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

Царева Василия Дмитриевича

Тема работы:

СТАТИЧЕСКИЙ АНАЛИЗ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ НА БАЗЕ ГРАФА ЗАВИСИМОСТЕЙ ПО ДАННЫМ

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В.Э. /……
(ФИО) / (подпись)
«31» мая 2024 г.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./……...
(ФИО) / (подпись)
«31» мая 2024 г.

Соруководитель ВКР
к.т.н.,
ст. преп. каф. ПВ ФИТ
Перепёлкин В. А./……..
(ФИО) / (подпись)
«31» мая 2024 г.

Новосибирск, 2024

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)
«03» ноября 2023 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Цареву Василию Дмитриевичу, группы 20202
(фамилия, имя, отчество, номер группы)

Тема: Статический анализ фрагментированных программ на базе графа зависимостей по данным
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 03 ноября 2023 № 0413

Срок сдачи студентом готовой работы 20 мая 2024 г.

Исходные данные (или цель работы): разработать алгоритмы статического анализа LuNA-программ для наиболее типовых классов семантических ошибок, опираясь на графовые представления программ, и реализовать данные алгоритмы в программном средстве.

Структурные части работы: обзор и анализ существующих методов и инструментов статического анализа, формулирование требований к модулю статического анализа, разработка и реализация модуля статического анализа, тестирование и интеграция модуля в комплекс автоматизированной отладки LuNA-программ.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./……...
(ФИО) / (подпись)
«03» ноября 2023 г.

Задание принял к исполнению
Царев В. Д. /……..
(ФИО студента) / (подпись)
«03» ноября 2023 г.

Соруководитель ВКР
к.т.н., ст. преп. каф. ПВ ФИТ
Перепёлкин В. А./……..
(ФИО) / (подпись)
«03» ноября 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**
Глава 1. Статический анализ и промежуточные представления программ **7**
1.1 Виды ошибок в программах 7
1.2 Динамический анализ 8
1.3 Статический анализ 8
1.4 Виды промежуточных представлений 9
1.5 Подходы к отладке методом статического анализа 12
1.6 Рассмотренные средства статического анализа 13
Глава 2. Система LuNA и граф зависимостей по данным для LuNA-программ **16**
2.1 Система LuNA 16
2.2 База типов ошибок в LuNA-программах 19
2.3 Граф зависимостей по данным LuNA-программы 19
2.4 Построение и применение графа зависимостей по данным LuNA-программы 21
2.5 Требования к создаваемому анализатору 26
Глава 3. Модуль статического анализа DeGSA **27**
3.1 Структура графа и вспомогательные объекты 27
3.2 Реализация алгоритмов построения графа 40
3.3 Реализация алгоритмов поиска ошибок 43
3.4 Выходные файлы 47
Глава 4. Тестирование и интеграция анализатора в комплекс автоматизированной отладки ADAPT **49**
4.1 Комплекс ADAPT и интеграция в него модуля DeGSA 49
4.2 Унифицированный формат вывода отчётов об ошибках 50
4.3 Утилита adapt_output_generator 50
4.4 Тестирование модуля DeGSA 51
ЗАКЛЮЧЕНИЕ **56**
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ **57**
ПРИЛОЖЕНИЕ А **61**
ПРИЛОЖЕНИЕ Б **63**
ПРИЛОЖЕНИЕ В **66**

ВВЕДЕНИЕ

Современное программирование требует большого количества усилий на создание необходимых приложений, а потребности человечества накладывают множество требований на характеристики программ, в частности, на скорость их работы. Для ускорения выполнения программы применяются различные концепции, одной из которых является параллельное программирование [1], направленное на написание программ, использующих возможности многопроцессорных машин и многомашинных комплексов. Такие программы создавать гораздо тяжелее, поскольку при их написании требуется учитывать особенности межпроцессорных коммуникаций, и для сокращения трудозатрат на разработку таких программ внутри ИВМиМГ СО РАН создаётся система автоматизированного конструирования параллельных программ LuNA (Language for Numerical Algorithms) [2, 3] и одноименный язык для неё, призванные облегчить работу программиста за счёт эффективного автоматизированного конструирования параллельных программ.

Несмотря на то, что применение системы LuNA проще применения других языков программирования, оно остаётся достаточно нетривиальным: разработчик всё ещё может допускать ошибки, которые бывают трудны в исправлении и на поиск которых уходит много человеко-часов. Примерами таких ошибок могут быть попытка использования неинициализированного фрагмента данных или повторная инициализация фрагмента данных. Они приводят к внештатному завершению программы, и к тому же являются недетерминированными, поскольку проявляются не при каждом запуске программы (чем больше в исходном коде ветвлений, тем тяжелее подобной ошибке проявиться).

Важным инструментом для уменьшения затрат на исправление ошибок в программах на различных языках программирования является статический анализ – анализ исходного кода программы (или промежуточных представлений, сгенерированных на его основе) без её запуска. Это позволяет существенно сократить трудозатраты на разработку программы за счёт

уменьшения времени диагностики и устранения ошибок. На текущий момент существует множество статических анализаторов, однако система LuNA в силу своей специфичности не способна работать с ними. Данная работа направлена на создание такого статического анализатора.

Цель

Разработать алгоритмы статического анализа LuNA-программ для наиболее типовых классов семантических ошибок, опираясь на графовые представления программ, и реализовать данные алгоритмы в программном средстве.

Задачи

- Принять участие в формировании базы типов ошибок в LuNA-программах.
- Провести обзор методов и средств статического анализа.
- Выбрать (и обосновать выбор) промежуточного представления LuNA-программ для проведения статического анализа с целью выявления семантических ошибок.
- Разработать и протестировать модуль статического анализа с использованием выбранного представления.
- Интегрировать разработанный модуль в комплекс автоматизированной отладки LuNA-программ ADAPT.

Разработанный метод статического анализа программ на языке LuNA заключается в построении графа зависимостей операторов из абстрактного синтаксического дерева и поиске по этому графу семантических ошибок, в том числе циклических зависимостей по данным, попыток использований неинициализированных фрагментов данных, повторных инициализаций фрагментов данных и пр.

Разработанное приложение используется для сокращения трудозатрат программистов на языке LuNA путём уменьшения количества требуемых усилий на отладку программ.

Работа содержит четыре главы. В первой главе представлены анализ существующих методов статического анализа и обзор системы LuNA, и сформирован список требований к создаваемому анализатору. Во второй главе обоснован выбор промежуточного представления LuNA-программы и описаны алгоритмы, позволяющие автоматизировать построение выбранного представления и поиск ошибок в LuNA-программе. Третья глава посвящена вопросам реализации спроектированного модуля статического анализа. Четвёртая глава описывает тестирование и интеграцию разработанного модуля в комплекс автоматизированной отладки LuNA-программ.

Глава 1. Статический анализ и промежуточные представления программ

1.1 Виды ошибок в программах

Ошибки, допускаемые программистом во время написания программы, можно разделить на два класса: синтаксические, и семантические.

Первый класс ошибок представляет собой некорректную с точки зрения синтаксиса языка ситуацию и гарантированно обнаруживается во время лексического и синтаксического этапов компиляции. Это означает, что создавать статический анализатор для поиска ошибок этого класса нецелесообразно – по крайней мере, в случае, если возвращаемая компилятором информация позволяет эффективно устранить ошибку.

Семантические ошибки же влияют на корректность результата работы программы, однако не препятствуют компиляции и запуску программы. Они часто являются более трудными в устранении чем синтаксические, поскольку требуют анализа работы программы и выяснению, при каких входных данных и где именно происходит нарушение работы. Примерами типов таких ошибок являются разыменование нулевых указателей, использование неинициализированных переменных, бесконечная рекурсия и др.

Учитывая разную природу семантических ошибок, для их поиска приходится использовать различные подходы – как ручные, так и автоматические. Например, для автоматического поиска циклических зависимостей по данным необходимо обнаружить циклы в графе зависимостей по данным, для обнаружения мёртвого кода нужно находить недостижимые вершины в графе потока управления, для проверки типов и значений переменных необходимо использовать методы анализа потока данных с помощью графа потока управления и т.д.

Существуют различные подходы к автоматизации отладки программ, среди которых выделяются динамический анализ и статический анализ.

1.2 Динамический анализ

Динамический анализ [4] – анализ программы во время её работы. Обладает следующими преимуществами:

- не выдаёт ложные срабатывания (любой доклад об ошибке это констатация факта её возникновения);
- не требует исходного кода анализируемой программы;

Недостатки динамического анализа:

- требует компиляции и запуска приложения;
- не может проанализировать весь код сразу (анализируется лишь путь выполнения программы, который был непосредственно пройден во время работы);
- страдает от “эффекта наблюдателя” (само средство отладки может непреднамеренно исказить работу анализируемой программы и испортить результаты анализа).

1.3 Статический анализ

Статический анализ [5, 6] – анализ исходного кода программы без запуска последней, в противоположность динамическому анализу. Он используется для поиска различных ошибок в коде программ, а также для подсчёта метрик [7] (примерами являются метрика Мак-Кейба, метрика Майерса, метрика Хансена и др.). Статический анализ обладает такими преимуществами, как:

- возможность применения анализа ко всему исходному коду одновременно;
- отсутствие необходимости компиляции и запуска приложения;
- отсутствие зависимости от используемого компилятора и среды выполнения программы;
- отсутствие влияния на анализируемую программу во время проведения анализа.

Минусами статического анализа являются порождение ложных срабатываний (доклад об ошибках, которых на самом деле нет) и

необходимость в наличии исходного кода программы или подходящего промежуточного представления.

Промежуточные представления [8] играют большую роль в работе анализатора и представляют собой некоторые математические модели, которые отражают структуру исходной программы. Существует множество различных представлений, которые обладают теми или иными преимуществами и удобствами для поиска ошибок различных видов.

1.4 Виды промежуточных представлений

1.4.1 Абстрактное синтаксическое дерево

Абстрактное синтаксическое дерево [9] (abstract syntax tree, AST) представляет собой ориентированный ациклический граф, чьими узлами являются операции, а дуги указывают участие в них результатов других операций. Листьями являются некоторые данные – константы или значения переменных.

Абстрактное синтаксическое дерево является упрощённым видом синтаксического дерева [10], которое, в свою очередь, является результатом синтаксического анализа программы. Несмотря на упрощения, абстрактное синтаксическое дерево содержит всю необходимую семантическую информацию (в предположении, что исходная программа является синтаксически корректной). Это делает его привлекательным для проведения анализа, поскольку это (теоретически) позволяет находить ошибки любого типа, которые в принципе возможно обнаруживать. Пример абстрактного синтаксического дерева представлен на рисунке 1.

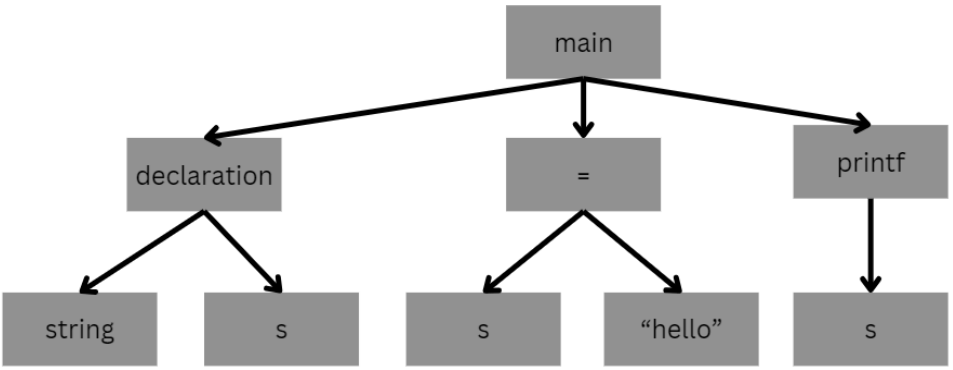

Рисунок 1 – Пример AST

Недостатком абстрактного синтаксического является громоздкость – относительно других представлений AST является большим и сложным объектом, и для анализа часто необходимо заново обходить его участки. Как следствие, возникает необходимость в представлении, которое будет более простым в обходе.

1.4.2 Трёхадресный код

Трёхадресный код [10] – последовательность команд, каждая из которых имеет одну операцию и не более трёх адресов. Строится путём обхода абстрактного синтаксического дерева и разделения сложных выражений на простые и с добавлением новых переменных. Команды выполняются последовательно, для обработки таких конструкций как if, for и while используются переходы по меткам.

Трёхадресный код используется во время компиляции программы для проведения оптимизаций. Простота команд и линейный порядок их выполнения позволяет удалять и заменять ненужные и неэффективные команды (удаление неиспользуемых переменных, упрощение выражений, и т.д.). Пример трёхадресного кода представлен на рисунке 2.

```
i = 1
one:
if i <= 10 goto two
goto next
two:
t1 = x + 2
x = t1
t2 = i + 1
i = t2
goto one
next:
...
```

Рисунок 2 – Пример трёхадресного кода

1.4.3 Граф потока управления

Граф потока управления [11-13] (control-flow graph, CFG) – ориентированный граф, чьими вершинами являются базовые блоки, а дуги указывают переходы управления между ними. Базовыми блоками называются последовательности трёхадресных команд, где последняя команда является переходом, а первая является меткой. Переходы могут быть условными и безусловными. Граф потока управления строится путём обхода трёхадресного кода программы. Пример CFG представлен на рисунке 3.

