Monotonic Clock).

В качестве альтернативы для замера времени имеется библиотека chrono. Данная библиотека помимо замера времени имеет поддержку даты и временных зон в соответствии с международным стандартом ISO 8601 [14]. Использование данной библиотеки для анализатора является избыточным, так как она имеет лишние структуры и модули, который не будут использоваться.

3.4 Аргументы пользовательской программы

Динамический анализатор должен генерировать входные аргументы согласно некоторому описанию, которое составляет пользователь.

Входные аргументы будут ограничены 3 вариантами:

– массив;

– матрица;

– диапазон натуральных чисел.

Массив и матрица — это самые распространенные контейнеры, которые используются при тестировании асимптотических временных сложностей. Диапазон значений использует как альтернативный вариант, если пользовательская программа использует более сложные контейнеры. Диапазон значений можно использовать как идентификатор размера пользовательского контейнера, который заполняет данными непосредственно в пользовательской программе или извне.

Значения, генерируемые в качестве содержимого массивов и матриц будут ограничены следующими типами:

– целое 64 битное число;

– вещественное 64 битное число;

– символы ASCII — a-z, A-Z, 0-9;

– логические значения — 0, 1.

Пользователь должен иметь возможность настраивать размеры соответствующих аргументов:

– у массива пользователь может настраивать начальную длину, конечную длину, а также множитель, который увеличивает начальную длину в несколько раз, до тех пор, пока начальная длина не будет равна конечной;

– у матрицы пользователь может настраивать по отдельности как строки, так и столбцы. Настройка строк и столбцов должна быть идентичной настройки длины массивов. То есть, также отдельно для строк и столбцов содержать начальную длину, конечную длин и множитель;

– у диапазона значений настройка должна быть идентична массиву, за исключением того, что диапазон не генерирует значения, а возвращает число — собственный размер.

Для упрощения реализации анализатора необходимо реализовать программный интерфейс генератора. Интерфейс должен содержать следующий функционал:

– получение текущей длины аргумента;

– генерация новых значений одинаковой длины (итерации);

– генерация новых значений с разной длиной (поколения).

3.5 Генерация входных аргументов

Для генерации значений используется библиотека rand. Данная библиотека предоставляет несколько криптографически стойких генераторов псевдослучайных чисел [15]:

– OsRng — генератор предоставляемый операционной системы;

– ThreadRng — локальный генератор для потока, начальное состояние которого задается операционной системой. Гораздо быстрее OSRng за счет локальности;

– StdRng — стандартный генератор. Схож с ThreadRng, но без периодического обновления состояния;

– SmalllRng — небезопасный генератор, нацеленный на скорость и небольшое использование оперативной памяти.

В качестве генератора выбран ThreadRng, так как выбранные алгоритмы для StdRng и SmallRng могут поменять в будущих версиях Rust и быть платформазависимыми.

Помимо генераторов, библиотека rand предоставляет на выбор несколько распределений для генерации. Некоторые из них:

– Normal — среднеквадратичное;

– Uniform — равномерное;

– Bernoulli — распределение Бернулли.

Для того, чтобы генерируемые значения были равномерно распределены

по всему диапазону, указанному пользователем, в качестве итогового распределения был выбран Uniform.

3.6 Сериализация и десериализация аргументов

При разработке анализатора необходимо предоставить понятный для описание входных аргументов формат, к примеру JSON.

В стандартной библиотеке Rust нет необходимых средств для сериализации и десериализации данных. Для этого используется сторонние библиотеки или фреймворки. В экосистеме Rust для сериализации и десериализации данных используется фреймворк serde.

Фреймворк serde имеет собственную экосистему. Serde предоставляет специальные инструменты для сериализации и десериализации структур. В их число входят процедурные макросы, типажи и функции. Однако, serde не выполняет сериализацию или десериализацию в какой-либо формат. Для этого необходимо подключить другую библиотеку, которая будет использовать функционал serde для сериализации и десериализации в нужный формат. То есть, serde выступает в виде слоя, который связывает данные и форматы данных между собой, предоставляя интерфейс для сериализации и десериализации данных [10].

Serde поддерживает множество форматов данных, начиная с самых популярных: JSON, YAML, URL; заканчивая бинарными форматами данных: MessagePack, Bincode, D-Bus, и другие.

Чтобы установить serde необходимо в конфигурационном файле Cargo.toml, находящимся в корне Rust проекта, в разделе dependencies ввести: serde = { version = "номер версии", features = ["derive"] } и serde_json = "номер версии" (если необходим формат данных JSON).

```
[dependencies]
serde = { version = "1.0.123", features = ["derive"] }
serde_json = "1.0.61"
```

Рисунок 1 — Установка serde

После чего, находясь в корне проекта, в терминале ввести cargo --build. Проект начнет собираться, скачивая и устанавливая serde и serde_json с репозиториев.

В отличие от многих языков программирования, которые для задачи сериализации и десериализации данных используют рефлексию среды выполнения, serde использует механизм типажей.

3.7 Разработка алгоритма

Для сбора подробной статистики о времени исполнения программы

необходимо сделать по N итераций в каждом из M поколений. Итерации необходимы для сбора минимального, максимального и среднего времени исполнения по каждому из поколений. Количество итераций и количество поколений пользователь сможет задать самостоятельно.

Анализ программы должен выполняться на основе минимального времени исполнения, так как накладные расходы сторонних программ и операционной системы негативно сказываются на собранную статистику. Беря минимальное время исполнения программы анализатор может точнее вывести асимптотическую временную сложность, так как чем меньше накладные расходы, тем ближе время исполнения программы к идеальному времени.

Основная цель алгоритма — это нахождение зависимости между временем исполнения и размером входных данных.

Один из самых простых алгоритмов — это анализ изменений между значениями, собранными по формуле ниже:

$$O_i = \frac{T_i}{T_{(i-1)}} / \frac{N_i}{N_{(i-1)}} \tag{1}$$

где T_i — минимальное время исполнения i — поколения;

N_i — длина аргументов i — поколения.

С помощью данного алгоритма, можно вывести предполагаемые зависимости между асимптотической временной сложностью и разницей между изменением времени исполнения и изменением длины аргументов.

Ниже приведены предполагаемые зависимости:

– O(1) — $(\forall O) O \approx \frac{1}{D}$;

– O(logN) — $(\forall O) O > \frac{1}{D} \wedge O < 1$;

– O(N) — $(\forall O) O \approx 1$;

– O(N Log N) — $(\forall O) O > 1 \wedge O < D$;

– O(N^2) — $(\forall O) O \geq D$.

Однако, данный метод плохо работает на маленьких значениях. Вместо него был выбран алгоритм, основанный на методе наименьших квадратов, также применяемый в Google Benchmark. Метод наименьших квадратов — метод нахождения подходящей функции для данных, путем минимизации суммы квадратичных отклонений [16].

В общем случае теоретическое время выполнения программы можно представить в следующей форме — T = C * f(n), T — прогнозируемое время работы программы, C — коэффициент, f(n) — это функция высшего порядка: N, Log N, N Log N и другие.

