МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Плешкова Андрея Владимировича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ РАСПРЕДЕЛЕННОЙ СБОРКИ МУСОРА В СИСТЕМЕ LUNA

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э./……..
(ФИО) / (подпись)
«27»…мая…2021 г.

Руководитель ВКР
к.т.н, доцент
доцент каф. ПВ ФИТ НГУ
Маркова В.П. /……...
(ФИО) / (подпись)
«27»…мая…2021 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В.А. /……...
(ФИО) / (подпись)
«27»…мая…2021 г.

Новосибирск, 2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)

…….
(подпись)

«17» декабря 2020 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студент*у(ке)*……Плешкову Андрею Владимировичу……..…, группы…17201……
(фамилия, имя, отчество, номер группы)

Тема…Разработка и реализация алгоритмов распределенной сборки мусора в системе LuNA……
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от…17.12.2020…№0451
Срок сдачи студентом готовой работы…31…мая…2021 г.

Исходные данные (или цель работы):..разработать и реализовать алгоритм сборки мусора для задач численного моделирования и итерационных процессов в системе фрагментированного программирования LuNA ……

Структурные части работы: …обзор литературы, постановка задачи, разработка и реализация алгоритма, тестирование……
Консультанты по разделам ВКР (при необходимости, с указанием разделов):
……
(раздел, ФИО)

Руководитель ВКР
доцент каф. ПВ ФИТ НГУ,
к.т.н, доцент
Маркова В.П. /……..
(ФИО) / (подпись)

«17»…декабря…2020 г.

Задание принял к исполнению
Плешков А. В./……..
(ФИО студента) / (подпись)

«17»…декабря…2020г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ,
Перепёлкин В.А. /……...
(ФИО) / (подпись)

«17»…декабря…2020 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Каждая программная система, использующая динамическую память, нуждается в механизмах её освобождения. К таким механизмам относится ручное управление памятью, из-за которого могут возникать такие ошибки как существование указателей на уже освобождённую память и, наоборот, её утечек. Также, языки программирования, использующие данный подход усложняют процесс написания программ, не содержащих вышеуказанных ошибок и имеют больший порог вхождения в систему для программиста, чем языки с автоматическим управлением памятью.

Для того, чтобы упростить процесс управления памятью в 60-е годы прошлого столетия был предложен такой механизм как сборка мусора. На данный момент этот подход автоматического освобождения памяти популярен в современных языках, так как упрощает работу программиста и предотвращает появление вышеописанных проблем.

Процесс сборки мусора существенно усложняется при переходе к задаче автоматического управления памятью в системах с распределённой памятью, по сравнению с нераспределёнными системами, поскольку узлы системы могут ссылаться на локальные объекты других узлов вычислительной системы. Применение последовательных алгоритмов не подходит из-за невозможности напрямую адресовать память других узлов, а в силу того, что все коммуникации проходят по сети, получение достоверной картины использования объектов на других узлах приведёт к большим коммуникационным нагрузкам, что, в свою очередь, усложняет масштабирование вычислительной системы. Несмотря на то, что к настоящему времени было разработано достаточно много алгоритмов сборки мусора в распределенной системе, которые могут работать без синхронизации, а также написано большое количество работ по этой теме, многие ее аспекты остаются недостаточно изученными, особенно если рассматривать вопросы практического применения.

Создание системы сборки мусора зачастую требует решения ряда задач,

обусловленных требованиями системы, в контексте которой разрабатываются алгоритмы и механизмы автоматического управления динамической памятью. В зависимости от задач, решаемых распределённой системой, сборщик мусора должен учитывать коммуникации между узлами, наличие связей между объектами, а также расположением и управлением распределёнными ресурсами.

В распределённых вычислениях, особенно в высокопроизводительных системах, из-за сложности написания программ для суперкомпьютеров, актуальна автоматизация программирования, следовательно и автоматическая сборка мусора. Особенную актуальность автоматическая сборка мусора имеет в системах, где ручное управление памятью требует слишком больших трудозатрат и отнимает у программиста больше времени, чем написание самой программы. В частности, именно там требуется решение проблемы сборки мусора. Система фрагментированного программирования LuNA, разработанная с целью упростить написание параллельных программ, является подходящей площадкой для исследования проблемы распределённой сборки мусора из-за особенностей архитектуры, позволяющей сосредоточиться на разработке самих алгоритмов. Одним из самых распространённых типов задач в системе, являются задачи численного моделирования и итерационные процессы на сетках, что относится к задачам, где каждую итерацию может происходить выделение больших объёмов памяти.

Целью работы была поставлена разработка и реализация алгоритма сборки мусора для задач численного моделирования и итерационных процессов в системе фрагментированного программирования LuNA.

Задачи:

1. Проанализировать существующие решения для сборки мусора.
2. Разработать алгоритм для осуществления автоматического освобождения неиспользуемой памяти в задачах численного моделирования и итерационных процессов.
3. Реализовать разработанный алгоритм в виде модуля для компилятора.

Практическая ценность решения заключается в создании сборщика мусора в распределённой системе в качестве автоматического механизма управления памятью вместо использования механизмов ручного управления, что позволит:

- снизить порог вхождения в систему для новых пользователей
- ускорить процесс написания программ, при этом не теряя эффективности по памяти
- снизить количество ошибок, связанных с ручным управлением памятью, таких как преждевременное или несвоевременное удаление неиспользуемых объектов

Научная новизна заключается в разработке алгоритма сборки мусора в распределённой системе, учитывающий особенности систем фрагментированного программирования, в частности, системы LuNA.

Проделанная работа вносит вклад в решение проблемы сборки мусора в распределённых системах.

Работа изложена в трёх главах, в первой описывается анализ существующих решений и их применимость в существующей системе, во второй поставленные для решения задачи и описание предложенного решения, в третьей представлены практические результаты.

1 Анализ предметной области

Самым распространённым решением для задачи сборки мусора является динамический сборщик мусора, специальный процесс, периодически освобождающий уже неиспользуемую память [15]. Сборщик мусора не должен освобождать ещё используемую память, что можно сформулировать как “действия сборщика мусора должны оставлять логическую структуру программы неизменной” [19]. В контексте распределённой системы, основным требованием, предъявляемым к динамическому алгоритму сборщика мусора, являются масштабируемость.

Наивным алгоритмом подсчёта ссылок в распределённой системе является хранение числа ссылок на каждый доступный объект, увеличиваемого каждый раз при создании или копировании объектов, с помощью отправки специального сообщения владельцу ссылки [5].

Отправка таких сообщений существенно увеличивает коммуникационные нагрузки, а обработка сообщений владельцем должна осуществляться в порядке их отправки, чтобы предотвратить состояние гонки. Гонка между сообщениями об увеличении и уменьшении числа ссылок может привести к тому, что объект будет скопирован, а затем сразу же удалён. Одним из способов избежать состояние гонки является подтверждение получения каждого сообщения об увеличении, прежде чем отослать ссылку.

Взвешенный алгоритм подсчёта ссылок был предложен как более эффективная альтернатива подсчёта ссылок. Владелец каждого удалённого объекта задаётся 2 веса: частичный и полный. Полный вес задаётся значением больше нуля, остаётся неизменным с момента создания ссылки и уменьшается при удалении. Частичный вес задаётся равным полному и каждый раз, когда создаётся ссылка на объект, уменьшается в 2 раза, отнятая половина отправляется клиенту, где используется как значение для инициализации частичного веса для ссылки на удалённый объект [7].

После того, как клиент удаляет свою копию, он отправляет сообщение со своим частичным весом, который прибавляется к частичному весу, хранящемуся у владельца объекта.

Контроль веса ссылок позволяет убедиться в отсутствии потерь сообщений, так как для каждого объекта справедлив инвариант: полный вес равен сумме всех частичных весов. Все ссылки на объект удалены, когда полный вес входного значения будет снова равен частичному.

Оптимизированный взвешенный алгоритм подсчёта ссылок улучшает предшественника в двух аспектах: устойчивость к потере сообщений и косвенные входные значения [9].

Это достигается за счёт нового инварианта: полный вес больше либо равен сумме частичных весов. Таким образом, потеря сообщения не нарушает новый вариант, однако двойное уменьшение ссылки может привести к тому, что сумма частичных весов будет больше либо равна полному весу.

Оптимизированный взвешенный алгоритм избегает создания косвенных ссылок, когда частичный вес не может быть разделён, используя специальные веса со значением null. В таком случае полный вес будет всегда больше, чем сумма частичных, однако это является слабой ссылкой на объект, то есть в любой момент времени ссылка может стать недействительной.

Косвенный подсчёт ссылок был придуман, чтобы справиться с основным недостатком взвешенного подсчёта ссылок, а именно с ограничением на копирование ссылок. При инициализации веса значением, равным 2^k, можно осуществить лишь k копирований, после чего значение 2^0 будет неразделимым [16].

В худшем случае, ссылки выстраивались в неэффективно длинную цепочку косвенных ссылок. Одним решением было увеличение количества битов под веса, однако это не позволит избежать создания косвенных ссылок.

Ключевая особенность косвенного подсчёта в том, чтобы хранить два локатора, аналог указателя, необходимого, чтобы ссылаться с одного узла вычислительной системы на другие. Сильный локатор указывает на объект,

находящийся у владельца, дополнительный слабый локатор срезает путь, указывая на лучшее, с точки зрения топологии сети, местонахождение объекта. При отсутствии миграций объектов, слабый локатор абсолютно точен. Сильный локатор должен использоваться только для распределённой сборки мусора, например для предотвращения удаления основного объекта. Копирование ссылок происходит локально, без обращения к владельцу объекта, чтобы избежать состояния гонки.

Копирование может создавать косвенные ссылки, поэтому чтобы избежать циклов, создаются уникальные идентификаторы. Идентификатор посылается вместе с ссылкой, чтобы распознать использование конкретного объекта. Отправка идентификаторов не является большой проблемой сама по себе, однако становится таковой с масштабированием системы.

Недостатком этого алгоритма, как и взвешенного подсчёта, являются длинные цепочки ссылок, избежать которых можно с помощью введения счётчиков на каждом узле, что, в свою очередь приводит к значительному использованию памяти.

Список ссылок отличается от подсчёта ссылок тем, как владелец управляет ссылками на объект. Для каждого клиента, владелец объекта хранит список клиентов, держащих ссылку на объект. Каждый элемент списка содержит идентификатор его клиента. Увеличение и уменьшение числа ссылок заменены на вставку и удаление из списка. Удаление последнего элемента из списка сигнализирует о том, что значением больше никто не пользуется [6].

Список ссылок увеличивает устойчивость к потере сообщений взамен на большие расходы памяти и постоянную коммуникацию клиента с владельцем объекта, что не учитывает топологию сети. Основным преимуществом перед подсчётом ссылок является идемпотентность сообщений об удалении и вставки.

Алгоритм Stub-Scion Pair Chains проектировался для распределённых систем без общей памяти, возможными сбоями и дорогими и ненадёжными сообщениями, что уже создаёт издержки при реализации. Ссылка с удалённого узла представляется в виде цепочки ссылок на узлы, с которых ссылка была

получена, таким образом отслеживая путь объекта. Цепочка ссылок позволяет знать как владельца объекта, так и короткий путь для копирования ссылки, с целью уменьшения числа коммуникаций. При отправке ссылки, получатель создаёт узел из ссылки на удалённый узел и локальной ссылки. Достижимость узлов проверяется через обмен специальными сообщениями, что увеличивает нагрузку на сеть. Устойчивость к потере сообщений достигается за счёт их идемпотентности [18].

Все вышеперечисленные алгоритмы не рассчитаны на работу в среде, где объекты могут ссылаться друг на друга и образовывать циклические зависимости. В системе фрагментированного программирования LuNA данные не могут содержать ссылки, что означает теоретическую применимость вышеуказанных алгоритмов. Для получения полной картины рассмотрим алгоритмы динамической сборки мусора, поддерживающие циклы данных.

Для решения этой проблемы были разработаны **гибридные сборщики мусора**, объединяющие два подхода, локальный подсчёт числа ссылок и глобальный циклический сборщик мусора, способный по графу объектов находить недостижимые циклы между объектами и собирать их. Обычно циклический распределённый сборщик мусора вызывается реже и большинство мусора удаляется с помощью ациклического сборщика [4].

Метод трассировки является расширением алгоритма Mark and Sweep, применяемого для нераспределённых систем. Алгоритм состоит из двух фаз, что приводит к необходимости синхронизации фаз узлов. После завершения синхронизированной фазы Mark, разметки всех доступных объектов, будет инициирована независимая фаза Sweep для удаления всех объектов, недостижимых с других узлов [11].