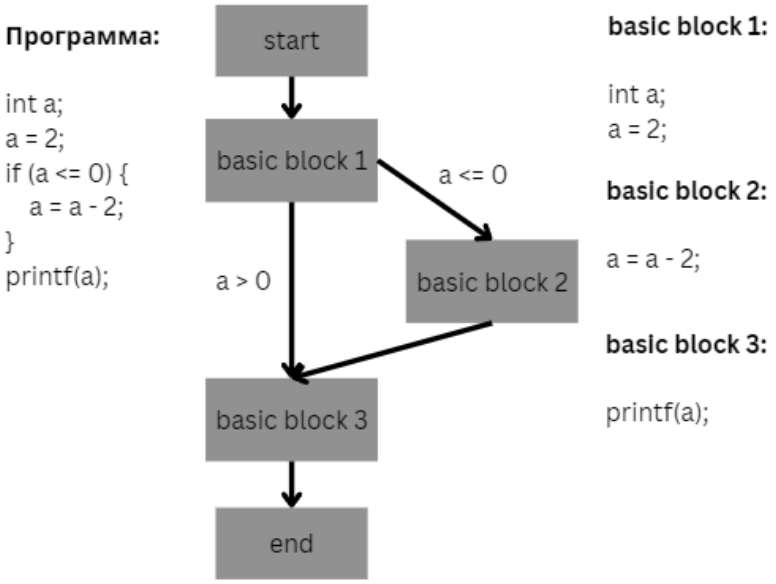

Рисунок 3 – Пример графа потока управления

1.4.4 Граф зависимостей по данным

Граф зависимостей по данным [14-17] (data-dependence graph, DDG) – ориентированный помеченный граф, чьими вершинами являются операции, а дугами указываются зависимости по данным между ними. Пример DDG представлен на рисунке 4.

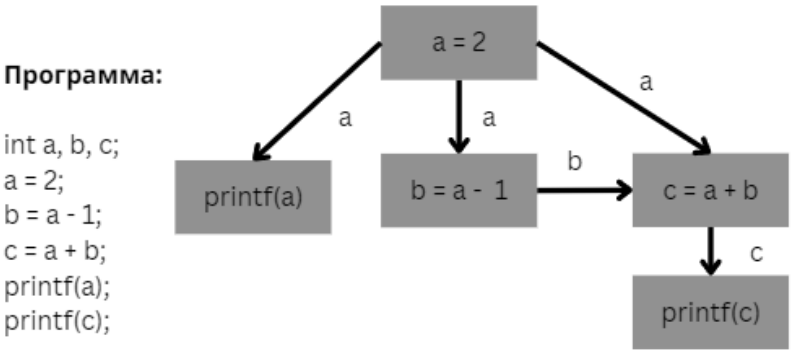

Рисунок 4 – Пример графа зависимостей по данным

1.5 Подходы к отладке методом статического анализа

В рамках одного и того же представления возможно использовать различные методы отладки, которые являются эффективными для поиска ошибок тех или иных видов.

1.5.1 Анализ AST

Анализ AST включает в себя обход графа в глубину и сбор фактов о программе. Такой анализ проводится компилятором во время проверки программы на синтаксическую корректность. Неудобство метода заключается в том, что для получения семантической информации часто приходится совершать несколько повторных обходов в глубину. Вместе с AST могут быть использованы различные стратегии – например, сравнение с шаблоном [18] (используется в [19]) .

1.5.2 Анализ потока управления

Анализ потока управления [20] использует CFG. Во время анализа проводится обход графа и анализируется, куда именно совершаются переходы согласно известным фактам о программе. В случае, если варианты переходов рассчитать нет возможности, некоторые переходы отсекаются (например, если известно, что последующий код точно безопасен и не требует анализа),

аппроксимируется количество итераций в цикле (ограничиваются сверху некоторым достаточным числом), и т. д. Соответственно, абсолютно точного ответа алгоритм может не предоставить, а также имеет зависимость от выбора интересующего множества переходов среди всех возможных. Анализ потока управления позволяет искать мёртвый код (с помощью поиска недостижимых вершин графа) или бесконечные циклы (существует цикл в графе, из которого нет возможности выйти).

1.5.3 Анализ потока данных

Анализ потока данных [21, 22] также применяет CFG, однако его целью является сбор фактов о данных в программе – например, возможные значения переменных, их типы, являются ли они константами, ссылаются ли указатели на один и тот же адрес в памяти и т. д. Это позволяет проводить оптимизации исходного кода (например, упрощение арифметических выражений) и докладывать об ошибках (например, указывать на несоответствие типов значений переменных в языках с динамической типизацией, или на использование неинициализированных данных).

Существует множество подходов к проведению анализа потока данных, среди которых можно выделить taint-анализ, T1-T2 анализ, интервальный анализ и структурный анализ.

1.6 Рассмотренные средства статического анализа

1.6.1 Cppcheck

Cppcheck [23] представляет собой минималистичный анализатор кода языков C и C++. Достаточно скачать и запустить его, после чего загрузить программу через контекстное меню. Пример вывода анализатора представлен на рисунке 5.

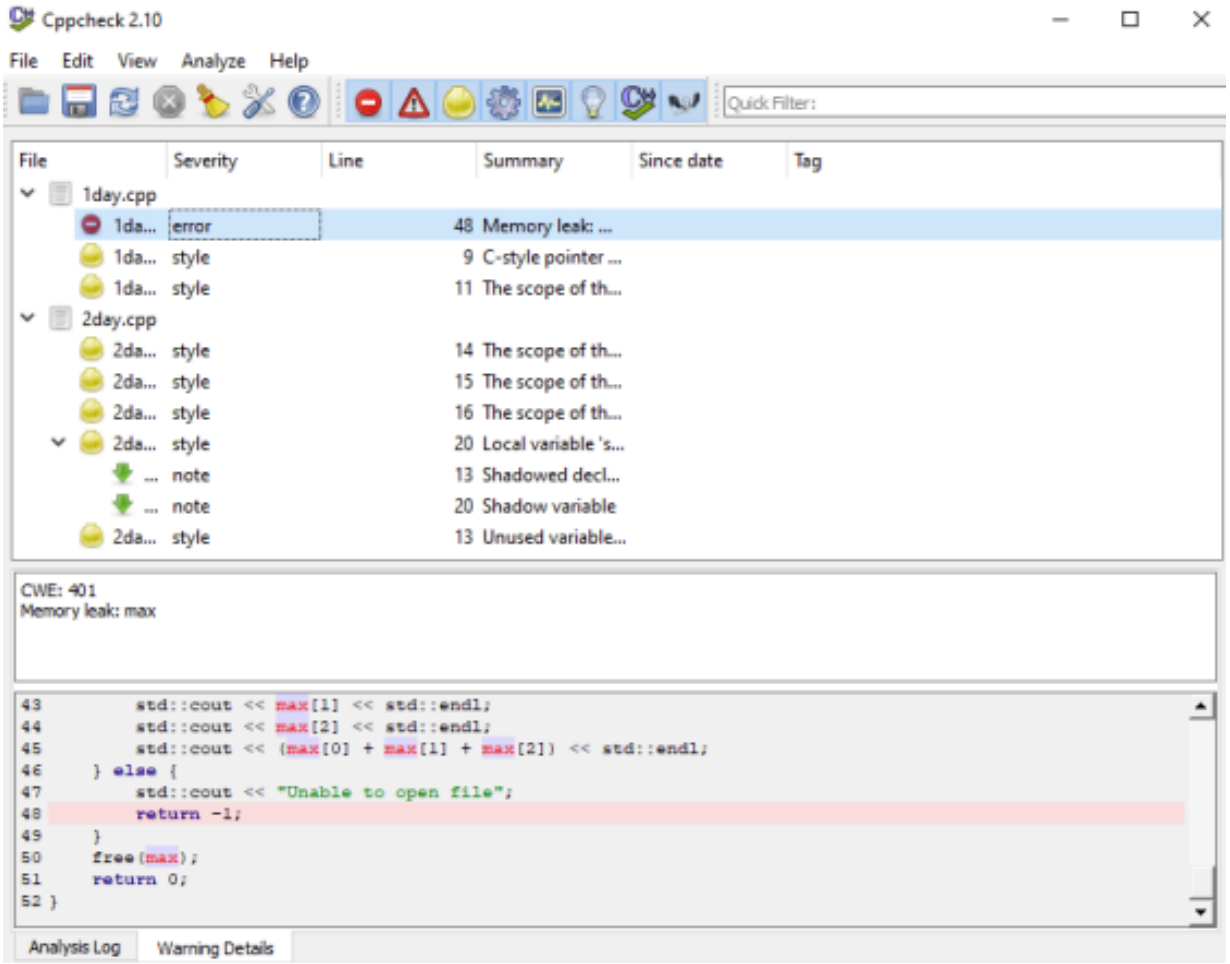

Рисунок 5 – Пример вывода Cppcheck

Cppcheck работает в рамках философии минимального количества ложных срабатываний. Например, во время анализа программы, представленной на рисунке 6, он не обнаружит недочёта (который заключается в утечке памяти). Это происходит из-за того, что Cppcheck предполагает, что оба оператора if выполнятся, и память будет освобождена.

```
void f(int x){
        char *a = 0;
        if (x == 10)
                a = malloc(10);
        if (x == 20)
                free(a);
}
```

Рисунок 6 — Входная программа для Cppcheck

1.6.2 Stan

Stan [19] – статический анализатор для функционального языка Haskell. Он использует HIE-файлы, которые функционально представляют собой AST.

Для поиска ошибок используется сравнение по шаблону [18] – наборы узлов, которые представляют собой некорректные ситуации, сравниваются с узлами AST входной программы. Для оптимизации работы анализатора используется шаблон Посетитель [24]. Схема работы анализатора проиллюстрирована на рисунке 7.

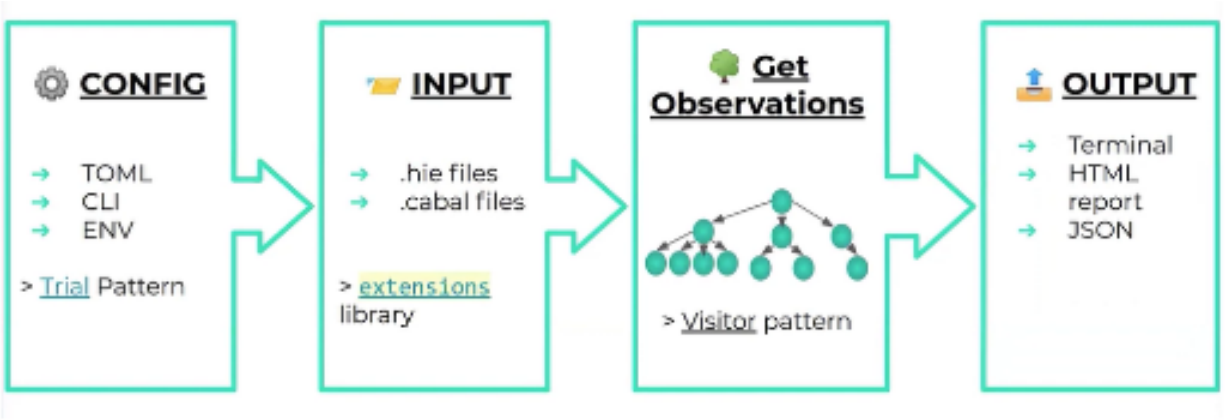

Рисунок 7 – Схема работы Stan

1.6.3 Weeder

Weeder [25] это статический анализатор для языка Haskell, направленный на поиск мёртвого кода. Использует граф зависимостей программы (указывает зависимости между объявлениями), который строится из HIE-файлов. После построения графа проводится обход, начиная с main. Недостижимые вершины графа считаются мёртвым кодом.

Глава 2. Система LuNA и граф зависимостей по данным для LuNA-программ

2.1 Система LuNA

Система LuNA [2, 3] (Language for Numerical Algorithms), разрабатываемая в лаборатории синтеза параллельных программ ИВМиМГ СО РАН, предназначена для автоматического создания эффективных программ.

Основные термины системы LuNA:

- Фрагмент данных (ФД) – единожды инициализируемая единица информации произвольной структуры. Может быть индексированным.
- Фрагмент кода (ФК) – пользовательская процедура. Может быть двух типов: атомарная или структурированная. Структурированный ФК представляет собой фрагментированный алгоритм, описанный на языке LuNA. Атомарный ФК является импортированной из файла C++ функцией, которая может иметь входные и выходные фрагменты данных.
- Фрагмент вычислений (ФВ) – единица работы, которая динамически распределяется по вычислительным узлам во время работы программы. Порождается во время вызова атомарного ФК, и для порождения требует удовлетворения всех входных информационных зависимостей.

Язык LuNA позволяет описывать фрагментированные алгоритмы [26], которые представляют собой рекурсивно-перечислимые множества триплетов <*in*, *mod*, *out*>, где *in* – множество входных ФД, *mod* – модуль, вычисляющий *out* по полученному *in*, *out* – множество выходных ФД. Описанный на языке LuNA фрагментированный алгоритм называется фрагментированной программой, и её запуск обрабатывается системой LuNA в целях эффективного распределения запускаемых ФВ по вычислительным узлам в целях достижения баланса нагрузки и, как следствие, высокой эффективности сгенерированной параллельной программы. Таким образом, система LuNA позволяет программисту ограничиться написанием последовательных модулей на языке C++ и достаточно простых программ на языке LuNA вместо создания полноценной параллельной программы, что часто сопряжено с допущением

различных ошибок, которые могут быть незаметными и недетерминированными в своём поведении (например, взаимная блокировка или состояние гонки).