Для нахождения правильной асимптотической сложности программы необходимо выбрать минимальное отклонение между фактическим временем выполнения программы и теоретическим. Для этого алгоритм должен

вычислить теоретическое время выполнения программы на разных функция высшего порядка. После чего, вычислить квадратичную ошибку между фактическим и полученным времен выполнения по формуле 2.

$$E=\sum_{i=0}^{k} t_i-f(n_i)^2 \quad (2)$$

где k — количество замеров времени.

Итоговая квадратичная ошибка аппроксимации будет минимальной, когда градиент E = 0:

$$E'=2*\sum_{i=0}^{k}(t_i-f(n_i))*f(n)' \quad (3)$$

С помощью формулы 3 можно вывести формулу для нахождения коэффициента C:

$$C=\sum_{i=0}^{k}(t_i*f(n_i))/\sum_{i=0}^{k} f(n_i)^2 \quad (4)$$

Применяя формулу 4 для нахождения коэффициента C, разработанный алгоритм находит теоретическое время выполнения программы для каждого из случаев: O(1), O(Log N), O(N), O(N Log N), O(N^2), O(N^3). Затем, полученные значения сравниваются с помощью формулы 2 с фактическими данными. Среди всех полученных ошибок выбирается минимальная. Функция, при которой была получена минимальная ошибка является асимптотической временной сложностью программы [13].

3.8 Выводы по главе

– произведена декомпозиция работы динамического анализатора;

– определены основные элементы анализатора, используемые библиотеки и модули;

– описан алгоритм нахождения асимптотических временных сложностей.

4 Разработка анализатора

4.1 Организация модулей программы

Программа была разбита на несколько модулей, каждый из которых отвечает за определенный функционал анализатора.

Диаграмма программы представлена в приложении Б.

Модуль main являет связующим модулем между модулями program, complexity и report. В данном модуле находится главная функция main, которая принимает в качестве аргументов пути до конфигурационных файлов. В цикле проходит по все путям, производя следующие действия:

– вызов статического метода Program::from_config;

– вызов метода exec у объекта структуры Program;

– вызов статического метода LeastSquares::computate_big_o;

– вызов статического метода Report::new;

– вывод объекта структуры report в консоль.

4.2 Вызов пользовательской программы

Для запуска пользовательской программы был написан отдельный программный модуль. На рисунке 2 продемонстрирована диаграмма данного модуля.

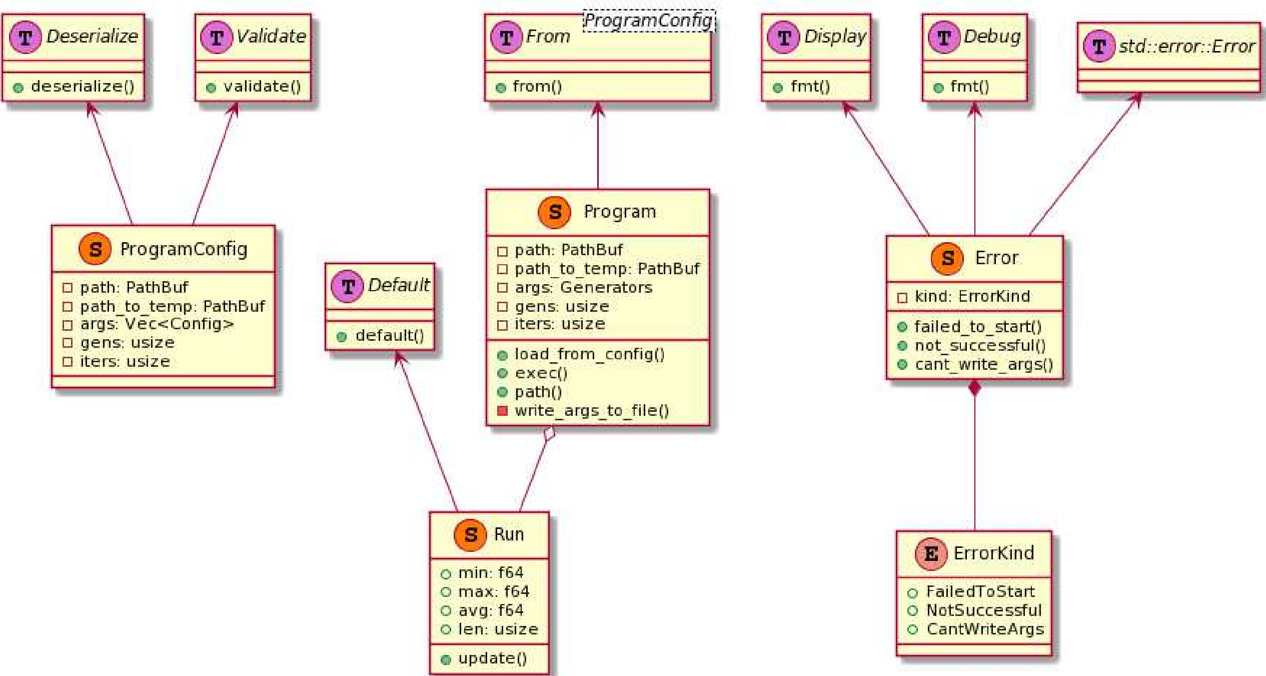


Рисунок 2 — Диаграмма модуля program

Структура Program — основной элемент модуля. Данная структура

используется для запуска пользовательской программы и замеров времени её выполнения. Реализует типаж From<ProgramConfig>, который позволяет делать преобразование из ProgramConfig в Program.

Поля структуры:

– path — путь до пользовательской программы;

– path_to_temp — путь, куда будут генерироваться промежуточные файлы;

– args — массив структур, которые реализуют типаж ArgumentGenerator Generators — псевдоним типа Vec<Box<dyn ArgumentGenerator>;

– gens — количество поколений;

– iters — количество итераций.

В листинге 4 представлен пример использование Command.

Листинг 4 — Пример использования Command

```
Command::new("ls")
        .arg("-l")
        .arg("-a")
        .spawn()
        .expect("ls command failed to start");
```

В данном примере статический метод new создает новый дочерний процесс, где в качестве аргумента передается путь до исполняемого файл. Метод arg передает процессу аргумент, который предается в качестве строки. Метод spawn запускает созданный процесс вместе с переданными ему аргументами. Синхронно ожидает его завершения, после чего возвращает результат работы пользовательского процесса.

Если процесс не удалось запустить, то вызывается метод expect, который завершает текущую программу с текстом ошибки, передаваемым в качестве аргумента.

Если процесс успешно запустился, то возвращается структура Child. С помощью неё можно управлять запущенным процессом:

– завершить процесс;

– захватить потоки ввода/вывода/вывода ошибок;

– синхронно дождаться завершения процесса со статусом выхода и данными потоков вывода/вывода ошибок.

В листинге 5 представлен пример использования Instant для замера времени работы функции.

Листинг 5 — Пример использования Instant

```
let instant = Instant::now();
let three_secs = Duration::from_secs(3);
std::thread::sleep(three_secs);
assert!(instant.elapsed() >= three_secs);
```

В данном примере создается новый объект Instant путем вызова

статического метода now. С помощью вызова статического метода from_secs в переменную three_secs записывается объект Duration промежутком в 3 секунды.

Duration — структура, которая хранит промежуток времени. Duration состоит из целого числа секунд и дробной части представленной в наносекундах.

Статический метод sleep приостанавливает действие текущего потока на время хранящиеся в переменной three_secs. После чего, с помощью декларативного макроса assert проверяется, что время, возвращенное методом elapsed, больше или равно времени, хранившиеся в переменной three_secs.