Оптимизированный взвешенный алгоритм подсчёта ссылок, объединённый с циклическим глобальным сборщиком мусора использует множества графов Эйлера для организации всех достижимых объектов, обход каждого из которых занимает линейное время [8]. Позднее алгоритм был усовершенствован с помощью метода трассировки: глобальный циклический

сборщик, основанный на распределённой трассировке, окрашивал удалённые ссылки на объекты и отслеживал их через распределённый граф объектов [17].

В результате анализа существующих решений динамической сборки мусора можно сделать вывод о том, что динамическая распределённая сборка мусора является тяжёлой задачей с точки зрения её исполнения, а любой алгоритм будет создавать издержки, так как был спроектирован и разработан с расчётом на применение в определённой системе. Исходя из всего вышесказанного, было решено рассмотреть способы отслеживания жизненного цикла объектов на этапе компиляции.

Для отслеживания жизненного цикла, сначала необходимо решить задачу поиска зависимости по данным, самым популярным и используемым алгоритмом для решения которой является построение **графа зависимостей**.

В общем случае, задача построения графа зависимостей сводится к выявлению в программе всех возможных состояний, а также выяснению зависимостей между ними. Программа, в таком случае, рассматривается как система, имеющая конечное число состояний [1]. Если программа удовлетворяет этому требованию,такой подход сможет полностью заменить динамическую сборку мусора без создания лишней нагрузки на систему исполнения.

Среди алгоритмов поиска зависимостей между данными во время статического анализа, сложность алгоритма разделяют в зависимости от типа поддерживаемых данных:

1. Zero Index Variable (ZIV) – в ссылке отсутствует индексация
2. Single Index Variable (SIV) – ссылка является одномерным массивом и содержит только один индекс
3. Multiple Index Variable (MIV) – ссылка является N-мерным массивом и содержит N индексов [13].

В случае MIV, важную роль играет разделимость индексов – в случае когда один и тот же индекс участвует в нескольких частях выражения, неразделимость может давать потерю точности анализа данных. Разработанный

для решения этой проблемы подход **Subscript-by-subscript** предлагает хранить многоразмерные индексы как вектора, что позволяет оперировать над ними в любом удобном виде, а для поиска зависимостей можно воспользоваться поиском множества пересечений. Единственным недостатком этого подхода является случай, когда индекс представляет несуществующий вектор [20].

Векторный подход способствовал разделимости индексов, но не учитывал расстояния между векторами, поэтому позднее алгоритм был доработан, чтобы указывать расстояние между элементами массива, что может быть использовано для оптимизации [12].

Алгоритм на основе разделов распознаёт зависимости по данным группируя индексы на минимально разделяемые группы, содержащие одинаковые индексы, маркирует их, а затем используя вектора для каждой группы, либо определяют отсутствие зависимостей между группами, либо сливают все вычисленные направления векторов несколько групп в одну [10].

Существующие алгоритмы статического анализа не позволяют отслеживать использование объектов, соответственно не решают проблему в полной мере. Необходимость исследовать проблему приводит к разработке собственного алгоритма, в основе которого можно использовать концепцию подсчёта ссылок на этапе компиляции программы и статический анализ зависимостей между данными.

2 Алгоритм автоматической сборки мусора

2.1 Термины и определения

Для понимания задачи, необходимо ввести следующие понятия:

Фрагмент данных – агрегат из переменных [14]. Каждый фрагмент является неизменяемым и инициализируется единожды. В абстрактном синтаксическом дереве представлен в виде идентификатора.

Фрагмент вычислений – единица программы, содержащая описание входных и выходных фрагментов данных и исполняемого кода фрагмента без побочных эффектов [14]. В абстрактном синтаксическом дереве представлен в виде идентификатора.

Выражение – синтаксическая единица языка программирования, выполнение которой приводит к вычислению её значения. Является комбинацией одной или более констант, фрагментов данных и операций, которая может быть интерпретирована в соответствии с правилами конкретного языка [21].

Рекомендация – механизм ручного управления поведением системы исполнения, используется чтобы сообщить компилятору или системе дополнительную информацию [2]. В зависимости от назначения рекомендации, может иметь список аргументов, описанных в виде выражений.

Оператор – синтаксическая единица языка программирования, означающая некоторое исполняемое действие [22]. Оператором являются объявление фрагментов данных, вызовы подпрограмм и процедур, условные операторы, циклы и другое. С оператором может быть ассоциировано порождение одного или нескольких фрагментов вычислений. В абстрактном синтаксическом дереве программы оператор состоит из следующих блоков:

- список аргументов, каждый из которых является выражением
- тело оператора, является списком операторов
- список рекомендаций, применимых к оператору

Подпрограмма – идентифицированная часть программы, содержащая описание определённого набора действий, может быть многократно вызвана из разных частей программы. В абстрактном синтаксическом дереве, описание подпрограммы состоит из следующих блоков:

- список аргументов
- тело подпрограммы, является списком операторов
- список рекомендаций, применимых к подпрограмме

Абстрактное синтаксическое дерево – представление программы, доступное в момент компиляции для чтения и модификации. На верхнем уровне, дерево состоит из списка процедур и подпрограмм, объявленных в программе. Точкой входа в программу является подпрограмма main.

Таким образом, абстрактное синтаксическое дерево программы можно представить следующим образом:

- подпрограмма
  - список аргументов, пустой, в случае main
  - тело подпрограммы, состоит из списка операторов
    - объявление фрагментов данных
    - вызов подпрограммы
      - идентификатор фрагмента вычислений
      - вызываемое имя
      - список аргументов, каждый является выражением
      - список рекомендаций
    - условный оператор
      - идентификатор фрагмента вычислений
      - выражение
      - тело, также состоит из списка операторов
        - ...
      - список рекомендаций
        - в зависимости от рекомендации, аргументы
    - …

- ...

2.2 Постановка задачи

Для того, чтобы изложить постановку задачи, рассмотрим следующие положения:

1. Фрагментированная программа состоит из множества фрагментов данных и фрагментов вычислений. Процесс выполнения программы рассматривается как исполнение фрагментов вычислений, динамически потребляющих и вырабатывающих фрагменты данных. Фрагменты вычислений отображаются на узлы вычислительной системы, которая, в процессе исполнения, может автоматически удалять фрагменты данных и перемещать их между узлами.
2. Важной особенностью распределённой системы является тот факт, что данные могут храниться на различных узлах, следовательно, для вычисления значения выражения, в котором участвуют фрагменты данных, необходимо подготовить их, предварительно запросив. Запрос является рекомендацией, применимой к оператору, содержащему вычисляемое выражение, и совершает потребление фрагмента данных.
3. В технологии фрагментированного программирования, каждый фрагмент данных имеет свой жизненный цикл, который можно представить в виде некоторого числа потреблений или области видимости, ограниченной телом оператора или подпрограммы, в которой объявлен фрагмент данных.

Таким образом, под созданием решения для сборки мусора в системе фрагментированного программирования подразумевается разработка автоматического механизма освобождения памяти, выделенной для хранения значения фрагментов данных, потребления которых уже не произойдёт.

2.2.1 Требования, предъявляемые к алгоритму

1. Алгоритм не должен вносить модификации в исходную программу, в случае, когда нет уверенности в том, количество потреблений фрагмента данных вычисленно правильно.
2. В идеальном случае, пользователь системы не должен задумываться об управлении памятью, и всё же, нужно явно выделить случаи, в которых сборка мусора не произойдёт. Выделение таких случаев позволит пользователям системы обрабатывать только эти случаи вручную.
3. Любой язык программирования постоянно развивается, обрастая новыми конструкциями. В контексте системы LuNA очевидной точкой расширения является введение новых рекомендаций и операторов. Алгоритм не обязан поддерживать конструкции, не существовавшие на момент его разработки, но необходимо учитывать их возможное появление. Одним из возможных решений является не пытаться вычислить количество потреблений фрагмента данных, задействованного в неподдерживаемой конструкции.

2.3 Описание решения

Основной идеей алгоритма является построение специального дерева потреблений для каждого фрагмента данных, используемого в программе. Вычисление каждого дерева потребления позволит определить количество потреблений фрагмента данных, с которым ассоциировано дерево.

Входными параметрами алгоритма являются абстрактное синтаксическое дерево программы, в котором содержится вся необходимая информация о потреблениях фрагментов данных, и список, содержащий идентификаторы фрагментов данных, которые освобождаются с помощью других механизмов, чтобы не отслеживать их жизненный цикл.

На рисунке 1 представлен псевдокод алгоритма построения множества деревьев потреблений в результате обхода абстрактного синтаксического дерева. Алгоритм оперирует такими понятиями, как область видимости (scope),

структура, содержащая ссылку на родительскую область видимости, если такая имеется, хранящая все объявленные фрагменты данных и их потребления.

```
1   # обход списка подпрограмм
2   procedure traverse_ast(ast) {
3       foreach (sub_prog in ast) {
4           # создаём область видимости для подпрограммы
5           sub_scope
6           sub_scope.dfs = sub_prog.args
7           traverse_body(sub_prog.body, sub_scope)
8       }
9   }
10
11  # обход списка операторов из которых состоит тело
12  procedure traverse_body(body, parent_scope) {
13      # область видимости для локальных фрагментов данных
14      local_scope
15      local_scope.parent = parent_scope
16
17      foreach (statement in body) {
18
19          # добавить потребления для всех запрошенных фрагментов данных
20          add_usages(local_scope, statement.requests)
21
22          # если тело существует
23          if (statement.body) {
24              # traverse_body возвращает информацию о фрагментах,
25              # использованных в теле оператора
26              sub_scope = traverse_body(statement.body, local_scope)
27          }
28
29          # всем фрагментам данных, использованных в теле оператора
30          # добавить использования в зависимости от типа оператора
31          apply_statement_type(sub_scope, statement.type)
32      }
33      return local_scope
34  }
35
36  procedure add_usages(scope, usages) {
37
38      foreach (usage in usages) {
39          temp_scope = scope
40          # поднимаемся наверх, расширяя область видимости
41          # пока не найдём нужный фрагмент данных
42          while (!temp_scope.contains(usage.df)) {
43              temp_scope = temp_scope.parent
44          }
45          temp_scope.add(usage)
46      }
47  }
```

Рисунок 1 – Псевдокод обхода абстрактного синтаксического дерева

Чтобы иметь наглядное представление о работе алгоритма, рассмотрим его на примере программы на рисунке 2, где A является некоторым фрагментом данных. Данной программе соответствует абстрактное синтаксическое дерево, изобаражённое на рисунке 3.

```
1   sub main()
2   {
3       df A;
4       init(A);
5
6       if (A > 5)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      }
13  }
```

Рисунок 2 – Пример фрагментированной программы

Чтобы понимать, в каких случаях происходит потребление фрагмента данных, алгоритм использует запросы из абстрактного синтаксического дерева программы. Алгоритм учитывает тип оператора, в теле которого происходит потребление. Исходя из примера, где в теле цикла (оператор for), происходит два потребления, cf_c1 и cf_c2 на рисунке 3, для правильного подсчёта числа потреблений нужно умножить потребления в теле на число итераций цикла.

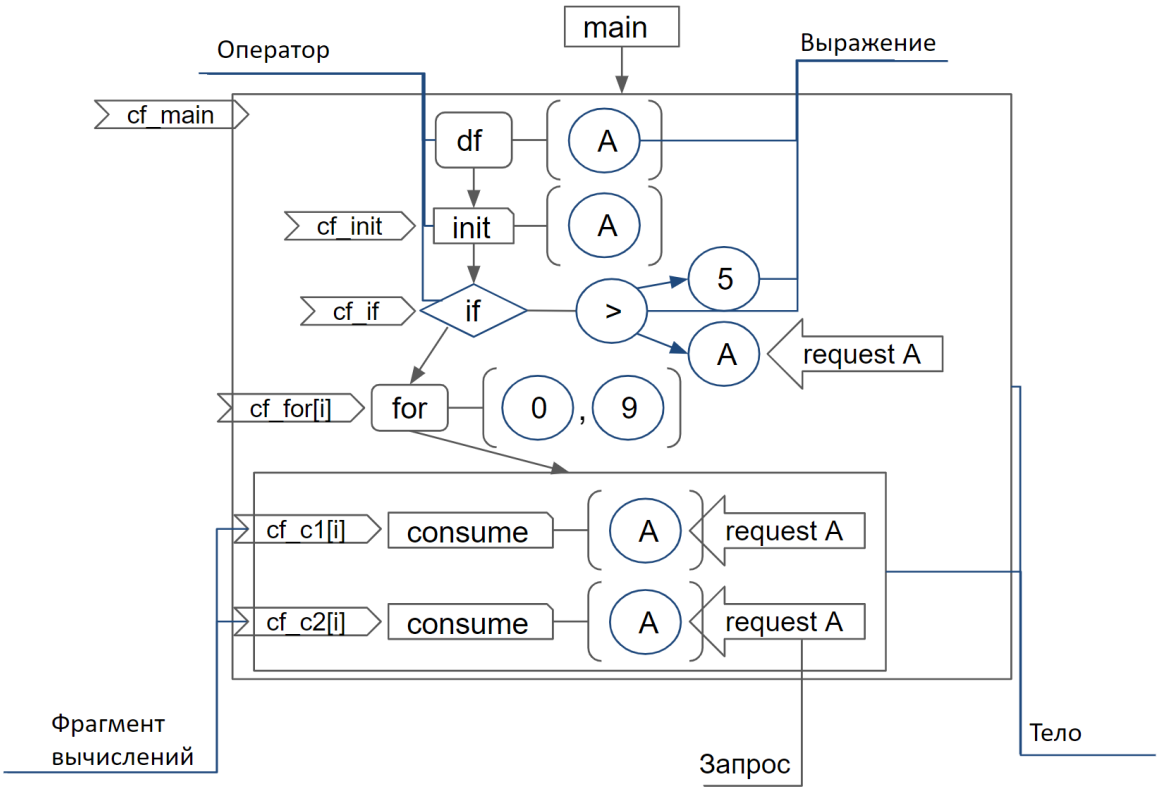

Рисунок 3 – Пример абстрактного синтаксического дерева

Промежуточным результатом работы алгоритма будет построенное дерево потреблений для фрагмента данных A, изображённое на рисунке 4.

