

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 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Мичурова Михаила Антоновича

Тема работы:

**РАЗРАБОТКА СТАТИЧЕСКОГО АНАЛИЗАТОРА LUNA-ПРОГРАММ С
ИСПОЛЬЗОВАНИЕМ ЯЗЫКА ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ PROLOG**

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

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

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ
Киреев С. Е. /.....
(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: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В.Э.

(фамилия, И., О.)

.....

(подпись)

«18» января 2024 г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студенту Мичурову Михаилу Антоновичу, группы 22221

(фамилия, имя, отчество, номер группы)

Тема Разработка статического анализатора LuNA-программ с использованием языка логического программирования Prolog

(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 07.11.2022 № 0330

скорректирована распоряжением проректора по учебной работе от 18.01.2024 № 0009

Срок сдачи студентом готовой работы 20 мая 2024 г.

Исходные данные (или цель работы) Разработка алгоритмов статического анализа LuNA-программ, использующих их логическое представление для обнаружения семантических ошибок, реализация разработанных алгоритмов в программном средстве, а также интеграция разработанного средства в комплекс ADAPT

Структурные части работы: Обзор подходов к отладке параллельных программ, проектирование статического анализатора LuNA-программ, разработка статического анализатора, тестирование.

Руководитель ВКР

доц. каф. ПВ ФИТ,

к.т.н.

Власенко А. Ю./.....

(ФИ О) / (подпись)

«18» января 2024 г.

Задание принял к исполнению

Мичуров М. А./.....

(ФИО студента) / (подпись)

«18» января 2024 г.

Соруководитель ВКР

ст. преп. каф. ПВ ФИТ,

Киреев С. Е./.....

(ФИ О) / (подпись)

«18» января 2024 г.

СОДЕРЖАНИЕ

Определения, обозначения и сокращения	5
Введение	6
Глава 1. Обзор подходов к отладке параллельных программ	9
1.1 Ошибки в параллельных программах	9
1.2 Проблемы отладки параллельных программ	11
1.3 Подходы к отладке	12
1.4 Оценка подходов к отладке	19
1.5 Применение языков логического программирования в анализе программ	20
Глава 2. Проектирование статического анализатора LuNA-программ	24
2.1 Технология фрагментированного программирования	24
2.2 Система LuNA	25
2.3 Ошибки в LuNA-программах	27
2.4 База типов ошибок в LuNA-программах	32
2.5 Формат представления информации об ошибках	35
2.6 Требования	37
2.7 Характеристики предлагаемого решения	39
Глава 3. Разработка статического анализатора prolog-analyzer	40
3.1 Анализ на основе графа зависимостей по данным	40
3.2 Анализ высокоуровневых семантических конструкций языка	46
3.3 Трансляция AST программы и выделение анализируемых абстракций в процессе анализа	53
3.4 Интеграция prolog-analyzer в ADAPT	66
Глава 4. Тестирование	69
4.1 Проверка функциональных возможностей	69
4.2 Оценка времени анализа программ	76

Заключение	81
Список использованных источников и литературы	83
Приложение А	91
Приложение Б	93
Приложение В	101

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ТФП — технология фрагментированного программирования.

Фрагмент кода (ФК) — подпрограмма на языке LuNA или C++.

Фрагмент данных (ФД) — объект данных в ТФП.

Фрагмент вычислений (ФВ) — запущенный экземпляр ФК, содержащий описание входных/выходных фрагментов данных.

Фрагментированный алгоритм (ФА) — представление алгоритма в виде не более чем счетного множества фрагментов вычислений.

ВВЕДЕНИЕ

Для решения задач численного моделирования как правило используются параллельные программы, запускаемые на суперкомпьютерах. Создание эффективных реализаций численных алгоритмов является сложной задачей, требующей компетенций в решении различных проблем, типичных для распределенных параллельных вычислений [1]. Помимо реализации непосредственно параллельной программы, что уже требует высокой квалификации, программисту необходимо учитывать нефункциональные требования, такие как эффективность использования процессорного времени и памяти, настройку на доступные ресурсы, отказоустойчивость и многие другие. Решение этих проблем требует больших временных затрат, а также знаний и навыков, которыми пользователи суперкомпьютеров могут не обладать в достаточной степени.. Одним из выходов в таком случае является автоматическая генерация параллельных программ.

Системы автоматического конструирования параллельных программ обычно принимают на вход высокоуровневую спецификацию вычислительного алгоритма. Такая спецификация должна позволять настраивать алгоритм на доступное аппаратное обеспечение и данные, а также должна состоять из вычислимых независимо частей, которые можно было бы динамически отображать на доступные вычислительные ресурсы.

Одной из таких систем является LuNA (Language for Numeric Algorithms) [1-3]. LuNA — разрабатываемая в ИВМиМГ СО РАН система автоматического конструирования параллельных программ для суперкомпьютеров, поддерживающая технологию фрагментированного программирования. Программа для системы описывается в виде фрагментированного алгоритма на одноименном языке. Написание таких программ не требует навыков реализации параллельных приложений, что существенно упрощает работу пользователей, решающих прикладные задачи. В то же время средства поддержки программирования для системы LuNA слабо развиты, что усложняет работу над проектами. В частности, средства отладки традиционных параллельных

приложений (TotalView, MPI-CHECK, Intel Trace Analyzer and Collector и другие) неприменимы для отладки LuNA-программ, а разработанные специализированные инструменты (luna_trace [4, 5]) не решают проблему отладки полностью.

Одним из подходов к отладке программ является статический анализ. Средства статического анализа позволяют идентифицировать проблемы в кодовой базе без запуска программ, что упрощает их применение, а ранняя идентификация ошибок повышает производительность разработки. Таким образом, наличие средств статического анализа для системы LuNA упростило бы ее использование для пользователей.

Целью данной работы является разработка алгоритмов статического анализа LuNA-программ, использующих их логическое представление для обнаружения семантических ошибок, реализация разработанных алгоритмов в программном средстве, а также интеграция разработанного средства в комплекс ADAPT.

Для достижения данной цели были поставлены следующие **задачи**:

1. Выполнить анализ существующих средств отладки параллельных программ, а также средств отладки, использующих логическое представление анализируемой программы.
2. Систематизировать типы ошибок в LuNA-программах, выделить трудные для обнаружения “вручную” семантические ошибки, принять участие в составлении базы ошибок в LuNA-программах.
3. Сформулировать требования к разрабатываемому средству статического анализа.
4. Разработать структуры данных и алгоритмы обнаружения выбранных семантических ошибок.
5. Интегрировать разработанное средство анализа в комплекс автоматизированной отладки LuNA-программ ADAPT.
6. Выполнить тестирование разработанного средства анализа автономно и как части комплекса автоматизированной отладки.

Научная новизна работы состоит в том, что был предложен подход к обнаружению семантических ошибок во фрагментированных программах по их логическому представлению.

Практическая ценность обусловлена облегчением процесса отладки LuNA-программ и повышением эффективности разработки.

Работа изложена в четырех главах. Первая глава содержит обзор существующих средств отладки параллельных программ, и отдельно — средств отладки, использующих логическое представление анализируемых программ. Вторая глава посвящена анализу предметной области и формулированию требований к разрабатываемому программному средству. В третьей главе описываются рассмотренные в процессе разработки алгоритмы и структурам данных, реализация программного средства статического анализа и его интеграции в ADAPT. В четвертой главе речь идет о тестировании разработанного средства.

Глава 1. Обзор подходов к отладке параллельных программ

Был проведен обзор существующих подходов к отладке параллельных программ, а также программных средств, в которых эти подходы реализованы. Также был проведен обзор средств анализа исходного кода, применяющих логические языки программирования, такие как Prolog и Datalog.

1.1 Ошибки в параллельных программах

Традиционно ошибки разделяются на **синтаксические** и **семантические** [6, 7]. Синтаксические ошибки, как правило, не вызывают особого интереса, поскольку их обнаружение полностью автоматизировано в компиляторах и средах разработки. Пользователь получает исчерпывающую информацию о таких ошибках, что позволяет исправлять их быстро и без дополнительных затрат. С другой стороны, семантические ошибки сложны в обнаружении, и процесс их выявления и исправления может быть очень трудозатратным. Далее под ошибками будут пониматься именно семантические ошибки как представляющие наибольший интерес.

С одной стороны, при написании параллельных программ пользователь может допускать специфические ошибки, не свойственные последовательным программам. Такие ошибки могут быть связаны как с некорректной реализацией взаимодействия параллельно взаимодействующих процессов или потоков (гонки данных, мертвые блокировки), так и с используемой технологией параллельного программирования. С другой стороны, свойственные последовательным программам ошибки могут проявляться совершенно новым образом.

Таким образом, можно классифицировать ошибки в параллельных программах следующим образом:

1. Ошибки, свойственные последовательным программам, но имеющие иные проявления в параллельных программах.
2. Ошибки, связанные с некорректным использованием выбранной технологии параллельного программирования.
3. Ошибки взаимодействия параллельно выполняющихся процессов или потоков [8]:

- a. Мертвые блокировки (“дедлоки”).
- b. Гонки данных.
- c. “Ливлоки”.
- d. и др.

Данные классы условны и могут пересекаться. Более того, ошибки одних классов могут быть причинами ошибок других классов. Часто некорректное использование технологии параллельного программирования может приводить к мертвым блокировкам и гонкам данных. Примерами таких ситуаций может быть выполнение над разделяемыми данными операций, не являющихся атомарными, вне критических секций при использовании OpenMP [9, 10], или вызов блокирующей операции коммуникации типа точка-точка лишь в одном из взаимодействующих процессов при использовании MPI [11].

Тем не менее, такое разделение все же имеет смысл. При проектировании средства отладки необходимо выбрать, ошибки какого класса будут обнаруживаться. Средства отладки должны давать информацию, помогающую исправить ошибку как можно быстрее. Поэтому в примерах некорректного использования OpenMP и MPI, приведенных ранее, информация о том, что в директиве `omp for` отсутствует выражение `reduction` для некоторой разделяемой переменной, или информация о том, что некоторой операции `MPI_Send` не соответствует `MPI_Recv`, окажется более полезной, нежели сообщение о наличии гонки данных или мертвой блокировки.

Отдельно стоит остановиться на ошибках, свойственных и последовательным программам. Проблема отладки таких программ хорошо изучена, и статические анализаторы уже стали частью компиляторов [12] и сред разработки. Разумеется, для выявления ошибок, не связанных с параллелизмом, могут быть использованы существующие средства отладки. Однако, некоторые ошибки, типичные для последовательных программ, в параллельных программах могут проявляться иначе. Например, распространенная ошибка выхода за границу массива в последовательной программе может вызвать исключение (C#, Java) или аварийное завершение процесса из-за попытки

обращения к памяти, к которой у процесса нет доступа (C, C++), в то время как попытка использования несуществующих элементов распределенного массива (что может быть синтаксически идентично выходу за границу) в зависимости от реализации может привести к зависанию программ [13, 14]. В разделе 2.4 будет показано, что при обнаружении специфических для ТФП [1] ошибок в программах на языке LuNA требуется обнаруживать ситуации, очень напоминающие ошибки, свойственные последовательным программам. Также будет показано, что популярные статические анализаторы (PVS-Studio [15], Clang-Tidy [16]) не обладают достаточной “чувствительностью” для обнаружения таких ошибок в нетривиальных случаях, аналогичных типичным фрагментам LuNA-программ.

1.2 Проблемы отладки параллельных программ

Отладка параллельных программ — трудная задача, осложняемая различными факторами [17]. Главной проблемой является то, что для обнаружения ошибок часто требуется анализировать поведение и взаимодействие множества параллельно выполняющихся потоков или процессов. Если подразумевается мониторинг поведения работающей программы, отладка осложняется недетерминированностью выполнения (или нестабильностью воспроизведения) и “эффектом наблюдателя”. Эффект наблюдателя заключается в том, что сам факт наблюдения за состоянием программы может вносить изменения в ее поведение. Так, например, гонки данных будет очень трудно обнаружить при выполнении кода по шагам, поскольку действия потоков будут разделены во времени [10]. Нестабильность воспроизведения заключается в том, что различные запуски программы с одними и теми же данными могут давать разные результаты. Еще одной проблемой является то, что анализ профиля, собранного во время выполнения, может занимать очень большое (в сотни или тысячи раз дольше) время по сравнению со временем работы программы [10].

1.3 Подходы к отладке

В [18] подходы к отладке параллельных программ разделяются на *автоматический контроль корректности* выполнения программы, *сравнительную отладку* и *диалоговую отладку*, причем автоматический контроль корректности включает в себя как динамический анализ в процессе реального выполнения программы, так и анализ трасс, собранных при выполнении. В работе [11] также отдельно выделяется *верификация модели программы*. В [19] подходы к анализу программ глобально разделяются на *динамический анализ* и *статический анализ*, то есть такой, при котором анализируемая программа не запускается. Далее, в [20] выделяется еще несколько подходов к статическому анализу, таких как сопоставление с шаблоном, анализ потока управления и другие.

Для целей данной работы будет удобно классифицировать подходы к отладке следующим образом:

1. Методы, требующие запуска программы:
 - a. Диалоговая отладка — TotalView [21], PD [22], Distributed Debugging Tool [23].
 - b. Сравнительная отладка — работы [24-26].
 - c. Автоматизированный контроль корректности:
 - i. Во время выполнения программы — Intel Trace Analyzer and Collector [27], MARMOT [28] и MPI-CHECK [29].
 - ii. Анализ трассы после завершения программы — Intel Trace Analyzer and Collector, DVM [18], luna_trace [4, 5].
2. Методы, не требующие запуска анализируемой программы:
 - a. Статический анализ — Cppcheck [30], Clang Static Analyzer [31], Clang-Tidy [16] и PVS-Studio [15].
 - b. Верификация (формальной) модели программы — Spin [32], MPI-Spin [33].

Существуют некоторые разногласия о том, является ли верификация модели программы одной из разновидностей статического анализа, либо

отдельным подходом к отладке. Так, например, в [19] и [34] верификацию формальных моделей программ относят к статическому анализу, а в [6] статический анализ и вовсе не упомянут как отдельный подход, в то время как верификация модели программы рассматривается наряду с другими методами. В данной работе верификация модели не относится к статическому анализу. Статические анализаторы, как правило, не требуют от пользователя дополнительных действий помимо запуска самого анализатора, в то время как для верификации модели программы эта модель должна быть описана отдельно от анализируемой программы. Разумеется, автоматическая генерация формальных моделей программ стирает данное разделение, однако такие системы [35, 36] остаются за рамками рассмотрения этой работы.

1.3.1 Диалоговая отладка

Средства диалоговой отладки параллельных программ предоставляют пользователям как традиционные для диалоговых отладчиков возможности (приостановка и возобновление работы, просмотр значений переменных и др.), так и ориентированные на работу с параллельными программами функции, такие как управление группами процессов и потоков.

Диалоговая отладка — хорошее средство функциональной отладки программ. Типичными представителями этого направления являются, например, TotalView, PD и Distributed Debugging Tool. Однако, у диалоговой отладки есть ряд недостатков. Во-первых, требуется непосредственное участие программиста. Отслеживать поведение большого количества процессов вручную очень сложно, а суперкомпьютерные центры могут не предоставлять возможности интерактивной работы с запущенными приложениями. Во-вторых, при помощи диалогового отладчика трудно или вовсе невозможно обнаружить многие типы ошибок, связанных с логикой взаимодействия параллельно выполняющихся процессов.

1.3.2 Сравнительная отладка

Сравнительная отладка подразумевает сравнение промежуточных результатов работы отлаживаемой программы с некоторой эталонной

программой. Такой эталонной программой может быть, например, последовательная программа, в корректности которой нет сомнений [24]. Результаты сравнения и информация о найденных расхождениях могут быть предоставлены пользователю.

В [25] процесс сравнительной отладки выполняется в два этапа: сначала параллельная программа, использующая OpenMP, компилируется с модулем трассировки и запускается в режиме сбора трассы; затем, та же программа запускается на одном потоке (или перекомпилируется без OpenMP и запускается) и выполняет проверку трассы. Интерактивный визуализатор может быть использован для просмотра различий.

Несомненным плюсом такого метода отладки является то, что расхождения с эталонной программой могут быть выявлены в автоматическом режиме. Недостатки же включают в себя необходимость существования эталонной программы, которая также должна быть отлажена, и тот факт, что рассматривается только один конкретный запуск программы. Для больших программ актуальна проблема бесконтрольного роста трассы, которая частично решается путем сокращения числа контрольных точек [26]. Такой компромисс, однако, может привести к тому, что некоторые расхождения останутся незамеченными.

1.3.3 Автоматизированный контроль корректности

Автоматизированный контроль корректности — автоматическое выявление ошибочного поведения параллельной программы. Данный подход использует информацию, собранную во время выполнения программы, для выявления ошибок.

Можно выделить два подхода к автоматизированному контролю корректности:

1. Анализ во время выполнения программы;
2. Анализ собранной трассы (“посмертный анализ”).

Первый подход реализован в Intel Trace Analyzer and Collector, MARMOT и MPI-CHECK. Такой подход удобен тем, что информация об ошибках собирается за один запуск программы.

Второй подход реализован в Distributed Virtual Machine (DVM) и luna_trace. Этот подход подразумевает сбор трассы во время выполнения программы и последующий анализ отдельным программным средством. Схема работы приведенных средств отладки примерно следующая: во время работы программы трассировщик накапливает информацию о существенных с точки зрения анализа событиях (например, в DVM — это обращения к функциям и их параметры, в luna_trace — информация о создании объектов данных и вычислений); после завершения работы программы для каждого MPI-процесса создается отдельный файл трассы; наконец, программа-анализатор агрегирует трассы, производит анализ и выводит пользователю информацию о найденных ошибках, включая такие полезные сведения, как стеки вызовов функций, имена и строки файлов исходного кода.

Плюсы применения автоматизированного контроля корректности очевидны: от программиста требуется минимум действий (как правило, достаточно изменить флаги команды компиляции), а весь анализ происходит автоматически. Но то же можно сказать и о минусах: рассматривается единственный запуск программы, а трасса, как и в случае со сравнительной отладкой, может расти бесконтрольно, что приводит как к увеличению времени анализа, так и к чрезмерному использованию дискового пространства.

1.3.4 Статический анализ

Статический анализ подразумевает разбор и анализ исходного кода программы без ее запуска. Данный подход не подвержен эффекту наблюдателя и потенциально обеспечивает полное покрытие кода, не зависит от конкретных компиляторов и реализаций используемых библиотек параллельного программирования, но может давать как ложноположительные результаты (отмечать корректные фрагменты кода, идентифицированные как ошибочные), так и ложноотрицательные (пропускать ошибки). Тем не менее, статический

анализ может выполняться многократно в процессе разработки, что позволяет обнаруживать и исправлять ошибки по мере их возникновения, а проблема обнаружения не всех ошибок может быть нивелирована параллельным использованием отладчиков, реализующих различные методы.