Метод elapsed возвращает объект Duration, которая содержит разницу во времени между вызовами Instant::now и instant.elapsed.

В таблице 2 представлено описание методов структуры Program.

Таблица 2 — Описание методов Program

Название метода	Описание
load_from_config	Принимает в качестве аргумента объект, которые реализует типаж AsRef<Path>. Данный типаж позволяет произвести дешевое преобразование из ссылки одного типа в ссылку другого типа, в данном случае в Path. Происходит попытка открыть файл по преобразованной ссылке. Затем происходит попытка считывания содержимого файла в строку. После считывания данных происходит попытка десериализации строки в объект ProgramConfig. После чего, у десериализованного объекта вызывается метод validate, которой проверяет поля объекта на нарушение инвариантов. Если во время любой из операции произошла ошибка, то она возвращается вызывающему данный метод коду. Иначе с помощью типажа From<ProgramConfig> десериализованный объект преобразуется в объект Program, который возвращается в качестве итогового результата.
path	Возвращает неизменяемую ссылку на path.
exec	Данный метод в цикле генерирует новые значения на основе конфигурационного файла, используя типаж ArgumentGenerator. После чего, вызывает пользовательскую программу. Так как передача большого количества аргументов может переполнить стек функции пользовательской программы, было принято решение изменить механизм вызов пользовательской программы. С помощью метода write_args_to_file сгенерированные значения записывает в промежуточный файл в указанную пользователем директорию. После чего, путь до файла передается в качестве аргумента при вызове пользовательской программы. При вызове программы собирается статистика (минимальное, максимально и среднее время) её времени выполнения по поколениям. Если во время выполнения данного метода произошла ошибка, то она преобразуется в Error и возвращается вызывающему данный метод коду. Если ошибок не было, то возвращается собранная статистика в виде массива.
write_args_to_file	Записывает сгенерированные значения во временный файл по передаваемому в качестве аргумента пути. Если во время записи произошла ошибка, то она возвращается вызывающему данный метод коду.

Структура ProgramConfig — вспомогательная структура к Program. Используется для преобразования в Program, так как Rust и библиотека serde накладывает ограничение на десериализацию объектов-типажей (Box<dyn ArgumentGenerator) из-за возможного нарушения trait object safety. Реализует типажи Deserialize и Validate. Типаж Validate, сторонней библиотеки validator, позволяет проверять инварианты полей структур, которые программист указывает вручную с помощью процедурных макросов. Используется для проверки gens > 0 и iters > 0, и возвращении соответствующей ошибки.

Строковое представление ProgramConfig представлено в листинге 6.

Листинг 6 — Строковое представление ProgramConfig

```
{
   "path": "/path/to/bin/file.out",
   "path_to_temp": "/path/to/tmp/",
   "args": [],
   "gens": 1,
   "iters": 1
}
```

Структура Run — структура для сбора статистики времени выполнения пользовательской программы. Реализует типаж Default, который позволяет получить значение по умолчанию.

Поля структуры:

– min — минимальное время выполнения в поколении. По умолчанию: 1.7976931348623157E+308;

– max — максимальное время выполнения в поколении. По умолчанию: -1.7976931348623157E+308;

– avg — среднее время выполнения в поколении. По умолчанию: 0,0;

– len — сумма длин аргументов в поколении. По умолчанию: 0.

Метод update принимает в качестве аргумента 64-битное вещественное число. Если число меньше min или больше max, то заменяет его соответственно, увеличивает avg на число.

Структура Error — структура ошибки, которая может случиться при выполнении метода Program::exec. Структура Error реализует типаж std::error::Error. Данный типаж стандартной библиотеки является типажом-маркером ошибки. У него нет своих методов, но он требует реализовать типажи Display и Debug. Данные типажи позволяют выводить информацию об объекте в консоль. Display используется для пользовательского вывода. Debug используется для вывода подробной информации об объекте, его полях и другие данные. Типаж Display программист должен реализовывать вручную, в то время как Debug можно вывести автоматически с помощью процедурного макроса derive — для необходимо, чтобы все поля также реализовывали данный типаж.

Поле структуры kind — перечисление типа ошибки. Возможные значения:

– FailedToStart(std::io::Error) — не удалось запустить пользовательскую

программу. Ошибка из модуля io стандартной библиотеки;

– NotSuccessful(Option<i32>) — пользовательская программа завершилась неудачно. Код, возвращаемый программой, если он имеется;

– CantWriteArgs(PathBuf, std::io::Error) — не удалось записать аргументы в промежуточный файл. Ошибка из модуля io стандартной библиотеки.

В таблице 3 представлено описание методов структуры Error.

Таблица 3 — Описание методов Error.

Название метода	Описание
failed_to_start	Принимает в качестве аргумента st::io::Error. Возвращает Error::FailedToStart.
not_successful	Принимает в качестве аргумента Option<i32>. Возвращает Error::NotSuccessful.
cant_wrtie_args	Принимает в качестве аргумента st::io::Error. Возвращает Error::CantWriteArgs.

4.3 Пользовательские аргументы

Самый просто способ сериализовать и/или десериализовать Rust структуру — это написать процедурный макрос #[derive(Serialize, Deserialize)] над определением структуры. Данный макрос будет раскрыт во время компиляции, и для указанной структуры будут автоматически определены типажи Serialize и Deserialize. Однако, для этого необходимо, чтобы каждое поле структуры рекурсивно для себя также реализовывало данные типажи, иначе, при компиляции, будет выдана соответствующая ошибка.

Программист может самостоятельно определить типажи Serialize, Deserialize, но данный метод гораздо сложнее.

Помимо этого, serde предоставляет ряд процедурных макросов для дополнительной настройки процесса сериализации и десериализации. Например, процедурный макрос #[serde(rename = "name")], который сериализует и десериализует соответствующее поле структуры с указанным именем, вместо стандартного имя поля структуры.

В листинге 7 показан пример использования serde.

Листинг 7 — Пример использования serde

```
#[derive(Serialize, Deserialize)]
struct Point {
    x: i32,
    y: i32,
}
let point = Point { x: 1, y: 2 };
let serialized = serde_json::to_string(&point).unwrap();
let deserialized: Point = serde_json::from_str(&serialized).unwrap();
```

В данном примере с помощью процедурного макроса derive выводятся два типажа: Serialize и Deserialize.

В переменную point записывает новый объект Point. Затем, в статический метод to_string из библиотеки serde_json, передается ссылка на point. Данный метод использует выведенный типаж Serialize и сериализует переменную point в его строковое представление в формате JSON. Так как в структуре нет никакого перечисления, которое мог бы содержать ошибку — использует метод unwrap, который сразу вернет строковое представление.

Затем, в статический метод from_str из библиотеки serde_json, передается ссылка на point. Данный метод использует выведенный типаж Deserialize и десериализует строковое представление point обратно в объект Point. Так как используется уже сериализованное строковое представление, то ошибок при десериализации быть не должно.

В данном примере был использован формат данных JSON, предоставляемый библиотекой serde_json. При необходимости, данную библиотеку можно заменить на другую — к примеру, serde_yaml, которая сериализует и десериализует данные в формат YAML.