Пример программы на языке LuNA представлен на рисунке 8. Исходный код импортированных модулей (атомарных ФК) представлен на рисунке 9.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4           df a, b, c;
5           init(1, a);
6           foo(a);
7           for iterator = 1..2 {
8                   let index = iterator {
9                           init(iterator, b[index]);
10                          print(b[index]);
11                  }
12          }
13  }
14  sub foo(name f){
15          if (f < 5){
16                  print(f);
17          }
18  }
```

Рисунок 8 – Пример программы на языке LuNA

```
1   #include <cstdio>
2   #include "ucenv.h"
3
4   extern "C" {
5
6           void printfl(double d){
7                   printf("%.14f\n", d);
8           }
9
10          void c_init_int(int val, OutputDF &df){
11                  df.setValue(double(val));
12          }
13
14  }
```

Рисунок 9 – Импортируемые модули на языке C++, использованные в программе на рисунке 8

Язык LuNA имеет 5 различных операторов, каждый из которых имеет свою специфику. Их поведение подробно описано в [27].

1. Оператор cf – указывает применение ФК (атомарный или структурированный) к конкретным ФД.

2. Оператор for (оператор примитивно-рекурсивного перечисления) – предписывает породить все ФВ внутри своего блока, причём каждый ФВ будет порождён столько раз, сколько позволит условие for, и каждый ФВ имеет доступ к конкретному значению итератора for.
3. Оператор while (оператор частично-рекурсивного перечисления) – аналогичен for, однако ограничивается не конкретными лимитами на значение итератора, а логическим условием с его участием. Также оператор инициализирует указанный ФД (называемый “выходным ФД”) числом прошедших итераций.
4. Оператор if (условный оператор) – порождает ФВ внутри своего блока в случае выполнения условия.
5. Оператор let (оператор внесения значения в контекст) – порождает ФВ внутри своего блока и заменяет определённые в нём имена указанными значениями.

Помимо операторов важными элементами синтаксиса являются объявления базовых имён (df). В начале каждого блока возможно указать, какие базовые имена в нём объявляются. Они будут видны для всех операторов этого блока и всех вложенных операторов (исключая блоки структурированных ФК). Базовые имена используются для порождения ФД.

Система LuNA, несмотря на достаточно простой синтаксис, позволяет программисту допускать малозаметные и трудноустранимые ошибки [28], поскольку обладает такими особенностями, как:

- недетерминированный порядок выполнения операций;
- единожды инициализируемые ФД;
- требование инициализации входных ФД.

Для облегчения процесса отладки программ на языке LuNA возможно использовать статический анализатор, однако в настоящее время готовых к использованию и доступных в открытом доступе статических анализаторов для LuNA-программ не существует. Важно заметить, что параллельно с данной работой велась разработка двух иных статических анализаторов, применяющих

различные методы анализа. В целях создания более удобного и качественного инструмента было принято решение провести работы по объединению всех трёх модулей в единый комплекс автоматизированной отладки, работа и структура которого описана в 4 главе.

Важно заметить, что, помимо поиска семантических ошибок, необходимо также иметь возможность обнаруживать и синтаксические ошибки. Это связано с ограничениями компилятора системы LuNA, который часто выводит информацию об обнаруженных недочётах в плохо читаемом формате. Статический анализатор должен быть способен предоставить полную информацию о найденной ошибке – её местонахождение, тип и причину. Такие ошибки будут рассмотрены в дальнейшем.

2.2 База типов ошибок в LuNA-программах

Для разработки алгоритмов поиска ошибок была создана база типов ошибок в программах на языке LuNA. База продолжает активно расширяться; на текущий момент она содержит 37 типов ошибок, среди которых есть как семантические, так и синтаксические. В дальнейшем для описываемых типов ошибок будет указываться код, присвоенный этому типу в базе. Сама база типов ошибок находится в открытом репозитории (https://github.com/LuNA-Static-Analysis/LuNA-Static-Analysis-Repository/wiki/База-ошибок).

2.3 Граф зависимостей по данным LuNA-программы

В качестве используемого промежуточного представления был выбран граф зависимостей по данным. Выбор обоснован недетерминированностью порядка выполнения ФВ – использовать граф потока управления становится невозможно из-за его требования к последовательному выполнению команд (оно необходимо, поскольку его требует трёхадресный код, который нужен для построения графа потока управления). Также граф зависимостей по данным позволяет обнаруживать различные семантические ошибки, связанные с данными (в случае системы LuNA – с ФД). Параллельно с данной работой велось создание двух иных анализаторов, использующих AST и логическое

программирование соответственно, и повторять эти разработки не было необходимости. В этой ситуации самым перспективным промежуточным представлением остался граф зависимостей по данным.

Специфика языка LuNA требует уточнения модели графа зависимостей по данным. В качестве вершин используются операторы языка LuNA, а дугами являются зависимости по фрагментам данных между ними (сплошные дуги, дуги типа “use-def”). В дальнейшем термины “вершина” и “оператор” могут быть использованы взаимозаменяемо. Также, в целях упрощения создания и применения графа используются дуги, указывающие участие операторов в блоках (пунктирные дуги, дуги типа “inside”). Созданное представление сочетает в себе полноту описания исходной программы, удобство использования и универсальность – к нему могут быть применены различные методы анализа (свёртка констант, анализ указателей, сравнение с шаблоном), некоторые из них будут рассмотрены в дальнейшем.

В качестве примера работы алгоритма построения графа зависимостей по данным рассмотрим программу на языке LuNA, содержащую несколько ошибок (рисунок 10).

```
1   import printint(int) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4           df multipleInitName, nonInitUsedName, unusedName;
5           init(1, multipleInitName);
6           init(2, multipleInitName);
7           print(multipleInitName);
8           print(nonInitUsedName);
9           let constOne = 1, nonIndexableName = 2 {
10                  foo(constOne);
11                  init(1, nonIndexableName[0]);
12          }
13  }
14  sub foo(int f){
15          if (f){
16                  print(f);
17          }
18  }
```

Рисунок 10 – Пример программы на языке LuNA с ошибками

Программа на рисунке 10 некорректна по следующим причинам:

1. Повторная инициализация ФД (код ошибки: LUNA03). На строчках 5 и 6 инициализируется один и тот же неиндексированный ФД *multipleInitName*.
2. Попытка использования неинициализированного ФД (код ошибки: LUNA05). На строчке 8 атомарный ФК print использует значение неинициализированного ФД *nonInitUsedName*.
3. Неиспользуемый ФД (код ошибки: LUNA10). Объявленное на строчке имя *unusedName* не используется и не инициализируется нигде. Если имя где-то инициализируется, но нигде не используется, то такая ситуация тоже считается ошибочной.
4. Тождественно истинная формула внутри if (код ошибки: LUNA23). На строчке 15 условие *f* представляет собой идентификатор, который содержит значение “1”, т.е. условие всегда выполняется.
5. Попытка индексации объекта, не являющегося ФД (код ошибки: LUNA36). На строчке 11 индексированный ФД *nonIndexableName[1]* содержит в качестве базового имени константу, скрытую под переменной let *nonIndexableName*, которую нельзя индексировать.

Эта программа будет использоваться для демонстрации алгоритма построения и применения графа зависимостей по данным.

2.4 Построение и применение графа зависимостей по данным LuNA-программы

Построение и применение графа состоит из четырёх этапов. Первые три отвечают за построение графа, а четвёртым этапом является его применение, т.е. поиск ошибок.

Этап 1: анализ сигнатур ФК

Первый этап представляет собой анализ сигнатур импортированных из C++ и описанных на языке LuNA процедур.

Анализ сигнатуры атомарного ФК сводится к сохранению имени фрагмента и типов его аргументов. Если тип указан как name, то этот аргумент

является инициализируемым, т.е. выходным; в ином случае аргумент является используемым, т.е. входным.

Анализ сигнатуры структурированного ФК сохраняет информацию об имени фрагмента, об именах и типах его аргументов, а также об их порядке в сигнатуре.

Для программы на рисунке 10 информация о ФК будет иметь следующий вид:

1. Структурированный ФК main без аргументов.
2. Структурированный ФК foo. Аргументы: первый – int *f*.
3. Атомарный ФК print. Аргументы: первый – использующийся.
4. Атомарный ФК init. Аргументы: первый – использующийся, второй – инициализируемый.

Этап 2: порождение вершин

На втором этапе совершается обход в глубину AST исходной программы и проводится создание вершин графа.

Обход в глубину начинается со структурированного ФК main (поскольку с него начинается работа программы), и для каждого оператора создаётся вершина соответствующего ему типа. В случае встречи оператора, имеющего свой блок, или оператора cf, вызывающего структурированный ФК, проводится порождение вершин для всех операторов в блоке, и полученные вершины связываются с оператором-владельцем блока дугами типа “inside” (пунктирные).

В случае встречи оператора cf, вызывающего атомарный ФК, по сохранённой информации о его сигнатуре указывается, какие ФД, участвующие в вызове ФК, были инициализированы (множество “def”), а какие использованы (множество “use”). Помимо этого, “use” и “def” указываются также для:

1. операторов for: порождаемые ими итераторы указываются как “def”, а имена, участвующие в выражениях-границах цикла, помечаются как “use”;

2. операторов while: аналогично for, итераторы помечаются как “def”, и имена внутри выражения, задающего начальное значение оператора, указываются как “use”; также выходной ФД оператора while указывается как “def”;
3. операторов if: имена, участвующие в условии, помечаются как “use”.

Во время всего этапа для каждого оператора ведётся учёт всех видимых в этой области имён – базовые имена, аргументы ФК, итераторы for и while, имена let. При возможности указывается, на какое именно выражение ссылается то или иное имя – например, для переменной let в программе на рисунке 10 *constOne* будет указано, что оно ссылается на единицу. На этом этапе проводятся проверки на дубликаты имён.

После проведения второго этапа граф программы на рисунке 10 будет иметь вид, указанный на рисунке 11.

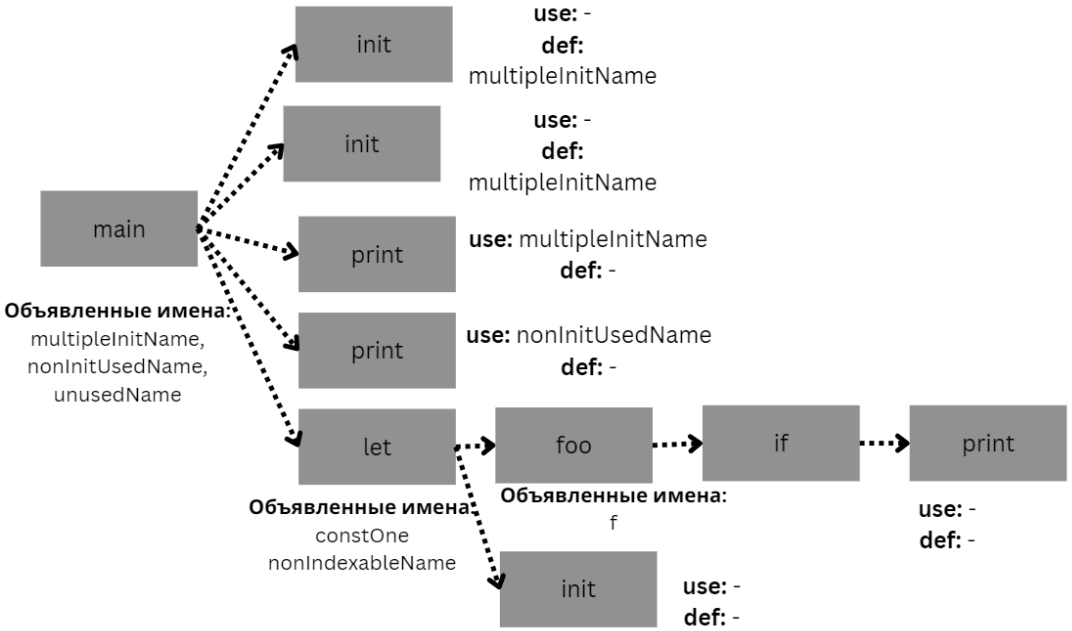

Рисунок 11 – Граф зависимостей программы на рисунке 10 после проведения второго этапа построения

У вершин указаны их имена; для операторов cf указываются имена вызываемых ими ФК, а для остальных операторов на вершинах указываются их названия.

Этап 3: связывание вершин

Третий этап состоит из связывания вершин дугами типа “use-def” (сплошные) в зависимости от инициализируемых и используемых ими ФД. Это достигается путём линейного обхода списка всех вершин графа. После проведения этого этапа граф считается построенным.

Итоговый вид графа программы, представленной на рисунке 10, продемонстрирован на рисунке 12.

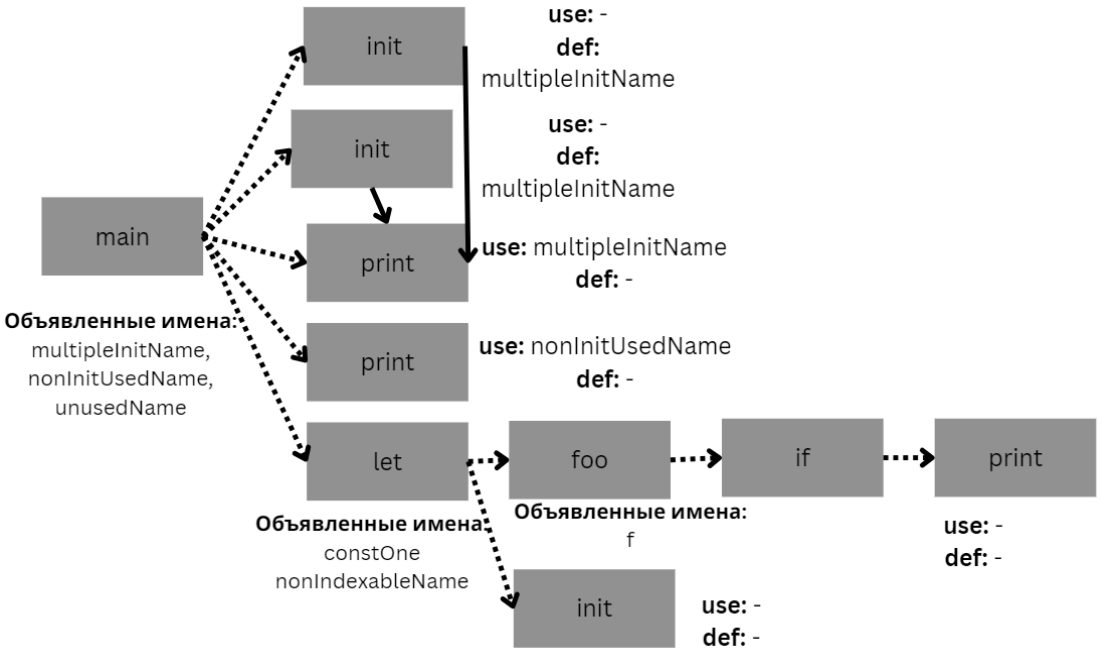

Рисунок 12 – Граф зависимостей по данным на рисунке 10

Этап 4: поиск ошибок

Различная природа ошибок в программах на языке LuNA диктует создание различных алгоритмов и функций для их поиска. На четвёртом этапе проводится запуск функций, которые обходят граф и обнаруживают различные недочёты. Рассмотрим, как именно будут обнаружены ошибки в программе на рисунке 10:

1. Повторная инициализация ФД. В случае, если ФД, инициализированный несколько раз, был где-то использован, то в использующую вершину буду вести несколько дуг типа “use-def” с одинаковыми метками. Однако даже если ФД не был использован, повторная инициализация приведёт к внештатному завершению программы, поэтому ошибку целесообразнее

искать с помощью отслеживания всех использований и инициализаций ФД для всех базовых имён. Отслеживание также позволит искать ошибки номер 2 и 3 (этот метод является альтернативой представленным ниже методам).

2. Попытка использования неинициализированного ФД. Вершина, использующая неинициализированный ФД, не будет иметь входящих дуг типа “use-def” с меткой используемого ФД.
3. Неиспользуемый ФД. Поиск таких ФД сводится к обходу вершин графа в любом порядке и проверкой, что каждый инициализированный ФД где-либо используется. Для этого достаточно проверить, что если вершина инициализирует некоторый ФД, то из неё выходит дуга типа “use-def” с соответствующей меткой, т.е. существует вершина, использующая этот ФД.
4. Тождественно истинная формула внутри if. Для поиска ошибки необходимо аппроксимировать значение выражения в условии оператора if. В случае, если выражение получается вычислить точно (используется метод свёртки констант [29]), ошибка считается обнаруженной; в иных случаях требуется применять методы анализа потока данных, которые будут рассмотрены в дальнейшем.
5. Попытка индексации объекта, не являющегося ФД. Для обнаружения такой ошибки необходимо использовать дополнительные структуры, которые позволяют отслеживать, какое именно выражение было присвоено данному ФД.

Граф зависимостей по данным также предоставляет возможность проводить анализ указателей [30, 31] – например, в некоторых случаях возможно предсказать значение того или иного ФД исходя из предположения о работе атомарного ФК. Это позволит точнее определять, какие значения хранятся в индексированных ФД.

2.5 Требования к создаваемому анализатору

Разрабатываемое средство должно представлять собой самостоятельную программу, способную проводить анализ произвольных программ на языке LuNA, а также докладывать о найденных ошибках пользователю в легко воспринимаемом формате. Помимо этого, необходимость интеграции разрабатываемого модуля в комплекс автоматизированной отладки требует соответствия вывода программы некоторому унифицированному формату, который также необходимо разработать, после чего создать утилиту для генерации текстовых отчётов о найденных ошибках.

Требования для разрабатываемого модуля статического анализа:

- Наличие функций для автоматического построения графа зависимостей по данным произвольной LuNA-программы.
- Наличие функций, использующих граф зависимостей по данным для поиска ошибок в программе.
- Наличие функций, позволяющих генерировать отчёты об ошибках в унифицированном формате.
- Наличие консольного интерфейса.
- Время работы модуля при анализе программы до 5 тысяч строк не должно превышать 1 минуту.

Глава 3. Модуль статического анализа DeGSA

Разработанная программа для проведения статического анализа носит название DeGSA (Dependence Graph Static Analyzer). Анализатор написан на языке C++ и применяет AST исходной программы, автоматическое построение которого реализовано Максимом Курбатовым. Использование отличного от используемого в системе LuNA AST обусловлено тем, что последнее не строится в случае попытки компиляции программы, содержащей синтаксические ошибки. Это неприемлемо, так как анализатор используется в том числе для вывода информации о синтаксических ошибках в удобном для пользователя формате.

Модуль использует описанные в главе 2 алгоритмы построения графа, а также применяет дополнительные структуры и алгоритмы в целях расширения спектра типов обнаруживаемых ошибок. Информация об их характере, местонахождении и прочих свойствах сохраняется в виде объектов JSON, после чего передаётся составляющей комплекса ADAPT, предназначенной для генерации текстовых отчётов, которые выводятся пользователю.

3.1 Структура графа и вспомогательные объекты

Помимо модели, реализующей структуру графа, рассмотренную в главе 2, для поиска более сложных в обнаружении ошибок были разработаны дополнительные структуры. Среди них необходимо сначала выделить систему выражений и идентификаторов (в дальнейшем называемой структурой “Identifier-Expression”), на которую опирается использование графа; затем будет описана реализация вершин и дуг графа, а также их порождение.

3.1.1 Выражения

Выражением называется любой арифметический оператор или идентификатор, который в указанном месте отдаёт своё значение. Например, условие внутри if или while является выражением; также выражениями являются аргументы в месте вызова ФК.

Для работы с выражениями создан класс Expression, который содержит в себе синтаксическое дерево выражения. Объект этого класса содержит либо:

1. тип оператора и указатели на операнды (один или два, поскольку в языке LuNA существуют бинарные и унарные операторы), которые в свою очередь являются объектами класса Expression (в случае, если операнды являются константами или арифметическими операторами) или производных классов Identifier;
2. тип значения константы (int, real или string) и её значение;
3. указатель на идентификатор.

Помимо этого, объект Expression содержит в себе ссылку на оператор (т.е. на объект-вершину), в котором выражение находится.

Пример программы на языке LuNA указан на рисунке 13; часть структуры из объектов Expression для этой программы показана на рисунке 14.