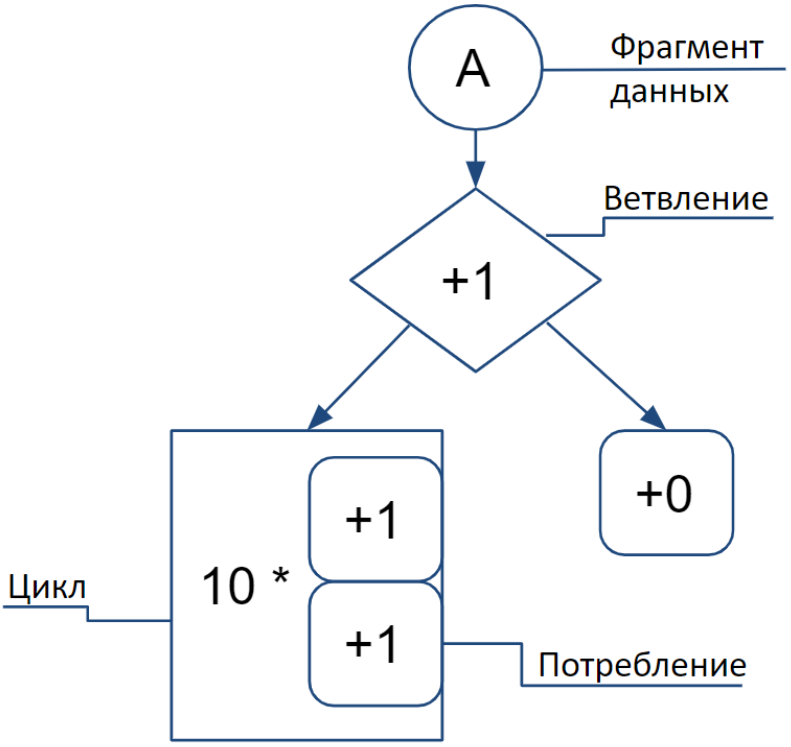

Рисунок 4 – Дерево потреблений для фрагмента данных A

Далее рассмотрим особенности работы алгоритма в контексте фрагментированного программирования.

2.3.1 Интерпретация результата алгоритма

Количество потреблений не всегда можно вычислить на этапе компиляции, именно для этого строится дерево потреблений, но и оно не является окончательным результатом работы алгоритма. Каждая ветвь дерева потреблений сопоставляется с некоторым ветвлением в анализируемой программе, соответственно, для вычисления количества потреблений, будет выбрана та ветвь, сопоставленный аналог которой выполнится во время работы программы. Обход дерева от корня до листа, выбирая все выполненные ветви программы позволяет посчитать конечное число потреблений фрагмента данных. Так, для программы на рисунке 2 и дерева потреблений на рисунке 4, если значение фрагмента данных A больше 5, число потреблений будет равняться 21, иначе 1.

Во время компиляции мы не можем определить какая из ветвей дерева будет выполнена, поэтому необходимо вставить всё дерево в код программы в виде формулы, понятной для системы, с помощью механизма рекомендаций. Таким образом, число потреблений фрагмента данных A из программы на рисунке 2 можно представить в виде формулы на рисунке 5.

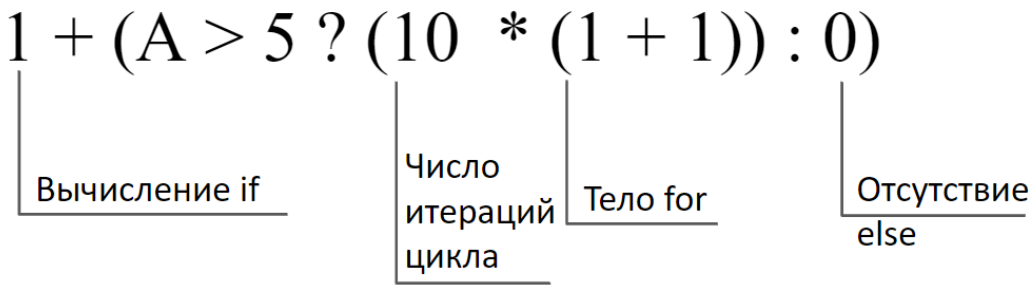

Рисунок 5 – Формула для вычисления количества потреблений фрагмента A

2.3.2 Косвенные потребления

Несмотря на то, что фрагменты данных не могут содержать ссылки на другие фрагменты данных, количество потреблений одного фрагмента данных может зависеть от значения другого. Можно разделить потребления на прямые и косвенные. Прямые потребления – потребления, необходимые для вычисления значения выражений, фигурирующих в операторах. Косвенные потребления – потребления, необходимые для вычисления количества потребления других фрагментов данных.

Так, на рисунке 6, количество употреблений фрагмента данных A зависит от значения фрагмента данных B.

```
1   sub main()
2   {
3       df A, B;
4       consume(A);
5
6       if (B > 5)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      } else {
13          consume(A);
14      }
15  }
```

Рисунок 6 – Пример фрагментированной программы с косвенным потреблением

Из текста программы видно, что значение фрагмента данных B используется только один раз, для вычисления, какая из ветвей условного оператора будет выполнена. Можно сказать, что фрагмент данных B имеет одно прямое потребление. На рисунке 7 построены деревья потреблений для фрагментов данных A и B. Из-за того, что в дереве потреблений фрагмента данных A используется фрагмент данных B, необходимо добавить к дереву потреблений фрагмента B ещё одно, косвенное, выделенное красным цветом.

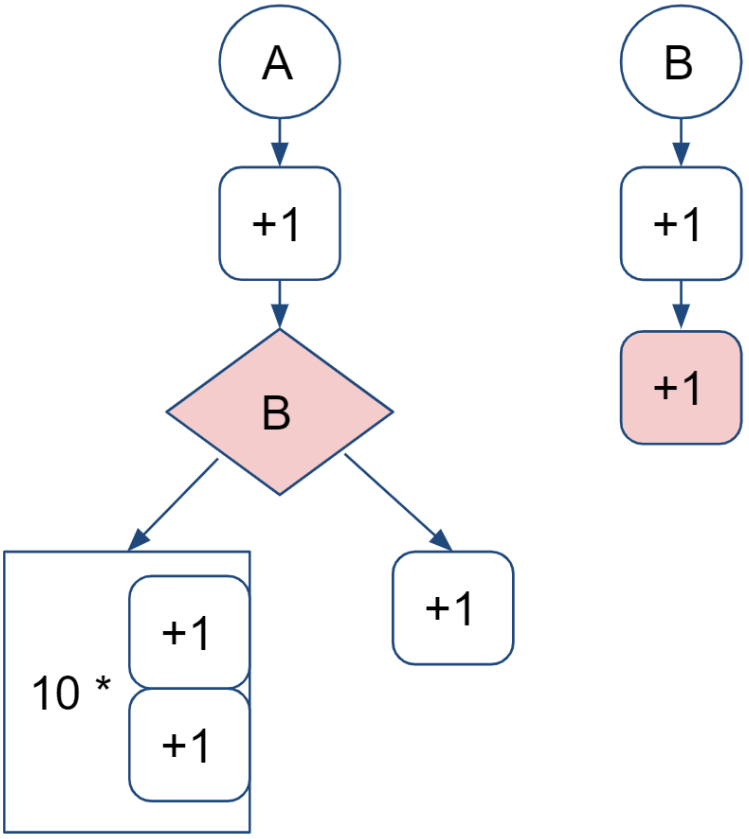

Рисунок 7 – Множество деревьев потреблений для фрагментов A и B

2.3.3 Массивы

Массивы во фрагментированной программе также состоят из фрагментов, что приводит к тому, что у разных элементов массива может быть разный жизненный цикл и место инициализации.

Не зная на этапе компиляции размер массива, мы не можем завести дерево потреблений на каждый элемент, не говоря уже о потенциальных проблемах с выделением большого количества памяти под все деревья. Разработанный мной алгоритм решает эту задачу следующим образом – с точки зрения алгоритма, при работе с массивом, обращение происходит по заранее неизвестному индексу и сохраняется как условное обращение. Например, для программы на рисунке 8, будет построено дерево, представленное на рисунке 9.

```
1   sub main()
2   {
3       df A;
4       consume(A[0]);
5   }
```

Рисунок 8 – Пример обращения к элементу массива по заранее известному индексу во фрагментированной программе

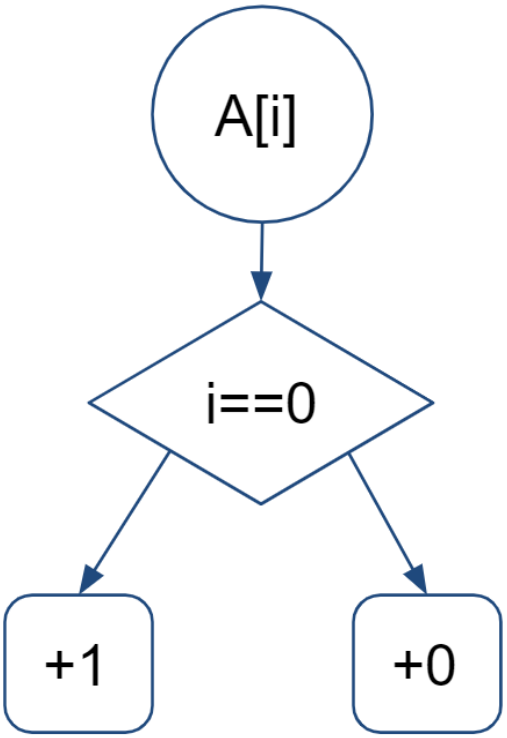

Рисунок 9 – Пример дерева потреблений для одномерного массива с обращением к элементу с индексом 0

Результат работы алгоритма можно интерпретировать как “обращение будет совершено, если элемент является нулевым”.