Для реализации пользовательский аргументов был написан специальный модуль. Диаграмма структур модуля config представлена на рисунке 3.

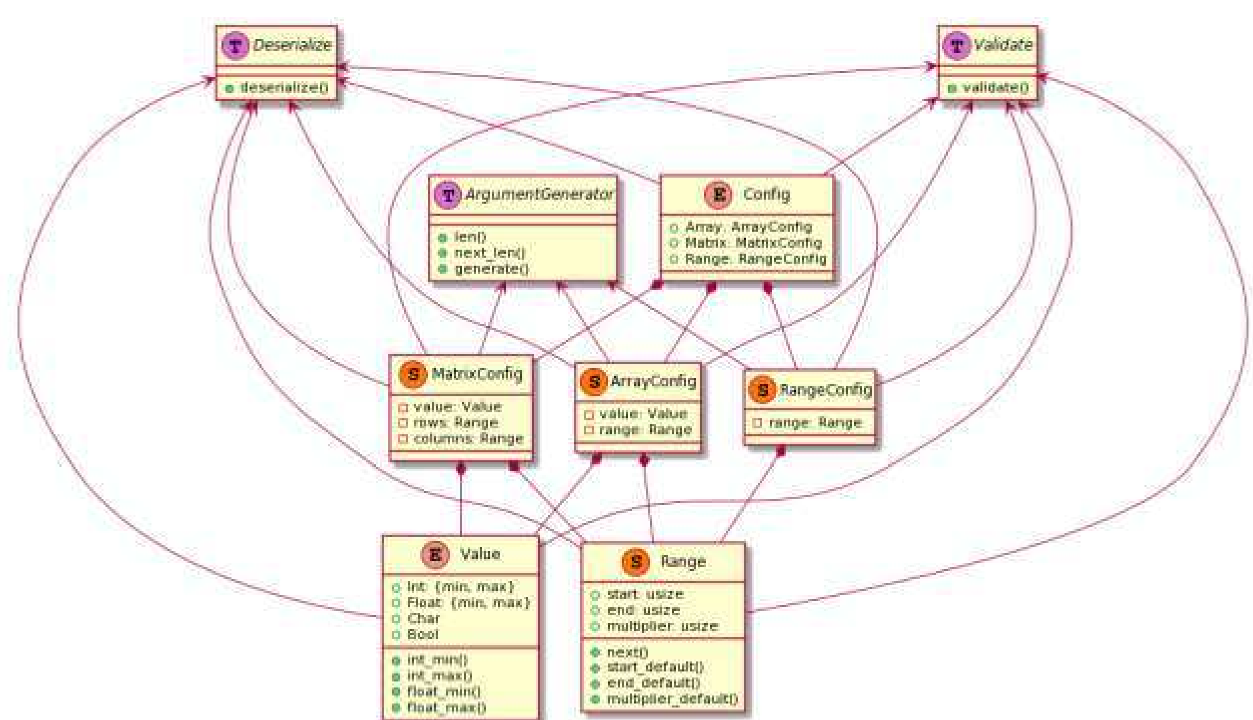


Рисунок 3 — Диаграмма классов модуля config

Перечисление Value описывает типы содержимого массивов и матриц. Реализует типаж Deserialize. Возможные варианты:

– Value::Int — анонимная структура. Целое 64-битное число. Содержит 2 поля: min и max, которые ограничивают диапазон генерируемых значений;

– Value::Int — анонимная структура. Вещественное 64-битное число; Содержит 2 поля: min и max, которые ограничивают диапазон генерируемых значений;

– Value::Char — маркер. ASCII символы;

– Value::Bool — маркер. Булевы значения.

Поля min, max в обоих случаях пользователь может не указывать. Вместо них будут подставлены значения по умолчанию, путем вызова соответствующих статических методов.

В таблице 4 представлены описание методов Value.

Таблица 4 — Описание методов Value

Название метода	Описание
int_min	Возвращает минимально возможное целое число, помещающееся в 64 бита: -9223372036854775808. Используется, если пользователь не указал поле min в Value::Int
int_max	Возвращает максимально возможное целое число, помещающееся в 64 бита: 9223372036854775807. Используется, если пользователь не указал поле max в Value::Int
float_min	Возвращает минимально возможное вещественное число, помещающееся в 64 бита: -1.7976931348623157E+308. Используется, если пользователь не указал поле min в Value::Float
float_max	Возвращает максимально возможное вещественное число, помещающееся в 64 бита: 1.7976931348623157E+308. Используется, если пользователь не указал поле max в Value::Float

В листингах 8, 9, 10, 11 продемонстрированы строковые представления перечисления Value во всех вариантов.

Листинг 8 — 64 битное целое число

```
{
   "type" : "Int",
   "min": 0,
   "max": 1000
}
```

Листинг 9 — 64 битное вещественное число

```
{
   "type" : "Float",
   "min": 10.0,
   "max": 1000.0
}
```

Листинг 10 — ASCII символы

```
{
   "type" : "Char"
}
```

Листинг 11 — Булевы значения

```
{
   "type" : "Bool"
}
```

Структура Range описывает диапазон натуральных значений. Реализует типажи Deserialize, Validate. Типаж Validate используется для проверки start >= 1 и multiplier >= 2, и возвращении соответствующей ошибки.

Поля структуры:

– start — начало диапазона;

– end — конец диапазона;

– multiplier — множитель увеличения начала диапазона.

Поля данной структуры пользователь может не указывать. Вместо них будут подставлены значения по умолчанию, путем вызова соответствующих статических методов.

В таблице 5 представлены описание методов Range.

Таблица 5 — Описание методов Range.

Название метода	Описание
next	Пытается увеличить начало диапазона согласно множителю. Если новое число больше, чем конец диапазона, то возвращает в качестве нового значения конец диапазона, иначе новое значение.
start_default	Возвращает начало диапазона по умолчанию: 10. Используется, если пользователь не указал поле start.
end_default	Возвращает максимально возможное число, помещающееся в usize — платформозависимый тип, равный размеру указателя. На 32 битных системах — 32 бита, на 64 битных системах — 64 бита. Используется, если пользователь не указал поле end.
multiplier_default	Возвращает множитель диапазона по умолчанию: 2.

Название метода	Описание
	Используется, если пользователь не указал поле multiplier.

В листинге 12 продемонстрировано строковое представление.

Листинг 12 — Строковое представление Range.
```
{
    "Range": {
        "start": 1024,
        "end": 160124,
        "multiplier": 2
    }
}
```

Структура ArrayConfig описывает тип аргумента в виде массива. Реализует типажи Deserialize и ArgumentGenerator.

Поля структуры:

– value — тип содержимого массива;

– range — диапазон размера массива.

В листинге 13 продемонстрировано строковое представление.

Листинг 13 — Строковое представление ArrayConfig.
```
{
   "Array" : {
       "value": {
           "type": "Int",
           "min" : 10,
           "max": 20000
       },
       "start": 1024,
       "end": 160124,
       "multiplier": 2
   }
}
```

Структура MatrixConfig описывает тип аргумента в виде матрицы. Реализует типажи Deserialize и ArgumentGenerator.

Поля структуры:

– value — тип содержимого матрицы;

– rows — диапазон размера строк матрицы;

– columns — диапазон размера столбцов матрицы.

В листинге 14 продемонстрировано строковое представление.