Хотя статические анализаторы в основном обнаруживают ошибки, свойственные последовательным программам, некоторые способны обнаруживать ошибки при использовании библиотек и средств параллельного программирования, таких как OpenMP (Cppcheck [37]), POSIX Threads (Clang Static Analyzer [38] и Clang-Tidy [39]), а также модуля threads стандартной библиотеки C++ (PVS-Studio [40]). К сожалению, поддержка параллельного программирования в статических анализаторах как правило слабая.

Существует ряд подходов к статическому анализу — сопоставление с шаблоном, анализ потока данных, анализ потока управления, символический анализ и другие. Однако, глобально можно разделить средства статического анализа на две группы по тому, как организован процесс поиска ошибок (рисунок 1).



Рисунок 1 — Варианты анализа параллельной программы

В отличие от [10], под моделью программы понимается не только сеть Петри, а вообще любое представление программы, строящееся по абстрактному синтаксическому дереву и используемое при анализе.

Вариант слева на рисунке 1 является наиболее распространенным, и используется, например, в PVS-Studio. При анализе абстрактного

синтаксического дерева может быть использовано сопоставление с шаблоном. Такой подход эффективен при работе с большими кодовыми базами, но анализ оказывается “поверхностным”, из-за чего точность может страдать. Так, например, для программы из листинга на рисунке 2 PVS-Studio сигнализирует о выходе за границу массива (“warning: V557 Array overrun is possible. The value of 'i' index could reach 42”).

```
01: size_t get_size() {
02:     return 42;
03: }
04:
05: int main() {
06:     size_t size = get_size();
07:     int array[size];
08:     for (size_t i = 0; i < size; i++) {
09:         array[i] = (int) i;
10:     }
11:     for (size_t i = 0; i < size + 1; i++) {
12:         std::cout << array[i] << " ";
13:     }
14:     return 0;
15: }
```

Рисунок 2 — Фрагмент программы на C++ с выходом за границу массива

Тем не менее, достаточно вынести печать элементов массива в отдельную функцию, либо сделать размер массива переменным (даже если он может принимать лишь одно из нескольких значений), и PVS-Studio не будет сигнализировать об ошибке.

Другая типичная ошибка — перекрывающиеся условия внутри логического выражения. В листинге на рисунке 3 приведен фрагмент программы, содержащий несколько таких выражений.

```

01: int a;
02: std::cin >> a;
03: int array[a];
04:
05: if (a > 0 or a <= 0) { // 1
06:     std::cout << 1;
07: }
08:
09: if (a + a*0 > 0 or a <= 0) { // 2
10:     std::cout << 2;
11: }
12:
13: if (a > 0 or a == 0 or a < 0) { // 3
14:     std::cout << 3;
15: }
16:
17: if (array[1] > 0 or array[1] <= 0) { // 4
18:     std::cout << 4;
19: }
20:
21: if (array[1] + 1 > 0 or array[1] + 2 - 1 <= 0) { // 5
22:     std::cout << 5;
23: }

```

Рисунок 3 — Фрагмент программы на C++ с перекрывающимися условиями

PVS-Studio сообщает об ошибке только для выражений, отмеченных комментариями “1” (строка 5) и “4” (строка 17), хотя все выражения являются ошибочными.

Вариант справа на рисунке 1 дает возможность использовать более сложные методы анализа, работающие с некоторым абстрактным представлением (моделью) программы. Так, например, для программы может быть построен граф зависимостей по данным, что позволит обнаруживать мертвые блокировки, то есть циклы в этом графе [41].

Статический анализ удобен тем, что работает напрямую с исходным кодом программы. При этом обеспечивается полное покрытие кода, а саму программу не нужно запускать, что экономит время и упрощает интеграцию статического анализа в процесс разработки. Несмотря на привлекательность концепции, для параллельных программ статические анализаторы слабо развиты.

1.3.5 Верификация модели программы

Верификация модели программы подразумевает проверку некоторых свойств программы, описанных на языке формальной логики, на модели

программы. Параллельная программа представляется моделью вычислений, то есть системой переходов с конечным числом состояний, а верификация выполняется путем анализа выполнимости требуемых свойств на всем пространстве состояний модели программы. Этот процесс может быть автоматизирован [42].

Типичным представителем данного направления является Spin. Данный инструмент поддерживает высокоуровневый язык описания систем PROMELA (PROcess MEta LAnguage). Spin применялся для обнаружения ошибок в распределенных системах, таких как операционные системы, протоколы передачи данных, системы управления космическими аппаратами, и многих других. Spin проверяет логическую согласованность спецификации и сообщает о мертвых блокировках, гонках данных, различных типах неполноты и необоснованных предположениях об относительных скоростях процессов [32]. Для Spin также существует расширение MPI-Spin [33], упрощающее моделирование MPI-программ.

Верификация позволяет гарантировать необходимые свойства программы, что является главным плюсом данного подхода. В то же время программисту необходимо составить на некотором псевдоязыке модель программы, которая должна учитывать важные с точки зрения проверяемых свойств нюансы и абстрагировать несущественные детали, что в общем случае нетривиальная задача, требующая отдельного набора навыков.

1.4 Оценка подходов к отладке

Существует ряд подходов к отладке параллельных программ, каждый из которых имеет свои преимущества и недостатки. Подходы отличаются чувствительностью к ошибкам, а также требованиями к компетенциям и времени программиста.

С точки зрения программиста, конечно, очень важной является простота использования средства отладки, особенно если он решает прикладную задачу и не является специалистом в области параллельного программирования. С этой

точки зрения наиболее привлекательными подходами представляются автоматизированный контроль корректности и статический анализ.

К сожалению, для системы LuNA существующие средства отладки параллельных программ не подходят. Уровень абстракции, предоставляемый системой (фрагменты данных и вычислений), делает информацию об ошибках, связанных с нижележащей библиотекой параллельного программирования, практически бесполезной. Специально для LuNA было разработано средство анализа трассы `luna_trace`, однако оно не решает проблему отладки полностью как в силу ограниченного числа обнаруживаемых ошибок, так и в силу других ограничений, присущих системам автоматизированного контроля корректности.

Актуальной является разработка средства статического анализа LuNA-программ, учитывающего специфику семантики языка. Такой статический анализатор в дальнейшем может быть интегрирован в единый комплекс отладки, включающий набор средств, реализующих различные дополняющие друг друга подходы.

1.5 Применение языков логического программирования в анализе программ

В данном разделе отдельное внимание уделяется работам и программным средствам, применяющим языки логического программирования при анализе исходного кода.

Важной концепцией здесь является понятие логического метапрограммирования [43-48]. В [47] языком логического метапрограммирования называется язык логического программирования, использующийся как метаязык. Метасистема — система, предметной областью которой является другая вычислительная система (называемая базовой); метапрограмма — программа, описывающая метасистему для некоторой вычислительной системы; метаязык — язык программирования, предназначенный для описания метапрограмм [47]. Логическое метапрограммирование, таким образом, это применение языков логического программирования для анализа и трансформации логического представления

программы. При этом программа на базовом языке, как правило, представляется в виде набора фактов на метаязыке, а анализ и трансформации выполняются с помощью предикатов (правил), выполняющих запросы к этому представлению.

Подход логического метапрограммирования используют многие системы запросов к коду (code querying systems), такие как JQuery [49], JTL [50], SOUL [51], Codequest [52], Astlog [53] и JTransformer [54]. JQuery, JTransformer и CodeQuest генерируют представление программ на языке Java в виде набора фактов. В JTransformer используется платформонезависимый и бесплатный SWI-Prolog [55] для хранения, анализа и трансформаций наборов фактов. JQuery использует TyRuBa (Type Rule Base) [56]. CodeQuest хранит факты в реляционной базе данных и реализует свой собственный механизм запросов, способный вычислять запросы на Datalog относительно содержимого базы данных. В работе [54] приводится сравнение ряда систем запросов к коду на Java, опирающихся на логическое метапрограммирование (рассматриваются JTransformer, JQuery и CodeQuest). В частности, рассматривается задача обнаружения в кодовой базе шаблона проектирования “наблюдатель”, и приводится пример правила для его поиска, опирающегося на семантику методов (например, метод *notify()* должен в цикле вызывать методы *update()* у зарегистрированных наблюдателей). Astlog — вариант Prolog, специально разработанный для работы с абстрактными синтаксическими деревьями программ на C и C++. К особенностям Astlog относится то, что программа не транслируется в набор фактов явно, а также наличие так называемого “текущего объекта”, относительно которого вычисляются предикаты.

Логическое метапрограммирование также применяется при анализе исходного кода. В работе [57] Prolog используется для верификации сформулированных в CTL (Computational Tree Logic) свойств файловой системы флеш-памяти, реализованной на C++. В частности, рассматриваются ограничения, связанные с использованием примитивов синхронизации (например, семафоров). В данной работе абстрактное синтаксическое дерево представляется в виде набора фактов, по которому строятся

недетерминированные деревья последовательностей выполнения, на которых проверяются сформулированные с помощью предметно-ориентированного языка свойства. Авторы отмечают плюсы такого подхода: Prolog позволяет описывать верифицируемые свойства в компактном и декларативном виде; существующие в Prolog подходы и библиотеки для работы с графами существенно облегчают работу с древовидными структурами; такие особенности Prolog, как недетерминизм, поиск с возвратом (backtracking) и сопоставление с образцом (pattern matching) позволяют легко рассматривать различные пути исполнения программы.

В [58] Prolog используется как язык запросов, в то время как для представления программы используется SCML. Авторы отмечают, что логика предикатов хорошо подходит для представления ограничений.

Еще одним программным средством, опирающемся на Prolog при выполнении анализа, является SQUASH (SQL Query Analyzer and Schema Enhancer) [59]. На SWI-Prolog и Perl реализованы алгоритмы визуализации, анализа, изменения и оптимизации схем баз данных. Также SQUASH способен обнаруживать семантические ошибки в запросах, например, операции join без внешних ключей, и дефекты дизайна схемы базы данных, такие как изолированные части схемы, циклические зависимости внешних ключей, неиспользуемые столбцы/таблицы, и многие другие.

Datalog также используется при анализе исходного кода, например, в [60] и JavaDL [61].

JavaDL — это основанный на Datalog декларативный язык спецификаций для описания шаблонов дефектов в коде на Java. Существует множество средств анализа исходного кода, автоматически обнаруживающих дефекты, подходящие под обобщенные шаблоны. Такие средства, как правило, включают большое количество шаблонов, а также позволяют разработчикам добавлять новые шаблоны, специфичные для их проектов. Главная проблема таких программных средств, по словам авторов, заключается в том, что детекторы ошибок описываются в виде императивных спецификаций, содержащий избыточные

детали не только того, *что* нужно найти, но и того, *как* это сделать. Это усложняет поддержку и расширение функциональности, а также мешает программному средству выбирать способ обнаружения дефектов, например, сделать процесс постепенным (incremental). JavaDL позволяет использовать при описании детекторов ошибок как синтаксические шаблоны, так и логические правила на Datalog. По сравнению с существующими средствами поиска дефектов в коде, описания шаблонов на JavaDL оказываются существенно меньше, показывая при этом сопоставимую эффективность. Авторы утверждают, что такой подход демонстрирует потенциал так называемых clear-box статических анализаторов, ограничивающих язык спецификации шаблонов и дающих программному средству большую свободу в применении этих шаблонов.

Таким образом, логическое метапрограммирование является перспективным подходом к реализации программных средств статического анализа кода. Использование языков логического программирования позволяет описывать правила поиска ошибок или иных дефектов кода в декларативном стиле, абстрагируясь от самого процесса поиска. Это упрощает поддержку, а также дает возможность с легкостью добавлять новые правила и расширять функциональность существующих правил. Сами возможности языков логического программирования, такие как поиск с возвратом, сопоставление с шаблоном и недетерминизм, делают удобным их применение для работы с древовидными представлениями программ и рассмотрения различных путей исполнения. Еще одним преимуществом языков логического программирования является поддержка программирования в ограничениях, широко применяющихся в данной работе, что позволяет реализовать символический анализ, не опираясь на внешние зависимости.

Глава 2. Проектирование статического анализатора LuNA-программ

Одной из главных проблем использования суперкомпьютеров для численного моделирования является разработка эффективной параллельной программы. Одним из решений данной проблемы является автоматическое конструирование параллельных программ с нужными нефункциональными свойствами по некоторой высокоуровневой спецификации вычислительного алгоритма.

2.1 Технология фрагментированного программирования

Технология фрагментированного программирования (ТФП) разрабатывается в ИВМиМГ СО РАН [1]. Основу подхода составляет представление алгоритма в виде не более чем счетного множества *фрагментов вычислений* (ФВ) — вычислительных задач, представляемых триплетом (*in*, *op*, *out*), где *in* и *out* — конечные множества входных и выходных *фрагментов данных* (ФД), а *op* — некоторая операция, вычисляющая значения всех ФД *out* из значений ФД *in*. Это представление называется фрагментированным алгоритмом (ФА). На рисунке 4 приведено схематическое представление ФВ, принимающего *n* входных ФД и вырабатывающего значения *k* выходных ФД.

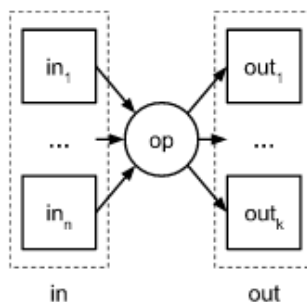


Рисунок 4 — Схематическое представление ФВ

ФД — объекты единственного присваивания. Система может дублировать ФД, отправлять их по сети или сохранять на диск, что накладывает некоторые ограничения на данные, которые могут быть представлены таким образом.

Операция *op* — последовательная процедура без побочных эффектов, для одних и тех же значений входных ФД вычисляющая одни и те же значения

выходных ФД. Выполнение ФВ — применение *op* к ФД *in* и инициализация таким образом значений ФД *out*. Процесс вычисления ФА состоит в выполнении всех ФВ по мере готовности (инициализации) их входных ФД.

Описанные выше свойства ФД и ФВ позволяют автоматически распределять и перераспределять их по узлам суперкомпьютера для обеспечения динамической балансировки нагрузки, выбирать порядок выполнения ФВ согласно информационным зависимостям, дублировать ФД на узлах для повышения доступности данных и отказоустойчивости, а также иными методами обеспечивать выполнение требуемых нефункциональных свойств.

ТФП также позволяет использовать *рекомендации* для управления ходом исполнения программы. Рекомендации позволяют задавать дополнительную информацию об алгоритме и его отображении на ресурсы суперкомпьютера во времени. Рекомендации влияют на нефункциональные свойства исполнения ФА, не изменяя при этом вычисленных значений, что позволяет использовать их для повышения эффективности.

ТФП позволяет описывать прикладной алгоритм в виде высокоуровневой спецификации, абстрагируясь от деталей реализации параллельного взаимодействия.

2.2 Система LuNA

LuNA (Language for Numerical Algorithms) — экспериментальная система автоматического конструирования параллельных программ для суперкомпьютеров, поддерживающая ТФП [1]. Она предоставляет язык LuNA, позволяющий описывать ФА как множество ФД и ФВ. В листинге на рисунке 5 приведен пример программы на языке LuNA, а на рисунке 6 — возможный вывод данной программы.

```

01: C++ sub assign_int(name x, int val) ${{ x = val; $}}
02:
03: C++ sub print_string(string s) ${{ std::printf("%s\n", s);
$}}
04:
05: sub assign_squares(name a, int n) {
06:     for j = 1..n {
07:         assign_int(a[j], j * j);
08:     }
09: }
10:
11: sub main() {
12:     df N, x;
13:     for i = 1..N {
14:         print_string("" + i);
15:         print_string("x[" + i + "] = " + x[i]) @ {
16:             delete x[i];
17:         };
18:     }
19:     assign_squares(x, N);
20:     assign_int(N, 4);
21: }

```

Рисунок 5 — Пример LuNA-программы

```

1
2
x[3] = 9
4
3
x[2] = 4
x[1] = 1
x[4] = 16

```

Рисунок 6 — Возможный вывод LuNA-программы на рисунке 5

Данный пример демонстрирует одно из главных свойств системы LuNA: во время выполнения программы все операторы языка порождают ФВ, которые затем выполняются по мере удовлетворения информационных зависимостей, то есть фрагментированная структура программы во время выполнения сохраняется. Например, на рисунке 6 можно увидеть, что порядок вывода сообщений не соответствует порядку “итераций” оператора *for* и порядку операторов внутри тела *for*.

Для описания ФА пользователям предоставлен высокоуровневый язык, имеющий Си-подобный синтаксис. Язык LuNA не требует явного задания порядка вычислений, типов вычисляемых значений (строка 12), или отображения вычислительных задач на ресурсы системы, на которой программа

будет выполняться, предоставляя в то же время возможность через рекомендации управлять нефункциональными свойствами генерируемой параллельной программы (строка 16). Язык LuNA позволяет разделять описание прикладного алгоритма на подпрограммы, называемые фрагментами кода (ФК). Фрагменты кода могут быть атомарными, то есть последовательными процедурами, реализованными на C++ (например, `assign_int` и `print_string`), и *структурированными* (`assign_squares` и `main`), то есть реализованными на языке LuNA. Атомарные ФК соответствуют операциям в терминах ТФП, поскольку только они имеют выходные параметры, которые они обязаны инициализировать. Для ФВ, порожденных оператором применения структурированного ФК, операция порождает дочерние ФВ в окружении, заданном переданными параметрами. Задействованный в примере на рисунке 5 оператор примитивно-рекурсивного перечисления *for* сначала вычисляет граничные значения для переменной-счетчика, после чего порождает копии ФВ, определенных в его теле, для каждого целочисленного значения переменной-счетчика из заданного граничными значениями диапазона. В языке LuNA присутствуют и другие операторы, семантика которых будет раскрыта по мере необходимости в следующих разделах.

Эксперименты показывают, что производительность параллельных программ, конструируемых системой LuNA, обычно ниже производительности MPI-программ, реализующих те же алгоритмы. В то же время, написание программ на языке LuNA существенно проще разработки параллельных программ с использованием традиционных библиотек и подходов, а использование рекомендаций и других методов оптимизации позволяет добиться требуемых нефункциональных свойств [1-3].