Такой способ также позволяет работать с заранее неизвестным индексом. Так, например, для программы на рисунке 10, где в цикле по индексу от 3 до N происходит одно обращение к i-тому элементу, для массива A и фрагмента данных N будут построены деревья, изображённые на рисунке 11. Важно заметить то, что N имеет одно прямое потребление и N косвенных, по одному на каждый элемент массива A, учитывая, что размер массива равен N.

```
1   sub main()
2   {
3       df A, N;
4
5       for i=3..N-1 {
6           consume(A[i]);
7       }
8   }
```

Рисунок 10 – Пример обращения к элементу массива по счётчику цикла во фрагментированной программе

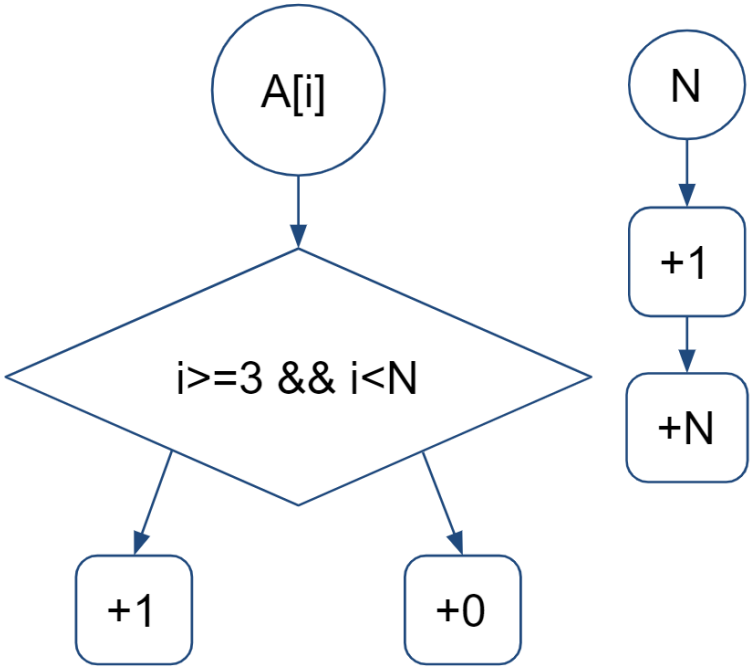

Рисунок 11 – Пример деревьев потреблений для одномерного массива с обращением к некоторому числу элементов в цикле и фрагмента данных, являющегося размером массива

2.3.4 Несколько использований в одном выражении

Рассмотрим тот случай, когда жизненный цикл фрагмента данных в нескольких местах зависит от другого фрагмента, как на рисунке 12. В наивной реализации, для вычисления формулы фрагмента A, представленной в виде дерева на рисунке 13, могло бы потребоваться несколько запросов, но, так как дерево потреблений транслируется в одно выражение, для его вычисления нужно совершить лишь одно косвенное потребление фрагмента данных B.

```
1   sub main()
2   {
3       df A, B;
4       consume(A);
5
6       if (B > 0)
7       {
8           for i=0..9 {
9               consume(A);
10              consume(A);
11          }
12      } else {
13          if (B == 0)
14          {
15              consume(A);
16          }
17      }
18  }
```

Рисунок 12 – Пример фрагментированной программы с несколькими ветвлениями, зависящими от значения одного фрагмента данных

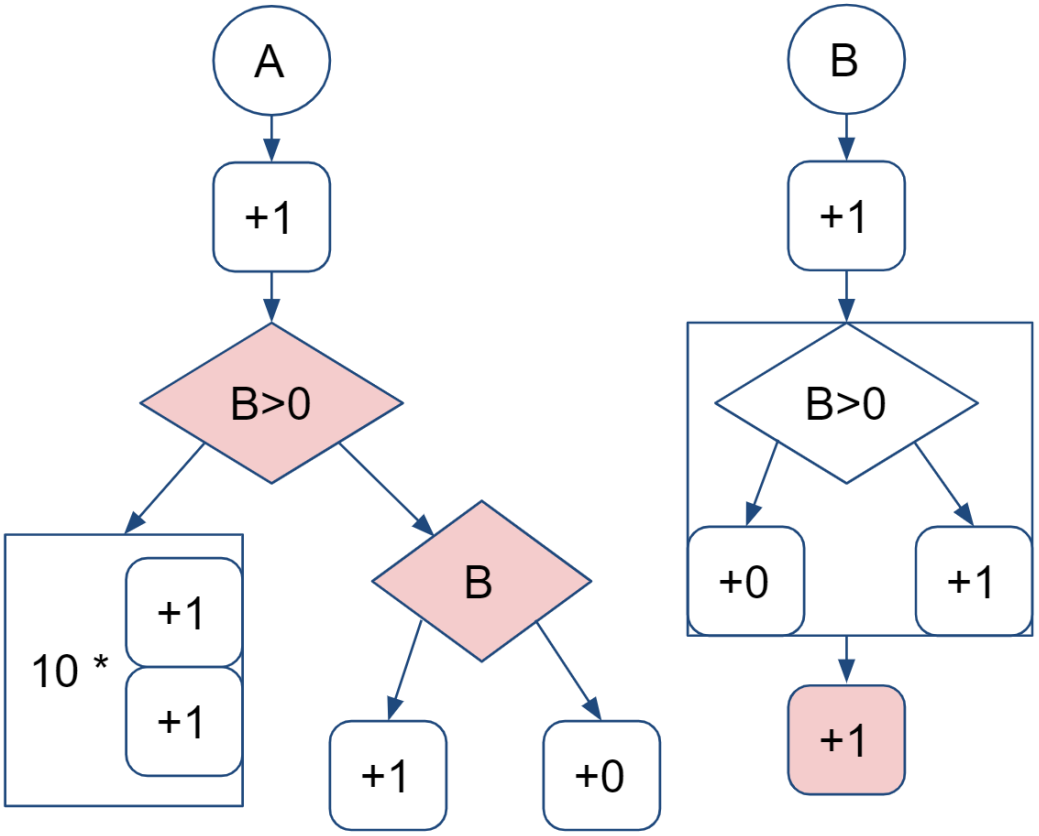

Рисунок 13 – Множество деревьев потребления для фрагментов данных A и B

2.3.5 Вызовы подпрограмм

Вызовы подпрограмм также требуют дополнительного рассмотрения. Передача фрагмента данных в подпрограмму не является потреблением, однако операторы, содержащиеся в теле подпрограммы могут совершать обращения к переданному фрагменту. Так как подпрограмма может использоваться неограниченное количество раз с разными фрагментами данных в качестве входных параметров, возникает необходимость построения дерева потреблений для каждого входного параметра подпрограммы и локального фрагмента данных.

На рисунке 14 представлен пример множества построенных деревьев для некоторой подпрограммы, принимающей на вход один фрагмент данных A и создающей B локально, с областью видимости, ограниченной телом подпрограммы.

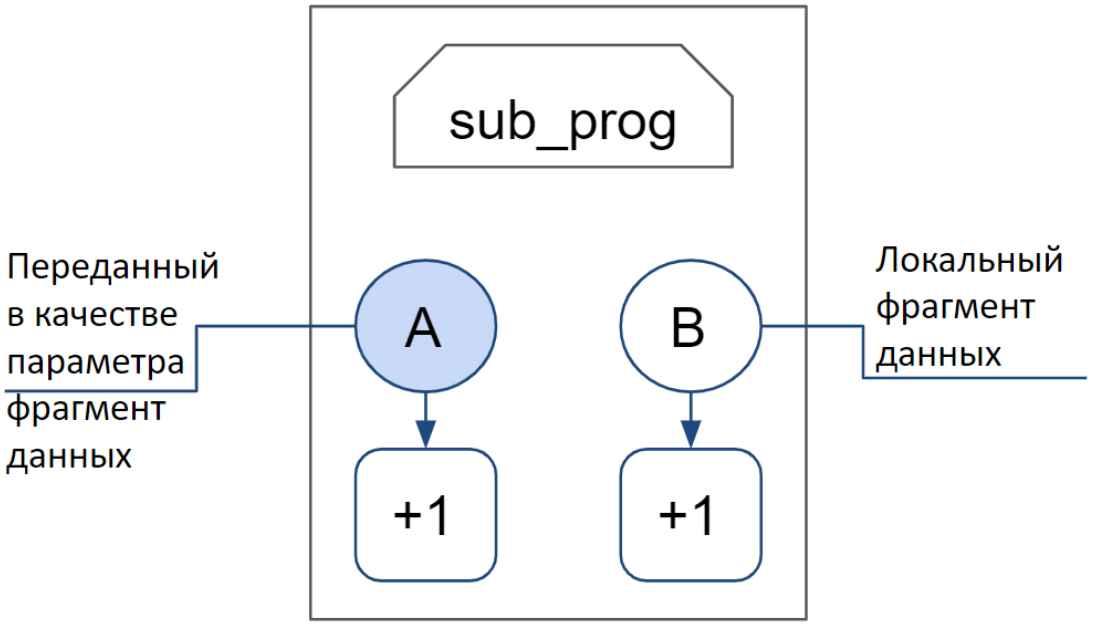

Рисунок 14 – Множество деревьев потреблений, построенных для подпрограммы

Таким образом, при вызове подпрограммы, останется лишь сопоставить фрагмент данных, переданный в подпрограмму и его дерево потреблений в подпрограмме, как на рисунке 15.

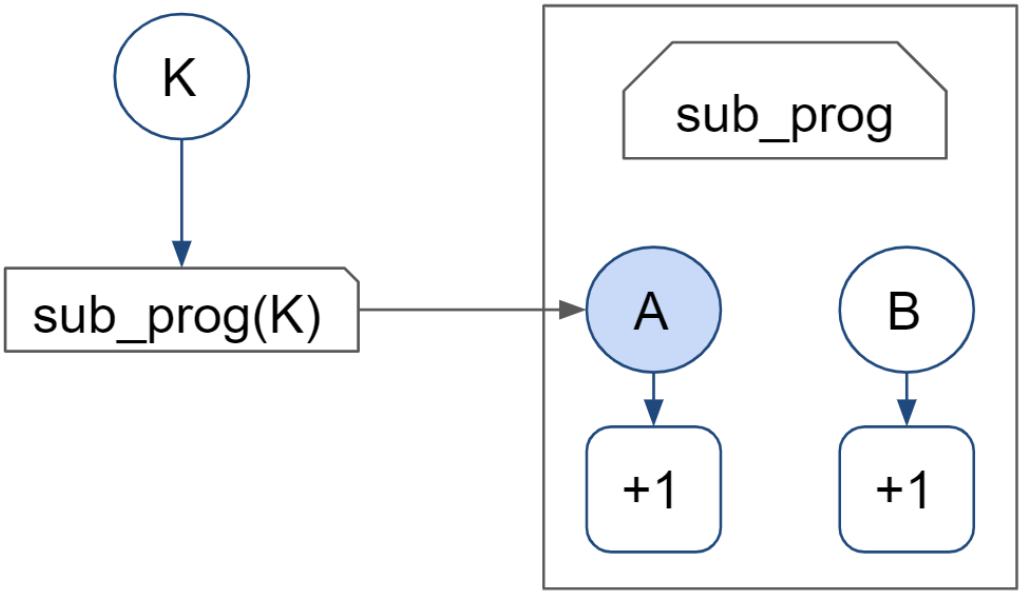

Рисунок 15 – Пример сопоставления дерева потреблений фрагмента данных с деревом потребления аргумента подпрограммы

Сопоставление аргументов является важной частью алгоритма, хоть это и кажется очевидным в случае, когда фрагмент данных сопоставляется напрямую, на рисунке 16 приведён пример подпрограммы, принимающей двумерный массив. Эта подпрограмма может быть вызвана как для матрицы, так и для элемента трёхмерного массива. В случае элемента массива важно сопоставить вызов подпрограммы таким образом, чтобы дерево потреблений соответствовало рисунку 17.

```
1   sub matrix(name M)
2   {
3       for i=0..9 {
4           consume(M[i][i]);
5       }
6   }
7
8   sub main()
9   {
10      df Array3D, Matrix;
11
12      matrix(Array3D[0]);
13      matrix(Matrix);
14  }
```

Рисунок 16 – Пример подпрограммы, работающей с двумерными массивами

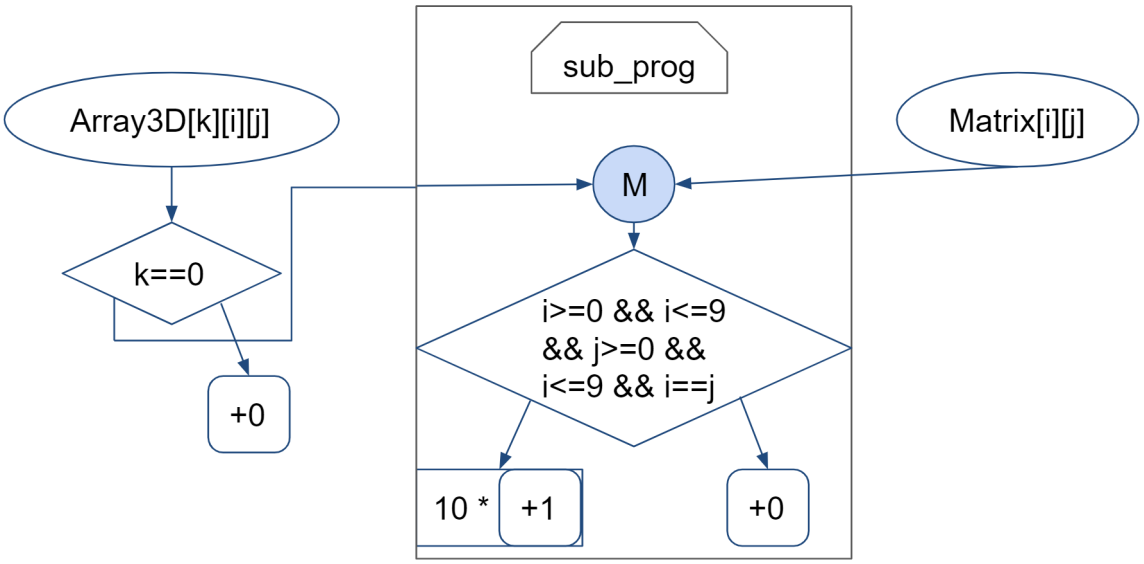

Рисунок 17 – Пример правильного сопоставления для двумерного массива, взятого из трёхмерного массива

2.3.6 Зависимость от локального фрагмента данных

У каждой рекомендации есть своя область применения, и в случае выставления количества потреблений фрагмента данных, этой областью является оператор, в котором происходит инициализация фрагмента данных.