Листинг 14 — Строковое представление MatrixConfig.

```
{
   "Matrix" : {
       "value": {
           "type": "Int",
           "min" : 10,
           "max": 20000
       },
       "rows": {
           "start": 1024,
           "end": 160124,
           "multiplier": 2
       },
       "columns": {
           "start": 1024,
           "end": 160124,
           "multiplier": 2
       }
   }
}
```

RangeConfig описывает тип аргумента в виде диапазона. Реализует типажи Deserialize и ArgumentGenerator.

Поле структуры range — диапазон значений.

В листинге 15 продемонстрировано строковое представление.

Листинг 15 — Строковое представление RangeConfig.

```
{
   "Range" : {
       "start": 1024,
        "end": 160124,
        "multiplier": 2
   }
}
```

Перечисление Config перечисляет все доступные варианты генерируемых аргументов.

ArgumentGenerator — типаж генератора аргументов. Описание методов данного типажа представлено в таблице 6.

Таблица 6 — Описание методов типажа ArgumentGenerator

Название метода	Описание
len	Возвращает текущую длину аргументов
next_len	Увеличивает текущую длину, и возвращает новое значение
generate	Генерирует новые значения. Так как генерируемые значения необходимо записывать в файл, то данный метод возвращает значения в виде строки.

В листинге 16 продемонстрировано пример итогового конфигурационного файла.

Листинг 16 — Итоговый конфигурационный файл.

```
{
   "path": "/path/to/bin/file.out",
   "path_to_temp": "/path/to/tmp/",
   "args": [
       {
           "Array" : {
               "value": {
                   "type": "Int"
               },
               "start": 1024,
               "multiplier": 2
           }
       }
   ],
   "gens": 6,
   "iters": 1
}
```

4.4 Алгоритм нахождения асимптотических временных сложностей

Для реализации алгоритма нахождения асимптотических временных сложностей был написан специальный модуль. Диаграмма структур модуля complexity представлена на рисунке 4.

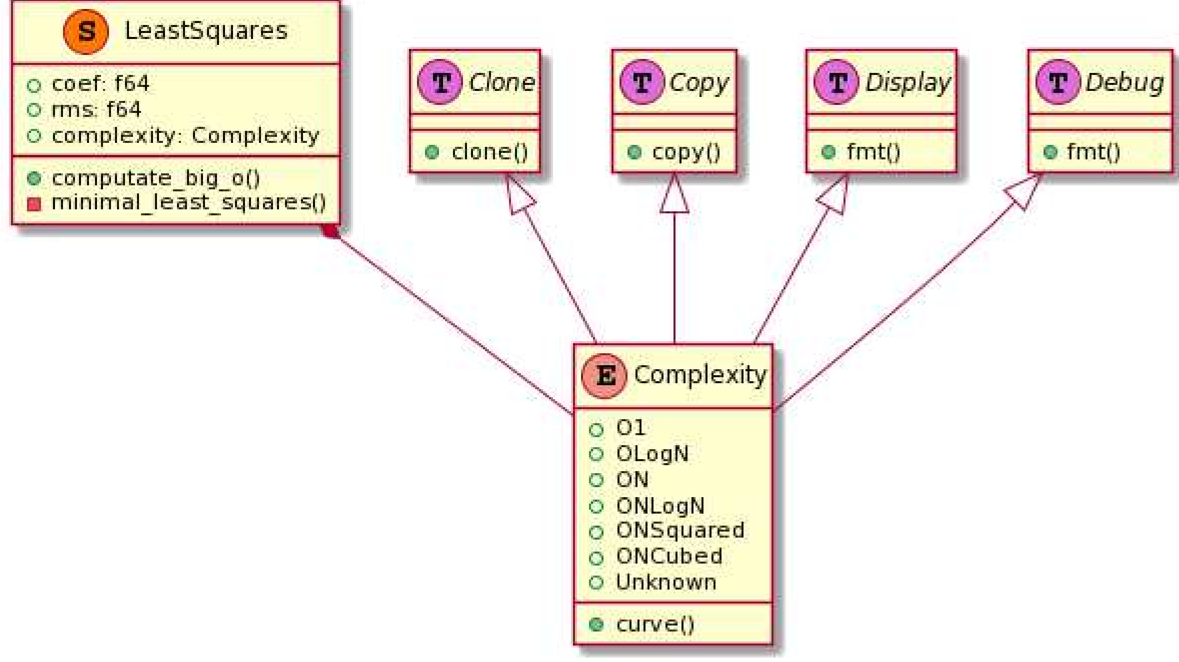


Рисунок 4 — Модуль complexity

Перечисление Complexity описывает возможные асимптотические временные сложности, которые анализатор может определить. Реализует типажи Clone, Copy, Display, Debug. Возможные варианты:

– O1 — константное время;
– OLogN — логарифмическое время;
– ON — линейное время;
– ONLogN — линейно-логарифмическое время;
– ONSquared — квадратичное время;
– ONCubed — кубическое время;
– Unknown — неизвестное время.

Метод Complexity::curve возвращает лямбда-функцию соответствующей асимптотической сложности: O1 – 1.0, OLogN – Log N, ON – N и другие.

Структура LeastSquares описывает значения, которые используются для вычисления асимптотической сложности.

Поля структуры:

– coef — коэффициент асимптотической сложности;
– rms — итоговая нормализованная среднеквадратичная ошибка;
– complexity — асимптотическая временная сложность.

В таблице 7 представлены методы структуры LeastSquares.

Таблица 7 — Описание методов LeastSquares

Название метода	Описание
computate_big_o	Принимает в качестве аргумента ссылку на массив, содержащий время выполнения пользовательской программы. В цикле для каждого варианта Complexity, кроме Unknown, вызывает метод minimal_least_squares. После чего, находит и возвращает структуру LeastSquares, у которой поле rms является минимальным.
minimal_least_squares	Принимает в качестве аргумента ссылку на массив, содержащий время выполнения пользовательской программы, и лямбда-функцию высшего порядка. С помощью описанного в главе 2.7 алгоритме рассчитывает coef, rms. Возвращает структуру LeastSquares, где complexity равно Complexity::Unknown.

4.5 Вывод в консоль

Для вывода в консоль была написана дополнительная структура Report. Данная структура собирает всю итоговую информацию, позволяя выводить её в

удобном для пользователя виде. Реализует типажи Display и Debug.

Диаграмма модуля Report продемонстрирована на рисунке 5.