2.3 Ошибки в LuNA-программах

Несмотря на то, что язык LuNA довольно простой и позволяет полностью абстрагироваться от реализации параллельного взаимодействия, пользователь все же может допустить целый ряд ошибок, как синтаксических, так и семантических, включая ошибки, свойственные параллельным программам.

Проблема синтаксических ошибок в LuNA-программах является актуальной, поскольку компилятор не всегда способен дать информативное сообщение об ошибке, и эта проблема рассматривается в рамках проектов AST-analyzer и DeGSA. Данная же работа фокусируется на семантических ошибках.

В параллельных программах, автоматически конструируемых системой LuNA, могут присутствовать как гонки данных, так и бесконечное ожидание. Причиной бесконечного ожидания может быть как ошибка в коде атомарного ФК (например, бесконечный цикл), так и мертвая блокировка, вызванная использованием ФД, который не инициализируется во время выполнения программы. Для удобства далее такие ФД будут называться просто неинициализированными. Простейший пример приведен в листинге на рисунке 7 — ФД объявлен и используется, но не инициализируется.

```
01: C++ sub consume(value v) ${{ $ }}
02:
03: sub main() {
04:     df x;
05:     consume(x);
06: }
```

Рисунок 7 — Пример использования неинициализированного ФД

Такая программа зависнет при попытке ее выполнить. Другим примером является программа из листинга на рисунке 8. Здесь в коде присутствуют операторы, инициализирующие все используемые ФД, однако эта программа также зависнет, поскольку в графе информационных зависимостей присутствует цикл, что приводит к мертвой блокировке во время выполнения программы.

```
01: C++ sub assign_int(name x, int val) ${{ x = val; $ }}
02:
03: sub main() {
04:     df x, y;
05:     assign_int(x, y);
06:     assign_int(y, x);
07: }
```

Рисунок 8 — Пример мертвой блокировки

Наконец, к бесконечному ожиданию может привести использование ФД с индексированным именем внутри оператора *for* с некорректными границами. Пример такой программы приводится на рисунке 9.

```

01: C++ sub assign_int(name x, int val) ${{ x = val; $}}
02: C++ sub consume(value v) ${{ $}}
03:
04: sub main() {
05:     df x;
06:     for i = 1..5 { assign_int(x[i], i * i); }
07:     for i = 1..8 { consume(x[i]); }
08: }

```

Рисунок 9 — Пример некорректной границы при использовании for

Здесь $x[i]$ инициализируется для i от 1 до 5 (строка 6), однако из-за того, что используются значения с индексами от 1 до 8 (строка 7), данная программа также зависнет.

Допустить гонку данных при написании LuNA-программы так, чтобы она осталась незамеченной, сложно. ФД — объекты единственного присваивания, и при попытке инициализировать один и тот же ФД дважды выполнение программы аварийно завершится. Тем не менее, один из вариантов, при которых возникает гонка данных — использование рекомендации *delete*. Данная рекомендация говорит исполнительной системе о том, что некоторый ФД может быть удален после выполнения ФВ, к которому она относится. На рисунке 10 представлен листинг программы, в которой ФД инициализируется дважды и удаляется после каждой инициализации.

```

01: C++ sub assign_int(name x, int val) ${{ x = val; $}}
02:
03: sub main() {
04:     df x;
05:     assign_int(x, 42) @ { delete x; };
06:     assign_int(x, 0) @ { delete x; };
07: }

```

Рисунок 10 — Пример повторной инициализации ФД

Такая программа может завершиться успешно, если удаление значения x произойдет до повторной инициализации. Интересным является и то, что само использование *delete* дважды становится причиной гонки данных, поскольку при некоторых запусках программа завершалась аварийно с сообщением о повторном удалении (“Duplicate destroy”).

Возможна также ситуация, когда повторная инициализация ФД может как происходить, так и не происходить в зависимости от условий. Оператор *if* в языке

LuNA сначала вычисляет условие и приводит его к целочисленному типу. Аналогично языку Си, ложью считается 0, а любое другое значение — истинной. Если условие истинно, порождаются ФВ, определенные в теле оператора. На рисунке 11 представлен листинг LuNA-программы, содержащей инициализацию повторяющихся индексов для x внутри условий, которые не являются взаимоисключающими.

```
01: C++ sub assign_int(name x, int val) ${{ x = val; }}
02:
03:
04: sub main() {
05:     df x;
06:     assign_int(x[0], 7);
07:     if x [0] > 0 {
08:         for i = 1..5 { assign_int(x[i], i * i); }
09:     }
10:     if x[0] <= 10 {
11:         for i = 5..10 { assign_int(x[i], i * i); }
12:     }
13: }
```

Рисунок 11 — Пример повторной инициализации ФД в зависимости от истинности условий

Условия $x[0] > 0$ и $x[0] \leq 10$ в зависимости от значения $x[0]$ могут быть истинны одновременно, и в этом случае $x[5]$ будет инициализирован дважды (строки 8 и 11).

Приведенные выше примеры показывают, что пользователь при написании программ на языке LuNA не застрахован от типичных для традиционного параллельного программирования ошибок. Тем не менее, разнообразие способов допустить эти ошибки говорит о том, что с точки зрения пользовательского опыта при использовании гипотетического средства статического анализа гораздо полезнее будут сообщения об ошибках, ссылающиеся на конкретные языковые абстракции, нежели говорящие о самом факте наличия ошибок, связанных с некоторыми ФД или ФВ. Особенно это иллюстрируют примеры на рисунках 9 и 11, где ошибки связаны с границами *for*.

Для дальнейшего изложения будет удобно ввести следующие понятия:

Линейное выражение от переменной *var* — это арифметическое выражение, эквивалентное $step * var + offset$, где *step* (шаг) и *offset* (смещение) — целые числа;

Цикл инициализации для имени *name* — это оператор *for var = begin..end*, в теле которого (возможно, внутри вызываемых ФК) инициализируются значения *name[expr]*, где *expr* — линейное выражение от *var*;

Цикл использования для имени *name* — это оператор *for var = begin..end*, в теле которого (возможно, внутри вызываемых ФК) потребляются значения *name[expr]*, где *expr* — линейное выражение от *var*;

Диапазон индексов для имени *name* — это множество всех значений индексов *name*, задаваемых данным циклом инициализации или использования.

Таким образом, имеет смысл следующая классификация ошибок в LuNA-программах:

1. Синтаксические ошибки;
2. Семантические ошибки:
 - а. Ошибки низкого уровня:
 - i. Использование неинициализированного ФД;
 - ii. Повторная инициализация ФД;
 - iii. Условие тождественно истинно/ложно;
 - и др.
 - б. Ошибки высокого уровня:
 - i. Пересечение диапазонов инициализированных индексов для некоторого индексированного имени;
 - ii. Несоответствие границ циклов инициализации и использования;
 - iii. Циклическая зависимость по данным;
 - и др.

Конечно, как и в предложенной в главе 1 классификации ошибок в параллельных программах, классы не являются непересекающимися. Тем не менее, полезнее всего для пользователя средств статического анализа будет

информация об ошибках высокого уровня, объединяющих несколько ошибок низкого. Так, например, сообщение о том, что в программе на рисунке 9 имеет место выход за границу диапазона инициализированных индексов, так как верхняя граница цикла использования, 8 (строка 7), всегда больше верхней границы цикла инициализации, 5 (строка 6), будет гораздо более информативным и полезным при исправлении ошибки, чем набор сообщений о том, что ФД $x[6]$, $x[7]$ и $x[8]$ используются, но не инициализируются. Более того, не всегда возможно сформулировать суть ошибки в терминах низкого уровня: достаточно заменить 8 на некоторый ФД в строке 7, и перечислить все неинициализированные ФД станет невозможно.

2.4 База типов ошибок в LuNA-программах

Как было упомянуто ранее, данная работа не единственная по теме статического анализа LuNA-программ. Это, а также планы по интеграции разрабатываемых средств статического анализ в единый программный комплекс, делает актуальной задачу создания единой базы типов ошибок в LuNA-программах.

В рамках работы было принято участие в создании такой базы. На данный момент она объединяет 37 типов синтаксических и семантических ошибок, каждой из которых был присвоен код в формате “LUNANN”, где NN — номер типа ошибки. В приложении А приводится список типов ошибок с короткими пояснениями.

Типы ошибок, рассматриваемые в данной работе — это:

- LUNA18 — нижняя граница цикла использования может быть как больше, так и меньше нижней границы цикла инициализации.
- LUNA19 — нижняя граница цикла использования всегда строго меньше нижней границы цикла инициализации.
- LUNA20 — верхняя граница цикла использования может быть как меньше, так и больше верхней границы цикла инициализации.
- LUNA21 — верхняя граница цикла использования всегда строго больше верхней границы цикла инициализации.

- LUNA22 — шаг цикла использования не кратен шагу цикла инициализации.
- LUNA23 — формула в *if* тождественно истинна/ложна.
- LUNA25 — операторы, возвращающие логическое значение, используется в целочисленном контексте.
- LUNA35 — пересечение диапазонов инициализированных индексов.
- LUNA37 — присутствуют циклы использования, но нет циклов инициализации.

Нетрудно заметить, что ошибки LUNA18-LUNA21 синтаксически выглядят похоже на типичную для последовательных программ ошибку выхода за границу массива. Тем не менее, как было показано в подразделе 1.3.4, популярные статические анализаторы не способны выполнить достаточно глубокий анализ для обнаружения ошибок в случаях, аналогичных типичным сценариям использования циклов инициализации и использования в языке LuNA. Рассмотрим довольно простую LuNA-программу, листинг которой представлен на рисунке 12, выполняющую суммирование чисел от 1 до 5.

```

01: C++ sub assign_int(name x, int val) ${{ x = val; $}}
02: C++ sub print_int(int v) ${{ std::printf("%d\n", v); $}}
03:
04: sub sum(name a, int n, name result) {
05:     df acc;
06:     assign_int(acc[1], a[1]);
07:     assign_int(result, acc[n]);
08:     for i = 2..n {
09:         assign_int(acc[i], acc[i - 1] + a[i]);
10:     }
11: }
12:
13: sub main() {
14:     df x, s, N;
15:     print_int(s);
16:     assign_int(N, 5);
17:     for i = 0..N-1 { assign_int(x[i], 1 + i); }
18:     sum(x, N, s);
19: }

```

Рисунок 12 — Пример выхода за границу диапазона инициализированных индексов в LuNA

В данной программе допущена ошибка: внутри ФК *sum* предполагается, что для базового имени *a* индексация начинается с *1*, в то время как в *main* значения инициализируются начиная с *0*. На рисунке 13 представлен листинг аналогичной программы на языке C++.

```
01: #include <iostream>
02:
03: void sum(int const *a, int n, int *result) {
04:     int acc = a[1];
05:     for (int i = 0; i <= n; i++) {
06:         acc += a[i];
07:     }
08:     *result = acc;
09: }
10:
11: int main() {
12:     int N = 5, s, x[N];
13:     for (int i = 0; i < N; i++) { x[i] = 1 + i; }
14:     sum(x, N, &s);
15:     std::cout << s << std::endl;
16:     return 0;
17: }
```

Рисунок 13 — Пример выхода за границу массива в C++

Для программы на рисунке 13 PVS-Studio не выдает никаких сообщений об ошибках. Сообщение о возможном выходе за границу массива появляется лишь в том случае, если заменить вызов функции *sum* ее телом. Если же *N* не инициализируется статически (то есть известным на момент компиляции значением), PVS-Studio не сообщает об ошибке и в этом случае.

То же можно сказать и про ошибку LUNA25: некоторые тождественно истинные/ложные формулы не распознаются средствами статического анализа вообще. Однако, анализ выражений в условных операторах важен при обнаружении других семантических ошибок, связанных с инициализацией и использованием ФД.

Таким образом, при написании программ на языке LuNA пользователь может допустить ряд трудно обнаруживаемых семантических ошибок. Некоторые из этих ошибок могут иметь аналоги в программах на других языках программирования, для которых существуют средства статического анализа, но

чувствительность этих программных средств оказывается неудовлетворительной в контексте рассматриваемых ошибок.

2.5 Формат представления информации об ошибках

В рамках данной работы было принято участие в разработке формата вывода информации об ошибках в LuNA-программах. Единый формат вывода дает возможность отделить логику представления информации об ошибках от логики их поиска. Во-первых, это дает возможность с легкостью объединять вывод различных средств отладки LuNA-программ. Во-вторых, визуализация информации о найденных ошибках может быть реализована в отдельных программных средствах, представляющих сообщения об ошибках в удобном для восприятия пользователем виде.

Отчет об ошибках — это файл в формате JSON [62], содержащий список описаний ошибок. Каждое описание ошибки — это объект, имеющий следующие поля:

- “error_code” — код ошибки в виде строки, например, “LUNA21”;
- “details” — объект, содержащий специфическую для данного кода ошибки информацию.

Для каждого кода ошибки были определены поля объекта “details”, были зафиксированы типы и структура объектов, которые могут быть использованы при описании ошибки в данном формате. Далее приводится информация о полях типовых объектов, используемых при описании различных ошибок.

Элемент стека вызовов имеет следующие поля:

- “file” — имя файла (строка);
- “line” — номер строки (число);
- “name” — имя вызываемого ФК или имя оператора (строка).

Стек вызовов является списком таких элементов.

Фрагмент данных описывается полями:

- “name” — имя при объявлении (строка);
- “declared” — места объявления (список стеков вызовов);

- “initialized” — места инициализации, может отсутствовать (список стеков вызовов);
- “used” — места использования, может отсутствовать (список стеков вызовов);

Под *ссылкой на ФД* понимается использование имени ФД в конкретном месте в коде. Ссылка на ФД имеет поля:

- “df” — информация о месте объявления базового имени (ФД, описанный выше);
- “local” — выражение в локальной области видимости (строка);
- “true” — реальное выражение, значение которого скрыто за “local”, как если бы использовался нормальный порядок вычислений (строка). Например, если на место параметра с именем *x* передать *y[i]*, то “local” будет иметь значение “x”, а “true” — “y[i]”.
- “where” — место использования ФД (стек вызовов).

Цикл for описывается следующим образом:

- “var” — имя переменной-счетчика (строка);
- “first” — выражение, использованное в качестве нижней границы (строка);
- “last” — выражение, использованное в качестве верхней границы (строка);
- “where” — информация о том, где данный цикл находится (стек вызовов).

Диапазон индексов имеет поля:

- “df_ref” — место использования имени (ссылка на ФД);
- “loop” — информация о цикле, который задает данный диапазон (цикл for);
- “step” — шаг (строка);
- “offset” — смещение (строка).

Большинство из рассматриваемых в работе ошибок описываются в терминах типовых объектов, приведенных выше. Для ошибок LUNA18-LUNA22 “details” имеет поля “initialized” и “used”, являющиеся диапазонами индексов. Для LUNA35 “details” содержит поле “ranges” — список пересекающихся диапазонов индексов. При описании LUNA37 “details” всегда имеет поле “used” (диапазон индексов), а также опционально поля “use_conditions” (список строк),

“initialized” (диапазон индексов) и “init_conditions” (список строк). При описании ошибки LUNA23 в “details” присутствуют поля “condition” (строка), “type” (логическое значение) и “where” (элемент стека вызовов), а при описании LUNA25 — “bad_expr” (строка), “arg_index” (число, номер операнда, имеющего логическое значение) и “where” (элемент стека вызовов).

Вывод информации о найденных в LuNA-программах ошибках в фиксированном формате дает возможность вести разработку программных средств, отвечающих за представление этой информации, независимо от средств отладки. В частности, это позволило использовать шаблоны сообщений об ошибках, в которые подставлялась необходимая информация из описания ошибки, причем использование в описаниях типовых объектов позволило переиспользовать логику их визуализации для многих ошибок.

2.6 Требования

Для разрабатываемого программного средства статического анализа, названного prolog-analyzer, были сформулированы следующие требования:

1. Программный комплекс prolog-analyzer должен состоять из двух компонентов:
 - a. генератор представления LuNA-программы в виде набора фактов на языке логического программирования Prolog (логического представления LuNA-программы);
 - b. программа-анализатор на языке Prolog, выполняющая поиск семантических ошибок.

Логическое представление LuNA-программы может быть использовано не только для анализа. Например, на базе генератора логического представления может быть реализована система запросов к коду (code querying system), позволяющая выполнять поиск элементов программы с помощью запросов на языке логического программирования. Другой возможный вектор развития — автоматизация рефакторинга программы. Логическое представление может быть использовано для поиска мест в коде, подлежащих рефакторингу, может быть динамически изменено, а также конвертировано обратно в код на языке LuNA.

2. Генератор логического представления должен поддерживать подмножество синтаксиса языка LuNA, включающее:

a. Операторы:

- i. *if* — условный оператор;
- ii. *for* — оператор примитивно-рекурсивного перечисления.

b. Выражения:

- i. целочисленные константы;
- ii. арифметические выражения (операторы +, -, *, /);
- iii. сравнения (операторы >, >=, <, <=, ==, !=);
- iv. логические операторы (&& и ||);
- v. индексированные имена (например, $x[i]$ или $y[z[k + 1]][7]$).

c. Оператор применения ФК.

Описанные здесь элементы синтаксиса языка LuNA достаточны для реализации итерационных численных алгоритмов.

3. Программа, выполняющая поиск ошибок, должна обнаруживать ошибки LUNA18-LUNA23, LUNA25, LUNA35 и LUNA37.

4. Вывод информации о найденных ошибках должен осуществляться в формате, описанном в разделе 2.5.

5. При запуске *prolog-analyzer* должна присутствовать возможность указать имя файла, куда нужно записать информацию о найденных ошибках (файл отчета об ошибках):

- a. если файл существует, он должен быть дополнен с сохранением описанной в разделе 2.5 структуры;
- b. если файл не существует, он должен быть создан.

6. *Prolog-analyzer* должен при запуске предоставлять выбор, следует ли удалить промежуточные файлы, генерируемые и используемые при анализе.

7. *Prolog-analyzer* должен автоматизировать генерацию логического представления LuNA-программы, поиск ошибок и генерацию отчета об ошибках.

2.7 Характеристики предлагаемого решения

Данное решение позволяет автоматизировать обнаружение ряда семантических ошибок в LuNA-программах. Основным преимуществом решения является автоматизация процесса обнаружения ошибок — пользователю необходимо лишь запустить `prolog-analyzer` с желаемыми параметрами, после чего будет сгенерирован файл отчета об ошибках в представленных файлах исходного кода. `Prolog-analyzer` не является частью системы LuNA, что дает возможность вносить в него изменения и расширять функциональность и без вмешательства в систему. Стандартизация формата вывода информации о найденных ошибках и, как следствие, отделение логики представления информации об ошибках от логики их поиска предоставляет широкий простор в реализации программных средств, ответственных за предоставление пользователю данной информации в пригодном для восприятия виде. Такие программные средства могут разрабатываться независимо и использовать различные способы визуализации: генерация HTML-страницы, вывод информации в консоль, предоставление подсказок в IDE и др.