Из-за необходимости рассчитывать ветвь дерева во время инициализации фрагмента данных возникает проблема, в случае когда для вычисления жизненного цикла фрагмента данных необходимо совершить косвенное потребление локального фрагмента данных.

Следовательно, для программы на рисунке 18, дерево потреблений, изображённое на рисунке 19, для фрагмента данных K, является некорректным, так как обход дерева будет совершаться во время инициализации фрагмента данных K, когда фрагмента данных B ещё не существует. Попытка запросить фрагмент данных B приведёт к ошибке во время исполнения программы.

```
1   sub sub_prog(name A)
2   {
3       df B;
4       consume(A);
5
6       if (B > 0) {
7           consume(A);
8       }
9   }
10
11  sub main()
12  {
13      df K;
14      sub_prog(K);
15  }
```

Рисунок 18 – Пример фрагментированной программы, содержащей зависимость от значения локального фрагмента данных

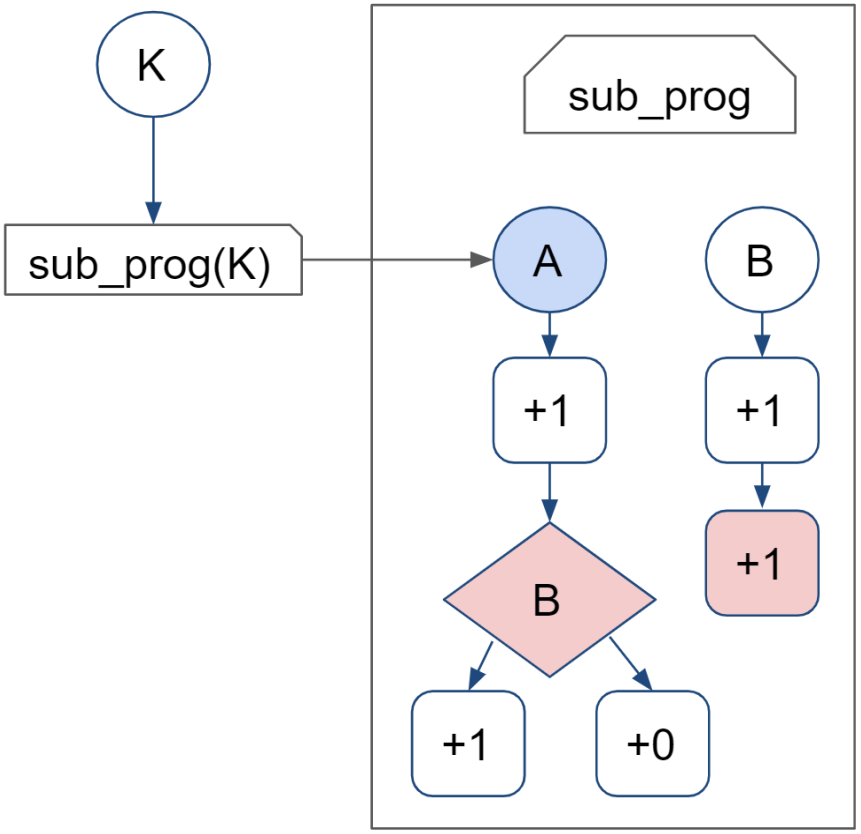

Рисунок 19 – Неправильное дерево потреблений, позволяющая фрагментам с большей областью видимости использовать локальные фрагменты данных

Есть два пути решения этой проблемы. Для первого варианта, нужно рассмотреть, как именно инициализируется фрагмент B. В случае, когда

фрагмент данных инициализируется константным выражением, известным на этапе компиляции, как на рисунке 20, это выражение можно подставить в ветвление вместо запроса значения фрагмента B, как в случае, изображённом на рисунке 21.

```
1   sub sub_prog(name A)
2   {
3       df B;
4       B = 1;
5       consume(A);
6
7       if (B > 0) {
8           consume(A);
9       }
10  }
11
12  sub main()
13  {
14      df K;
15      sub_prog(K);
16  }
```

Рисунок 20 – Пример инициализации локального фрагмента константным выражением

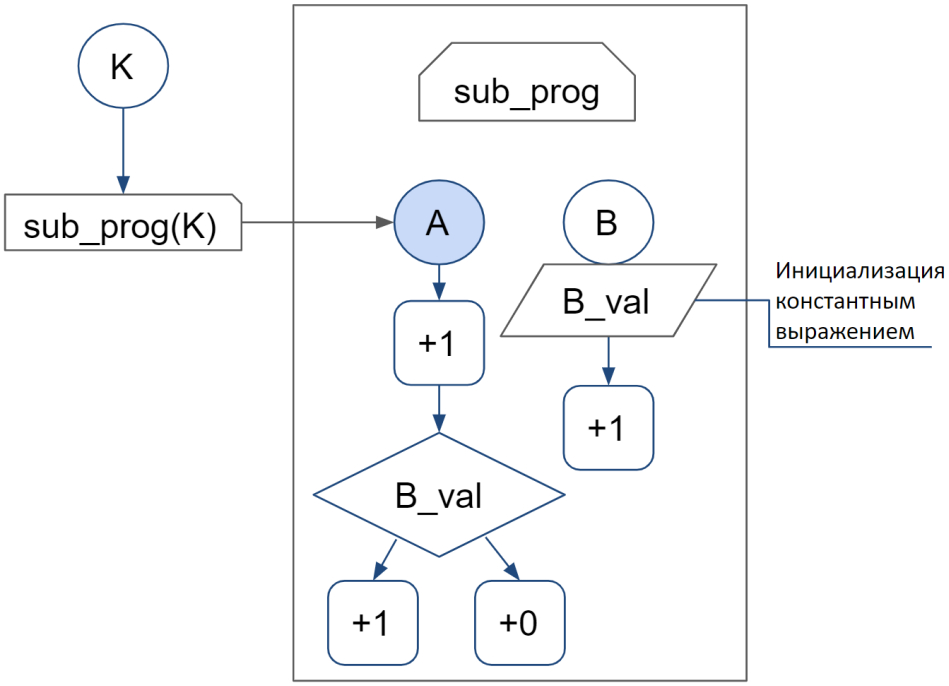

Рисунок 21 – Пример дерева потреблений для инициализации фрагмента данных константным выражением

Если B инициализируется значением другого фрагмента данных, как на рисунке 22, фрагментом данных L, с областью видимости равной области видимости фрагмента K, переданным в подпрограмму как фрагмент данных C, тогда нужно транзитивно запросить значение фрагмента L, минуя обращение к фрагменту B. Несмотря на то, что в дереве потреблений фрагмента A на рисунке 23 стоит фрагмент данных C, во время сопоставления деревьев, на его место будет подставлен фрагмент данных L, переданный как C.

```
1   sub sub_prog(name A, name C)
2   {
3       df B;
4       B = C;
5       consume(A);
6
7       if (B > 0) {
8           consume(A);
9       }
10  }
11
12  sub main()
13  {
14      df K, L;
15      sub_prog(K, L);
16  }
```

Рисунок 22 – Пример инициализации локального фрагмента данных значением другого фрагмента

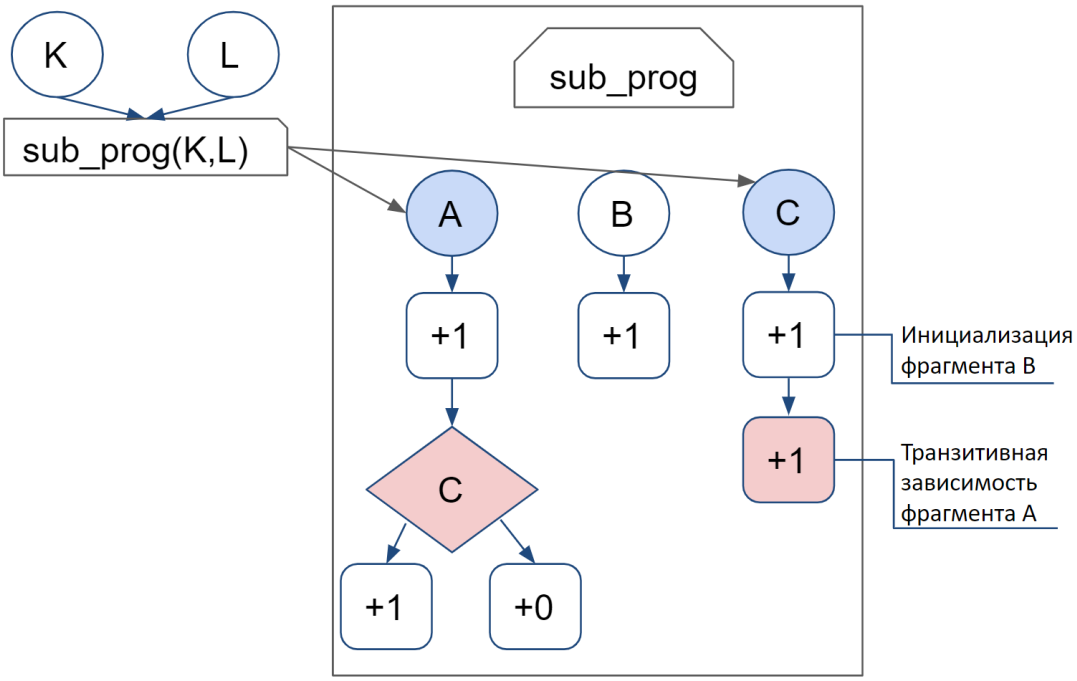

Рисунок 23 – Пример дерева потреблений для инициализации фрагмента данных значением другого фрагмента данных

Для определения того факта, что локальный фрагмент данных равен значению другого фрагмента или некоторому константному выражению, необходимо ввести рекомендацию или оператор присваивания. Таким образом, алгоритм будет способен отличить, в каких именно случаях происходит присваивание.

Второй вариант решения проблемы – добавление глобальных идентификаторов, доступных во время компиляции. Во многих языках программирования, недоступность переменной за пределами некоторой области видимости означает то, что эта переменная уже была удалена. Во фрагментированном программировании, локальный фрагмент данных B может оставаться в памяти после окончания вызова подпрограммы. Основная проблема, с которой связана невозможность ссылаться на B после вызова подпрограммы, заключается в том, что подпрограмма может быть вызвана больше одного раза, соответственно, для каждого вызова подпрограммы, будет создаваться новый локальный фрагмент данных B. Добавление глобальных идентификаторов решает проблему обращения к локальным фрагментам данных. Так как во время каждого вызова подпрограммы, создаётся новый

фрагмент вычислений, можно считать, что глобальным идентификатором каждого фрагмента данных можно составить из идентификаторов фрагментов вычислений, в контексте которых существует фрагмент данных и его локального идентификатора. Таким образом, глобальный идентификатор можно рассматривать как стек вызовов, за исключением того, что вместо вызова в фрагментированной программе будет порождение фрагмента вычислений.

Воспользовавшись глобальными идентификаторами, для программы на рисунке 24, можно построить дерево потреблений, изображённое на рисунке 25.

```
1   sub sub_prog(name A)
2   {
3       df B;
4       consume(A);
5
6       if (B > 0) {
7           consume(A);
8       }
9   }
10
11  sub main()
12  {
13      df K;
14      cf cf_call: sub_prog(K);
15  }
```

Рисунок 24 – Пример программы с явно объявленным идентификатором фрагментов вычислений

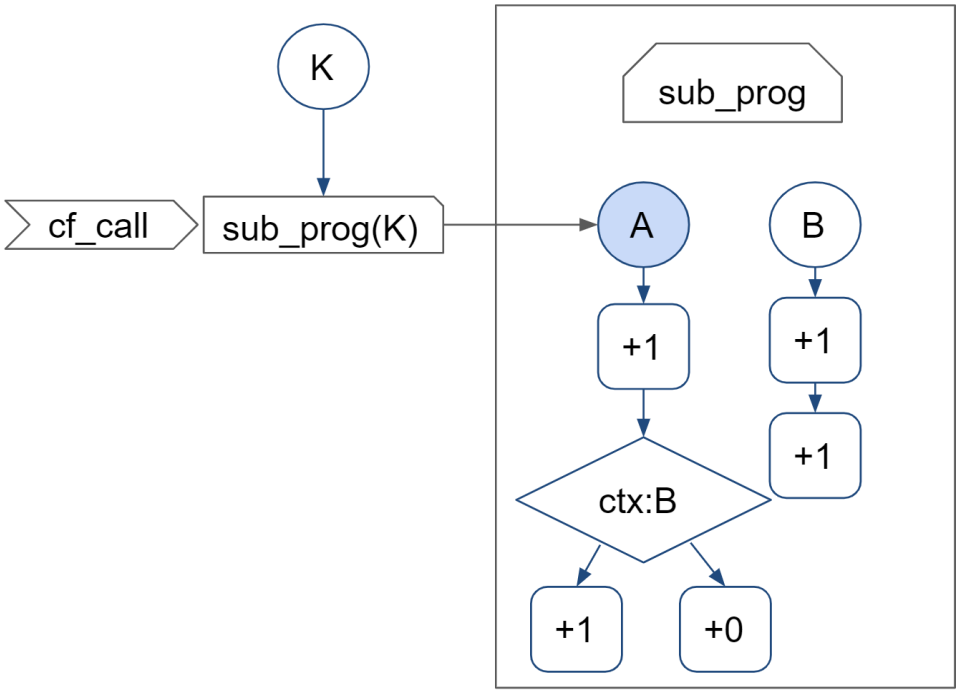

Рисунок 25 – Пример дерева потреблений, содержащего информацию о контексте

В случае K, контекстом является cf_call. Глобальный идентификатор для фрагмента B, созданного в рамках этого вызова, будет равен cf_call:B.