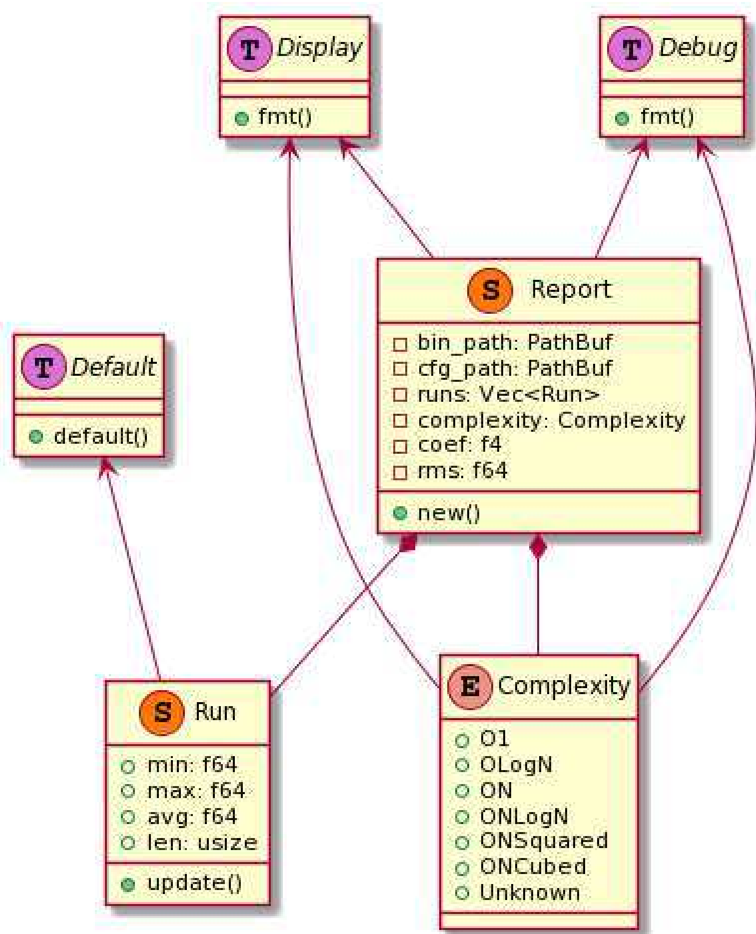


Рисунок 5 — Диаграмма модуля Report

Поля структуры:

– bin_path — путь до пользовательской программы;

– cfg_path — путь до конфигурационного файла;

– runs — статистика времени выполнения пользовательской программы;

– complexity — асимптотическая временная сложность пользовательской программы;

– coef — коэффициент асимптотической сложностью;

– rms — процент ошибки.

Метод new возвращает новое значение Report. Принимает в качестве аргументов LeastSquares, путь до исполняемого файла, путь до конфигурационного файла, массив со временем исполнения программы.

Пример вывода в консоль продемонстрирован на рисунке 6.

```
[naymoll@naymoll time_analyzer]$ ./target/release/time_analyzer -c /
Binary file: /home/naymoll/Projects/Clion/find.out
Config file: /home/naymoll/Projects/time_analyzer/target/cfg2.json
Len            Min time(sec)   Avg time(sec)   Max time(sec)
-------------------------------------------------------------
1024                 0.00858         0.01058         0.01258
2048                 0.01099         0.01278         0.01457
4096                 0.01144         0.01148         0.01152
8192                 0.01539         0.01600         0.01661
16384                0.00864         0.01591         0.02317
32768                0.00839         0.00860         0.00881
65536                0.01480         0.01522         0.01565
131072               0.02811         0.02872         0.02934
262144               0.05453         0.05585         0.05717
524288               0.10630         0.10971         0.11311
Complexity: 0.0000002054610049073296 O(N)
RMS: 26.95%
```

Рисунок 6 — Пример вывода Report в консоль

4.6 Выводы по главе

Был разработан динамический анализатор асимптотических временных сложностей:

– представлено описание пользовательских аргументов в формате json;

– организован процесс генерации пользовательских аргументов согласно описанию

– запуск пользовательской программы с пользовательскими аргументами и замеры времени её выполнения;

– анализ времени выполнения программы по статистики её запуска и вывод итоговой асимптотической сложности;

– вывод запусков программы и её асимптотической сложности.

5 Тестирование и документация

5.1 Модульные тесты

Для тестирования функционала программы были написаны модульные тесты. Модульные тесты в Rust встроены на уровне языка и его экосистемы. Для написания теста необходимо в файле написать новый модуль. Над модулем необходимо написать процедурный макрос #[cfg(test)]. Данный макрос необходим, чтобы убрать данный модуль из debug и release сборок. В данном модуле необходимо написать новую функцию. Над данной функцией необходимо написать процедурный макрос #[test], чтобы зарегистрировать данную функцию как тестовую функцию.

Пример модульного теста продемонстрирован в листинге 17.

Листинг 17 — Пример модульного теста

```
#[cfg(test)]
mod tests {

    #[test]
    fn my_test() {
         assert_eq!(2 + 2, 4);
    }
}
```

В данном примере в функции my_test декларативный макрос assert_eq проверяет значение слева до запятой — 2 + 2, на соответствие значения справа — 4. Так как левое значение равно правому, то происходит обычный выход из функции. В таком случае тест считается успешно пройденным. Если бы левое и правое значения отличались друг от друга, то assert_eq «бросал» панику, что считается, как провальный тест.

Для запуска тестов необходимо в командной строке, в корне проекта ввести cargo test. Данная команда запустить все зарегистрированные тесты, после чего выведет итоговую информацию о результатах тестах в консоль.

На рисунке 7 продемонстрирован пример команды cargo test.

```
C:\Users\Naymoll\Videos\test_example>cargo test
    Blocking waiting for file lock on build directory
   Compiling test_example v0.1.0 (C:\Users\Naymoll\Videos\test_example)
    Finished test [unoptimized + debuginfo] target(s) in 17.52s
     Running target\debug\deps\test_example-4458bf399ec224a0.exe

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

   Doc-tests test_example

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s


C:\Users\Naymoll\Videos\test_example>
```

Рисунок 7 — Пример запуска тестов.

Для анализатора были написаны 10 тестов. Большинство тестов — это проверка десериализации и валидации с разными корректными и некорректными данными.

Запуск тестов анализатор продемонстрирован на рисунке 8.

```
PS C:\Users\Naymoll\Desktop\time_analyzer> cargo test
   Compiling time_analyzer v0.1.0 (C:\Users\Naymoll\Desktop\time_analyzer)
    Finished test [unoptimized + debuginfo] target(s) in 18.88s
     Running target\debug\deps\time_analyzer-7da08b207954811b.exe

running 10 tests
test program::tests::des_test ... ok
test program::tests::des_test_failed ... ok
test program::tests::validate_test ... ok
test configs::tests::deserialization_test ... ok
test program::tests::validate_test_2 ... ok
test program::tests::validate_test_failed ... ok
test program::tests::validate_test_failed_2 ... ok
test report::tests::debug_report ... ok
test report::tests::display_report ... ok
test run::tests::update_test ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

PS C:\Users\Naymoll\Desktop\time_analyzer>
```

Рисунок 8 — Тесты анализатора

5.2 Тестирование на программах

Второй этап тестирования заключался в тестировании анализатора на программах, которые считывают данные с файла, после чего используют на них разные алгоритмы.

Программы представляли из себя C++ программы, с разными

асимптотическими сложностями. Всего их было:

– умножение числа в 2 раза — O(1);

– поиск минимального числа — O(N);

– быстрая сортировка, с помощью стандартной функции sort — O(N Log N);

– сортировка пузырьком — O(N^2).