Глава 3. Разработка статического анализатора prolog-analyzer

В ходе работы рассматривались различные подходы к реализации средства статического анализа LuNA-программ. Все они объединены тем, что информация о программе представляется в виде набора фактов на языке логического программирования Prolog (используется реализация SWI-Prolog [55]), а для поиска ошибок применяются предикаты (правила). Таким образом, проверка на наличие ошибок сводится к проверке выполнимости правил. Удобно и то, что при этой проверке Prolog инстанцирует входящие в правила свободные переменные, что позволяет получить как информацию о наличии ошибки, так и информацию о том, почему ошибка имеет место.

3.1 Анализ на основе графа зависимостей по данным

Первым из рассматриваемых подходов был анализ графа зависимостей по данным. При определенных условиях по исходному коду LuNA-программы возможно определить множество ФВ, которые будут порождены во время ее выполнения, а также информационные зависимости между ними — множества входных и выходных ФД.

3.1.1 Представление LuNA-программы

В качестве примера будет использоваться программа, листинг которой представлен на рисунке 14:

```

01: C++ sub set(name x, int v) ${{
02:     x.setValue<int>(v);
03: $}}
04:
05: C++ sub print(int v) ${{
06:     printf("%d\n", v);
07: $}}
08:
09: sub main() {
10:     df a, b, c;
11:     set(b, a * a);
12:     set(c, b + 1);
13:     print(c);
14: }

```

Рисунок 14 — LuNA-программа, содержащая использование
неинициализированного ФД

Программа представляется в виде набора фактов о фрагментах данных и вычислений в формате *cf(Id, InDFs, OutDFs)*. Здесь *Id* — уникальный идентификатор ФВ, а *InDFs* и *OutDFs* — списки имен входных и выходных ФД данного ФВ. Например, в листинге на рисунке 15 приводится набор сгенерированных для программы на рисунке 14 фактов на языке Prolog.

```

01: cf(0, ["a"], ["b"]).
02: cf(1, ["b"], ["c"]).
03: cf(2, ["c"], []).

```

Рисунок 15 — Пример набора фактов, описывающего LuNA-программу

Для генерации данного представления используется промежуточное представление LuNA-программы в формате .ja (JSON Algorithm), генерируемое компилятором LuNA в процессе работы. Была реализована генерация логического представления для ограниченного подмножества языка LuNA: вызовов фрагментов кода (как атомарных, так и структурированных) с параметрами типа *name* и *int*, и оператора *for* с известными на этапе компиляции границами переменной-счетчика.

При генерации логического представления также сохранялась информация о соответствии идентификаторов ФВ, используемых в фактах, подробной информации о ФВ (имя, стек вызовов, место в коде). Эта информация использовалась для предоставления подробного отчета об обнаруженных семантических ошибках.

3.1.2 Обнаружение ошибок

Для обнаружения ошибок были реализованы два правила: *non_computable*(*CF*, *MissingDFs*) и *multiple_init*(*DF*, *CF*). Правило *non_computable* принимает значение истина, если ФВ с идентификатором *CF* не может быть вычислен из-за того, что ФД из списка *MissingDFs* не получают свои значения. Определение правила *non_computable* представлено в листинге на рисунке 16:

```
01: non_computable(CF, MissingDFs) :- (  
02:   deadlock(CF), !;  
03:   (  
04:     cf_in(CF, CFIn),  
05:     exclude(df_computed, CFIn, MissingDFs),  
06:     MissingDFs \= []  
07:   )  
08: ).
```

Рисунок 16 — Определение правила *non_computable*

Тело данного правила означает, что ФВ невычислим, если он имеет прямую или косвенную информационную зависимость от самого себя (*deadlock*(*CF*), строка 2), либо не пуст список ФД, которые входят в набор входных ФД данного ФВ и для которых правило *df_computed* принимает значение ложь. Правила *deadlock*, *cf_in* и *df_computed* также были определены в рамках этой работы, *exclude* — часть стандартной библиотеки SWI-Prolog.

Например, для программы на рисунке 14 запрос “*non_computable*(*CF*, *MissingDFs*).” будет иметь следующий результат (рисунок 17):

```
3 ?- non_computable(CF, MissingDFs).  
CF = 0,  
MissingDFs = ["a"] ;  
CF = 1,  
MissingDFs = ["b"] ;  
CF = 2,  
MissingDFs = ["c"].
```

Рисунок 17 — Вывод запроса *non_computable*

Правило *multiple_init*(*DF*, *CF*) принимает значение истина, если ФД с именем *DF* является выходным для ФВ с идентификатором *CF*, и существует один или несколько других ФВ, для которых данный ФД также является

выходным. Таким образом, набор всех значений переменных (DF , CF), на которых правило *multiple_init* истинно, будет содержать информацию обо всех ФД, инициализированных несколько раз, а также обо всех ФВ, выполняющих повторную инициализацию.

3.1.3 Анализ LuNA-программ

На языке программирования Python было реализовано средство анализа LuNA-программ, автоматизирующее процесс генерации логического представления и выполнения запросов. Для работы средству анализа требовались файлы, генерируемые компилятором LuNA (`luna --compile-only --no-cleanup`). В качестве параметров оно принимало путь до директории с LuNA-программой, а также, опционально, путь до директории, в которую компилятор сохранил промежуточные файлы.

Средство анализа генерировало и сохраняло подробную информацию о ФВ (входные и выходные ФД, стек вызовов, реальные значения параметров, информация о месте в коде LuNA-программы). Затем генерировалось логическое представление программы. Наконец, к базе знаний выполнялись запросы (проверяется выполнимость правил *non_computable* и *multiple_init*), и выводилась подробная информация об обнаруженных ошибках.

Рассмотрим пример программы из листинга на рисунке 18:

```

01: import c_init( name, int) as init;
02: import c_print( value) as print;
03:
04: sub init_1(name arr, int ind) {
05:     init(arr[ind], 0);
06: }
07:
08: sub init_2(name arr, int ind) {
09:     init(arr[ind-1], 1);
10: }
11:
12: sub main() {
13:     df x;
14:     for i = 0..250 {
15:         init_1(x, 2*i);
16:         print(x[2*i]) @{
17:             delete x[2*i];
18:         };
19:         init_2(x, 2*i+1);
20:         print(x[2*i+1]) @{
21:             delete x[2*i+1];
22:         };
23:     }
24: }

```

Рисунок 18 — Пример LuNA-программы с ошибками

Данная LuNA-программа содержит ошибку повторной инициализации: *init_1* вызывается для инициализации *x* с индексами $2*i$, а *init_2* — с индексами $2*i+1$, но *init_2* перед инициализацией отнимает единицу от переданного индекса (строка 9). Из-за данной ошибки индексации ФД *x* по индексам $2*i+1$ никогда не инициализируются, из-за чего ФВ, соответствующие *print(x[2*i+1])*, не могут быть выполнены.

На рисунках 19 и 20 представлены фрагменты вывода средства анализа, соответствующие данным ошибкам.

```

CF cannot be computed
print(x[2*i+1]) {x[((2 * i) + 1)] is x[1]} [./main.fa:20]
in for i = 0..250 {i is 0} [./main.fa:14]
in sub main() [./main.fa:12]

Missing x[1]

```

Рисунок 19 — Фрагмент вывода для ФВ, который не может быть выполнен из-за отсутствующих ФД

```

DF x[0] initialized multiple times in
init(arr[ind-1], 1) {arr[(ind - 1)] is x[0]} [./main.fa:9]
  in init_2(x, 2*i+1) {((2 * i) + 1) is 1} [./main.fa:19]
  in for i = 0..250 {i is 0} [./main.fa:14]
  in sub main() [./main.fa:12]

init(arr[ind], 0) {arr[ind] is x[0]} [./main.fa:5]
  in init_1(x, 2*i) {(2 * i) is 0} [./main.fa:15]
  in for i = 0..250 {i is 0} [./main.fa:14]
  in sub main() [./main.fa:12]

```

Рисунок 20 — Фрагмент вывода для ФД, который инициализируется несколько раз

Приведенные выше примеры показывают, что использование данного подхода позволяет предоставить пользователю подробную информацию об ошибках, включающую стеки вызовов, имена файлов исходного кода и номера строк в них, а также информацию о соответствии локально видимых идентификаторов реальным выражениям.

3.1.4 Тестирование производительности и оценка решения

Было выполнено тестирование с целью оценки времени анализа LuNA-программы. Использовалась программа на рисунке 18 (“программа с ошибками”) с различными значениями верхней границы цикла (строка 14), а также ее вариант, не содержащий ошибку индексации в `init_2` (“программа без ошибок”). Результаты представлены на графике на рисунке 21.

Время анализа программы

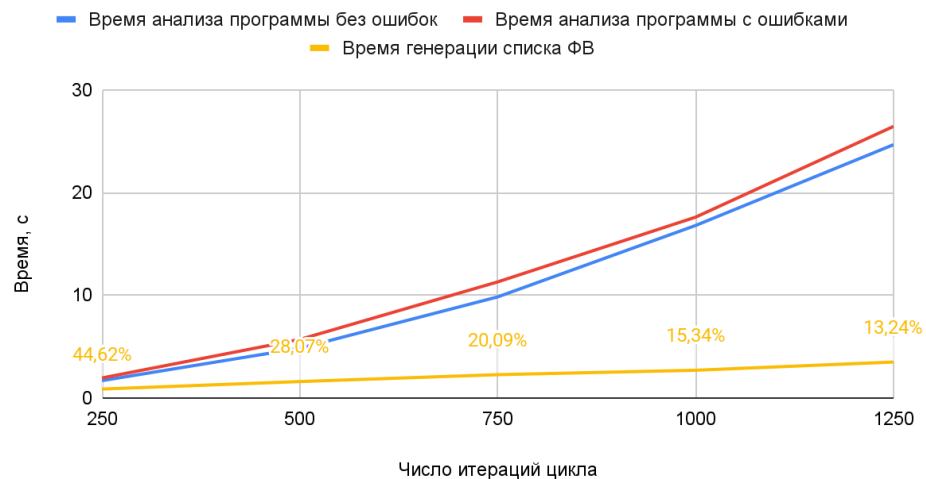


Рисунок 21 — График зависимости времени анализа от числа итераций цикла в программе

Можно заметить, что время анализа практически не зависит от наличия либо отсутствия ошибок, однако растет нелинейно с увеличением числа итераций цикла (и, как следствие, числа ФВ, которые нужно проанализировать). Можно заметить и то, что доля времени, которое средство анализа тратит на генерацию логического представления (“Время генерации списка ФВ”), уменьшается с ростом числа итераций, что свидетельствует о том, что именно анализ с помощью Prolog оказывает наибольшее влияние на общее время работы.

Несмотря на преимущества, которые предлагает описанный подход, он имеет и ряд недостатков. Во-первых, как показало тестирование, время работы может быть очень велико даже на сравнительно небольших и простых LuNA-программах. Во-вторых, в текущей реализации анализ возможен только для программ, написанных на сильно ограниченном подмножестве языка LuNA. Необходимость возможности вычислять выражения на этапе компиляции делает невозможным использовать такие полезные функции языка, как, например, переменные границы циклов и ветвления.

3.2 Анализ высокоуровневых семантических конструкций языка

Прототип, описанный в разделе 3.1, был представлен на семинаре "Математическое обеспечение высокопроизводительных вычислительных

систем" (студенческая серия). Одной из предложенных идей был переход от анализа отдельных ФД и ФВ к анализу высокоуровневых семантических конструкций. Так, например, программист при написании программ, использующих операторы `for`, рассматривает множества ФД с индексированными именами в целом, то есть как диапазоны индексов. Исходя из данных соображений, был разработан второй прототип средства статического анализа.

3.2.1 Представление программы

Для описания формата представления LuNA-программы потребуются следующие определения:

Одиночный ФД — ФД, имя которого не содержит индексов (например, M , N , x , но не $x[i]$).

Место инициализации/использования ФД x — стек вызовов от `main` и до выражения, потребляющего значение ФД x .

Для представления LuNA-программы используются следующие факты:

1. $df_init_single(InitID, Name)$ — инициализация одиночного ФД;
2. $df_use_single(UseID, Name)$ — использование одиночного ФД;
3. $df_init_range(InitId, Name, boundary(InitStart, InitStartType), boundary(InitEnd, InitEndType))$ — инициализация диапазона индексов для ФД $Name$ начиная от $InitStart$ и заканчивая $InitEnd$ с шагом 1 и смещением 0;
4. $df_use_range(UseId, Name, boundary(UseStart, UseStartType), boundary(UseEnd, UseEndType))$ — использование диапазона индексов для ФД $Name$ начиная от $UseStart$ и заканчивая $UseEnd$ шагом 1 и смещением 0.

Во всех четырех видах фактов $InitID$ и $UseId$ — уникальные идентификаторы мест инициализации и использования ФД в программе, а $Name$ — имя ФД. Параметры $\{Init|Use\}\{Start|End\}Type$ принимали значения “const”, “df” и “expr”.

В программе объявленные с одним и тем же именем ФД могут быть фактически разными во время выполнения (например, при нескольких вызовах одной и той же подпрограммы, внутри которой объявляется ФД). Поскольку в программе на языке Prolog ФД идентифицируются именами, использование имен ФД из текста LuNA-программы могло бы привести к коллизиям. Чтобы этого избежать, для ФД генерировались уникальные имена на основе имени, использованного в программе, и позиций в коде программы элементов стека вызовов места объявления ФД.

Например, для LuNA-программы на рисунке 22, для ФД x будут сгенерированы имена *Var174_244_x*, *Var174_260_x* и *Var174_277_x* для первого, второго и третьего вызова подпрограммы *init* соответственно.

```
01: import c_init(name, int) as set_int;
02: import c_print(value) as print;
03:
04: #define Offset 5
05:
06: sub init(int start, int end) {
07:     df x;
08:
09:     for i = start..end {
10:         set_int(x[i], i);
11:     }
12: }
13:
14: sub main() {
15:     df N, M;
16:
17:     set_int(M, 0);
18:     set_int(N, 9);
19:
20:     init(0, 9);
21:     init(1, 10);
22:     init(M, N);
23: }
```

Рисунок 22 — LuNA-программа, содержащая несколько вызовов подпрограммы, в которой объявляется ФД

На рисунке 23 приводится фрагмент набора фактов для программы на рисунке 22, содержащий факты о инициализации x в цикле для трех вызовов *init* — от 0 до 9, от 1 до 10 и от M до N .

```

01: df_init_range(2, "Var174_244__x", boundary("0", "const"),
boundary("9", "const")).
02: df_init_range(3, "Var174_260__x", boundary("1", "const"),
boundary("10", "const")).
03: df_init_range(6, "Var174_277__x",
boundary("Var__Var174__M", "df"), boundary("Var__Var174__N",
"df")).

```

Рисунок 23 — Факты, описывающие инициализацию диапазона индексов для
ФД x

В отличие от первого прототипа, границы *for* могут быть как константами, так и выражениями, содержащими константы и одиночные ФД (например, 0 , $1 + 1$, $2 * N - 1$). Не поддерживались вложенные операторы *for*, операторы *while*, условные операторы и многоуровневая индексация.

3.2.2 Обнаружение ошибок

Были реализованы два правила для обнаружения ошибок:

- *use_of_uninitialized_df*(Name, Details) — использование неинициализированного одиночного ФД;
- *invalid_df_use_range*(Name, Details) — некорректные границы цикла использования ФД.

Правило *use_of_uninitialized_df* имеет три варианта, соответствующие следующим трем сценариям использования неинициализированного ФД:

1. ФД зависит сам от себя, то есть присутствует циклическая зависимость по данным.
2. ФД не инициализируется в принципе, то есть не существует ФВ, для которого ФД был бы выходным.
3. ФД x зависит от другого неинициализированного ФД y , то есть ФВ, инициализирующему x , необходимо значение y .

Правило *invalid_df_use_range* имеет два варианта для проверки нижней и верхней границы цикла использования. Выражения, используемые в качестве границ, сравниваются с помощью библиотеки CLP(FD) (Constraint Logic Programming over Finite Domains) [63]. Например, если верхняя граница цикла использования строго больше верхней границы цикла инициализации (инициализация до 9, использование до 10, или инициализация до N , но

использование до $N + 1$), то цикл использования некорректен, и пользователю будет выведено сообщение об ошибке. Если же однозначно упорядочить границы нельзя (например, M может быть как меньше или равно N , так и больше), то пользователю будет выведено предупреждение. Нижние границы проверяются аналогично.

3.2.3 Анализ LuNA-программ

Аналогично первому прототипу, программа на Python использовалась для генерации логического представления LuNA-программы и выполнения запросов к нему (проверялась выполнимость правил *invalid_df_use_range* и *use_of_uninitialized_df*), после чего выводилась подробная информация об обнаруженных ошибках.

В качестве примера будет использоваться программа, листинг которой приводится на рисунке 24.

```

01: import c_init(name, int) as set_int;
02: import c_print(value) as print;
03:
04: sub init(name it, int from, name to) {
05:     for i = from..to {
06:         set_int(it[i], i);
07:     }
08: }
09:
10: sub use(name it, name from, int to) {
11:     for i = from..to {
12:         print(it[i]);
13:     }
14: }
15:
16: sub main() {
17:     df x, y, N, M;
18:
19:     set_int(N, 10);
20:     set_int(M, 10);
21:
22:     // Initialization
23:     init(x, N - 10, N);
24:     init(y, N, M);
25:
26:     // Consumption
27:     use(x, M, N);
28:     use(y, x, M + 1);
29: }

```

Рисунок 24 — LuNA-программа, содержащая ошибки, связанные с некорректными границами циклов использования

В данной программе, например, для y инициализируются индексы от N до M , но используются индексы от x до $M + 1$. Вывод для верхней границы представлен на рисунке 25.

```

[error] DF y: initialized up to M but used up to (M + 1) (error:end_out_of_bounds_df_expr)
Initialized:
  in set_int(it[i], i) [./main.fa:6]
  in for i = from..to [./main.fa:5]
  in sub init(name it, int from, name to) [./main.fa:4]
  in init(y, N, M) [./main.fa:24]
  in sub main() [./main.fa:16]
    note: M declared here
    note: y declared here

Used:
  in print(it[i]) [./main.fa:12]
  in for i = from..to [./main.fa:11]
  in sub use(name it, name from, int to) [./main.fa:10]
  in use(y, x, M + 1) [./main.fa:28]
  in sub main() [./main.fa:16]
    note: M declared here
    note: y declared here

```

Рисунок 25 — Фрагмент вывода анализатора для верхней границы цикла использования у

Поскольку $M + 1$ строго больше M , использование у с индексами до $M + 1$ — ошибка (в выводе анализатора — “error”). ФД x , однако, может иметь такое значение, что использование индексов начиная с x не приводит к выходу за границы диапазона инициализированных индексов, поэтому для для нижней границы анализатор выведет предупреждение (“warning”).