```
1   import printint(int) as print;
2   sub main(){
3           foo(1 * 2);
4   }
5   sub foo(int f){
6           print(f + 3 + 4.0);
7   }
```

Рисунок 13 – LuNA-программа

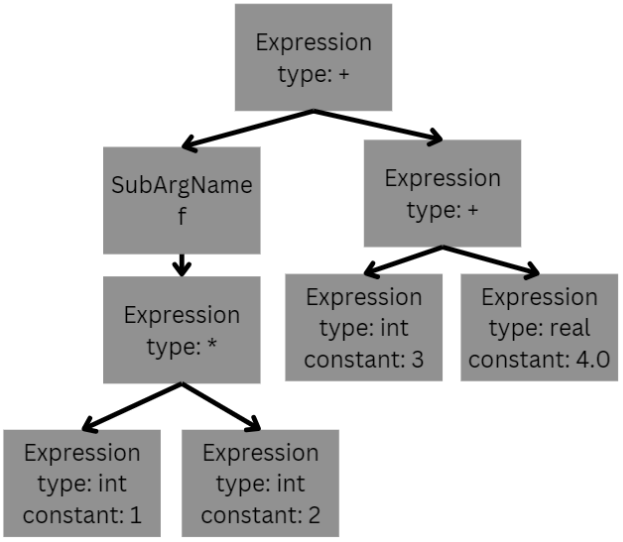

Рисунок 14 – Структура “Identifier-Expression” для программы на рисунке 13

3.1.2 Идентификаторы

Идентификатором называется любой именованный объект, который может быть использован для получения значения. Например, идентификатором

считается базовое имя, поскольку оно позволяет порождать индексированные и неиндексированные ФД, которые содержат данные непосредственно. Идентификаторами также считаются итераторы for и while, переменные let, аргументы структурированных ФК, индексированные и неиндексированные ФД, а также аргументы main (они отделяются от других аргументов ФК, поскольку не могут быть инициализированы).

Идентификаторы представлены абстрактным классом Identifier, и для каждого из вышеперечисленных типов идентификаторов существует соответствующий класс-наследник. Для каждого идентификатора в программе создаётся ровно один объект. Класс Identifier содержит следующие данные:

- указатель на вершину, в которой идентификатор был порождён (зависит от типа идентификатора);
- имя идентификатора;
- тип идентификатора.

Для идентификаторов, описывающих итераторы for и while, указатель на вершину содержит соответствующие операторы; для идентификаторов базовых имён указывается вершина, в блоке которой объявлено соответствующее базовое имя; для идентификаторов аргументов структурированных ФК указывается вершина, соответствующая оператору cf, который вызывает данный ФК; для аргументов main указывается вершина main, для переменных let указывается вершина let.

Всего на текущий момент существует 8 классов, описывающих идентификаторы.

SubArgName – идентификатор, описывающий аргумент структурированного ФК внутри блока самого ФК. Порождается в начале обхода блока вызванного ФК. Его особенностью является привязка к выражению, которое представляет собой аргумент в месте вызова этого ФК. Для этого он содержит поле *reference*: указатель на Expression. Примеры представлены на рисунках 15 и 16.

```
1   import printint(int) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4       df a;
5       foo(a);
6   }
7   sub foo(name f){
8       init(1, f);
9       print(f);
10  }
```

Рисунок 15 – LuNA-программа

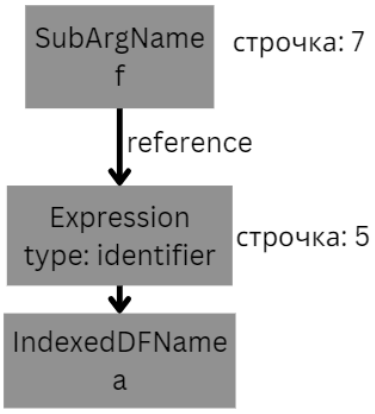

Рисунок 16 – Часть структуры “Identifier-Expression” для программы на рисунке 15

BaseDFName – идентификатор, описывающий базовое имя. Порождается в начале обхода блока, где были указаны базовые имена (имена после ключевого слова df). Он содержит в себе информацию о каждом индексированном и неиндексированном ФД, которые участвуют в программе, а также хранит информацию о том, были они использованы или инициализированы. Примеры представлены на рисунках 17 и 18.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4       df a;
5       init(1, a);
6       init(2, a[0]);
7       init(3, a[1]);
8       print(a);
9   }
```

Рисунок 17 – LuNA-программа

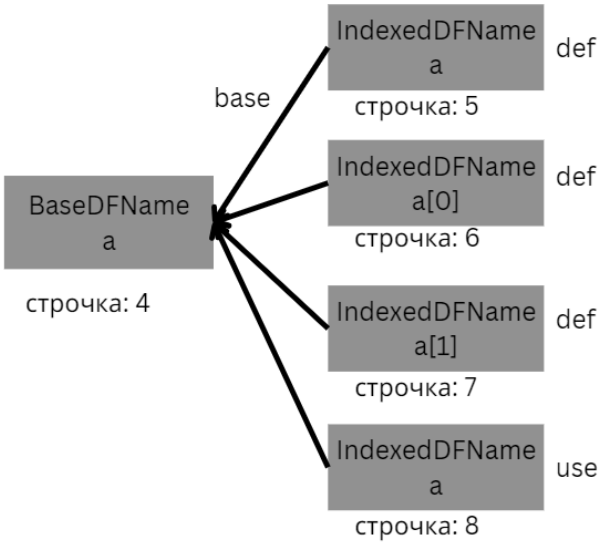

Рисунок 18 – Часть структуры “Identifier-Expression” для программы на рисунке 17

IndexedDFName – идентификатор, описывающий индексированные и неиндексированные ФД (можно считать, что неиндексированные ФД это индексированные ФД с нулём индексов). Порождается во время парсинга выражений и сопутствующего создания объектов Expression. Содержит в себе указатель на своё базовое имя, а также массив указателей на объекты Expression, представляющие его индексы. Примеры представлены на рисунках 19 и 20.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4       df a;
5       let b = a[0] {
6           init(1, a[0]);
7           init(2, b[0]);
8           foo(b[0]);
9       }
10  }
11  sub foo(name f){
12      init(3, f[0]);
13  }
```

Рисунок 19 – LuNA-программа

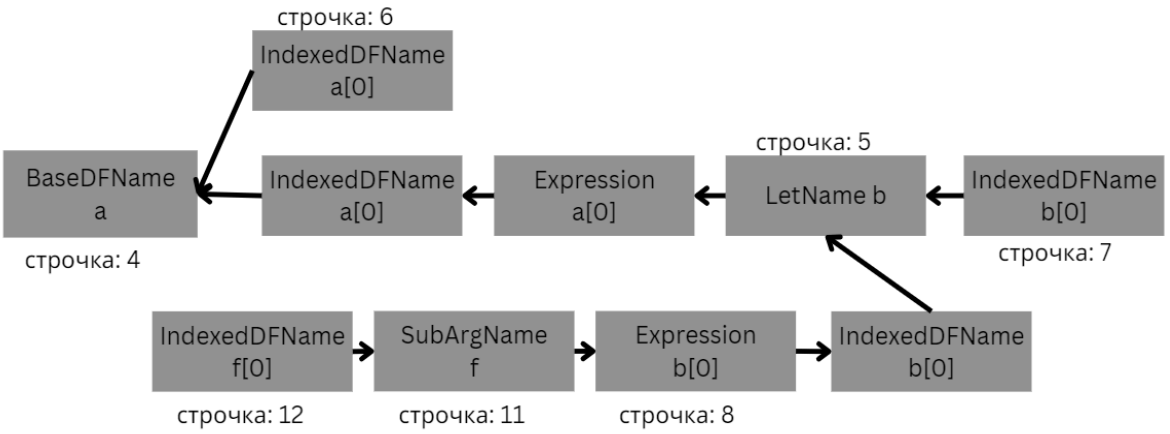

Рисунок 20 – Часть структуры “Identifier-Expression” для программы на рисунке 19

Необходимо отметить, что базовым именем индексированного ФД в тексте программы не обязательно является имя, соответствующее BaseDFName – например, можно породить индексированный ФД с помощью переменной let (рисунок 19, строчка 7) или аргумента структурированного ФК (рисунок 19, строчка 12).

При порождении объекта IndexedDFName проводятся проверки на возможность использования предлагаемого базового имени. Несмотря на возможность использования аргументов ФК и переменных let в качестве базовых имён, последние должны в любом случае приводить к базовому имени, объявленному после ключевого слова df; в противном случае программа является некорректной. Проверка заключается в обходе в глубину дерева-подграфа структуры “Identifier-Expression”, чьим корнем является создаваемый объект IndexedDFName.

ForIteratorName – идентификатор, описывающий итератор оператора for. Порождается во время анализа операторов for и порождения соответствующих вершин. Он не содержит в себе дополнительной информации (помимо заложенной в базовый класс Identifier), но предоставляет методы для доступа к границам цикла. Также объекты этого класса автоматически помечаются как “def” при порождении вершин. Примеры представлены на рисунках 21 и 22.