На рисунке 9 продемонстрирован результат работа программы, которая работает за O(1). Анализатор правильно нашел итоговую асимптотическую сложность. По выводу можно заметить, что время выполнения данной программы практически не меняется в независимости от количества сгенерированных значений.

```
Binary file: C:/Users/Naymoll/Desktop/time_analyzer/target/const.exe
Config file: const.json
Len            Min time(sec)   Avg time(sec)   Max time(sec)
-------------------------------------------------------------
100                  0.01025         0.01025         0.01025
200                  0.01323         0.01323         0.01323
400                  0.01195         0.01195         0.01195
800                  0.01054         0.01054         0.01054
1600                 0.01045         0.01045         0.01045
Complexity: 0.01128442 O(1)
RMS: 10.15%
```

Рисунок 9 — Постоянное время

На рисунке 10 продемонстрирован результат работа программы, которая работает за O(N). Время выполнения данной программы меняется в зависимости от количества сгенерированных значений. Анализатор правильно нашел итоговую асимптотическую сложность, несмотря на то что в первых тестах изменения времени выполнения не соответствует изменению длины аргументов.

```
Binary file: C:/Users/Naymoll/Desktop/time_analyzer/target/find.exe
Config file: find.json
Len            Min time(sec)   Avg time(sec)   Max time(sec)
-------------------------------------------------------------
1024                 0.01217         0.01217         0.01217
2048                 0.01367         0.01367         0.01367
4096                 0.01767         0.01767         0.01767
8192                 0.02791         0.02791         0.02791
16384                0.04460         0.04460         0.04460
Complexity: 0.000002998411152859237 O(N)
RMS: 27.26%
```

Рисунок 10 — Линейное время

На рисунке 11 продемонстрирован результат работа программы, которая работает за O(N Log N). Время выполнения данной программы меняется в

зависимости от количества сгенерированных значений. Анализатор правильно нашел итоговую асимптотическую сложность. Однако, для этого потребовалось большее количество запусков.

```
Binary file: C:/Users/Naymoll/Desktop/time_analyzer/target/qsort.exe
Config file: qsort.json
Len            Min time(sec)   Avg time(sec)   Max time(sec)
------------------------------------------------------------
1024                 0.02726         0.02726         0.02726
2048                 0.01449         0.01449         0.01449
4096                 0.01855         0.01855         0.01855
8192                 0.02754         0.02754         0.02754
16384                0.04614         0.04614         0.04614
32768                0.08252         0.08252         0.08252
65536                0.17528         0.17528         0.17528
131072               0.31687         0.31687         0.31687
262144               0.65143         0.65143         0.65143
524288               1.39502         1.39502         1.39502
Complexity: 0.00000014011003801433691 O(NlogN)
RMS: 5.62%
```

Рисунок 11 — Линейно-логарифмическое время

На рисунке 12 продемонстрирован результат работа программы, которая работает за O(N^2). Время выполнения данной программы меняется в зависимости от количества сгенерированных значений. Анализатор правильно нашел итоговую асимптотическую сложность.

```
Binary file: C:/Users/Naymoll/Desktop/time_analyzer/target/sort.exe
Config file: sort.json
Len            Min time(sec)   Avg time(sec)   Max time(sec)
------------------------------------------------------------
1024                 0.01383         0.01383         0.01383
2048                 0.01799         0.01799         0.01799
4096                 0.03454         0.03454         0.03454
8192                 0.10882         0.10882         0.10882
16384                0.35464         0.35464         0.35464
Complexity: 0.0000000013422941938165207 O(N^2)
RMS: 12.20%
```

Рисунок 12 — Полиномиальное время

5.3 Документация

В экосистеме Rust инструменты для документации исходного кода присутствуют по умолчанию. Документации пишутся напрямую в исходном коде программе с помощью комментариев и процедурных макросов. Данные инструменты позволяют:

– добавлять описание к каждому элементу программу;

– скрывать не публичные структуры и методы, к которым у программиста

нет прямого доступа;

– ссылаться на сторонние библиотеки и модули программы;

– использовать markdown разметку.

Для того чтобы добавить описание элементу программы — над ним необходимо написать комментарий, начинающийся с трех знаков «///». После чего помощью команды cargo doc сгенерировать документацию. Cargo doc генерирует документацию вместе со всеми зависимостями, для того что сгенерировать документацию только для собственной программы или библиотеки — необходимо использовать команды cargo doc --no-deps.

После завершения работы команды документация будет находиться в папке target/doc/название_программы/. В данной папке будет находится файл index.html — это главная страница документации. На ней будут отображены основные элементы программы. На рисунке 13 представлена главная страница документации анализатора.

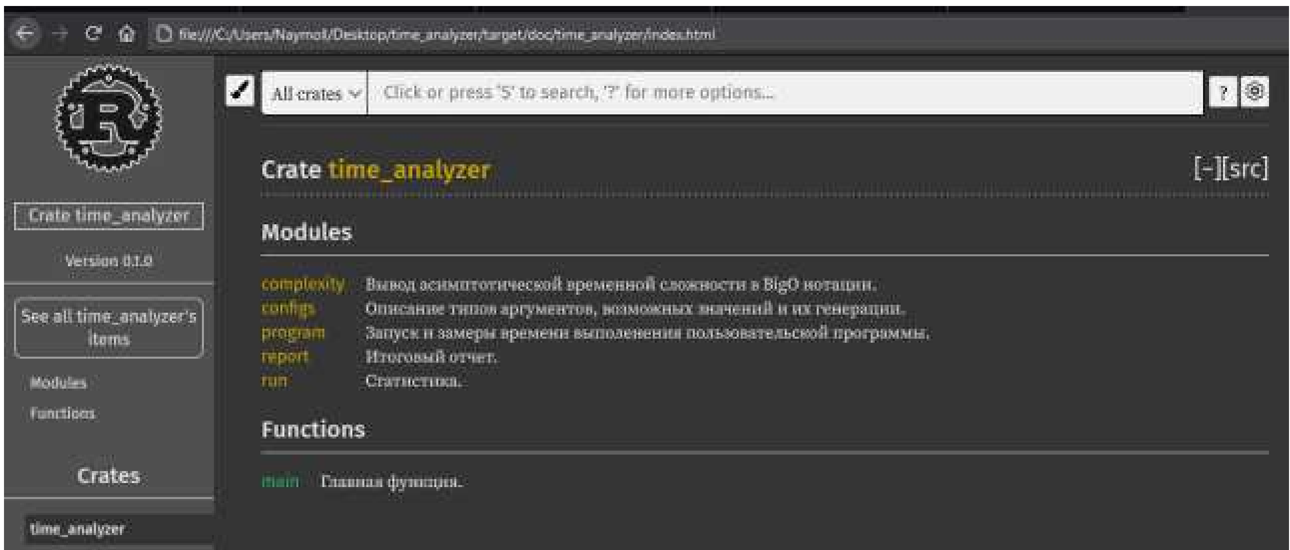


Рисунок 13 — Главная страница документации

Документации представляет из себя набор статических html-файлов, по которым можно свободно переходить. При необходимости, можно создать сервер с документацией, который будет доступен в сети интернет.

5.4 Выводы по главе

– были написаны, и успешно выполнены модульные тесты анализатора;

– в ходе тестирования анализатора на программах — анализатор успешно вывел их асимптотические временные сложности. У анализатора могут быть проблемы при выводе асимптотических временных сложностей у некоторых программ, как это было с линейно-логарифмической. Для решения подобной проблемы необходимо увеличивать размер тестируемой выборки — чем больше итоговая выборка, тем точнее анализатор сможет вывести асимптотическую

временную сложность;

– написана документация для анализатора. Её можно сгенерировать самостоятельно с помощью соответствующих команд.