3.2.4 Оценка решения

Был изменен подход к поиску неинициализированных ФД. Во втором прототипе ФД, которые инициализируются и используются в циклах, обрабатывались отдельно от одиночных, а циклы рассматривались целиком, то есть сравнивались диапазоны инициализированных и используемых индексов. Благодаря такому подходу время анализа не зависело от числа итераций циклов. В границах циклов также допускались арифметические выражения, содержащие ФД.

Главный минус этого подхода — анализируемые абстракции выделяются отдельной программой-генератором. Это препятствует расширяемости средства статического анализа, поскольку возможная функциональность программы-

анализатора ограничена набором сущностей, которые используются программой-генератором при создании логического представления LuNA-программы.

3.3 Трансляция AST программы и выделение анализируемых абстракций в процессе анализа

Для преодоления ограничений описанных выше прототипов было решено использовать близкое к абстрактному синтаксическому дереву логическое представление программы и выделять анализируемые абстракции в процессе анализа. В итоговой версии prolog-analyzer представляет собой программу на Python, последовательно вызывающую компилятор LuNA, утилиту генерации логического представления программы и программу-анализатор на Prolog. Итоговая схема работы prolog-analyzer представлена на рисунке 26.

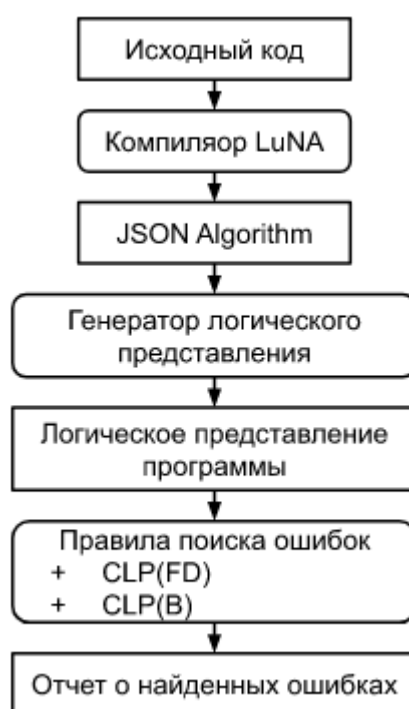


Рисунок 26 — Схема работы prolog-analyzer

В приложении Б приведено руководство оператора prolog-analyzer, а в приложении В - описание программы.

3.3.1 Представление LuNA-программы

Как и в предыдущих версиях, LuNA-программа представляется в виде набора фактов на языке Prolog. В отличие от предыдущих версий программа не

описывается как множество ФВ (первый прототип), или информацией об инициализациях и использовании диапазонов индексов или одиночных ФД (второй прототип). На Prolog транслируется внутреннее представление LuNA-программы, используемое компилятором.

Преобразование кода фрагментированного алгоритма на языке LuNA в набор фактов на Prolog выполняется в два этапа:

1. Компилятор LuNA используется для генерации представления LuNA-программы в формате JSON Algorithm с помощью команды `luna --compile-only`.
2. Утилита `luna2pro`, разработанная в рамках данной работы, используется для генерации логического представления программы — трансляции сгенерированного компилятором представления в формате `.ja` в набор фактов, описывающих LuNA-программу на языке Prolog.

Стоит отметить, что логическое представление LuNA-программы является близким к абстрактному синтаксическому дереву, но не является им в чистом виде. Во-первых, не всем вершинам абстрактного синтаксического дерева соответствуют отдельные факты. Вместо этого, как будет показано далее, факты представляют собой поддеревья. Во-вторых, формат JSON Algorithm допускает наличие дополнительной семантической информации, как, например, списки имен входных ФД для вызовов ФК.

Утилита `luna2pro` представляет собой небольшую Python-программу, которая принимает на вход представление LuNA-программы в формате `.ja` и файл с информацией о расположении элементов программы в файлах исходного кода в формате `.ti` (Text Info), и генерирует представление LuNA-программы в виде набора следующих фактов:

1. `luna_subroutine(Id, File, Line, Name, Type, Params, Body)`. — фрагмент кода:
Id — уникальный идентификатор данного факта (целое число);
File — имя файла (строка);
Line — номер строки (целое число);
Name — имя фрагмента кода (строка);

Type — тип, структурированный — *struct*, или атомарный — *extern* (атом);

Params — список формальных параметров; для структурированных ФК каждый элемент — это пара [*PType*, *PName*], где *PType* — тип параметра — *name* для *name* в коде, и *value* для всех остальных типов (атом), а *PName* — имя параметра (строка); для атомарных ФК имя не указывается;

Body — список идентификаторов выражений в теле ФК (список целых чисел).

2. *luna_statement*(*Id*, *File*, *Line*, *Type*, *TypeSpecificInfo*, *Requests*). —

выражение:

Id — уникальный идентификатор данного факта (целое число);

File — имя файла (строка);

Line — номер строки (целое число);

Type — тип данного выражения — *ехес* для вызова ФК, *for*, *dfs* и т.д. (атом);

TypeSpecificInfo — специфическая для данного выражения информация, например, имя вызываемого ФК и список фактических параметров для *ехес*, имя переменной-счетчика и выражения-границы для *for* и т.п.;

Requests — имена ФД, значения которых необходимы для выполнения данного оператора (список выражений).

Поддерживаются операторы вызова ФК, *for* и *if*, а также объявления ФД с помощью *df*. Поддерживаются выражения, состоящие из имен ФД (возможно, индексированных), целочисленных констант, арифметических операций (+, -, *, /), сравнений (>, >=, <, <=, ==, !=) и логических операций (&&, ||).

В таблице 1 приводится пример последовательности преобразований структурированного ФК *init_2* из исходного кода на LuNA в набор фактов на Prolog. Как было отмечено ранее, сгенерированные факты соответствуют поддеревьям абстрактного синтаксического дерева программы. Так, например, в столбце слева *luna_statement*, соответствующий вызову *init*, содержит дерево выражений для *arr[ind-1]* (выделено жирным).

Таблица 1 — Пример последовательности преобразования исходного кода в AST на Prolog

Исходный код на языке LuNA	Представление в формате .ja (сокращенное)	Представление в виде набора фактов на Prolog
<pre>sub init_2(name arr, int ind) { init(arr[ind-1], 1); }</pre>	<pre>"init_2": { "args": [{"id": "arr", "type": "name"}, {"id": "ind", "type": "int"}], "begin": 153, "body": [...], "end": 211, "type": "struct", "where": {"type": "luna"} }</pre>	<pre>luna_subroutine(4, "./main.fa", 8, "init_2", struct, [[name, "LUNA_arr"],[value, "LUNA_ind"]], [5,6]). luna_statement(5, "./main.fa", 8, dfs, [[]], []). luna_statement(6, "./main.fa", 9, exec, ["init", [luna_ref(["LUNA_arr", ["- ", "LUNA_ind", 1]]), 1]], []).</pre>

3.3.2 Дерево вызовов

Для анализа программы сначала строится дерево вызовов (call tree, CT). Оно определяется следующим образом:

$CT = (V_{CT}, A_{CT}, Aliases_{CT}, Conditions_{CT}, Info_{CT}, a_{CT}, c_{CT}, i_{CT})$, где

V_{CT} — вершины дерева вызовов: вызовы ФК, операторы *for* и *if*;

A_{CT} — дуги дерева вызовов; две вершины op_1 и op_2 соединены дугой (op_1, op_2), если 1) op_2 находится в теле op_1 , или 2) op_1 — вызов ФК F , и op_2 находится в теле F ;

$Aliases_{CT}$ — информация о соответствии видимых идентификаторов выражениям (например, соответствие формальных и фактических параметров) в вершинах;

$Conditions_{CT}$ — информация об условиях, истинность которых необходима для выполнения вершин;

$Info_{CT}$ — специфическая информация, которая может иметь различную структуру для разных типов вершин;

a_{CT} — отображение $V_{CT} \rightarrow Aliases_{CT}$;

c_{CT} — отображение $V_{CT} \rightarrow Conditions_{CT}$;

i_{CT} — отображение $V_{CT} \rightarrow Info_{CT}$.

При анализе дерево вызовов не строится в явном виде. Рассматриваются отдельные пути в нем, называемые *последовательностями выполнения* (execution sequence), и генерируемые с помощью предиката *walk(Body, Aliases, CallStack, ExecutionSequenceAcc, ExecutionSequence)*:

Body (входной параметр) — список идентификаторов выражений в теле текущего оператора/ФК;

Aliases (входной параметр) — текущая информация о соответствии видимых идентификаторов выражениям (элемент *Aliases_{СТ}*);

CallStack (входной параметр) — список идентификаторов выражений из *V_{СТ}* в текущем пути;

ExecutionSequenceAcc (входной параметр) — аккумулятор для *ExecutionSequence*, некоторая часть пути от начала до текущего оператора/ФК;

ExecutionSequence (выходной параметр) — путь; одно из возможных продолжений *ExecutionSequenceAcc* с учетом *Body*.

Предикат *walk* использует обход в глубину для построения путей, вызывая себя рекурсивно после добавления текущего элемента пути в аккумулятор. Были реализованы варианты, обрабатывающие:

1. вызовы атомарных ФК;
2. вызовы структурированных ФК;
3. операторы *for*;
4. операторы *if*.

Вариант, обрабатывающий атомарные ФК, не выполняет рекурсивных вызовов, а помещает текущее значение аккумулятора в *ExecutionSequence*.

Вариант, обрабатывающий *if*, добавляет условие, использованное в операторе, в элементы *Conditions_{СТ}* для всех дочерних вершин.

При анализе применение *walk* заключается в вызове его со следующими параметрами: *Body* — тело ФК *main* (список идентификаторов выражений в теле); *CallStack* — пустой список или *Id* ФК *main* для симуляции вызова *main* в начале стека вызовов; *Aliases* — пустой словарь; *ExecutionSequenceAcc* — пустой список; *ExecutionSequence* — свободный параметр. В этом случае параметр

ExecutionSequence будет принимать значения всех путей из *main* до вызовов атомарных ФК.

Элементы *ExecutionSequence* будут иметь следующий вид:

1. Для вызовов ФК — *[StatementId, exec, Aliases, Conditions, Callee]*, причем *Aliases* расширяется отображением формальных и фактических параметров.
2. Для оператора *for* — *[StatementId, for, Aliases, Conditions, Var]*, причем *Aliases* расширяется именем переменной-счетчика.
3. Для оператора *if* — *[StatementId, if, Aliases, Conditions]*, причем *Conditions* также содержит условие данного оператора.

Во всех вариантах, кроме *if*, условия (*Conditions*) наследуются у последнего элемента *ExecutionSequenceAcc*. Если же аккумулятор пуст, список условий считается пустым. Элементы *Aliases_{CT}*, *Conditions_{CT}* и *Info_{CT}* не хранятся явно в отдельных множествах, а являются частью вершин, которым соответствуют.

Во избежание коллизий имен при построении последовательностей выполнения для идентификаторов, использованных при объявлении ФД с помощью оператора ключевого слова *df*, внутри предиката *walk* генерируются псевдонимы. Псевдоним является строкой в формате “Var_<CallStack>__<Name>”, где <CallStack> — строка, полученная конкатенацией элементов параметра *CallStack* предиката *walk* с использованием “_” в качестве разделителя, а <Name> — имя ФД, использованное при объявлении. Пусть, например, *CallStack* имеет значение *[1, 2, 3]*, и объявляется ФД с именем *x*. Тогда для *x* будет сгенерирован псевдоним “Var_1_2_3__x”.

3.3.2 Поиск ошибок, связанных с индексированными ФД

Для поиска циклов инициализации и использования ФД применяются предикаты *df_init_loop* и *df_use_loop* соответственно, реализованные через мета-предикат *df_loop*, код которого представлен на рисунке 27. Предикат *df_loop* принимает начальный контекст для *walk* (строка 8), а также предикат *GetDf*, принимающий последовательность выполнения и возвращающий описание ФД, и возвращает диапазон индексов в виде словаря (строки 2-7).

```

01: df_loop(RootCtx, GetDf, IndexRange) :-
02:     IndexRange = index_range{
03:         'df': Df,
04:         'loop': Loop,
05:         'step': Step,
06:         'offset': Offset
07:     },
08:     execution_sequence:walk(RootCtx, Es),
09:     call(GetDf, Es, Df),
10:     get_dict('true', Df, TrueName),
11:     get_dict('where', Df, InitEs),
12:     arithmetic:indexed_name(TrueName, _, Var, Step,
Offset),
13:     for_loop(InitEs, Loop), get_dict('var', Loop, Var).

```

Рисунок 27 — Код предиката *df_loop*

Предикаты *df_init_loop* и *df_use_loop* при вызове *df_loop* на место *GetDf* передают предикаты, возвращающие выходные и входные ФД соответственно. Выходными являются ФД, переданные в качестве параметров типа *name* при вызове атомарных ФК, а входными — ФД, находящиеся в списке *Requests* факта *luna_statement*, соответствующего некоторой вершине дерева вызовов.

Предикат *arithmetic:indexed_name* (строка 12) пытается представить выражение *TrueName* в виде линейного выражения. Были реализованы предикаты для упрощения арифметических выражений, содержащих операции умножения, сложения, вычитания, и смены знака. Так, например, и $x[i]$ (переменная i , шаг и смещение равны нулю), и $x[2*j+1]$ (переменная j , шаг равен 2, смещение — 1), и $x[(2*k+1)-1]$ (переменная k , шаг равен 2, смещение равно нулю) будут корректно распознаны как индексированные имена ФД. Необходимость в такой функциональности исходит из того факта, что в программах итоговые выражения-индексы могут иметь довольно сложный вид из-за различных преобразований, выполняемых над выражением в цепочке вызовов, хотя итоговое выражение равносильно линейному преобразованию над переменной-счетчиком, как, например, при инициализации $x[2*i+1]$ с помощью *init_2* в программе на рисунке 16.

Предикаты *df_use_loop_of* и *df_init_loop_of* в качестве параметра также принимают базовое имя ФД и внутри вызывают *df_init_loop* и *df_use_loop*.

Для поиска ошибок, связанных с диапазонами инициализированных и использованных индексов, используется предикат *index_range_mismatch*. В листинге на рисунке 28 приводится код данного предиката за исключением создания словаря *Error*, содержащего информацию об ошибке.

```
01: index_range_mismatch(RootCtx, Error) :-
02:     df_use_loop_of(RootCtx, BaseName, UseRange),
03:     bagof(
04:         InitRange,
05:         (
06:             df_init_loop_of(RootCtx, BaseName, InitRange),
07:             index_range_implies(UseRange, InitRange)
08:         ),
09:         InitRanges
10:     ),
11:     \+ (
12:         member(InitRange, InitRanges),
13:         index_range_covers(InitRange, UseRange)
14:     ),
15:     member(InitRange, InitRanges),
16:     % ...
```

Рисунок 28 — Фрагмент кода предиката *index_range_mismatch*

Данный предикат работает следующим образом: сначала выбирается цикл использования для некоторого базового имени (строка 2); затем все циклы инициализации для этого же базового имени собираются в список (строки 3-10), причем рассматриваются только те циклы инициализации, которые выполняются при условиях, которые следуют из условий цикла использования (строка 7, предикат *index_range_implies*); наконец, выполняется проверка того, что ни один из циклов инициализации не покрывает данный цикл использования (строки 11-14). Дерево вызовов в данном случае позволяет распознавать циклы инициализации и использования независимо от глубины стека вызовов и локального имени ФД, поскольку *BaseName* (строки 2, 6) — реальное базовое имя для ФД, полученное с помощью *Aliases_{СТ}*.

Код далее (начиная со строки 15) отвечает за создание *Error* — словаря, содержащего подробную информацию об ошибке, включающую циклы инициализации и использования, а также код ошибки. В частности, именно здесь выделяются ошибки LUNA18-LUNA22, поскольку для одной пары циклов

инициализации и использования могут иметь место различные ошибки (например, выход как за нижнюю, так и за верхнюю границы).

Предикат *index_range_covers* применяется для сравнения границ циклов с учетом шага и смещения, а также для проверки того, что шаг цикла использования кратен шагу цикла инициализации, и использует библиотеку CLP(FD). Данная библиотека позволяет выполнять сравнения произвольных арифметических выражений, однако в языке LuNA также возможно использовать индексированные имена (например, $x[y[z]][42]$), что делает прямое применение библиотеки невозможным. Поэтому перед выполнением сравнений выражения переписываются без индексированных имен, которые заменяются новыми переменными, причем одним и тем же переменным соответствуют эквивалентные индексированные имена. Процесс переписывания реализован набором предикатов, обрабатывающих различные типы выражений (бинарные операции, переменные, константы и т. д.). Эти предикаты принимают словарь уже введенных переменных и возвращают этот же словарь, в который, возможно, были добавлены новые переменные. В этих словарях ключами выступают имена введенных переменных, а значениями — выражения, которым они соответствуют. Это позволяет при встрече индексированного имени выполнить поиск переменной, соответствующей эквивалентному выражению, и ввести новую переменную только в том случае, если данное индексированное имя ранее не встречалось. Этот процесс рекурсивен, так как для проверки эквивалентности двух индексированных имен, имеющих одно и то же базовое имя и количество индексов, необходимо проверить эквивалентность выражений-индексов на соответствующих позициях, которые в свою очередь тоже могут содержать индексированные имена. Тем не менее, в итоге все сводится к сравнениям арифметических выражений, содержащих только переменные, что реализовано с помощью динамической генерации термов CLP(FD).