```
1   import printint(int) as print;
2   sub main(){
3       for i = 1..10 {
4           print(i + 1);
5       }
6   }
```

Рисунок 21 – LuNA-программа

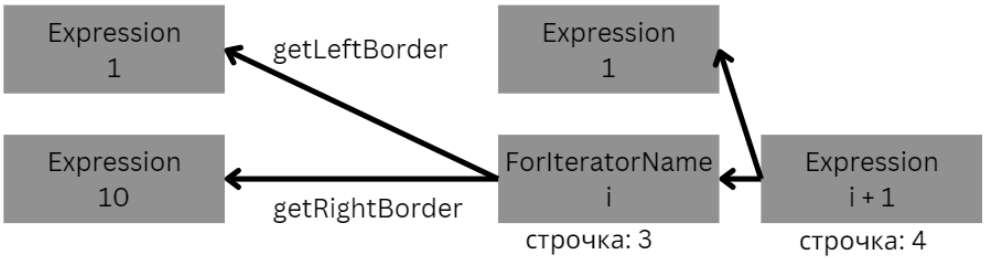

Рисунок 22 – Часть структуры “Identifier-Expression” для программы на рисунке 21

WhileIteratorName – идентификатор, описывающий итератор оператора while. Порождается во время анализа операторов while и порождения соответствующих вершин. Аналогично ForIteratorName, не содержит дополнительной информации, но предоставляет методы для доступа к начальному значению и условию оператора while. Примеры представлены на рисунках 23 и 24.

```
1   import printint(int) as print;
2   sub main(){
3       df a;
4       while (i < 10), i = 0 .. out a {
5           print(i + 1);
6       }
7   }
```

Рисунок 23 – LuNA-программа

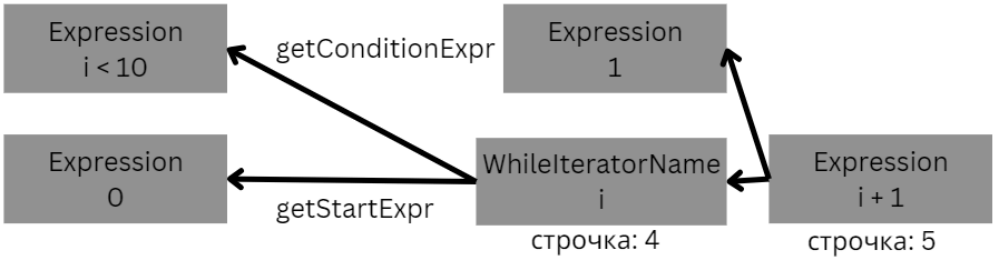

Рисунок 24 – Часть структуры “Identifier-Expression” для программы на рисунке 23

LetName – идентификатор, описывающий переменную оператора let. Порождается во время анализа операторов let и порождения соответствующих вершин. Содержит указатель на объект Expression, представляющий значение переменной let в программе. Примеры представлены на рисунках 25 и 26.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4       df a;
5       let b = a, c = a + 1.0 {
6           init(1, b);
7           print(b);
8           print(c);
9       }
10  }
```

Рисунок 25 – LuNA-программа

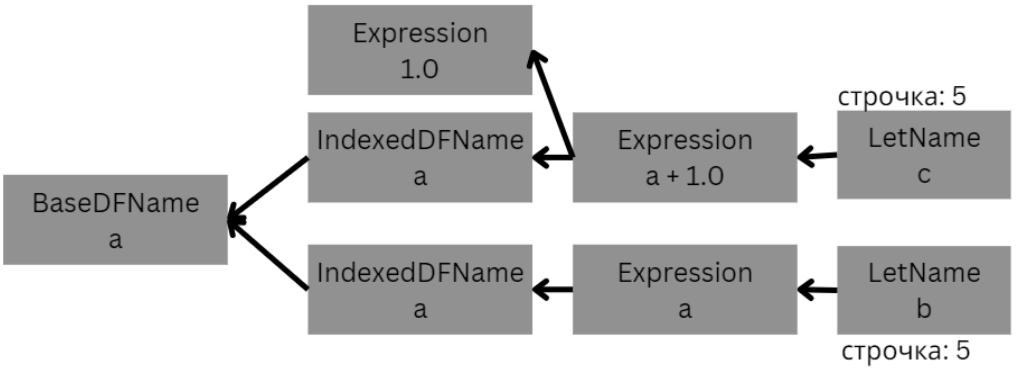

Рисунок 26 – Часть структуры “Identifier-Expression” для программы на рисунке 25

MainArgName – идентификатор, описывающий аргумент функции main. Порождается во время начала анализа блока main. Не содержит дополнительной информации. Примеры представлены на рисунках 27 и 28.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(int mainArg){
4       df a;
5       init(mainArg + 1, a);
6       print(a);
7   }
```

Рисунок 27 – LuNA-программа

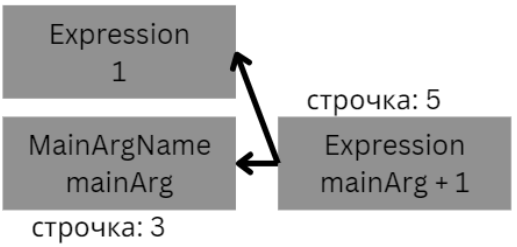

Рисунок 28 – Часть структуры “Identifier-Expression” для программы на рисунке 27

ValueId – идентификатор, отвечающий за имена типа value. В настоящее время не реализован, но в дальнейшем будет использоваться.

3.1.3 Структура “Identifier-Expression”

Порождённые таким образом объекты классов Identifier и Expression образуют граф, который идёт параллельно графу зависимостей по данным. Её связность позволяет реализовать методы “отметок” markAsUse и markAsDef, а также вычислять значения некоторых выражений.

Методы markAsUse и markAsDef присутствуют у обоих классов и всех их наследников, и представляют собой рекурсивные вызовы себя же у всех объектов, на которые ссылаются текущий, до тех пор, пока не будет обнаружен конечный объект. Все объекты, у которых был вызван метод, будут указаны как используемые (“use”) или инициализируемые (“def”), в зависимости от метода. Пример работы метода markAsUse, вызванного у объекта Expression, соответствующего выражению на строчке 8 программы на рисунке 29, представлен на рисунке 30.

```
1   import printfl(real) as print;
2   import c_init_int(int, name) as init;
3   sub main(){
4       df a;
5       for i = 1..2 {
6           let b = a + 5.0 {
7               init(1, a);
8               print(a + b * i);
9           }
10      }
11  }
```

Рисунок 29 – LuNA-программа

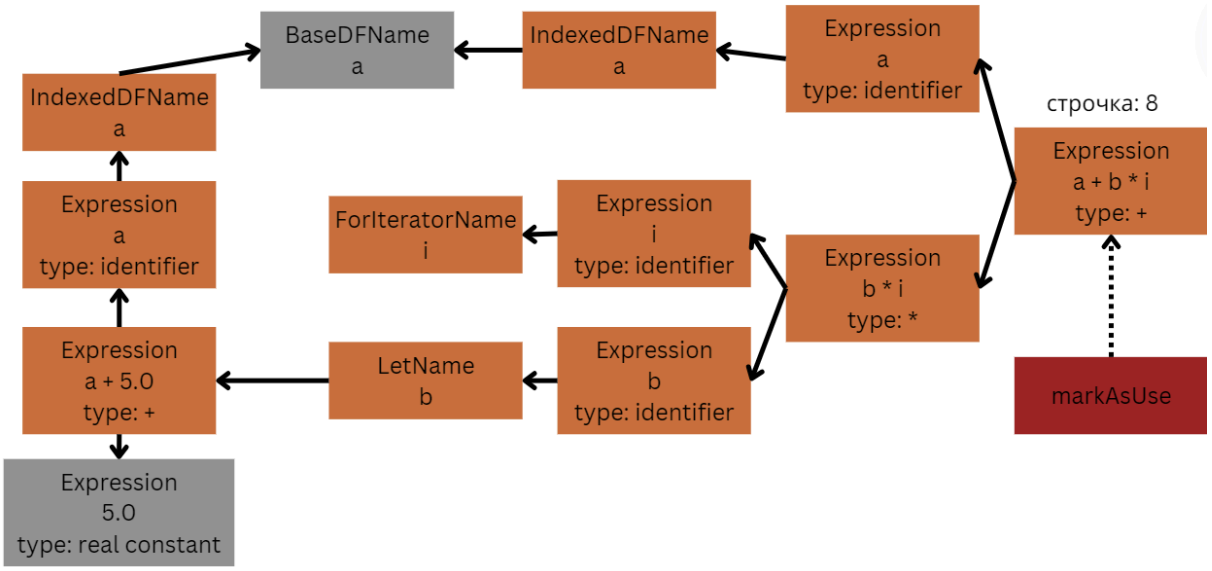

Рисунок 30 – Часть структуры “Identifier-Expression” и работа метода markAsUse для программы на рисунке 29

Все указанные оранжевым объекты помечаются как “use”; константы не помечаются, поскольку в этом нет необходимости; BaseDFName технически не помечаются, поскольку содержат информацию о всех использованиях и инициализациях порождаемых ими ФД в ином формате, но markAsUse сохраняет в них эту информацию. Метод используется, когда в исходном коде программы встречается выражение, значение которого используется; такими ситуациями являются:

1. выражения внутри for: начальные и конечные значения итератора;
2. выражения внутри while: начальное значение итератора, условное выражение;
3. условное выражение внутри if;
4. выражение, являющееся аргументом атомарного ФК, который указан как используемый.

Аналогичным образом работает метод markAsDef, помечающий идентификаторы и выражения как инициализируемые, однако он отличается наличием проверок на валидность: в системе LuNA возможно инициализировать лишь ФД, имеющие в качестве базового имени имя, объявленное после ключевого слова df, а также итераторы for и while.

Соответственно, метод проводит проверки и возвращает доклады об обнаружении ошибки при встрече:

1. бинарных и унарных операций;
2. аргументов main;
3. итераторов for и while (поскольку они автоматически считаются инициализированными).

Метод используется, когда в исходном коде программы встречается выражение, которое инициализируется; такими ситуациями являются:

1. выражение, являющееся выходным ФД оператора while;
2. выражение, являющееся аргументом атомарного ФК, который указан как инициализируемый.

Также инициализированными считаются итераторы for и while, но они указываются как инициализированные вручную.

Методы markAsUse и markAsDef возвращают массив строк, представляющих собой доклады о найденных ошибках в формате JSON; эти доклады собираются централизованно во время построения графа и в дальнейшем записываются в файл.

Помимо этого, класс Expression содержит метод getAsConstant, который представляет собой рекурсивную функцию, которая обходит структуру “Identifier-Expression” в глубину и пытается вычислить значение выражения, если все листья структуры является константами. Структура необходима для получения значений переменных let и аргументов структурированных ФК. Пример его работы для программы на рисунке 31 представлен на рисунке 32.