ЗАКЛЮЧЕНИЕ

В результате проделанной работы был спроектирован и разработан динамический анализатор асимптотических временных сложностей в виде программы без интерфейса. Разработанный анализатор позволяет генерировать данные, запускать пользовательскую программу и замерять её время работы на основе конфигурационного файла. Анализатор позволяет выводить наиболее распространенные асимптотические временные сложности. Для анализатора была написана документация.

Исходный код анализатора доступен для скачивания с GitHub репозитория [11].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. СТО 4.2—07—2014 Система менеджмента качества. Общие требования к построению, изложению и оформлению документов учебной и научной деятельности. Красноярск: СФУ, 2014. 60 с.
2. Справочное руководство LLVM IR [Электронный ресурс] — Режим доступа: https://llvm.org/docs/LangRef.html.

1 Ознакомительная статья, посвященная FFI [Электронный ресурс] — Режим доступа: http://wiki.c2.com/?ForeignFunctionInterface.

2 Домашняя страница Google Benchmark [Электронный ресурс] — Режим доступа: https://opensource.google/projects/benchmark.

3 Репозиторий BigO Calculator [Электронный ресурс] — Режим доступа: https://github.com/Alfex4936/python-bigO-calculator

3. Официальный сайт языка программирования Rust [Электронный ресурс] — Режим доступа: https://www.rust-lang.org/
4. Новость об образовании Rust Foundation [Электронный ресурс] — Режим доступа: https://habr.com/ru/news/t/541600/
5. Описание структуры Command [Электронный ресурс] — Режим доступа: https://doc.rust-lang.org/std/process/struct.Command.html
6. Описание счетчика Instant [Электронный ресурс] — Режим доступа: https://doc.rust-lang.org/std/time/struct.Instant.html
7. Официальный сайт serde [Электронный ресурс] — Режим доступа: https://serde.rs/
8. Репозиторий анализатора [Электронный ресурс] — Режим доступа: https://github.com/Naymoll/time_analyzer.git
9. Документация стандартной библиотеки Rust [Электронный ресурс] — Режим доступа: https://doc.rust-lang.org/std/
10. Репозиторий с описанием алгоритма и примером программы на C++ [Электронный ресурс] — Режим доступа: https://github.com/ismaelJimenez/cpp.leastsq
11. Документация библиотеки chrono [Электронный ресурс] — Режим доступа: https://docs.rs/chrono/0.4.19/chrono/
12. Документация библиотеки rand [Электронный ресурс] — Режим доступа: https://docs.rs/rand/0.8.3/rand/
13. Метод наименьших квадратов [Электронный ресурс] — Режим доступа: https://web.williams.edu/Mathematics/sjmiller/public_html/BrownClasses/54/handouts/MethodLeastSquares.pdf

ПРИЛОЖЕНИЕ А

Листинги тестируемых программ

const.exe

```
#include <fstream>

int main(int argc, char *argv[]) {
    auto stream = std::fstream(argv[1], std::fstream::in);

    size_t size;
    stream >> size;

    size_t double_size = size * size;
    return 0;
}
```

find.exe

```
#include <vector>
#include <fstream>
#include <algorithm>

int main(int argc, char *argv[]) {
    auto stream = std::fstream(argv[1], std::fstream::in);

    size_t size;
    stream >> size;

    std::vector<int64_t> buffer(size);
    for (size_t i = 0; i < size; i++) {
        int64_t value;
        stream >> value;

        buffer.push_back(value);
    }

    int64_t min = buffer[0];
    for (size_t i = 1; i < size; i++) {
        if (buffer[i] < min) {
            min = buffer[i];
        }
    }

    return 0;
}
```

qsort.exe

```
#include <vector>
```

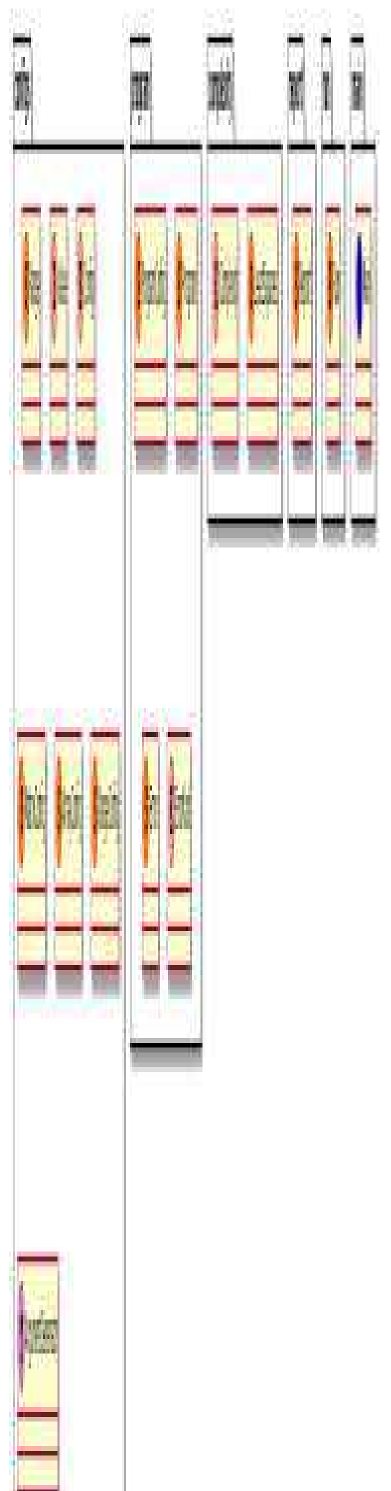
```
#include <fstream>
#include <algorithm>

int main(int argc, char *argv[]) {
    auto stream = std::fstream(argv[1], std::fstream::in);

    size_t size;
    stream >> size;

    std::vector<int64_t> buffer(size);
    for (size_t i = 0; i < size; i++) {
        int64_t value;
        stream >> value;

        buffer.push_back(value);
    }

    std::sort(buffer.begin(), buffer.end());
    return 0;
}
```

sort.exe

```
#include <vector>
#include <fstream>
#include <algorithm>

int main(int argc, char *argv[]) {
    auto stream = std::fstream(argv[1], std::fstream::in);

    size_t size;
    stream >> size;

    std::vector<int64_t> buffer(size);
    for (size_t i = 0; i < size; i++) {
        int64_t value;
        stream >> value;

        buffer.push_back(value);
    }

    for (size_t i = 0; i < size - 1; i++) {
        for (size_t j = i + 1; j < size; j++) {
            if (buffer[i] > buffer[j]) {
                std::swap(buffer[i], buffer[j]);
            }
        }
    }

    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Диаграмма модуле



Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт

Вычислительная техника
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой

______ О.В. Непомнящий
подпись инициалы, фамилия
« 10 » 06 20 21 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.01 — Информатика и вычислительная техника
код – наименование направления

Динамический анализатор асимптотических временных сложностей
тема

Руководитель 8.06.21 ______ ст. преподаватель И.В. Матковский
подпись, дата должность, ученая ступень инициалы, фамилия

Выпускник ______ 08.06.21 М.В. Соколов
подпись, дата инициалы, фамилия

Нормоконтролер 8.06.21 ______ И.В. Матковский
подпись, дата инициалы, фамилия

Красноярск 2021