Помимо сравнения границ, *index_range_covers* также проверяет, что недостающие индексы, если их число конечно, не инициализируются отдельно

вне циклов. Рассмотрим для примера фрагмент LuNA-программы, приведенный на рисунке 29.

```
01: init(x[0], 0);
02: init(x[1], 1);
03: init(x[N-1], N-1);
04: init(x[N], N);
05: for i = 2..N-2 { init(x[i], i); }
06: for j = 0..N { print(x[j]); }
```

Рисунок 29 — Фрагмент LuNA-программы с инициализацией некоторых индексированных ФД вне цикла

В данном случае для x присутствует цикл использования индексов от 0 до N (строка 6), однако единственный цикл инициализации присваивает значение $x[i]$ для i от 2 до $N-2$ (строка 2). В то же время, $x[0]$, $x[1]$ (строки 1-2), а также $x[N-1]$ и $x[N]$ (строки 3-4) инициализируются отдельно. В данной ситуации предикат *index_range_covers* определит, что с учетом инициализаций одиночных ФД цикл инициализации покрывает цикл использования, поэтому такой код не будет считаться ошибочным. Если же, например, не все недостающие индексы инициализированы, или некоторые индексы инициализированы внутри операторов if, которые имеют условия, выполнение которых не гарантируется при выполнении цикла использования, или нельзя установить диапазон недостающих индексов, то инициализации одиночных ФД игнорируются, а в описании ошибки будет говориться о несоответствии границ циклов.

Также были реализованы предикаты для поиска ошибок LUNA35 (пересечение диапазонов инициализированных индексов) и LUNA37 (отсутствие цикла инициализации). Для обнаружения ошибки LUNA35 также используется библиотека CLP(FD). Рассматриваются пары диапазонов индексов $(Lower_1, Upper_1, Step_1)$ и $(Lower_2, Upper_2, Step_2)$, и динамически генерируется и проверяется ограничение, истинное, когда для некоторых положительных N_1 и N_2 выполнено:

1. $Lower_1 + N_1 * Step_1 = Lower_2 + N_2 * Step_2$;
2. $Lower_1 + N_1 * Step_1 \leq Upper_1$;
3. $Lower_2 + N_2 * Step_2 \leq Upper_2$.

Причем *Lower* и *Upper* — это реальные верхняя и нижняя граница диапазона индексов, а не границы переменной-счетчика соответствующего цикла. Рассматриваются только те циклы, условия выполнения которых не являются взаимоисключающими.

Ошибка LUNA37 имеет место для некоторого ФД, имеющего цикл использования, в одном из двух случаев:

1. В коде программы не существует циклов инициализации для данного ФД. В этом случае в “details” в описании данной ошибки будет присутствовать только поле “used”.
2. Для данного ФД есть циклы инициализации, однако ни один из них не выполняется гарантировано при выполнении данного цикла использования (*index_range_implies*) или не покрывает этот цикл использования полностью (*index_range_covers*). В этом случае в описание ошибки будет включен цикл инициализации,

О том, как реализована работа с условиями, речь пойдет далее.

3.3.3 Поддержка условных операторов

Реализована поддержка оператора *if*, операторов сравнения (*<*, *<=*, *=* и т. д.), а также логических операций (*&&*, *||*). Одной из простейших ошибок, связанных с логическими выражениями, является использование логического значения в числовом контексте. Под числовым контекстом понимается операнд арифметической операции, сравнения, или индекс для индексированного ФД. Классический пример такой ошибки — цепочка сравнений $x < y < z$. В этом случае выражение вычисляется как $(x < y) < z$, и его результат может быть неожиданным для программиста, ранее использовавшего языки, поддерживающие такие сравнения. Как было упомянуто в разделе 2.4, данной ошибке был присвоен код LUNA25, а ее поиск осуществляется локально, то есть на уровне синтаксиса, путем обхода дерева выражения. Стоит отметить, что рассматриваются не только условия операторов *if*, но и выражения, использованные в качестве границ циклов *for*, а также аргументы при вызовах ФК.

Сложной для обнаружения ошибкой является ситуация, когда условие всегда имеет одно и то же значение. В общем случае условия могут содержать константы, арифметические и логические выражения и индексированные ФД. Для работы с условиями применяется библиотека CLP(B) [64, 65], являющаяся частью стандартной библиотеки SWI-Prolog. Однако очевидной проблемой становится то, что, как было упомянуто ранее, условия на языке LuNA не обязательно представляют собой формулы булевой алгебры. Для приведения произвольных условий к виду, с которым способна работать библиотека CLP(B), применяется следующий алгоритм:

1. избавиться от индексированных имен;
2. избавиться от арифметических выражений;
3. выразить сравнения через ' $<$ ', отрицание и логические операции;
4. заменить сравнения логическими переменными.

Шаг 1 выполняется с помощью предикатов, упомянутых при описании *index_range_covers*, а шаг 2 реализуется аналогичным образом, только вместо индексированных имен переменными заменяются арифметические выражения.

Так, например, выражение $(x[N] \geq y + 1 \ \&\& \ x[N + y * 0] < 1 + y) \ \&\& \ y > 5$ превращается в $!Cond_1 \ \&\& \ Cond_1 \ \&\& \ Cond_2$. Целочисленные константы и статически вычисляемые арифметические выражения, используемые как операнды логических операций, заменяются значением своей истинности, что позволяет обрабатывать такие условия, как, например 0 , 1 , $0 * x$, $1 \parallel x$, и многие другие.

Полученная в шаге 4 формула может быть тривиально конвертирована в формат, используемый CLP(B), а для определения тождественной истинности/ложности используется предикат *clpb:taut* из этой библиотеки. Данный предикат имеет два аргумента, первый из которых – формула, а второй – значение ее истинности (0 или 1). Выражение *clpb:taut*(F , T) истинно, если формула F всегда принимает значение T .

Способность проверять тождественную истинность и ложность условий дает возможность проверять, следует ли одно условие из другого, проверяя

истинность импликации соответствующих условий. Это применяется при поиске ошибок LUNA18-LUNA22, и LUNA35: рассматриваются только те циклы инициализации, которые имеют более слабые условия, чем условия данного цикла использования. Один из вариантов ошибки LUNA37 — условия цикла инициализации более сильные, чем условия цикла использования, или не следуют из них.

При реализации данного алгоритма семантически эквивалентные выражения (например, $1 + y$ и $y + 1$) заменяются одними и теми же переменными, что позволяет получить формулу, тождественная истинность или ложность которой будет означать то же самое для изначального условия. К сожалению, обратное не всегда верно. В текущей реализации при замене сравнений переменными не учитываются задаваемые сравнениями интервалы значений переменных: например, $x > 2$, $x > 1$ и $x < -3$ будут тремя независимыми логическими переменными, хотя очевидно, что истинность $x > 2$ гарантирует истинность $x > 1$ и ложность $x < -3$. Тем не менее, предложенный подход позволяет устанавливать тождественную истинность или ложность нетривиальных условий, которые не отмечаются статическими анализаторами как ошибочные в других языках, как, например, $x < y+1 \parallel y+1 < x \parallel x == y+1$.

Одним из возможных способов обхода данного ограничения может быть анализ сравнений и генерация дополнительных ограничений. Так, например, для сравнений одних и тех же переменных с выражениями, которые в свою очередь сравнимы статически (например, целочисленные константы, или линейные выражения), можно при помощи предиката *clpb:sat* задавать отношение между кодирующими сравнения переменными. Рассмотрим условия $(x < 1) \&\& (y > 40)$ и $(x < 2) \&\& (y > 20)$. В настоящее время все четыре сравнения были бы заменены разными логическими переменными, которые можно обозначить как *Less_x_1*, *Greater_y_40*, *Less_x_2* и *Greater_y_20*. Очевидно, что из первого условия следует второе, однако это невозможно установить, рассматривая формулы *Less_x_1* $\&\&$ *Greater_y_40* и *Less_x_2* $\&\&$ *Greater_y_20*, то есть выражение *clpb:taut*(*Less_x_1* * *Greater_y_40* =< *Less_x_2* * *Greater_y_20*, 1) будет ложным

(здесь $*$ обозначает конъюнкцию, $=<$ — импликацию, а 1 вторым аргументом — тождественную истинность условия). В то же время, если добавить $clpb:sat(Less_x_1 =< Less_x_2)$ и $clpb:sat(Greater_y_40 =< Greater_y_20)$. то выражение с предикатом $clpb:taut$, приведенное выше, будет истинным. Реализация данного подхода — одно из возможных направлений дальнейшей работы.

3.4 Интеграция prolog-analyzer в ADAPT

Разработанный статический анализатор prolog-analyzer был интегрирован в единый комплекс автоматизированной отладки LuNA-программ ADAPT (Analyzer, Data dependency graph, Abstract syntax tree, Prolog, Trace), схема работы которого приводится на рисунке 30. Для этого в prolog-analyzer был реализован вывод информации о найденных ошибках в специальном формате, созданном совместно с разработчиками других программных средств статического анализа LuNA-программ (описание приводится в разделе 2.5)

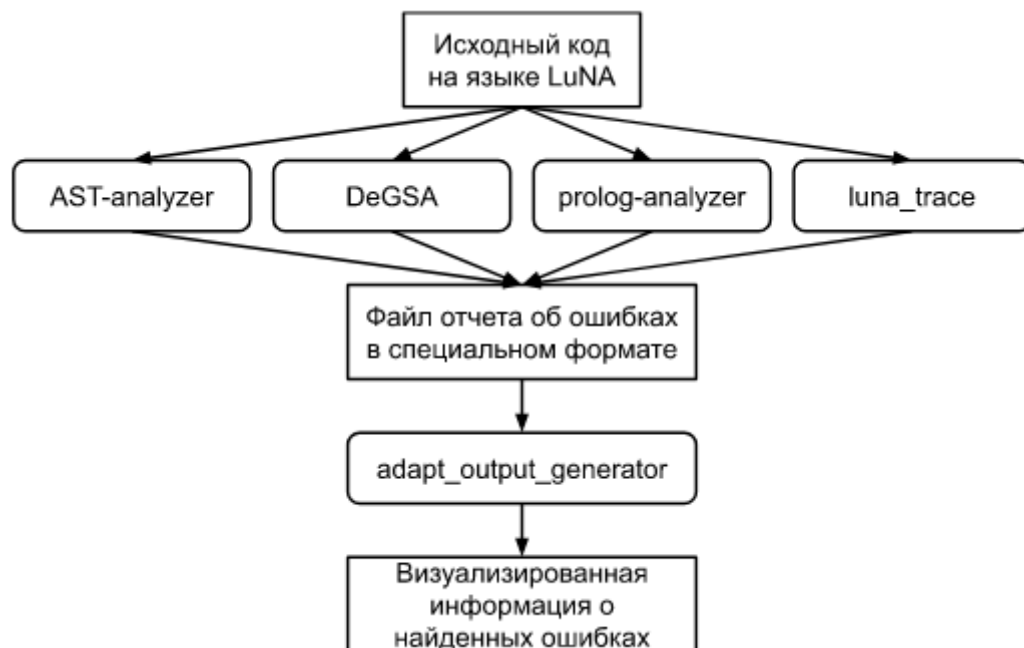


Рисунок 30 — Схема работы ADAPT

В настоящее время в ADAPT входят AST-analyzer, DeGSA и prolog-analyze. Интеграция luna_trace затруднена рядом обстоятельств, разрешение которых выходят за рамки данной работы.

Был создан Python-скрипт *adapt*, объединяющий AST-analyzer, DeGSA и prolog-analyzer. Adapt принимает в качестве аргумента командной строки имя исходного файла на языке LuNA, а также позволяет с помощью параметра командной строки “--run” выбирать, какие из анализаторов будут запущены. Для “--run” доступны значения “ast” (ast-analyzer), “degса” и “prolog” (prolog-analyzer), и если данный параметр не указан ни разу, запущены будут все анализаторы. Adapt также предоставляет возможность указать флаг “--no-cleanup” при запуске, который говорит о том, что сгенерированные в процессе анализа файлы не следует удалять. Adapt автоматизирует запуск выбранных анализаторов и визуализацию информации о найденных ошибках, также вызывая утилиту-визуализатор *adapt_output_generator*.

Prolog-analyzer доступен и как отдельное приложение, и как часть ADAPT (“adapt –run prolog”). Команда prolog-analyzer имеет следующие параметры:

1. “--project-dir” — директория, в которой находится анализируемая LuNA-программа;
2. “--build-dir” — передается компилятору LuNA, директория для генерируемых в процессе компиляции файлов;
3. “--output-dir” — директория для промежуточных файлов, создаваемых в процессе анализа;
4. “--errors-file” — имя файла, в который должен быть записан отчет о найденных ошибках в описанном ранее формате. Если файл не существует, он будет создан, иначе существующий отчет будет дополнен;
5. “--no-cleanup” — не удалять сгенерированные файлы (включая сгенерированные компилятором LuNA).

Остальные аргументы (как правило имена файлов исходного кода на LuNA) prolog-analyzer передает компилятору LuNA.

Было принято участие в разработке утилиты визуализации отчетов о найденных ошибках, названной *adapt_output_generator*. Данная утилита предоставляет интерфейс командной строки и в качестве входа принимает файл

отчета и путь до директории, в которой расположена LuNA-программа (“--project-dir”), а также через опциональный параметр “--output” имя выходного файла (по умолчанию используется стандартный поток вывода). В `adapt_output_generator` реализованы функции представления типовых объектов, о которых было упомянуто ранее, а для визуализации информации об ошибках результаты работы этих функций подставляются в шаблоны сообщений об ошибках, выбираемые по коду ошибки. Вклад данной работы состоит в добавлении вывода строк исходного кода LuNA-программ в вывод, реализации части функций представления объектов, а также добавлении информации о том, соответствуют ли коды ошибок предупреждениям (warning) или именно ошибкам (error).

В разделе 4.1 будут приведены примеры вывода ADAPT для типов ошибок, обнаруживаемых `prolog-analyzer`.

Глава 4. Тестирование

4.1 Проверка функциональных возможностей

Для проверки способности prolog-analyzer обнаруживать семантические ошибки была разработана утилита автоматизированного, аналогичная используемой для тестирования системы LuNA. Тест представляет собой директорию, содержащую файл конфигурации теста `test_config.json`. Конфигурация теста описывается в формате JSON и состоит из четырех основных частей:

1. Набор команд, которые нужно выполнить перед тестом (поле “`arrange`”).
2. Команда, работа которой тестируется (поле “`act`”).
3. Проверки, которые необходимо выполнить над выводом тестируемой команды (поле “`assert`”).
4. Список имен потоков вывода (“`stderr`”, “`stdout`”), которые нужно сохранить в файлы на диске (поле “`save`”).

Проверки могут производиться как над стандартным потоком вывода (`stdout`), так и над стандартным потоком ошибок (`stderr`). Проверки включают в себя:

- указание шаблонов как в виде регулярных выражений, так и в виде упрощенной версии шаблонов поиска (`wildcard pattern`, `glob pattern`) при использовании префикса “`ws:`” для отдельных строк или вывода целиком;
- указание пути к файлу, содержимое которого должно совпадать с выводом команды;
- должен ли быть поток вывода пустым/не пустым.

При выполнении теста сначала выполняются команды из п. 1. Затем выполняется команда из п. 2. После выполняются проверки из п. 3 над выводом из п. 2. Если одна из выполняемых команд завершается с ненулевым кодом возврата, либо одна из проверок не проходит, тест считается не пройденным. Если же все проверки проходят успешно, тест считается пройденным.

Для тестирования prolog-analyzer каждая тестовая директория содержит LuNA-программу и конфигурацию теста. В конфигурации командой, работа

которой проверяется, является вызов `adapt` с параметром `--run` равным `“prolog”` и именем файла исходного кода на языке LuNA. Для программ, в которых нет обнаруживаемых `prolog-analyzer` семантических ошибок, проверяется, что стандартный поток вывода содержит сообщение `“No errors found”`. Для программ, содержащих ошибки, конфигурация содержит набор шаблонов для стандартного потока вывода, проверяющих, что было найдено ожидаемое количество ошибок (например, `“wc:Found 4 errors”`), и что все важные части сообщения о заложенных в программу ошибках присутствуют в выводе (например, `“wc:error[LUNA19]*DF x”`).

В набор тестов были включены программы, приведенные на рисунках 9 (LUNA21), 11 (LUNA35), 12 (LUNA21), 18 (LUNA21, LUNA35) и 24 (LUNA18, LUNA21), а также не содержащие ошибок версии программ на рисунках 11 (условия циклов инициализации взаимоисключающие) и 18 (отсутствует смещение индекса на `-1` при инициализации). На рисунке 31 приводится фрагмент вывода ADAPT для программы на рисунке 11, включающий информацию об одном из циклов инициализации `x`.

```
Found 1 errors:
(1) warning[LUNA35]: Possible overlap of initialization loops for DF x.
Initialized:
DF x[i] as x[i] in:
  File "./main.fa", line 4, in main
    sub main()
  File "./main.fa", line 7, in if
    if x [0] > 0
  File "./main.fa", line 8, in for
    for i = 1..5 { assign_int(x[i], i * i); }
  File "./main.fa", line 8, in assign_int
    for i = 1..5 { assign_int(x[i], i * i); }

With i from 1 to 5, step 1 and offset 0
Note: x declared in:
  File "./main.fa", line 4, in main
    sub main()
```

Рисунок 31 — Фрагмент вывода ADAPT для ошибки LUNA35

Для других ошибок также были написаны тестовые программы, проверяющие различную функциональность prolog-analyzer при их обнаружении. Так, например, была реализована программа, в которой часть ФД с индексированными именами инициализируется вне циклов, как внутри операторов *if* с различными условиями, так и без условий. Для ошибки LUNA22 были реализованы две версии LuNA-программы, одна из которых содержала данную ошибку. Для ошибки LUNA37 были реализованы три тестовые программы: первая не содержит ошибок (для всех циклов использования есть циклы инициализации с подходящими условиями), во второй циклы инициализации отсутствуют, в третьей — циклы инициализации выполняются при условиях, которые не следуют из условий, при которых выполняются циклы использования соответствующих ФД.

Для ошибок LUNA23 и LUNA25 были реализованы пары корректных и ошибочных программ. Тесты с ошибочными программами проверяют способность prolog-analyzer обнаруживать различные варианты этих ошибок. На рисунке 32 приведен листинг LuNA-программы, в которой в качестве условий в операторах *if* использованы как тождественно истинные, так и тождественно ложные выражения (LUNA23).

```

01: C++ sub consume(int x) ${{{$}}
02:
03: sub main() {
04:     df x, y, N;
05:     if x || 1 { consume(x); }
06:     if x < y+1 || y+1 < x || x == y+1 {
07:         consume(x);
08:     }
09:     if (x[N] >= y + 1 && x[N + y * 0] < 1 + y) && y > 5 {
10:         consume(x);
11:     }
12: }

```

Рисунок 32 — LuNA-программа, содержащая ошибку LUNA23

На рисунке 33 приводится вывод анализатора для программы из рисунка 32.

```

Found 3 errors:
(1) warning[LUNA23]: Condition (x || 1) is always True.
In:
  File "./example.fa", line 5, in if
    if x || 1 { consume(x); }

(2) warning[LUNA23]: Condition (((x < (y + 1)) || ((y + 1) < x)) || (x == (y + 1)))
is always True.
In:
  File "./example.fa", line 6, in if
    if x < y+1 || y+1 < x || x == y+1

(3) warning[LUNA23]: Condition (((x[N] >= (y + 1)) && (x[(N + (y * 0))] < (1 + y)))
&& (y > 5)) is always False.
In:
  File "./example.fa", line 9, in if
    if (x[N] >= y + 1 && x[N + y * 0] < 1 + y) && y > 5

```

Рисунок 33 — Результат анализа программы на рисунке 32

На рисунке 34 приведен листинг LuNA-программы, в которой логическое значение используется в числовом контексте (LUNA25): цепочка сравнений в качестве условия в *if* (строка 6), а также использование результата сравнения в качестве индекса и операнда умножения (строка 10).

```

01: C++ sub consume(int x) ${{{$}}
02:
03: sub main() {
04:     df x, y, N;
05:
06:     if 0 < x < N {
07:         consume(x);
08:     }
09:
10:     consume(1 + x[42][x < y][(y == x) * 7]);
11: }

```

Рисунок 34 — LuNA-программа, содержащая ошибки LUNA25

На рисунке 35 приводится вывод анализатора для данной программы.