```
1   import printfl(real) as print;
2   sub main(){
3       let b = 1.0 {
4           foo(b);
5       }
6   }
7   sub foo(real f){
8       print(f + 1.0);
9   }
```

Рисунок 31 – LuNA-программа

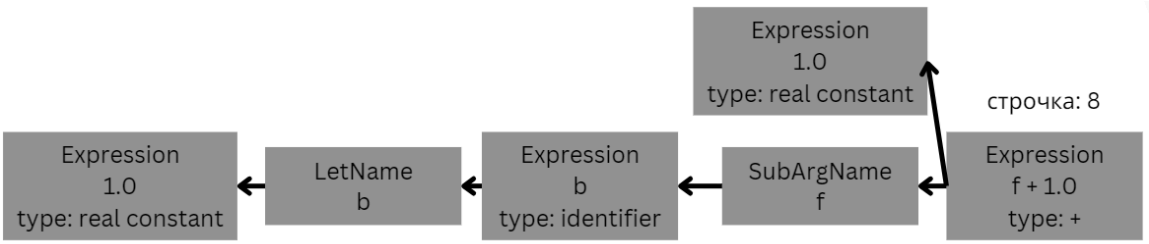

Рисунок 32 – Часть структуры “Identifier-Expression” для программы на рисунке 31

Метод getAsConstant был вызван у объекта, соответствующего выражению на строчке 8 указанной на рисунке 31 программы. Поскольку все листья являются константами, getAsConstant вернёт значение 2.0. Метод используется для поиска константных условий внутри операторов while и if.

3.1.4 Вершины и дуги

Для работы с вершинами создан абстрактный класс Vertex; для каждого типа операторов выделен свой класс-наследник Vertex. Содержимое объектов Vertex можно разделить на данные, необходимые для анализа, и данные, необходимые для возвращения информации о найденных ошибках. Данные, необходимые для анализа, включают в себя:

- множества входных и выходных вершин (т.е. вершины, соединённые с текущей дугами типа “use-def”);
- множество вершин-детей (т.е. вершины, соединённые с текущей выходящими дугами типа “inside”, чьи операторы находятся в блоке текущей вершины);
- ссылку на вершину-родителя (входящая дуга типа “inside”);
- множества инициализированных и использованных идентификаторов;
- множество видимых внутри блока идентификаторов;
- множество видимых самому оператору идентификаторов;
- имя (необходимо для анализа ФК);

Данные, необходимые для возвращения информации об ошибках:

- имя файла (поскольку система LuNA поддерживает многофайловые программы);
- строчка кода, на которой находится оператор;

Для генерации отчётов об ошибках могут использоваться данные из обеих групп.

Для реализации дуг типа “use-def” используется класс Binding, являющийся внутренним классом Vertex; его объекты содержат указатель на вершину, к которой ведёт дуга, а также указатель на идентификатор, который является меткой дуги.

Всего на текущий момент существует 5 классов, описывающих вершины.

CFVertex – вершина, представляющая оператор cf, т.е. вызов ФК. Содержит массив указателей на SubArgName или MainArgName, являющихся его аргументами при объявлении ФК в соответствующем порядке.

ForVertex – вершина, представляющая оператор for. Содержит указатели на левую и правую границы цикла (Expression), а также на итератор (ForIteratorName).

WhileVertex – вершина, представляющая оператор while. Содержит указатели на начальное значение итератора (Expression), условие (Expression), итератор (WhileIteratorName) и выходной ФД (Identifier).

IfVertex – вершина, представляющая оператор if. Содержит указатель на условие (Expression).

LetVertex – вершина, представляющая оператор let. Содержит массив указателей на объявленные переменные (LetName) в соответствующем порядке.

3.1.5 Генератор отчётов

Для предоставления пользователю информации об обнаруженных в LuNA-программе ошибках используется двухэтапная схема:

1. генерация отчётов в формате JSON;
2. вывод информации JSON-файла в удобном для пользователя формате.

Поскольку модуль DeGSA изначально создавался как часть комплекса автоматизированной отладки, необходимо было создать унифицированный формат хранения информации об ошибках, который использовался бы всеми модулями – это позволило бы в автоматическом режиме сравнивать результаты

работы различных анализаторов, а также упростить вывод информации в удобном формате. Разработанный унифицированный формат хранится в открытом репозитории (https://github.com/LuNA-Static-Analysis/LuNA-Static-Analysis-Repository/wiki/Формат-вывода-ошибок#форматы-описания-ошибок).

Генератор отчётов представляет собой класс JsonReporter, предоставляющий статические методы для генерации объектов типа string, который возможно автоматически конвертировать в объекты JSON. Для каждого типа ошибки существует свой метод, в который достаточно загрузить релевантную для этого вида ошибок информацию – зачастую достаточно загрузить объект класса Vertex или Identifier, после чего метод соберёт нужную информацию (например, стек вызовов) из структуры графа или “Identifier-Expression”. Возвращаемое методом значение сохраняется в массив отчётов, который является общим для всей программы и хранит информацию о всех обнаруженных ошибках; после завершения всех этапов построения и применения графа все объекты массива записываются в файл, общий для всего комплекса отладки; в дальнейшем этот файл используется утилитой adapt_output_generator, описанной в главе 4, которая генерирует отчёты в легко воспринимаемом формате. Отчёт модуля после анализа программы на рисунке 10 представлен в приложении А вместе с командой запуска модуля.

3.2 Реализация алгоритмов построения графа

Описанный в главе 2 алгоритм реализован поэтапно, по одной функции на каждый шаг (не считая вспомогательных).

Граф создаётся внутри одного объекта класса DDG, который содержит методы для проведения этапов построения и применения. Помимо генерации отчётов об обнаруженных ошибках в унифицированном формате по мере работы модуль также выводит краткую информацию о тех ошибках, которые препятствуют построению графа зависимостей по данным. Она выводится в терминал с префиксом “INTERNAL ERROR”.

3.2.1 Анализ сигнатур ФК

Анализ сигнатур ФК, представляющий собой первый этап построения графа, реализован методом findSubs. Функция анализирует верхние узлы AST, содержащие информацию об объявлениях ФК.

В случае встречи структурированного ФК сохраняется ссылка на узел AST, содержащая объект-блок этого ФК. Ссылка сохраняется в структуру “словарь”, чтобы в дальнейшем возможно было получить доступ к блоку ФК по имени последнего. Помимо этого в аналогичный словарь сохраняется массив, содержащий объекты-аргументы ФК. Важным элементом является проверка названия ФК на совпадение с “main”; если такого ФК не было найдено, программа завершается досрочно за неимением точки начала работы.

В случае встречи атомарного ФК алгоритм схож: в словарь, позволяющий получить значение по имени ФК, сохраняется словарь, позволяющий по позиции аргумента понять, инициализируется ли он (аргумент имеет тип name) или используется (аргумент имеет любой другой тип). Также имя атомарного ФК сохраняется в структуру “множество”.

3.2.2 Порождение вершин

Порождение вершин реализуется двумя методами: enterVF и enterBlock. Каждый из них вызывает второй, таким образом запуская рекурсивный обход AST в глубину. Во время работы каждый метод сохраняет информацию о всех видимых именах в своей области видимости – это позволяет отслеживать ошибка такого типа как попытка использования необъявленного ФД или ФД с одинаковыми названиями в одной области видимости.

Функция enterVF принимает на вход узел AST LuNA-программы, отвечающий за вызов оператора и возвращает указатель на порождённую вершину соответствующего типа.

В случае, если входной оператор представляет собой вызов атомарного ФК, он полностью анализируется внутри функции без вызова enterBlock – порождаются выражения, соответствующие аргументам вызова ФК, которые помечаются как “use” или “def” согласно сохранённой на первом этапе

построения графа информации, после чего функция завершается с возвратом указателя на объект-вершину в качестве результата.

Во всех остальных случаях входной оператор имеет блок, и функция вызывает метод enterBlock для анализа операторов внутри блока текущего. Помимо этого, enterVF порождает объекты-вершины, а также сопутствующие идентификаторы (например, для вершин while и for порождаются итераторы, а для вершин let – объявленные переменные), которые связываются с их вершинами в две стороны.

Помимо создания вершин, внутри enterVF также создаются объекты классов Expression и наследников Identifier. Они связываются с вершиной в две стороны; это позволяет как получать информацию об окружении выражения во время анализа данных, так и, например, анализировать поведение операторов согласно выражениям в их условиях.

Обработка вызовов атомарных ФК представляет собой порождение объектов Expression для каждого из аргументов ФК, а также запуск соответствующих методов – markAsUse вызываются для выражений, которые будут использованы, а markAsDef для выражений, которые будут инициализированы. Информация о том, какие аргументы являются используемыми, а какие инициализируемыми, была сохранена на первом этапе построения графа.

Функция enterBlock реализуют линейный обход операторов в указанном блоке; для каждого оператора вызывается функция enterVF, которая возвращает указатель на порождённый объект класса Vertex. Созданный объект добавляется в список вершин, выходящих из текущей, таким образом реализуя дуги типа “inside”. Перед началом линейного обхода проводится проверка на наличие объявлений базовых имён (ключевое слово df) с порождением соответствующих объектов класса BaseDFName; объявленные имена добавляются в список видимых идентификаторов (проводятся проверки на наличие повторяющихся имён).

3.2.3 Связывание вершин

Связывание вершин реализовано линейным обходом списка порождённых вершин и порождением соответствующих объектов класса Binding согласно множествам “use” и “def” анализируемых вершин. Важно заметить, что связывание реализовано лишь для вершин типа for, while и cf, вызывающих атомарные ФК. Эта особенность связана с необходимостью упростить структуру графа – несмотря на то, что вызовы структурированных ФК, а также операторы let порождают новые имена, информация об использовании или инициализации ими тех или иных ФД дублирует информацию, которая уже содержится в вершинах, отвечающих за for, while и вызовы атомарных ФК, так как последние являются первородными в плане инициализации; оператор if же не имеет возможности инициализировать ФД, но содержит конечное использование, поэтому он также участвует в связывании.

3.3 Реализация алгоритмов поиска ошибок

На текущий момент модуль DeGSA обнаруживает и докладывает об ошибках 9 различных типов:

1. Несуществующая LuNA-подпрограмма (код ошибки: LUNA02): ошибка представляет собой ситуацию, когда оператор cf вызывает ФК, сигнатура которого не совпадает ни с одним из объявленных или импортированных ФК. Эта ошибка будет обнаружена на 2 этапе построения графа внутри функции enterBlock – при вызове функции enterVF для анализа оператора cf проводится получение информации о вызываемом ФК из массива данных о ФК, собранном на первом этапе. В отчёте будет содержаться информация о месте вызова несуществующей программы, а именно файл LuNA-программы, строчка в нём, а также имя вызываемого ФК.
2. Повторная инициализация ФД (код ошибки: LUNA03): ошибка, возникающая при попытке инициализировать ФД, который был инициализирован до этого (т.е. участвовал в качестве инициализируемого аргумента в вызове атомарного ФК). Поиск ошибки реализован с помощью структуры “Identifier-Expression”, а конкретно с помощью

линейного обхода списка всех объектов типа BaseDFName, поскольку каждый из них отслеживает все использования и инициализации порождённых им ФД. Во время линейного обхода для каждого встреченного уникального ФД указывается, сколько раз он был инициализирован; в случае, если инициализаций было больше одной, запускается функция генерации отчёта об ошибке. В настоящее время не учитываются выражения в индексах ФД – например, если в программе дважды инициализируется ФД *a[1]*, ошибка не будет обнаружена. В дальнейшем эта проблема будет решена с помощью анализа индексов и поиска констант среди них, а также, в некоторых случаях, с помощью анализа значений ФД. В случае использований неиндексированных ФД ошибка обнаруживается во всех ситуациях, но лишь с учётом ограничений текущего моделирования поведения операторов (в настоящее время не учитывается, что, например, for и while имеют более чем одну итерацию). В отчёте об ошибке содержатся данные об инициализированном ФД – его имя, строчка и файл, где он был объявлен, а также стеки вызовов всех его инициализаций.

3. Попытка использования неинициализированного ФД (код ошибки: LUNA05): ошибка, возникающая при использовании выражения, содержащего неинициализированный ФД, в качестве используемого аргумента (например, в условии if или while). Её поиск производится вместе с поиском предыдущей ошибки – во время линейного обхода списка объектов BaseDFName исследуется, был ли инициализирован каждый из ФД, который был хоть раз использован. Если нет, то запускается генератор отчёта; отчёт содержит имя ФД, строчка и файл, где он был объявлен, а также стеки вызовов всех его использований.
4. Неиспользуемый ФД (код ошибки: LUNA10): недочёт, который не является ошибкой и не приводит к некорректному поведению программы, но является неоптимальной ситуацией и может сигнализировать о наличии ошибок иных типов. Его поиск также проводится во время

линейного обхода списка объектов BaseDFName, и отчёт об ошибке генерируется в двух случаях. Первый – объявленное после ключевого слова df имя не породило ни одного индексированного или неиндексированного ФД (т.е. это имя может быть удалено из программы). Об этой ситуации будет сигнализировать пустой список указателей на объекты IndexedDFName внутри объекта BaseDFName. Второй случай – порождённый ФД не имеет использований (вне зависимости от того, был он инициализирован или нет); это также будет замечено при анализе данных внутри объекта BaseDFName. Отчёт об ошибке будет содержать информацию об имени неиспользуемого ФД (или базового имени), строчке и файле, где он был объявлен, а также стеки вызовов всех его инициализаций.

5. ФД с одинаковыми названиями в одной области видимости (код ошибки: LUNA13): ошибка, в которой существует два идентификатора с одним именем в одной области видимости (например, оператор for создаёт итератор с именем *i* внутри блока структурированного ФК, у которого уже есть аргумент под именем *i*). Такая ситуация будет обнаружена на втором этапе построения графа во время работы функции enterBlock, а именно во время сохранения информации об объявленных базовых именах; помимо этого ошибка обнаруживается внутри функции enterVF во время сохранения информации об:
   - аргументах вызванного структурированного ФК;
   - итераторе for;
   - итераторе while;
   - переменных let.

   Отчёт об ошибке содержит информацию о всех ФД, имеющих одинаковые имена в одной области видимости: их название и строчка и файл, где они были объявлены. Важно заметить, что только первый объявленный ФД будет являться полноценным участником анализа – все

последующие идентификаторы с таким именем не будут учитываться во время дальнейшей работы программы.

6. Попытка использования необъявленного ФД (код ошибки: LUNA14): ошибка, которая заключается в участии в арифметическом выражении ФД с именем, которого нет в текущей области видимости. Ошибка будет обнаружена на 2 этапе построения графа внутри функции enterVF во время порождения объектов IndexedDFName (проводится проверка существования подходящего базового имени) и Expression (проверка существования всех участвующих идентификаторов). В случае отсутствия нужных имён объект Expression не будет иметь полностью инициализированную структуру, а объект IndexedDFName не будет создан в принципе; тем не менее, ошибка не остановит работу программы, а отсутствие объектов будет учитываться в дальнейшем. Отчёт об ошибке будет содержать информацию об имени, которое было использовано в выражении – его название, а также строчка и файл, где оно было использовано.
7. Формула в if/while тождественно истинна/ложна (код ошибки: LUNA23): ситуация, когда условие внутри операторов while или if является константным выражением, т.е. оператор if становится возможным убрать, а while необходимо отредактировать, поскольку он будет работать вечно. Это будет обнаружено во время проведения 4 этапа, т.е. применения графа, с помощью линейного обхода списка вершин графа. В зависимости от типа вершины анализируются те или иные выражения (условия if или while, границы for, аргументы cf). Алгоритм поиска этой ошибки использует описанный в пункте 3.1.3 метод getAsConstant, проводящий свёртку констант. Отчёт об ошибке будет содержать константное арифметическое выражение, его значение, а также строчку и имя файла, где оно находится.
8. Попытка инициализации объекта, не являющегося ФД (код ошибки: LUNA26): ошибка, возникающая при попытке подстановки выражения,

не являющегося индексированным или неиндексированным ФД, в качестве инициализируемого аргумента атомарного ФК или выходного ФД оператора while. Ошибка будет обнаружена во время 2 этапа построения графа внутри функции markAsDef, вызываемой во время анализа вызовов атомарных ФК и операторов while (описано в пункте 3.1.3). Проверка представляет собой обход в глубину структуры “Identifier-Expression”, начиная с текущего выражения, и, в случае встречи неподходящего узла (например, бинарная операция, или аргумент main), генерируется отчёт об ошибке. Он будет содержать использованное выражение, стек вызовов и ФК (или оператор while), который был использован для инициализации выражения.

9. Попытка индексации объекта, не являющегося ФД (код ошибки: LUNA36): аналогичная предыдущей ошибка, проявляющаяся при индексации объекта, чьё базовое имя не является подходящим для индексации. Её обнаружение также проводится во время 2 этапа построения графа внутри функции, порождающей объекты IndexedDFName – их порождение содержит проверки базового имени, которое должно быть именем, объявленным после ключевого слова df; использование структуры “Identifier-Expression” необходимо для проверок в случаях, когда непосредственным базовым именем индексированного ФД является переменная let или аргумент структурированного ФК, которые могут содержать в себе как подходящее базовое имя, так и, например, бинарную операцию. Отчёт об ошибке будет содержать использованное выражение и стек вызовов, где оно было использовано.

3.4 Выходные файлы

Помимо генерации JSON-файлов с отчётами о найденных ошибках, модуль DeGSA также генерирует текстовый файл degsa_output.txt, который содержит информацию о:

- всех порождённых вершинах: их имена, типы, строчки и файлы, где находятся соответствующие им операторы, входные и выходные

дуги типа “use-def”, вершина-родитель (входная дуга типа “inside”), вершины-наследники (выходные дуги типа “inside”), использованные ими выражения, объявленные и видимые имена, использованные и инициализированные идентификаторы, а также общее количество вершин;

- отчёты об ошибках в формате JSON, записанные построчно;
- время работы программы поэтапно: количество секунд, затраченных на построение AST, построение DDG и на поиск ошибок (4 этап работы, т.е. применение графа).

Глава 4. Тестирование и интеграция анализатора в комплекс автоматизированной отладки ADAPT

Тестирование модуля DeGSA проводилось в рамках работы комплекса ADAPT. Сначала необходимо описать схему работы комплекса и роль DeGSA в его функционировании.

4.1 Комплекс ADAPT и интеграция в него модуля DeGSA

Комплекс позволяет применить три различных инструмента для поиска ошибок. Первый из них – AST-analyzer, модуль построения и анализа абстрактного синтаксического дерева LuNA-программы. Поскольку он занимается созданием AST и (как и модуль DeGSA) не изменяет дерево во время анализа, было принято решение объединить компиляцию и запуск AST-analyzer и DeGSA в целях избежания создания двух одинаковых объектов. Компиляция требует запуска программ flex и bison, которые предоставляют инструментарий для генерации AST:

lex -o ./parser/lex.yy.cpp ./parser/lexics.l

bison -d ./parser/grammar.ypp -o ./parser/grammar.tab.cpp

После этого проводится компиляция файла adapt.cpp, содержащего вызовы конструкторов AST и DDG:

g++ -std=c++20 ./parser/lex.yy.cpp ./parser/grammar.tab.cpp adapt.cpp ./ast_analyzer/threadpool/threadpool.cpp -o ./bin/ast-ddg-analyzer -O2

Третий инструмент анализа – prolog-analyzer, на текущий момент взаимодействует с двумя другими модулями исключительно модификацией выходного JSON-файла с отчётами об обнаруженных ошибках, поэтому интеграция DeGSA в комплекс не требует учитывания специфики модуля prolog-analyzer. Он реализован на языке Python и не требует компиляции.

Запуск комплекса ADAPT представляет собой запуск программы на языке Python, предоставляющей выбор запуска конкретных анализаторов (опция *--run*), а также сохранение выходных файлов (опция *--no-cleanup*; например, по умолчанию файл degsa_output.txt не сохраняется).

4.2 Унифицированный формат вывода отчётов об ошибках

Для возможности интеграции модулей в единый комплекс было необходимо разработать унифицированный формат вывода отчётов об обнаруженных ошибках. Было принято решение создать форматы описания нескольких типовых объектов (например, информация о ФД или ФК), после чего использовать их для конструирования форматов отчётов об ошибках, по одному формату на каждый тип, для чего был выбран формат JSON. На текущий момент создано и используется 7 форматов описания типовых объектов:

- элемент стека вызовов (call_stack_entry);
- стек вызовов (call_stack);
- ФД (df);
- ссылка на ФД (df_ref);
- индексированный ФД/Диапазон индексов (index_range);
- ФК (sub);
- цикл for (for).

Их подробное описание находится в открытом репозитории (https://github.com/LuNA-Static-Analysis/LuNA-Static-Analysis-Repository/wiki/Формат-вывода-ошибок#формат-описания-типовых-объектов).

Описание ошибки представляет собой JSON-объект с двумя полями, первое из которых содержит код, присвоенный типу ошибки в базе, а второе содержит JSON-объект в соответствующем типу ошибки формате. Результатом работы запущенных модулей является массив JSON-объектов, каждый из которых содержит описание ровно одной ошибки.

4.3 Утилита adapt_output_generator

Утилита adapt_output_generator представляет собой программу на языке Python, принимающую на вход JSON-файл с отчётами об обнаруженных модулями ошибках, и подставляющую информацию из них в шаблоны докладов об ошибках, после чего она выводится в консоль. Для каждого типа ошибки создан свой шаблон, который хранится в JSON-файле report_templates.json. Это

необходимо для повышения читаемости отчётов, поскольку объекты JSON представляются в тяжело воспринимаемом человеком формате. Пример нескольких используемых шаблонов представлен на рисунке 33; пример вывода утилиты adapt_output_generator представлен на рисунке 34.