2.3.7 Вызов процедур

В отличии от вызова подпрограммы, фрагменты данных, переданные в атомарную процедуру, употребляются. Каждый из аргументов является выражением, выполняющимся в контексте оператора вызова процедуры, и если в процедуру передаются фрагменты данных, компилятор запросит каждый из использованных фрагментов. В случае, когда один и тот же фрагмент данных передаётся в несколько выражений, являющихся параметрами одной процедуры, система исполнения ограничивает число обращений одним разом, с целью оптимизации коммуникаций. Если компилятор не определит, что запрос происходит к одному и тому же фрагменту данных, количество ожидаемых запросов не совпадёт с реальным, у фрагмента данных останутся неиспользованные потребления, память не будет освобождена до окончания программы. Эта проблема особенно распространена для массивов, так индексом при передаче параметра в функцию может являться любое выражение, не обязательно вычислимое на этапе компиляции, а различные

выражения, выступающие в роли индекса элемента массива, могут давать один результат.

На рисунке 26 приведён пример дерева, не учитывающего оптимизации при обращении к одному и тому же элементу во время вызова процедуры.

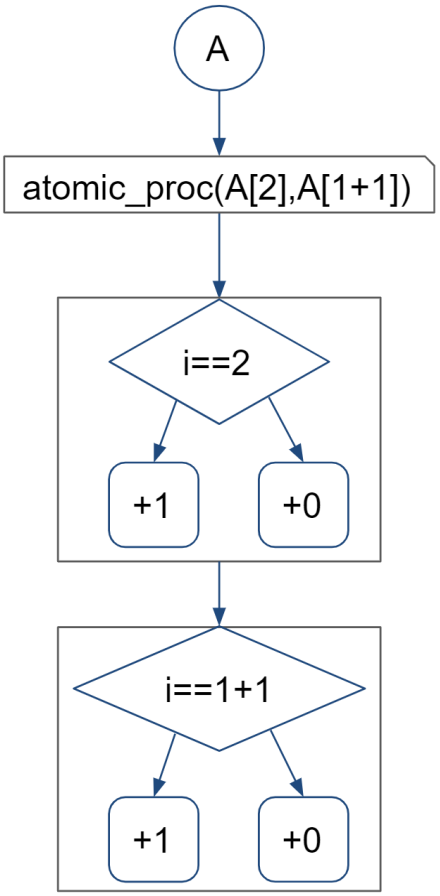

Рисунок 26 – Пример дерева потреблений, не учитывающего оптимизации запросов

На рисунке 27 изображён пример, как именно можно решить эту проблему дополнительным ветвлением.

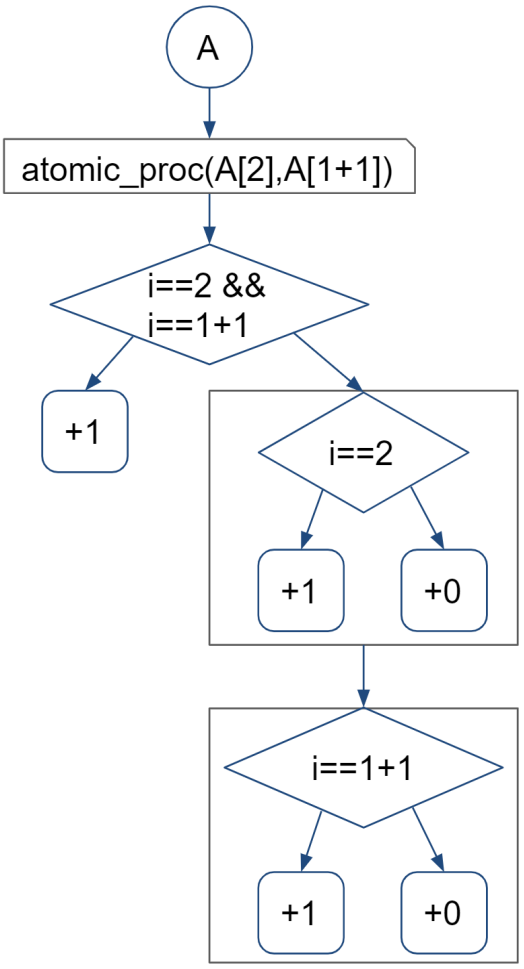

Рисунок 27 – Пример дерева потреблений, учитывающего оптимизации запросов

2.4 Характеристика предлагаемого решения

Данное решение позволит отслеживать жизненный цикл фрагментов данных на этапе компиляции для всевозможных существующих конструкций языка. Основным плюсом такого подхода является тот факт, что алгоритм будет отрабатывать на поддерживаемых конструкциях, но в случае расширения языка, новые конструкции не будут приводить к некорректному вычислению числа обращений, фрагменты данных, задействованные в таких конструкциях не будут собираться до окончания выполнения программы, чтобы избежать преждевременного удаления фрагмента данных. Также алгоритм позволяет сделать гибкую модульную реализацию, где для каждой конструкции языка будет существовать отдельный обработчик, с соответствующим интерфейсом.

Это позволит без особых проблем дополнять и изменять обработчики поддерживаемых конструкций.

3 Реализация и тестирование

3.1 Реализация

Для того, чтобы встроить алгоритм подсчёта числа потреблений фрагментов данных в систему LuNA, сначала нужно рассмотреть процесс компиляции в системе. Компиляция программы на языке LuNA производится несколькими модулями, написанными на языке программирования Python, разделяющими ответственность и поэтапно преобразующими код программы в программу на C++. Для более удобной работы с текстом программы, он переводится в абстрактное синтаксическое дерево, представленное в формате json, которое затем анализируется, модифицируется и транслируется в программу на C++. Исходя из вышеперечисленного, с целью сохранения модульности и разделения ответственности компилятора, для реализации алгоритма был выбран формат модуля и язык программирования Python. Для реализованного модуля было создано руководство программиста в приложении А.

Так как разработанный алгоритм не делает предположений, в каком случае будет осуществлено потребление фрагмента данных, а использует запросы, необходимо встроить модуль в определённое место конвейера компилятора, после того, как все необходимые для подготовки данных запросы будут расставлены. В системе LuNA функцию запроса выполняет рекомендация request, принимающая на вход идентификатор фрагмента данных.

Формула для вычисления числа потреблений вставляется в абстрактное синтаксическое дерево программы с помощью рекомендации req_count, принимающей идентификатор фрагмента данных и выражение, вычисление которого и будет количеством потреблений.

В разделе 2.3.6 было описано два способа разрешения зависимости от локального фрагмента данных. Так как на момент разработки алгоритма в системе LuNA отсутствовали как явная инициализация фрагмента данных, так

и глобальные идентификаторы, доступные на этапе компиляции, для первой версии модуля сборки мусора было решено реализовать поддержку только первого варианта, рассматривающего инициализацию фрагмента данных. Для отображения инициализации фрагмента данных был выбран оператор “=”.

Разработанный модуль состоит из следующих частей:

1. Парсер абстрактного синтаксического дерева языка системы LuNA.
2. Интерпретатор результата алгоритма.
3. Алгоритма, сюда включаются все классы и логика, необходимая для перевода абстрактного синтаксического дерева во множество деревьев потреблений.
4. Модуль сопоставления аргументов подпрограмм.

На рисунке 28 представлена схема работы разработанного модуля.

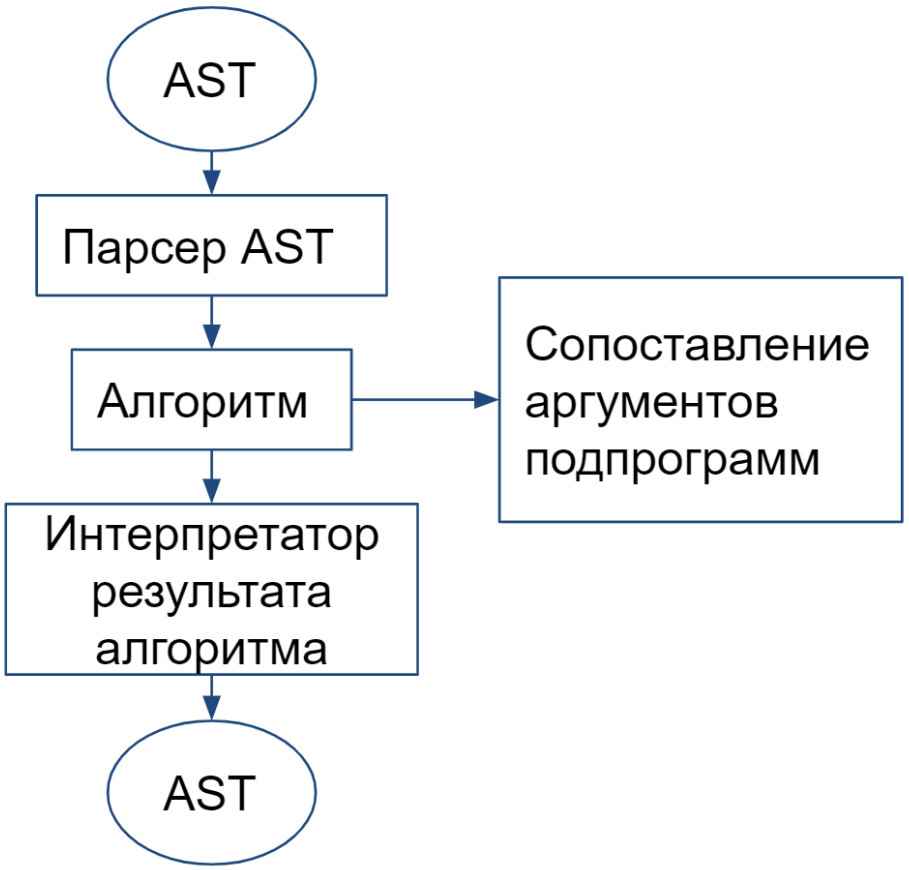

Рисунок 28 – Схема работы разработанного модуля для компилятора

Парсер и интерпретатор являются прослойкой между алгоритмом и системой, таким образом их выделение в отдельные сущности способствует переиспользованию алгоритма. Модуль сопоставления аргументов подпрограмм

был выделен в отдельную сущность для достижения большей тестируемости и является расширением возможностей алгоритма.

3.2 Тестирование

Проверить работоспособность модуля можно двумя способами, через сравнение абстрактного синтаксического дерева до и после работы алгоритма и сравнение потребления памяти во время выполнения программы.

3.2.1 Модификация абстрактного синтаксического дерева программы

Рассмотрим простую программу в системе LuNA. На рисунке 29, происходит объявление фрагмента данных, его инициализация и потребление.