Found 3 errors:

(1) warning[LUNA25]: Result of boolean operator is used where number is expected:
((0 < x) < N), argument at index 0.

In:

```
File "./example.fa", line 6, in if
  if 0 < x < N
```

(2) warning[LUNA25]: Result of boolean operator is used where number is expected:
((y == x) * 7), argument at index 0.

In:

```
File "./example.fa", line 10, in consume
  consume(1 + x[42][x < y][(y == x) * 7])
```

(3) warning[LUNA25]: Result of boolean operator is used where number is expected:
x[42][(x < y)][(y == x) * 7], argument at index 1.

In:

```
File "./example.fa", line 10, in consume
  consume(1 + x[42][x < y][(y == x) * 7])
```

Рисунок 35 — Результат анализа программы на рисунке 34

Разработанный статический анализатор также тестировался на программе из набора тестов системы LuNA, реализующей алгоритм блочного умножения матриц. В этой программе prolog-analyzer обнаруживает ошибку LUNA19 — выход за границу диапазона инициализированных индексов снизу. На рисунке 36 приведен фрагмент программы, содержащий ошибку, а на рисунке 37 — сокращенный вывод анализатора.

```

21: sub calc_mat(name A, name B, name C, int i, int j, int N)
22: {
23:     df Ctmp, Csum;
24:
25:     for k=0..N-1
26:     {
27:         cf f[i][j][k]: mult_mat(A[i][k], B[k][j],
Ctmp[k]);
28:     }
29:
30:     if N>1
31:         sum_mat(Ctmp[0], Ctmp[1], Csum[1]);
32:     if N==1
33:         copy_mat(Ctmp[0], Csum[0]);
34:
35:     for k=2..N-1
36:         cf sum[k]: sum_mat(Ctmp[k], Csum[k-1], Csum[k]);
37:
38:     copy_mat(Csum[N-1], C);
39: }

```

Рисунок 36 — Фрагмент LuNA-программы, реализующей алгоритм блочного умножения матриц

Found 1 errors:

(1) error[LUNA19]: Lower bound of consumption loop is always less than lower bound of initialization loop for DF Csum.

Used:

DF Csum[(k - 1)] as Csum[(k - 1)] in:

...

File "./main.fa", line 35, in for
for k=2..N-1

File "./main.fa", line 36, in sum_mat
cf sum[k]: sum_mat(Ctmp[k], Csum[k-1], Csum[k])

With k from 2 to (N - 1), step 1 and offset -1

...

Initialized:

DF Csum[k] as Csum[k] in:

...

File "./main.fa", line 35, in for
for k=2..N-1

File "./main.fa", line 36, in sum_mat
cf sum[k]: sum_mat(Ctmp[k], Csum[k-1], Csum[k])

With k from 2 to (N - 1), step 1 and offset 0

...

Рисунок 37 — Результат анализа программы на рисунке 36 (сокращенный)

Данный пример ложноположительного срабатывания интересен с точки зрения дальнейшего развития prolog-analyzer. В текущей реализации ошибка вне всяких сомнений должна быть обнаружена, ведь ФД *Csum[1]* инициализируется вне цикла при условии $N > 1$ (строки 30-31), тогда как цикл, в котором *Csum[1]* используется на первой итерации (строки 35-36) выполняется без условий. В то же время границы цикла задают неявное условие на его выполнение: необходимо, чтобы значение $N - 1$ было больше либо равно 2, или, что эквивалентно, чтобы значение N было больше 2 (предполагая целочисленный тип данных). Таким образом, учет неявных условий, задаваемых границами циклов, вместе с подходом к работе с условиями, задающими диапазоны значений, предложенным в разделе 3.3.3, могут помочь избавиться от ложноположительных срабатываний в ситуациях, аналогичных данной. Тем не менее, исходная LuNA-программа может быть модифицирована так, чтобы цикл использования *Csum* имел явное условие, и такая программа не будет содержать ошибок с точки зрения текущей версии prolog-analyzer. Пример модифицированного ФК *calc_mat* приведен на рисунке 38.

```

21: sub calc_mat(name A, name B, name C, int i, int j, int N)
22: {
23:     df Ctmp, Csum;
24:
25:     for k=0..N-1
26:         cf f[i][j][k]: mult_mat(A[i][k], B[k][j],
Ctmp[k]);
27:
28:     if N>1
29:     {
30:         sum_mat(Ctmp[0], Ctmp[1], Csum[1]);
31:         for k=2..N-1
32:             cf sum[k]: sum_mat(Ctmp[k], Csum[k-1],
Csum[k]);
33:     }
34:
35:     if N==1
36:         copy_mat(Ctmp[0], Csum[0]);
37:
38:     copy_mat(Csum[N-1], C);
39: }
```

Рисунок 38 — Фрагмент модифицированной LuNA-программы, реализующей алгоритм блочного умножения матриц

4.2 Оценка времени анализа программ

Время анализа было измерено на примере двух LuNA-программ, входящих в набор тестов системы LuNA. Первая программа реализует алгоритм блочного умножения матриц и включает в себя 84 строки исходного кода на языке LuNA. Вторая программа реализует метод прогонки, исходный код на языке LuNA насчитывает 484 строки. Для тестирования операторы LuNA, не поддерживаемые `prolog-analyzer`, были удалены или заменены поддерживаемыми без изменения итогового объема программы в строках.

Измерения производились на машине со следующей конфигурацией:

- ОС: Ubuntu 22.04.3 LTS.
- Процессор: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz.
- Число физических ядер: 2.
- Число логических процессоров: 4.
- RAM: 6,0 ГБ DDR3.

Производилось измерение:

1. общего времени работы команды `prolog-analyzer`;
2. времени этапов работы:
 - a. компиляции LuNA-программы;
 - b. генерации логического представления;
 - c. анализа и генерации отчета о найденных ошибках.

Для этапа анализа отдельно измерялось время загрузки логического представления программы, поиска отдельных ошибок, а также вывода информации о найденных ошибках. Для измерения времени в рамках программы `prolog-analyzer` использовалась функция `default_timer` модуля `timeit` стандартной библиотеки Python, возвращающая значения таймера с максимально доступным разрешением. В рамках программы-анализатора на Prolog использовался предикат `call_time`, возвращающий различные метрики выполнения переданного выражения, включая реально прошедшее время.

4.2.1 Блочное умножение матриц

Для LuNA-программы, реализующей блочное умножение матриц, общее время работы составило 5.5 секунд. Время компиляции составило 4.04 секунды, время генерации логического представления программы — 0.2, а время анализа — 0.84 секунды. Остальное время, по всей видимости, ушло на запуск интерпретатора Python, а также на удаление сгенерированных в процессе работы файлов. Время работы программы-анализатора, измеренное с помощью *call_time*, составило 0.044 секунды. На рисунках 39 и 40 представлено распределение времени выполнения этапов работы программы анализатора и время поиска отдельных ошибок соответственно.

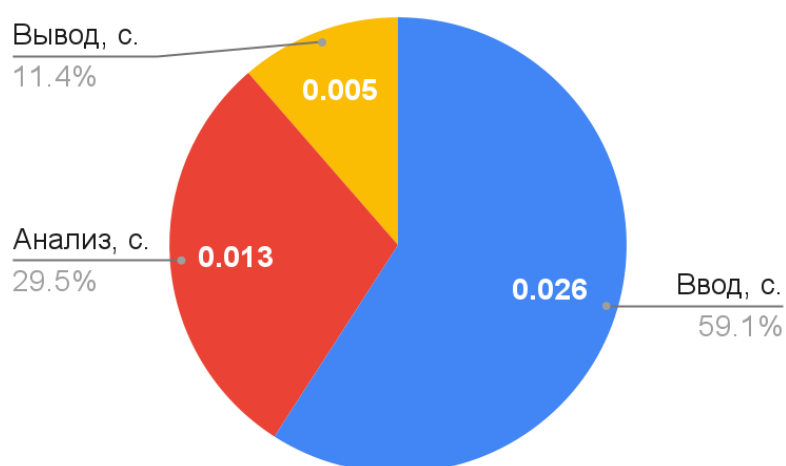


Рисунок 39 — Распределение времени этапов работы программы-анализатора

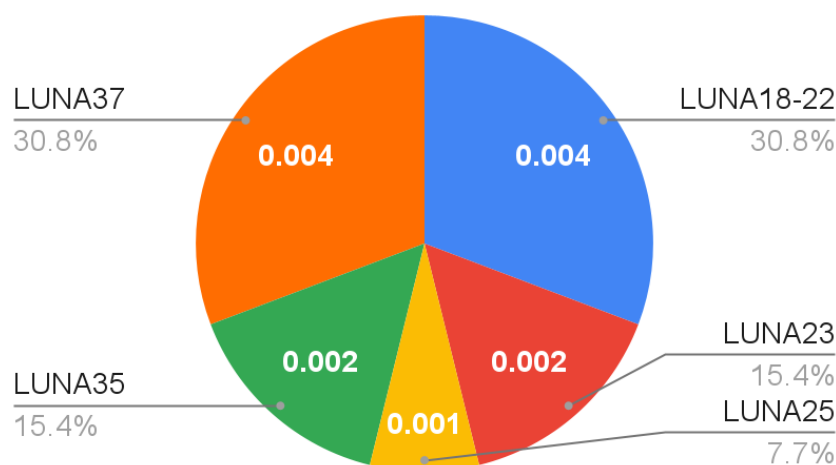


Рисунок 40 — Распределение времени поиска различных типов ошибок

Как можно заметить, общее время работы программы-анализатора (0.84 секунды) на порядок превосходит время, измеренное самой программой-анализатором (0.044 секунды). Это объясняется тем, что запуск SWI-Prolog и загрузка необходимых модулей оказываются весьма долгим процессом. Для сравнения было измерено время работы измененной версии программы-анализатора, из которой был удален весь код, отвечающий за анализ, а оставлены были только инструкции по загрузке модулей. Время работы такой программы составило порядка 0.80-0.85 секунд, что примерно соответствует приведенным ранее измерениям.

4.2.2 Метод прогонки

Всего время работы prolog-analyzer для данной работы составило 23.68 секунд, из них 22.1 секунду заняла компиляция, 0.27 секунд — генерация логического представления программы и 0.91 секунда ушла на анализ. Время работы, измеренное программой-анализатором, составило 0.092 секунды. На рисунках 41 и 42 представлено распределение времени выполнения этапов работы программы анализатора и время поиска отдельных ошибок соответственно.

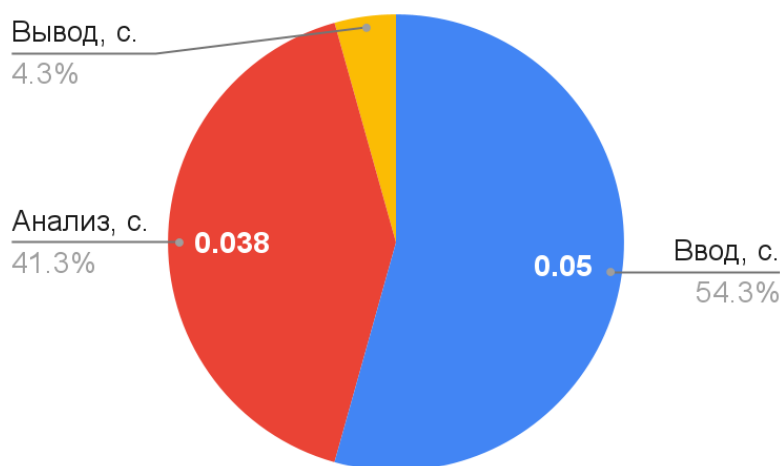


Рисунок 41 — Распределение времени этапов работы программы-анализатора

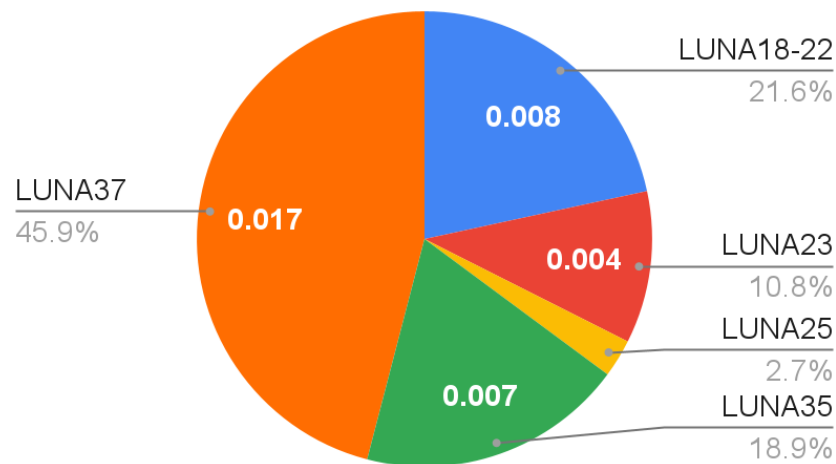


Рисунок 42 — Распределение времени поиска различных типов ошибок

4.2.3 Направления оптимизации

Большую часть времени работы prolog-analyzer занимает компиляция LuNA-программы (4 из 5.5 секунд и 22 из 23.7 секунды для приведенных программ). В настоящее время компилятор LuNA рассматривается как черный ящик, однако в будущем возможно провести исследование времени компиляции и выделить наиболее долгие этапы. Вероятно, причина большого времени компиляции кроется в необходимости сборки разделяемой библиотеки из сгенерированного кода ФВ на C++. Язык C++ известен как высокой производительностью программ, являющейся результатом оптимизаций, применяемых компиляторами, так и большим временем компиляции [66] даже для небольших программ. Если возможно использовать только те модули компилятора LuNA, которые ответственны за генерацию представления программы в формате JSON Algorithm, это может серьезно сократить время анализа.

Другим фактором, замедляющим анализ, является постоянное для всех программ время запуска программы-анализатора на Prolog, на которое уходит порядка 800 миллисекунд даже без выполнения каких-либо полезных действий. Возможным решением может быть использование клиент-серверной архитектуры: программа-анализатор на Prolog будет запущена единожды как фоновый процесс (сервер), а пользователь будет запускать клиент,

выполняющий генерацию логического представления LuNA-программ и отправку его серверу-анализатору.

Процесс поиска ошибок также может быть оптимизирован. Во-первых, дерево вызовов, используемое при поиске LUNA18-LUNA23, LUNA35 и LUNA37, можно строить единожды перед запуском анализа. В нем также можно сохранять часто используемую информацию, как, например, типы параметров ФК. Во-вторых, поиск различных типов ошибок может производиться параллельно с помощью доступных в стандартной библиотеке SWI-Prolog средств многопоточности.

ЗАКЛЮЧЕНИЕ

В рамках работы был выполнен обзор существующих средств и методов отладки параллельных программ, а также программных средств, использующих подход логического метапрограммирования при анализе программ. Было принято участие в составлении базы ошибок в LuNA-программах. Был разработан статический анализатор *prolog-analyzer*, использующий логическое представление LuNA-программ на языке Prolog. Данный анализатор способен обнаруживать ряд семантических ошибок, связанных с повторной инициализацией и использованием неинициализированных индексированных фрагментов данных. *Prolog-analyzer* также способен обнаруживать тождественно истинные и ложные формулы в условиях и случаи использования значений логического типа там, где ожидалось число. Разработанный статический анализатор был протестирован на ряде LuNA-программ, содержащих ошибки.

Было выполнено тестирование с целью оценки времени анализа программ и распределения времени выполнения отдельных шагов анализа. По результатам тестирования были предложены возможные пути оптимизации времени работы.

По теме работы была опубликована статья [13], а результаты работы были представлены на конференции ПаВТ-2024 в виде стендового доклада [14] и на МНСК-2024 в виде доклада, получившего диплом третьей степени.

Направления дальнейшей работы включают расширение функциональности разработанного статического анализатора, модификацию алгоритмов поиска ошибок для снижения числа ложноположительных результатов, а также оптимизацию отдельных шагов анализа для уменьшения общего времени работы.

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».

Мичуров Михаил Антонович

ФИО студента

Подпись студента

« ____ » _____ 20 ____ г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Перепелкин В. А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах // Проблемы информатики. — 2020. — №1 (46). — С. 66-74
2. Перепелкин В. А., Софронов И. В., Ткачева А. А. Автоматизация конструирования численных параллельных программ с заданными нефункциональными свойствами на базе вычислительных моделей // Проблемы информатики. — 2017. — №4 (37). — С. 47-60
3. Маркова В. П., Остапкевич М. Б. Сравнение возможностей MPI и LuNA на примере реализации модели клеточно-автоматной интерференции волн // Проблемы информатики. — 2017. — №. 2 (35). — С. 53-64
4. Власенко А. Ю., Мичуров М. А., Мустафин Д. Э. Автоматизация отладки и балансировки нагрузки во фрагментированных программах // Проблемы информатики. — 2022. — № 3. — С. 61-76
5. Malyshkin V., Vlasenko A., Michurov M. Automated Debugging of Fragmented Programs in LuNA System // Mathematical Modeling and Supercomputer Technologies. MMST 2022. Communications in Computer and Information Science. — 2022. — Volume 1750. — С. 266–280.
6. Власенко А. Ю. Автоматизированный контроль корректности MPI-программ на основе шаблонов ошибочного поведения / А. Ю. Власенко, Томск: [б. и.]. — 2014.
7. Афанасьев К. Е., Власенко А. Ю. Семантические ошибки в параллельных программах для систем с распределенной памятью и методы их обнаружения современными средствами отладки // Вестник КемГУ. — 2009. — №2. — С. 13-20.
8. Кропачева М. С. Формальная верификация параллельных программ // Космические аппараты и технологии. — 2012. — №. 2. — С. 35-38.