```
"09": "error[LUNA09]: Attempt to use DF after its deletion.",
"10": "error[LUNA10]: Unused DF $df_name.\n$df",
"11": "error[LUNA11]: Importing multiple functions with same aliases.\n- Duplicate functions information:\n$cfs",
"12": "error[LUNA12]: No function named \"main\"",
"13": "error[LUNA13]: Identifiers with same names in one namespace.\n- DFs information:\n$dfs",
"14": "error[LUNA14]: Attempt to use undeclared DF $df_name.\n$df",
```

Рисунок 33 – Пример шаблонов текстового описания обнаруженных ошибок

```
(3) error[LUNA26]: Attempt to initialize identifier a that is not a DF.
- Function information:
Name: init, type: extern, file "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 11
- Callstack:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 11, in init
    init(1, f)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 7, in foo
    foo(a)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 6, in let
    let b = a + 1
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 4, in main
    sub main()

(4) error[LUNA10]: Unused DF c.
Declared:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/sourceold.fa", line 4, in main
    sub main()
```

Рисунок 34 – Пример вывода утилиты adapt_output_generator

4.4 Тестирование модуля DeGSA

4.4.1 Тестирование корректности обнаружения ошибок

Для тестирование корректности обнаружения ошибок использовались программы, представленные в приложении Б и содержащие по одной ошибке каждая (типы ошибок описаны в пункте 3.3). Каждый запуск модуля породил файл JSON с объектом-докладом об ошибке со всей информацией, описанной в шаблоне описания ошибок. Вывод комплекса при анализе всех программ был корректен. Команда для запуска модуля: *adapt [путь к анализируемой LuNA-программе] --run degsa*

Вывод комплекса ADAPT по результатам работы модуля DeGSA для каждой из программ также приведён ниже.

1. Несуществующая LuNA-подпрограмма (код ошибки: LUNA02) – результат работы комплекса представлен на рисунке 35.

```
(1) error[LUNA02]: No function with name print exists.
- Called at:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna02.fa", line 2, in print
    print(1)
```

Рисунок 35 – Вывод комплекса ADAPT после анализа программы номер 1 в приложении Б

2. Повторная инициализация ФД (код ошибки: LUNA03) – результат работы комплекса представлен на рисунке 36.

```
(1) error[LUNA03]: Multiple initializing of a DF a.
- DF information:
Declaration callstacks:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 2, in main
    sub main()

Initialization callstacks:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 4, in init
    init(1, a)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 2, in main
    sub main()

  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 10, in init
    init(2, f)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 6, in foo
    foo(b)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 5, in let
    let b = a
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna03.fa", line 2, in main
    sub main()
```

Рисунок 36 – Вывод комплекса ADAPT после анализа программы номер 2 в приложении Б

3. Попытка использования неинициализированного ФД (код ошибки: LUNA05) – результат работы комплекса представлен на рисунке 37.

```
(1) error[LUNA05]: Attempt to use uninitialized DF a.
Declared:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna05.fa", line 3, in main
    sub main()

Used:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna05.fa", line 10, in print
    print(f)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna05.fa", line 6, in foo
    foo(b)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna05.fa", line 5, in let
    let b = a
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna05.fa", line 3, in main
    sub main()
```

Рисунок 37 – Вывод комплекса ADAPT после анализа программы номер 3 в приложении Б

4. Неиспользуемый ФД (код ошибки: LUNA10) – результат работы комплекса представлен на рисунке 38.

```
(1) error[LUNA10]: Unused DF a.
Declared:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 2, in main
    sub main()

Initialized:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 9, in init
    init(1, f)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 5, in foo
    foo(b)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 4, in let
    let b = a
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 2, in main
    sub main()


(2) error[LUNA10]: Unused DF c.
Declared:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna10.fa", line 2, in main
    sub main()
```

Рисунок 38 – Вывод комплекса ADAPT после анализа программы номер 4 в приложении Б

5. ФД с одинаковыми названиями в одной области видимости (код ошибки: LUNA13) – результат работы комплекса представлен на рисунке 39.

```
(1) error[LUNA13]: Identifiers with same names in one namespace.
- DFs information:
Name: i
Declared:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna13.fa", line 3, in main
    sub main()
```

Рисунок 39 – Вывод комплекса ADAPT после анализа программы номер 5 в приложении Б

6. Попытка использования необъявленного ФД (код ошибки: LUNA14) – результат работы комплекса представлен на рисунке 40.

```
(1) error[LUNA14]: Attempt to use undeclared DF d.
Declared:

Used:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna14.fa", line 8, in print
    print(d)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna14.fa", line 5, in foo
    foo(a)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna14.fa", line 3, in main
    sub main()
```

Рисунок 40 – Вывод комплекса ADAPT после анализа программы номер 6 в приложении Б

7. Формула в if/while тождественно истинна/ложна (код ошибки: LUNA23) – результат работы комплекса представлен на рисунке 41.

```
(1) warning[LUNA23]: Condition f > 1 is always true.
In:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna23.fa", line 8, in if
    if (f > 1)
```

Рисунок 41 – Вывод комплекса ADAPT после анализа программы номер 7 в приложении Б

8. Попытка инициализации объекта, не являющегося ФД (код ошибки: LUNA26) – результат работы комплекса представлен на рисунке 42.

```
(1) error[LUNA26]: Attempt to initialize identifier f + 1.0 that is not a DF.
- Function information:
Name: init, type: extern, file "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna26.fa", line 10
- Callstack:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna26.fa", line 10, in init
    init(1, b)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna26.fa", line 9, in let
    let b = f + 1.0
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna26.fa", line 5, in foo
    foo(a)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna26.fa", line 3, in main
    sub main()
```

Рисунок 42 – Вывод комплекса ADAPT после анализа программы номер 8 в приложении Б

9. Попытка индексации объекта, не являющегося ФД (код ошибки: LUNA36) – результат работы комплекса представлен на рисунке 43.