Рассмотрим ключевые места абстрактного синтаксического дерева программы до его модификации модулем.

```
1   #!/usr/bin/luna
2
3   import c_init(int, name) as init;
4   import c_print(value) as print;
5
6   sub main()
7   {
8       df x;
9       // инициализация
10      init(7, x);
11      // потребление
12      print(x);
13  }
```

Рисунок 29 – Программа на языке LuNA с единственным потреблением фрагмента данных

Как видно из рисунка 30, до модификации абстрактного синтаксического дерева, список рекомендаций для оператора инициализации фрагмента данных “x” пустой (строка 47). Именно в этот список алгоритм добавит рекомендацию, подсчитывающую число потреблений фрагмента данных.

```
26          {
27              "args": [
28                  {
29                      "is_expr": true,
30                      "type": "iconst",
31                      "value": 7
32                  },
33                  {
34                      "begin": 99,
35                      "is_expr": true,
36                      "ref": [
37                          "x"
38                      ],
39                      "type": "id"
40                  }
41              ],
42              "begin": 91,
43              "code": "init",
44              "id": [
45                  "_l9"
46              ],
47              "rules": [],
48              "type": "exec"
49          },
```

Рисунок 30 – Оператор инициализации фрагмента x до модификации

На рисунке 31 представлен оператор, потребляющий фрагмент данных x. Вывод о потреблении можно сделать из присутствия рекомендации request для фрагмента данных “x”.

```
50          {
51              "args": [
52                  {
53                      "begin": 112,
54                      "is_expr": true,
55                      "ref": [
56                          "x"
57                      ],
58                      "type": "id"
59                  }
60              ],
61              "begin": 106,
62              "code": "print",
63              "id": [
64                  "_l11"
65              ],
66              "rules": [
67                  {
68                      "begin": 106,
69                      "items": [
70                          [
71                              "x"
72                          ]
73                      ],
74                      "property": "request",
75                      "ruletype": "enum"
76                  }
77              ],
78              "type": "exec"
79          }
```

Рисунок 31 – Оператор, потребляющий фрагмент данных “x”

Основываясь на информации из абстрактного синтаксического дерева о числе запросов, в список рекомендаций оператора инициализации будет добавлена рекомендация req_count для фрагмента x со значением 1, что и показано на рисунке 32.

```
47            "rules": [
48                {
49                    "id": [
50                        "x"
51                    ],
52                    "property": "req_count",
53                    "ruletype": "assign",
54                    "type": "rule",
55                    "val": {
56                        "is_expr": true,
57                        "type": "iconst",
58                        "value": 1
59                    }
60                }
61            ],
```

Рисунок 32 – Число потреблений в виде рекомендации req_count

Чтобы проверить работоспособность модуля в случае с косвенными потреблениями рассмотрим программу на рисунке 33.

```
1   #!/usr/bin/luna
2
3   import c_init(int, name) as init;
4   import c_print(value) as print;
5
6   sub main()
7   {
8       df x, y;
9
10      init(7, x);
11      init(10, y);
12      if (y > 5)
13      {
14          print(x);
15      }
16      print(x);
17  }
```

Рисунок 33 – Программа на языке LuNA с косвенным потреблением фрагмента “y”

Для фрагмента данных y будет создана рекомендация на рисунке 34, включающая в себя одно прямое потребление для вычисления оператора if и одно косвенное, для вычисления числа потреблений фрагмента “x”.

```
141                 {
142                     "id": [
143                         "y"
144                     ],
145                     "property": "req_count",
146                     "ruletype": "assign",
147                     "type": "rule",
148                     "val": {
149                         "is_expr": true,
150                         "op_pos": 164,
151                         "operands": [
152                             {
153                                 "is_expr": true,
154                                 "type": "iconst",
155                                 "value": 1
156                             },
157                             {
158                                 "is_expr": true,
159                                 "type": "iconst",
160                                 "value": 1
161                             }
162                         ],
163                         "type": "+"
164                     }
165                 }
```

Рисунок 34 – Число потреблений “y”, равное 1 + 1

Для фрагмента данных “x” будут сгенерированы две рекомендации. На рисунке 35 представлен запрос, совершающая потребление фрагмента y, необходимое для вычисления выражения на рисунке 36, строка 82.

```
49                 {
50                     "begin": 93,
51                     "items": [
52                         [
53                             "y"
54                         ]
55                     ],
56                     "property": "request",
57                     "ruletype": "enum",
58                     "type": "rule"
59                 },
```

Рисунок 35 – Косвенное потребление фрагмента “y”, необходимое для вычисления

Второй рекомендацией будет req_count, вычисляющий число потреблений фрагмента данных x. Объект “val” из абстрактного синтаксического дерева является выражением, транслируемым в формулу числа потреблений “x”.

```
60          {
61              "id": [
62                  "x"
63              ],
64              "property": "req_count",
65              "ruletype": "assign",
66              "type": "rule",
67              "val": {
68                  "is_expr": true,
69                  "op_pos": 93,
70                  "operands": [
71                      {
72                          "is_expr": true,
73                          "operands": [
74                              {
75                                  "is_expr": true,
76                                  "op_pos": 93,
77                                  "operands": [
78                                      {
79                                          "begin": 93,
80                                          "is_expr": true,
81                                          "ref": [
82                                              "y"
83                                          ],
84                                          "type": "id"
85                                      },
86                                      {
87                                          "is_expr": true,
88                                          "type": "iconst",
89                                          "value": 5
90                                      }
91                                  ],
92                                  "type": ">"
93                              },
94                              {
95                                  "is_expr": true,
96                                  "type": "iconst",
97                                  "value": 1
98                              },
99                              {
100                                 "is_expr": true,
101                                 "type": "iconst",
102                                 "value": 0
103                             }
104                         ],
105                         "type": "?:"
106                     },
107                     {
108                         "is_expr": true,
109                         "type": "iconst",
110                         "value": 1
111                     }
112                 ],
113                 "type": "+"
114             }
115         }
```


Рисунок 36 – Число потреблений “x”, транслируется в (y > 5 ? 1 : 0) + 1

3.2.2 Сравнение потребления памяти во время работы программы

Помимо проверки модификаций абстрактного синтаксического дерева программы на соответствие ожиданиям, было проведено сравнение потребления памяти и времени работы программы. В качестве экспериментальной программы, была выбрана игра “Жизнь” Конвея.

Игра “Жизнь” является представителем своего класса задач, клеточный итеративный процесс на сетке, где из-за особенностей фрагментированного программирования каждую итерацию будет происходить выделение больших объёмов памяти, необходимых для хранения текущего состояния поля, а текущая итерация зависит от предыдущей в более, чем одном месте.

Для эксперимента был выбран размер поля 100x100, 10 итераций, достаточные параметры для того, чтобы задача выполнялась более одной минуты.

На рисунке 37 представлен график, позволяющий сравнить разные варианты выполнения программы. Ось Y отображает количество потребляемой памяти в килобайтах, каждые 0,1 секунду, отложенную по оси X.

В качестве пользовательских рекомендаций, было решено придерживаться следующих правил: граница решётки потребляется 3 раза, а центр решётки потребляется 9 раз, из них 8 раз соседними ячейками и 1 раз для вычисления следующего значения. Этот набор правил не является достаточным для полной сборки мусора, так как, например, угловые ячейки, потребляются лишь 1 раз, следовательно они, не будут удалены в течении всего времени работы программы.

Для сбора информации о потребляемой памяти была использована встроенная утилита top, позволяющая отображать системную информацию в реальном времени [23]. Так как утилита следит за потреблённой процессом памятью с точки зрения операционной системы, во время освобождения памяти с помощью free или delete, процесс не отдаёт память системе, а помечает как свободную для дальнейшего переиспользования. Из этого можно сделать вывод о том, что в результате измерений, мы получим неубывающий график.

Из графика на рисунке 37 можно увидеть, что темп потребления памяти с модулем и пользовательскими рекомендациями растёт быстрее, чем у варианта без рекомендаций. Можно решить, что варианты с рекомендациями и модулем работают хуже, чем вариант без рекомендаций, но это не так. Из графика также видно, что вариант без рекомендаций работает значительно медленнее. Ускорение работы программы и более быстрый темп роста потребляемой памяти обеспечивается переиспользованием памяти, выделенной для хранения состояния поля. Это обеспечивает бóльшую локальность по памяти, что и даёт прирост в производительности, как видно из таблицы 1.

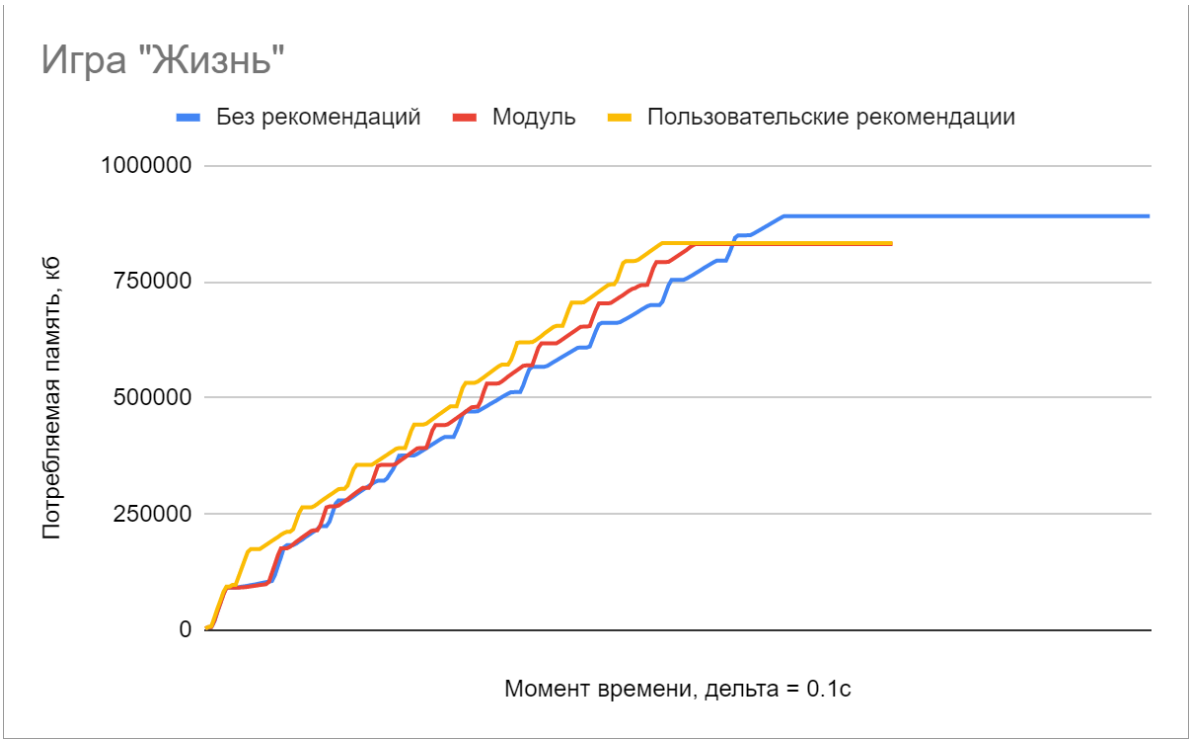

Рисунок 37 – График потребления памяти относительно времени работы программы для игры “Жизнь”

Таблица 1 – Сравнение времени работы разных вариантов программы, сек

Без рекомендаций	Пользовательские рекомендации	Модуль для компилятора
81,91	63,97	63,30

Рассмотрим рисунок 38, это тот же график, как и на рисунке 37, но диапазон потребляемой памяти сужен, чтобы более подробно рассмотреть различие между вариантами программы. В таблице 2 приведены пиковые

значения каждого из графиков. Оперируя данными из таблицы, получаем, что разница между программой без рекомендаций и с модулем составляет 59972 килобайт. Это огромный разрыв, вызванный копированием с одного узла на другой и хранением данных, без их последующего удаления.

Между пользовательскими рекомендациями и модулем разрыв в разы меньше: 2392 килобайта, что ожидаемо, так как пользовательские рекомендации удаляют большинство объектов, но не все.

Таблица 2 – Сравнение пикового значения потребляемой памяти для разных вариантов программы, кб

Без рекомендаций	Пользовательские рекомендации	Модуль для компилятора
891284	833504	831312

Рисунок 38 – График потребления памяти в диапазоне от 750 мб дл 900 мб

На рисунке 39 представлены результаты запуска трёх вариантов перемножения матриц размером 400 на 400 с разрезанных на матрицы размером 10 на 10. Как правило задания пользовательских рекомендаций было решено освобождать память только из под матриц, нетронутым в таком случае остаются два временных массива, используемых как агрегаторы. Именно за счёт

освобождения памяти используемой этим массивами, модуль для компилятора выигрывает с точки зрения потребления памяти. Из графика также можно увидеть, что не всегда своевременное освобождение памяти даёт прирост в производительности. Во-первых, это связано с тем, что большая часть данных тестируемой задачи, а именно две перемножаемые матрицы, выделяются в начале работы программы и не могут быть собраны до её окончания. Во-вторых, переиспользуемой памяти временных массивов недостаточно для эффективной локальности в памяти, нужно, чтобы задача выполнялась большее время, тогда ускорение, достигнутое за счёт высокой локальности в памяти, перевесит издержки освобождения памяти.