```
(1) error[LUNA36]: Attempt to index an identifier a that is not a DF.
- Callstack:
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 7, in print
    print(a[0])
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 6, in let
    let a = f + 1
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 3, in foo
    foo(1)
  File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 2, in main
    sub main()
```

Рисунок 43 – Вывод комплекса ADAPT после анализа программы номер 9 в приложении Б

4.4.2 Временные характеристики модуля

Для проверки времени работы модуля была использована программа, моделирующая поведение протопланетного диска, исходный код которой взят из открытого репозитория (https://gitlab.ssd.sscc.ru/luna-projects/adapt/adapt/-/blob/master/ast_analyzer/tests/test_time/disk.fa). Программа содержит более 10 тысяч строчек исходного кода.

Запуск производился командой */usr/bin/time -v adapt ast_analyzer/tests/test_time/disk.fa --run degsa.* Утилита */usr/bin/time* предписывает измерить время исполнения и максимальный расход оперативной памяти, опция *--run degsa* позволяет запустить исключительно модуль DeGSA (AST для него также будет построено). Время работы программы составило 62.41 секунды, максимальный расход памяти составил 110.7 Мб.

Параметры использованной для тестирования машины:

- Процессор: Intel Core i5-12500H (4 *p*-ядра, 12 *e*-ядер, максимальная частота 4.5 ГГц).
- Оперативная память: DDR5, 16 Гб.

Использовалась операционная система Ubuntu 22.04 LTS.

ЗАКЛЮЧЕНИЕ

В ходе работы был проведён обзор существующих методов и инструментов статического анализа. Был внесён вклад в заполнение базы типов ошибок в программах на языке LuNA, а также в создание комплекса автоматизированной отладки ADAPT, для которого был разработан модуль статического анализа DeGSA, использующий граф зависимостей по данным и способный обнаруживать девять различных типов ошибок в LuNA-программах. Модуль был протестирован на ряде ошибочных программ, оценка скорости работы показала, что анализ занимает приемлемое время.

Созданный модуль может быть в дальнейшем модифицирован с целью автоматизации поиска большего спектра типов ошибок в LuNA-программах, в том числе и методами статического анализа, альтернативными текущему. Руководство оператора разработанного модуля представлено в приложении В.

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».

Царев Василий Дмитриевич _____

ФИО студента *Подпись студента*

« ____ »_____20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Гергель В. П. Теория и практика параллельных вычислений [Текст] / В. П. Гергель. – М. : БИНОМ. Лаборатория знаний, 2007. – 424 с.
2. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах // Проблемы информатики. – 2020. – №1 (46). – С. 66-74.
3. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem // Parallel Computing Technologies. LNCS 6873. – 2011. – P. 53-61.
4. Катаев Н. А. Динамический анализ зависимостей по данным в системе SAPFOR / Н. А. Катаев, А. А. Смирнов, А. Д. Жуков // Научный сервис в сети Интернет. – 2019. – № 21. – С. 400-412.
5. Белеванцев А. А. Многоуровневый статический анализ исходного кода программ для обеспечения качества программ // Программирование. – 2017. – Том 43, №6. – С. 3-26.
6. Зубов М. В., Пустыгин А. Н., Старцев Е. В.. Применение универсальных промежуточных представлений для статического анализа исходного программного кода // Доклады ТУСУР. 2013. №1 (27).
7. Зубов М. В., Пустыгин А. Н., Старцев Е. В. Численное моделирование анализа исходного кода с использованием промежуточных представлений // Вестник Астраханского государственного технического университета. Серия: Управление, вычислительная техника и информатика. – 2014. – № 4. – С. 55-66.
8. Ицыксон В. М., Глухих М. И., Зозуля А. В., Власовских А. С. Исследование средств построения моделей исходного кода программ на языках C и C++ // Информатика, телекоммуникации и управление. – 2009. – №1 (72). – С. 122-129.

9. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition) / USA. Addison-Wesley Longman Publishing Co., Inc., 2006.
10. Levine J. Flex & Bison: Text Processing Tools / O'Reilly Media, Inc., 2009. P. 9-10.
11. Касьянов В. Н., Евстигнеев В. А. Графы в программировании: обработка, визуализация и применение / В. Н. Касьянов, В. А. Евстигнеев. – СПб: БХВ-Петербург, 2003. – 1104 с.
12. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб: БХВ-Петербург, 2002. – 608 с.
13. Arora V., Bhatia R. K., Singh M. Evaluation of flow graph and dependence graphs for program representation // International Journal of Computer Applications. – 2012. – 56(14). – P. 18-23.
14. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe. Dependence graphs and compiler optimizations // Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL). 1981. P. 207-218.
15. Ferrante J., Ottenstein K. J., Warren J. D. The program dependence graph and its use in optimization // ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 9, No. 3. 1987. P. 319–349.
16. Jens K. F. Information Flow Control and Taint Analysis with Dependence Graphs // Proceedings of the Third International Workshop on Code Based Software Security Assessments (CoBaSSA). 2007.
17. Harrold M. J., Malloy B., Rothermel, G. Efficient construction of program dependence graphs // ACM SIGSOFT Software Engineering Notes, 18(3). 1993. P. 160-170.
18. Emanuelsson P., Nilsson U. A comparative study of industrial static analysis tools // Electronic notes in theoretical computer science, 217. 2008. P. 5-21.
19. Stan – Haskell STatic ANalysis tool [Электронный ресурс]. URL: https://hackage.haskell.org/package/stan (дата обращения: 16.05.2024).

20. Frances E. Allen. Control flow analysis // In Proceedings of a symposium on Compiler optimization. Association for Computing Machinery, New York, NY, USA. 1970. P. 1–19.
21. Khedker U., Sanyal A., Sathe B. Data Flow Analysis: Theory and Practice (1st edition) / CRC Press. 2009.
22. Kreaseck B., Strout M. M., Hovland P. Depth analysis of MPI programs // In Proceedings of the First Workshop on Advances in Message Passing. 2010.
23. Cppcheck – static analysis tool for C/C++ code [Электронный ресурс]. URL: https://cppcheck.sourceforge.io/ (дата обращения: 15.05.2024)
24. Palsberg J., Jay C. B. The essence of the visitor pattern // In Proceedings of the Twenty-Second Annual International Computer Software and Applications Conference. 1998. P. 9-15.
25. Weeder – application to perform whole-program dead-code analysis [Электронный ресурс]. URL: https://hackage.haskell.org/package/weeder (дата обращения: 16.05.2024).
26. Малышкин В. Э. Технология фрагментированного программирования // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. – 2012. – №46 (305).
27. Описание языка LuNA [Электронный ресурс]. URL: https://gitlab.ssd.sscc.ru/luna/luna/-/wikis/luna_lang_v01 (дата доступа: 15.05.2024).
28. Malyshkin V., Vlasenko A., Michurov M. Automated Debugging of Fragmented Programs in LuNA System // D. Balandin et al. (Eds.): 22nd International Conference, MMST 2022, Nizhny Novgorod, Russia, November 14-17. 2022. P. 266–280.
29. Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K. Efficiently computing static single assignment form and the control dependence graph // ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4). 1991. P. 451-490.

30. Скрыпников И. С. Статический анализатор кода на основе взаимодействия интервального анализа и анализа указателей / И. С. Скрыпников. — Текст : непосредственный // Молодой ученый. — 2016. — № 23 (127). — С. 19-24.
31. Hind M. Pointer analysis: haven't we solved this problem yet? // In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01). Association for Computing Machinery, New York, NY, USA. 2001. P. 54–61.

ПРИЛОЖЕНИЕ А

Отчёт модуля DeGSA об обнаруженных в программе на рисунке 10 ошибках

Команда запуска модуля:

```
adapt DeGSA/luna/diplomaEx1.fa --run degsa --no-cleanup
```

Отчёт (набор JSON-объектов):


```
{"details": {"callstack": [{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "11","name": "init"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "9","name": "let"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}],"expression":
"nonIndexableName"},"error_code": "LUNA36"}

{"details": {"callstack": [{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "11","name": "init"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "9","name": "let"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}],"cf": {"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "11","name": "init","type": "extern"},"expression":
"nonIndexableName[0]"},"error_code": "LUNA26"}

{"details": {"df": {"declared": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]],"initialized": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "5","name": "init"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
```

fa","line": "3","name": "main"}],[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "6","name": "init"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]],"name":
"multipleInitName","used": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "7","name": "print"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]]}},"error_code": "LUNA03"}

{"details": {"df": {"declared": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]],"initialized": [],"name":
"nonInitUsedName","used": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "8","name": "print"},{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]]}},"error_code": "LUNA05"}

{"details": {"df": {"declared": [[{"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "3","name": "main"}]],"initialized": [],"name":
"unusedName","used": []}},"error_code": "LUNA10"}

{"details": {"condition": "f","type": "true","where": {"file":
"/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/diplomaEx1.
fa","line": "15","name": "if"}},"error_code": "LUNA23"}

ПРИЛОЖЕНИЕ Б

Листинги ошибочных LuNA-программ, использованных для тестирования модуля DeGSA на предмет корректности работы

1. *Файл luna02.fa (LUNA02: несуществующая LuNA-подпрограмма):*

```
sub main(){
        print(1);
}
```

2. *Файл luna03.fa (LUNA03: повторная инициализация ФД):*

```
import c_init_int(int, name) as init;
sub main(){
        df a;
        init(1, a);
        let b = a {
                foo(b);
        }
}
sub foo(name f){
        init(2, f);
}
```

3. *Файл luna05.fa (LUNA05: попытка использования неинициализированного ФД):*

```
import c_init_int(int, name) as init;
import printfl(real) as print;
sub main(){
        df a;
        let b = a {
                foo(b);
        }
}
sub foo(name f){
        print(f);
}
```

4. *Файл luna10.fa (LUNA10: неиспользуемый ФД):*

```
import c_init_int(int, name) as init;
sub main(){
        df a, c;
        let b = a {
                foo(b);
        }
}
sub foo(name f){
        init(1, f);
}
```

5. *Файл luna13.fa (LUNA13: ФД с одинаковыми названиями в одной области видимости):*

```
import c_init_int(int, name) as init;
import printint(int) as print;
sub main(){
        df i, i;
        print(1);
}
```

6. *Файл luna14.fa (LUNA14: попытка использования необъявленного ФД):*

```
import c_init_int(int, name) as init;
import printfl(real) as print;
sub main(){
        df a;
        foo(a);
}
sub foo(name f){
        print(d);
}
```

7. *Файл luna23.fa (LUNA23: формула в if тождественно истинна):*

```
import printint(int) as print;
sub main(){
        let a = 1 + 2 {
                foo(a);
        }
}
sub foo(int f){
        if (f > 1){
                print(f);
        }
}
```

8. *Файл luna26.fa (LUNA26: попытка инициализации объекта, не являющегося ФД):*

```
import c_init_int(int, name) as init;
import printfl(real) as print;
sub main(){
        df a;
        foo(a);
}
sub foo(name f){
        init(1, f);
        let b = f + 1.0 {
                init(1, b);
                print(b);
        }
}
```

9. *Файл luna36.fa (LUNA36: попытка индексации объекта, не являющегося ФД):*

```
import printint(int) as print;
sub main(){
        foo(1);
}
sub foo(int f){
        let a = f + 1 {
                print(a[0]);
        }
}
```

ПРИЛОЖЕНИЕ В

Модуль статического анализа DeGSA для программ на языке “LuNA”

Руководство оператора

Листов 7

Новосибирск 2024

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению модуля статического анализа LuNA-программ DeGSA.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

Аннотация **67**

1 Назначение программы **69**

1.1 Функциональное назначение программы 69

1.2 Эксплуатационное назначение программы 69

1.3 Состав функций 69

2. Условия выполнения программы **70**

2.1 Минимальный состав аппаратных средств 70

2.2 Минимальный состав программных средств 70

2.3 Требование к персоналу 70

3. Выполнение программы **71**

3.1 Загрузка и запуск программы 71

3.2 Выполнение программы 71

3.3 Завершение работы программы 72

1 Назначение программы

1.1 Функциональное назначение программы

Разработанная программа предназначена для анализа исходного кода программ на языке LuNA в целях поиска синтаксических и семантических ошибок.

1.2 Эксплуатационное назначение программы

Программа должна использоваться разработчиками программ на языке LuNA во время отладки последних.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Автоматическое построение и применение графа зависимостей по данным для входной программы на языке LuNA для поиска ошибок в исходном коде.
- Предоставление информации об обнаруженных в программе ошибках в формате JSON.
- Предоставление информации о построенном графе зависимостей по данным.

2. Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере. Устройство, на котором запускается программа, должно иметь следующие характеристики:

- Оперативная память объемом не менее 4 Гб;
- Жёсткий диск объёмом не менее 64 Гб;
- Процессор: два или более ядра с тактовой частотой не менее 1.5 ГГц;
- Монитор;
- Клавиатура.

2.2 Минимальный состав программных средств

- Операционная система семейств Windows или Linux;
- Компилятор языка C++20 (GCC, Clang, MSVC);
- Командный интерпретатор bash.

2.3 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования интерфейса командной строки.

3. Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется вызовом команды “adapt” с указанием пути к анализируемому файлу, который должен содержать текст программы на языке LuNA.

Пользователь может указать следующие опциональные параметры:

- --run [используемый анализатор] — запустить анализатор указанного типа; варианты – “degsa” (запуск модуля DeGSA, использующего граф зависимостей по данным), “ast” (запуск модуля ast_analyzer, использующего абстрактное синтаксическое дерево) и prolog (запуск модуля prolog-analyzer, использующего логическое представление LuNA-программы).
- --no-cleanup – не очищать порождённые во время работы анализатора файлы.
- --help – вывести информацию об опциях запуска комплекса и сразу завершить работу.

3.2 Выполнение программы

После запуска комплекс ADAPT проведёт анализ указанной программы на языке LuNA и напечатает в консоль информацию обо всех найденных ошибках. Пример его вывода представлен на рисунке B.1.

```
● conngent@Conngent:~/gitlabluna/ast-ddg-analyzer$ adapt DeGSA/luna/testingCorrectness/luna36.fa --run degsa
  Found 1 errors:
  (1) error[LUNA36]: Attempt to index an identifier a that is not a DF.
  - Callstack:
    File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 7, in print
      print(a[0])
    File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 6, in let
      let a = f + 1
    File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 3, in foo
      foo(1)
    File "/home/conngent/gitlabluna/ast-ddg-analyzer/DeGSA/luna/testingCorrectness/luna36.fa", line 2, in main
      sub main()
```

Рисунок В.1 — Вывод комплекса ADAPT

3.3 Завершение работы программы

Программа завершается автоматически после вывода информации об обнаруженных ошибках. В случае необходимости программу можно прервать в любой момент, воспользовавшись комбинацией клавиш Ctrl + C.