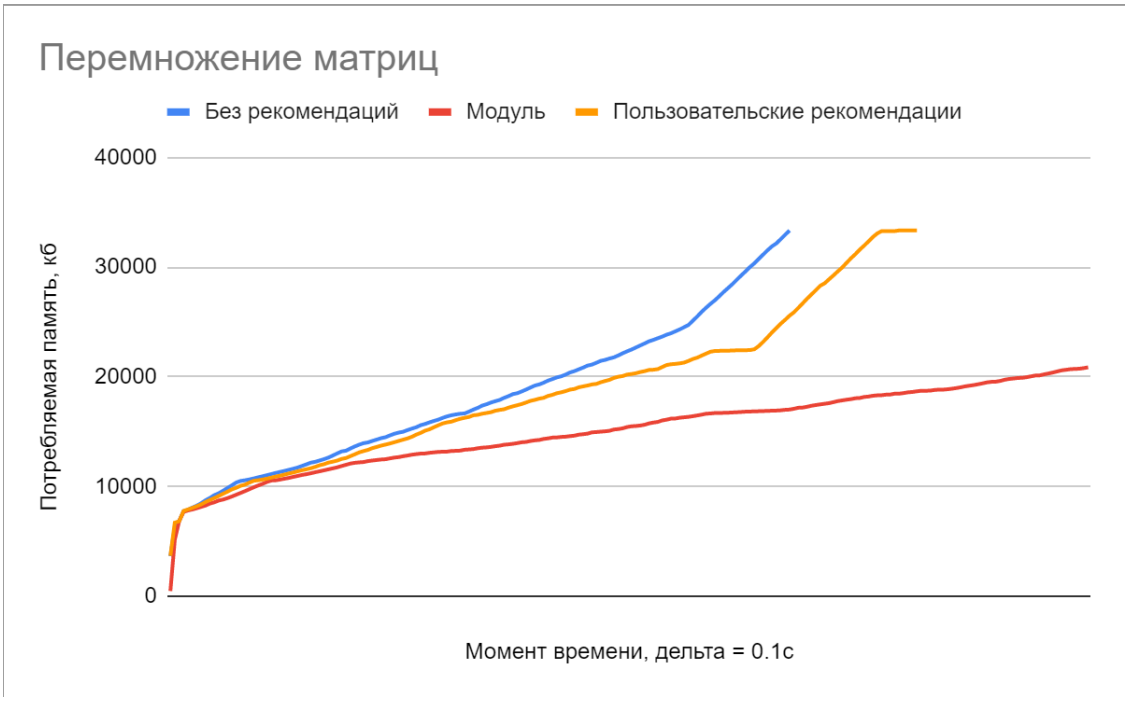

Рисунок 39 – График потребления памяти относительно времени работы программы для перемножения матриц

Вариант с модулем работал на 3,47 секунды дольше, чем вариант без рекомендаций и на 2 секунды дольше, чем вариант с пользовательскими рекомендациями. Результаты работы модуля ожидаемы, так как перемножение матриц не является репрезентативной задачей для демонстрации работы модуля сборки мусора, но даже на ней видно, что алгоритм справляется с освобождением неиспользуемой памяти.

Результаты тестирования подтвердили работоспособность алгоритма сборки мусора для задач численного моделирования и итерационных процессов. Разработанный алгоритм способен приблизить автоматическую сборку мусора к ручной для своего класса задач.

ЗАКЛЮЧЕНИЕ

Был разработан и реализован алгоритм автоматической сборки мусора, производящий анализ зависимостей, определение числа использований фрагментов данных и последующую модификацию программы для своевременного освобождения уже неиспользуемой памяти. Реализованный алгоритм был интегрирован в систему фрагментированного программирования LuNA в качестве модуля для компилятора, позволившего инкапсулировать сборку мусора. Алгоритм и модуль компилятора были задокументированы с целью их последующего переиспользования в случае изменения компилятора.

Тестирование продемонстрировало, что разработанный алгоритм справляется со сборкой мусора для задач численного моделирования и итерационных процессов на сетках и сопоставим с ручным управлением памятью.

Данная работы была опубликована на 59-й Международной научной студенческой конференции, г. Новосибирск, 2021 г. [3].

Защищаемые положения:

1. разработан алгоритм автоматического анализа зависимостей, определения числа использований фрагментов данных и последующей модификации программы для своевременного освобождения уже неиспользуемой памяти;
2. разработанный алгоритм был реализован в контексте системы фрагментированного программирования LuNA;
3. алгоритм был интегрирован в систему в качестве модуля для компилятора;
4. проведено экспериментальное исследование, которое показало работоспособность разработанного алгоритма.

В дальнейшем планируется продолжить работу над сборкой мусора в системе фрагментированного программирования LuNA. Возможными направлениями для развития являются:

1. расширение множества поддерживаемых операторов и конструкций языка;
2. реализация глобальных идентификаторов фрагментов данных, доступных на этапе компиляции;
3. разработка динамического алгоритма в качестве альтернативного механизма сборки мусора.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Плешков Андрей Владимирович _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Дараган, Е. И. Метод выявления информационных связей в программном обеспечении / Е. И. Дараган. — Текст : непосредственный // Молодой ученый. — 2012. — № 12 (47). — С. 125–130.
2. Перепелкин, В. А. Проблема распределения ресурсов мультикомпьютера в технологии фрагментированного программирования // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск). – М.: Изд-во МГУ, 2012. – С. 398 – 401.
3. Плешков А. В. Разработка алгоритмов распределенной сборки мусора в системе фрагментированного программирования LuNA / Плешков А. В. // Информационные технологии : Материалы 59-й Междунар. науч. студ. конф. 12–23 апреля 2021 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2021. — С. 118.
4. Abdullahi S. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection / Saleh E. Abdullahi, Graem A. Ringwood // ACM Computing Surveys – Queen Mary and Westfield College, University of London, 1998. — Vol. 30. – P. 330–373
5. Bevan D. Distributed garbage collection using reference counting. / Bevan D. // PARLE Parallel Architectures and Languages Europe. PARLE 1987. Lecture Notes in Computer Science, — Vol 259. – P. 176–187
6. Birrel A. Distributed garbage collection for network objects. / Birrel A., Evers D., Nelson G., Owicki S., Wobber E. // Technical report SRC 116, Digital — Systems Research Center, 1993.
7. Corporaal H., Veldman T., van de Goor A. J. An efficient, reference weight-based garbage collection method for distributed systems //Proceedings. PARBASE-90: International Conference on Databases, Parallel Architectures, and Their Applications. – IEEE, 1990. – P. 463-465.

8. Dickman P. Incremental, distributed orphan detection and actor garbage collection using graph partitioning and euler cycles. // Babaoğlu Ö., Marzullo K. (eds) Distributed Algorithms. WDAG 1996. Lecture Notes in Computer Science, — Vol 1151. – P. 141–158
9. Dickman P. Optimising weighted reference counts for scalable fault-tolerant distributed object-support systems //Submitted for HICSS. – 1992. – Vol. 26.
10. Goff G., Kennedy K., Tseng C. W. Practical dependence testing //ACM SIGPLAN Notices. – 1991. – Vol. 26. – №. 6. – P. 15-29.
11. Hughes J. A distributed garbage collection algorithm //Conference on Functional Programming Languages and Computer Architecture. – Springer, Berlin, Heidelberg, 1985. – P. 256-272.
12. D. Kuck, The Structure of Computers and Computations // John Wiley and Sons, New York, NY, 1978. – Volume 1
13. Libby J. et al. A survey of data dependence analysis techniques for automated parallelization. – 2007.
14. Malyshkin, V. E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V. E. Malyshkin, V. A. Perepelkin // Proceedings of the 11th Conference on Parallel Computing Technologis. – 2011. – Vol. 6873. – P. 53–61.
15. McCarthy J. Recursive functions of symbolic expressions and their computation by machine, part I //Communications of the ACM. – 1960. – Vol. 3. – №. 4. – P. 184-195.
16. Piquer J. M. Indirect reference counting: A distributed garbage collection algorithm //PARLE'91 Parallel Architectures and Languages Europe. – Springer, Berlin, Heidelberg, 1991. – P. 150-165.
17. Plainfossé D., Shapiro M. A survey of distributed garbage collection techniques //International Workshop on Memory Management. – Springer, Berlin, Heidelberg, 1995. – P. 211-249.
18. Shapiro M., Dickman P., Plainfossé D. SSP chains: Robust, distributed references supporting acyclic garbage collection : дис. – inria, 1992.

19. Wiseman S. R. Garbage collection in distributed systems : дис. – Newcastle University, 1988.
20. Wolfe M. J. Optimizing supercompilers for supercomputers : дис. – University of Illinois at Urbana-Champaign, 1982.
21. The Online Tech Dictionary for Students, Educators and IT Professionals. Expression definition — [Электронный ресурс] / – Режим доступа: https://www.webopedia.com/definitions/expression/ (дата обращения: 12.05.2021).
22. The Online Tech Dictionary for Students, Educators and IT Professionals. Statement definition — [Электронный ресурс] / – Режим доступа: https://www.webopedia.com/definitions/statement/ (дата обращения: 12.05.2021).
23. Linux manual page. top(1) — Linux manual page — [Электронный ресурс] / – Режим доступа: https://man7.org/linux/man-pages/man1/top.1.html (дата обращения: 20.05.2021).

ПРИЛОЖЕНИЕ А

МОДУЛЬ АВТОМАТИЧЕСКОЙ СБОРКИ МУСОРА ДЛЯ КОМПИЛЯТОРА В СИСТЕМЕ «LuNA»

РУКОВОДСТВО ПРОГРАММИСТА

Листов 9

Новосибирск 2021

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данном документе приведено руководство программиста для модуля автоматической сборки мусора во время компиляции программы в системе LuNA. Исходным языком программы является Python. Средство разработки – редактор исходного кода PyCharm от компании JetBrains.

Основной функцией программы является определение числа использований фрагментов данных с целью их последующего своевременного удаления.

Оформление программного документа «Руководство оператора» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

1 Назначение и условия программы

1.1 Назначение программы

Модуль сборки мусора входит в состав компилятора системы LuNA. Пользователь не осуществляет явной работы модулем, а пользуется компилятором системы как обычно.

1.2 Функции, выполняемые программой

Программа позволяет расширить компиляцию языка LuNA с помощью анализа кода, отслеживания числа использований фрагментов данных и автоматической генерации кода рекомендаций для освобождения памяти, тем самым упрощая написание программ.

1.3 Условия, необходимые для выполнения программы

Чтобы программа могла исполняться, необходимо наличие у программиста интерпретатора Python версии 3.7 и выше, а также пакета typing.

2 Характеристика программы

2.1 Описание основных характеристик программы

Модуль сборки мусора входит в состав компилятора системы LuNA и производит анализ кода компилируемой программы с целью подсчёта числа потреблений фрагментов данных в программе для своевременного освобождения памяти.

2.2 Описание основных особенностей программы

Разработанная версия модуля на момент написания руководства не поддерживает освобождение памяти для фрагментов данных, зависящих от значения локальных фрагментов данных, проинициализированных без использования оператора “=”.

3 Обращение к программе

3.1 Описание процедур вызова программы

Определение числа потреблений и модификация абстрактного синтаксического дерева программы выполняются с помощью вызова функции compile_time_req_count из файла cmp_time_ref_counter.

Для использования функции необходимо импортировать её из файла с исходным кодом программы.

Для чтения абстрактного синтаксического дерева программы используется класс AstParser, для его модификации класс AstModifier. Во время чтения дерева, вся необходимая для анализа потреблений фрагментов данных информация записывается в класс Scope. Сопоставлением аргументов подпрограмм осуществляет класс ExecMatcher.

4 Входные и выходные данные

4.1 Организация используемой входной информации

Входная информация – абстрактное синтаксическое дерево компилируемой программы на языке системы LuNA в формате json и список идентификаторов фрагментов данных, потребления которых не нужно отслеживать, так как они будут освобождены другими способами.

4.2 Организация используемой выходной информации

Выходной информацией программы является модифицированное абстрактное синтаксическое дерево программы на языке системы LuNA в формате json.

5 Сообщения

В случае, когда программа в системе LuNA компилируется с флагом --verbose, позволяющим компилятору выводить информацию в терминал, если во время анализа программы не получилось определить количество потреблений некоторых фрагментов данных, для всех таких фрагментов, будет показано сообщение “Unable to track usages and generate req_count for data fragment: <ИДЕНТИФИКАТОР ФРАГМЕНТА ДАННЫХ>”. Это сообщение не влияет на работу программы, однако для достижения большей эффективности по памяти, программисту предлагается либо обработать незатронутые фрагменты данных вручную, либо переписать программу без использования неподдерживаемых конструкций языка.

6 Лист регистрации изменений

Таблица 3 – Лист регистрации изменении руководства программиста

Лист регистрации изменений									
Номера листов (страниц)					**Всего листов (страниц) в докум.**	**№ докумен -та**	**Входящий № сопроводит. докум. и дата**	**Подп.**	**Дата**
Изм.	**изменен-ных**	**заменен-ных**	**новых**	**аннули-рованных**					