9. Рыжков Е. А., Середин О. С. Применение технологии статического анализа кода при разработке параллельных программ // Известия Тульского государственного университета. Технические науки. — 2008. — №. 3. — С. 191-197.
10. Колосов А. П. Методы отладки параллельных программ // Известия Тульского государственного университета. Технические науки. — 2010. — №. 2-2. — С. 172-180.
11. Власенко А. Ю. Модель масштабируемой системы автоматического контроля корректности параллельных программ // Вестник Новосибирского государственного университета. Серия: Информационные технологии. — 2009. — Т. 7. — №. 4. — С. 53-65.
12. Static Analyzer Options (Using the GNU Compiler Collection (GCC)) [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: <https://gcc.gnu.org/onlinedocs/gcc-10.1.0/gcc/Static-Analyzer-Options.html> — Загл. с экрана (дата обращения: 20.05.2024).
13. Мичуров М. А. Статический анализ фрагментированных программ с использованием Prolog // сборник материалов IV Международной научно-практической конференции «Вызовы глобализации и развитие цифрового общества в условиях новой реальности». — 2022. — С. 142-152.
14. Мичуров М. А. Разработка инструмента статического анализа фрагментированных программ с использованием Prolog // Параллельные вычислительные технологии – XVIII всероссийская конференция с международным участием, ПаВТ'2024, г. Челябинск, 2–4 апреля 2024 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2024. — 196 с. — С. 188.
15. PVS-Studio Messages [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: <https://pvs-studio.com/en/docs/warnings/> — Загл. с экрана (дата обращения: 20.05.2024).
16. Clang-Tidy — Extra Clang Tools 19.0.0 git documentation [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа:

- <https://clang.llvm.org/extra/clang-tidy/> — Загл. с экрана (дата обращения: 20.05.2024).
17. Andreev D. Yu., Antonov A. S., Voevodin Vad. V., Zhumatii S. A., Nikitenko D. A., Stefanov K. S., Shvets P. A. A system for the automated finding of inefficiencies and errors in parallel programs // Numerical methods and programming. — 2013. — Volume 14, Issue 3. — С. 48–53.
 18. Средства отладки MPI-программ. Анализатор корректности [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: https://keldysh.ru/papers/2006/prep28/prep2006_28.html — Загл. с экрана (дата обращения: 20.05.2024).
 19. Wögerer W. A survey of static program analysis techniques // Tech. rep., Technische Universität Wien. [Электронный ресурс]. — 2005. — Режим доступа: https://www.academia.edu/73238641/A_Survey_of_Static_Program_Analysis_Techniques. — Загл. с экрана. (дата обращения: 20.05.2024).
 20. Martignano M. A "New" C Static Analyzer: the Compiler // ACM SIGAda Ada Letters. — 2020. — № 2 (39). — С. 25–28.
 21. TotalView Debugger [Электронный ресурс] — Режим доступа: <https://totalview.io/free-trial> (дата обращения: 20.05.2024).
 22. Киселев А. Б., Киселев С. Н. Отладчик параллельных программ для ОС Linux // Труды Института системного программирования РАН. — 2020. — Т. 32. — №. 4. — С. 97-114.
 23. DDT - Distributed Debugging Tool [Электронный ресурс] — Режим доступа: <https://www.hpc2n.umu.se/resources/software/ddt> — Загл. с экрана (дата обращения: 20.05.2024).
 24. Крюков В. А. Разработка параллельных программ для вычислительных кластеров и сетей // Информационные технологии и вычислительные системы. — 2003. — Т. 1. — С. 2.

- 25.Бахтин В. А. [и др.]. Сравнительная отладка OpenMP-программ // Препринты Института прикладной математики им. МВ Келдыша РАН. — 2009. — №. 0. — С. 76-24.
- 26.Алексахин В. А. [и др.]. Средства отладки OpenMP программ в DVM-системе // Научный сервис в сети Интернет: решение больших задач. — 2008. — С. 281-285.
- 27.Get Started with Intel® Trace Analyzer and Collector [Электронный ресурс] — Режим доступа: <https://www.intel.com/content/www/us/en/docs/trace-analyzer-collector/get-started-guide/2021-10/overview.html> — Загл. с экрана (дата обращения: 20.05.2024).
- 28.Krammer B. et al. MARMOT: An MPI analysis and checking tool // Advances in Parallel Computing. – North-Holland, 2004. – Т. 13. – С. 493-500.
- 29.Luecke G. et al. MPI-CHECK: a tool for checking Fortran 90 MPI programs // Concurrency and Computation: Practice and Experience. — 2003. — Т. 15. — №. 2. — С. 93-100.
- 30.Cppcheck A tool for static C/C++ code analysis [Электронный ресурс] — Режим доступа: <http://cppcheck.net/> (дата обращения: 20.05.2024).
- 31.Clang Static Analyzer [Электронный ресурс] — Режим доступа: <https://clang-analyzer.llvm.org/> (дата обращения: 20.05.2024).
- 32.Spin, open-source verification software [Электронный ресурс] — Режим доступа: <https://spinroot.com/spin/whatispin.html> (дата обращения: 20.05.2024).
- 33.Siegel S. F. Verifying parallel programs with MPI-Spin //European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting. — Springer, Berlin, Heidelberg, 2007. — С. 13-14.
- 34.Трифанов В. Ю., Цителов Д. И. Статические и post-mortem методы обнаружения гонок в параллельных программа // КИО. — 2011. — №5. — С. 3-13

35. Kavi M., Moshtahi A. Modeling Multithreaded Applications Using Petri Nets. International Journal of Parallel Programming. — 2002. — Volume 30, № 5. — P. 353-371.
36. Voron J., Kordon F. Evinrude: A Tool To Automatically Transform Program's Sources into Petri Nets // Petri Nets Newsletter. — 2008. — Volume 75. — P. 19-38.
37. Cppcheck: cfg/openmp.cfg | Fossies [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: <https://fossies.org/linux/cppcheck/cfg/openmp.cfg> (дата обращения: 20.05.2024).
38. Alpha Checks [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: https://clang-analyzer.llvm.org/alpha_checks.html (дата обращения: 20.05.2024).
39. clang-tidy - concurrency-thread-canceltype-asynchronous — Extra Clang Tools 19.0.0git documentation [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: <https://clang.llvm.org/extra/clang-tidy/checks/concurrency/thread-canceltype-asynchronous.html> (дата обращения: 20.05.2024).
40. V1089. Waiting on condition variable without predicate. A thread can wait indefinitely or experience a spurious wake-up. [Электронный ресурс] — Электрон. текстовые дан. — Режим доступа: <https://pvs-studio.com/en/docs/warnings/v1089/> (дата обращения: 20.05.2024).
41. Иртегов, Д. В. Введение в операционные системы. — 2-е изд. — Санкт-Петербург: БХВ-Петербург, 2008.
42. Карпов, Ю. Г. MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010. — 560 с.
43. Tantius R., Speicher D., Behrend A. Towards an Application of Update Propagation on Logic Programs Representing Java Source Code // 26th Workshop on Logic Programming. — 2013.

- 44.Appeltauer M., Kniesel G. Towards Concrete Syntax Patterns for Logic-based Transformation Rules // Electronic Notes in Theoretical Computer Science. — 2008. — (219). — C. 113–132.
- 45.Roover C. D. A logic meta-programming foundation for example-driven pattern detection in object-oriented programs // Proceedings of the 2011 27th IEEE International Conference on Software Maintenance. — 2011. — C. 556–561.
- 46.Tourwe T., Mens T. Identifying refactoring opportunities using logic meta programming // Software Maintenance and Reengineering, 2003. Proceedings. — 2003. — C. 91–100.
- 47.Wuyts R. A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation // PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory. — 2001.
- 48.De Volder K. Type-Oriented Logic Meta Programming // PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory. — 1998.
- 49.McCormick E., De Volder K. JQuery: finding your way through tangled code // Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. — 2004. — C. 9–10.
- 50.Cohen T., Gil J. (Yossi), Maman I. JTL: the Java tools language // ACM SIGPLAN Notices. — 2006. — № 10 (41). — C. 89–108.
- 51.De Roover C. [и др.]. The SOUL tool suite for querying programs in symbiosis with Eclipse // Proceedings of the 9th International Conference on Principles and Practice of Programming in Java. — 2011. — C. 71–80.
- 52.Hajiyev E., Verbaere M., De Moor O. codeQuest: Scalable Source Code Queries with Datalog // Lecture Notes in Computer Science / под ред. D. Thomas, Berlin, Heidelberg: Springer Berlin Heidelberg. — 2006. — C. 2–27.
- 53.Crew R. F. ASTLOG: a language for examining abstract syntax trees // Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997. — USA: USENIX Association. — 1997. — C. 18.

54. Kniesel G., Hannemann J., Rho T. A comparison of logic-based infrastructures for concern detection and extraction // Proceedings of the 3rd workshop on Linking aspect technology and evolution. — 2007. — С. 6.
55. SWI-Prolog [Электронный ресурс] — Режим доступа: <https://www.swi-prolog.org/> (дата обращения: 20.05.2024).
56. TyRuBa -- Logic Meta Programming for Java [Электронный ресурс] — Режим доступа: <https://tyruba.sourceforge.net/> (дата обращения: 20.05.2024).
57. Flederer F. [и др.]. Source Code Verification for Embedded Systems using Prolog // Electronic Proceedings in Theoretical Computer Science. — 2017. — (234). — С. — 88–103.
58. 58 Ciraci S., Van Den Broek P., Aksit M. Graph-based verification of static program constraints // Proceedings of the 2010 ACM Symposium on Applied Computing. — Sierre Switzerland: ACM. — 2010. — С. 2265–2272.
59. Boehm A. M. [и др.]. A Tool for Analyzing and Tuning Relational Database Applications: SQL Query Analyzer and Schema EnHancer (SQUASH) // Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken (18th GI-Workshop on the Foundations of Databases), Wittenberg, Sachsen-Anhalt, Deutschland, 6.-9. Juni 2006 / под ред. S. Brass, A. Hinneburg, Institute of Computer Science, Martin-Luther-University. — 2006. — С. — 45–49.
60. Smaragdakis Y., Bravenboer M. Using Datalog for Fast and Easy Program Analysis // Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers / под ред. O. De Moor [и др.], Berlin, Heidelberg: Springer Berlin Heidelberg. — 2011. — С. 245–251.
61. Dura A., Reichenbach C., Söderberg E. JavaDL: automatically incrementalizing Java bug pattern detection // Proceedings of the ACM on Programming Languages. 2021. — № OOPSLA (5). — С. — 1–31.
62. Введение в JSON [Электронный ресурс] — Режим доступа: <https://www.json.org/json-ru.html> (дата обращения: 20.05.2024).

- 63.SWI-Prolog -- library(clpfd) // SWI-Prolog [Электронный ресурс] — Режим доступа: <https://www.swi-prolog.org/pldoc/man?section=summary-lib-clpfd> (дата обращения 20.05.2024)
- 64.Triska M. Boolean constraints in SWI-Prolog: A comprehensive system description // Science of Computer Programming. — 2018. — (164). — С. 98–115.
- 65.Triska M. The Boolean Constraint Solver of SWI-Prolog (System Description) // International Symposium on Functional and Logic Programming / под ред. О. Kiselyov, A. King, Cham: Springer International Publishing. — 2016. — С. 45–61.
- 66.Mivelli D. Analyzing and Reducing Compilation Times for C++ Programs (Dissertation) [Электронный ресурс] — Электрон. дан. — Режим доступа: <https://www.diva-portal.org/smash/get/diva2:1671584/FULLTEXT01.pdf> (дата обращения 20.05.2024)

ПРИЛОЖЕНИЕ А

Типы ошибок в LuNA-программах

- LUNA01. Несоответствие типов аргументов при вызове атомарного ФК
- LUNA02. Несуществующая LuNA-подпрограмма
- LUNA03. Повторная инициализация ФД
- LUNA04. Несоответствие типов аргументов при вызове структурированного ФК (неверный порядок аргументов)
- LUNA05. Попытка использования неинициализированного ФД
- LUNA06. Несовпадение количества аргументов при объявлении ФК и его вызове
- LUNA07. Два или более объявления ФД в подпрограмме
- LUNA08. Несоответствие типов LuNA при присваивании
- LUNA09. Использование ФД после его удаления
- LUNA10. Неиспользуемое имя
- LUNA11. Включение нескольких разных функций по одним именам
- LUNA12. Отсутствие функции main
- LUNA13. ФД с одинаковыми названиями в одной области видимости
- LUNA14. Попытка использования необъявленного ФД
- LUNA15. Циклическая зависимость по данным
- LUNA16. Повторное определение ФК
- LUNA17. Включение несуществующей C++-функции
- LUNA18. Нижняя граница цикла использования может быть как больше, так и меньше нижней границы цикла инициализации
- LUNA19. Нижняя граница цикла использования строго меньше нижней границы цикла инициализации
- LUNA20. Верхняя граница цикла использования может быть как меньше, так и больше верхней границы цикла инициализации
- LUNA21. Верхняя граница цикла использования строго больше верхней границы цикла инициализации
- LUNA22. Шаг цикла использования не кратен шагу цикла инициализации

- LUNA23. Формула в if тождественно истинна/ложна
- LUNA24. Формула в if истинна/ложна во всех путях выполнения
- LUNA25. Операторы, возвращающие логическое значение, используются в целочисленном контексте
- LUNA26. Попытка инициализации объекта, не являющегося ФД
- LUNA27. Попытка запросить неинициализированный ФД при помощи request
- LUNA28. Попытка использования ФД после превышения допустимого числа запросов
- LUNA29. Использование ФД после его удаления при помощи соответствующего оператора
- LUNA30. Использование оператора информационной зависимости для структурированного ФК
- LUNA31. Не передать значение для nparam
- LUNA32. Попытка запросить ФД из узла, где его нет
- LUNA33. Неправильный параметр для nparam
- LUNA34. Безусловная рекурсия
- LUNA35. Пересечение диапазонов инициализированных индексов
- LUNA36. Попытка индексации объекта, не являющегося ФД
- LUNA37. Нет (гарантированного) цикла инициализации

ПРИЛОЖЕНИЕ Б

СТАТИЧЕСКИЙ АНАЛИЗАТОР PROLOG-ANALYZER РУКОВОДСТВО ОПЕРАТОРА

Листов 8

Новосибирск 2024

93

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению и эксплуатации статического анализатора фрагментированных программ на языке LuNA prolog-analyzer.

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

1 Назначение программы	96
1.1 Функциональное назначение программы	96
1.2 Эксплуатационное назначение программы	96
1.3 Состав функций	96
2 Условия выполнения программы	97
2.1 Минимальный состав аппаратных средств	97
2.2 Минимальный состав программных средств	97
2.3 Требования к персоналу	97
3 Выполнение программы	98
3.1 Загрузка и запуск программы	98
3.2 Выполнение программы	98
3.3 Завершение программы	98
4 Сообщения оператору	99
4.1 Сообщения в рамках нормальной работы программы	99
4.2 Сообщения о сбое	99
5 Лист регистрации изменений	100

1 Назначение программы

1.1 Функциональное назначение программы

Разработанная программа предназначена для анализа фрагментированных программ, написанных на языке LuNA, с целью обнаружения различных типов семантических ошибок.

1.2 Эксплуатационное назначение программы

Статический анализатор prolog-analyzer предназначен для использования разработчиками программ на языке LuNA для обнаружения семантических ошибок в автоматизированном режиме.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Анализ исходного кода на языке LuNA для поиска семантических ошибок.
- Генерация логического представления программ на языке LuNA.
- Вывод информации о найденных ошибках в формате JSON.
- Дополнение существующих файлов информацией о найденных ошибках.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере или вычислительном кластере. Устройство, на котором запускается программа, должно иметь:

- Оперативная память объемом не менее 4 Гб.
- Жёсткий диск объёмом не менее 64 Гб.
- Центральный процессор, имеющий два или более ядер и тактовую частоту не менее 1.5 ГГц.
- Монитор.
- Клавиатура.

2.2 Минимальный состав программных средств

- ОС семейства Linux.
- Интерпретатор языка Python версии 3.10 или позднее, пакет click версии 3.8.1 или совместимой с ней.
- Система LuNA.
- SWI-Prolog.

2.3 Требования 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 `prolog-analyzer` в командном интерпретаторе `bash`.

Пользователь должен указать значение обязательного параметра “`--project-dir`” — имени директории, содержащей файлы исходного кода на языке LuNA, а также передать набор имен файлов исходного кода на языке LuNA.

Пользователь может указать следующие опциональные параметры:

- “`--build-dir`” — имя директории, которая должна быть использована компилятором LuNA при сборке программы.
- “`--output-dir`” — имя директории, в которой должны быть размещены созданные в процессе анализа файлы.
- “`--errors-file`” — имя файла, в который должна быть записана информация о найденных ошибках. Если файл существует, новая информация будет записана в конец, иначе файл будет создан.
- “`--no-cleanup`” — не удалять созданные в процессе анализа файлы, не являющиеся файлом с информацией о найденных ошибках.

3.2 Выполнение программы

После вызова, `prolog-analyzer` вызывает компилятор LuNA, генерирует логическое представление переданных файлов исходного кода, после чего выполняет его анализ с целью обнаружения семантических ошибок.

3.3 Завершение программы

`Prolog-analyzer` завершается после выполнения анализа и записи информации о найденных ошибках в файл без необходимости дополнительных действий со стороны пользователя.

4 Сообщения оператору

4.1 Сообщения в рамках нормальной работы программы

Во время работы программы в консоль будут последовательно напечатаны следующие информационные сообщения:

- “Compiling LuNA program” — идет компиляция программы на языке LuNA.
- “Generating facts” — выполняется генерация логического представления программы.
- “Analyzing” — выполняется анализ программы.
- “Results have been written to <имя_файла>” — информация о найденных ошибках была записана в файл с именем <имя_файла>.

Перечисленные выше сообщения не требуют действий со стороны оператора.

4.2 Сообщения о сбое

Если во время одного из этапов работы программы произошел сбой, в консоль будет выведено сообщение, начинающееся с “ERROR”. Это сигнализирует о том, что prolog-analyzer с использованными параметрами не может выполнить анализ программы, и оператор должен изменить параметры запуска.

5 Лист регистрации изменений

Таблица Б.1 — Лист регистрации изменений руководства оператора

[illegible]

ПРИЛОЖЕНИЕ В

СТАТИЧЕСКИЙ АНАЛИЗАТОР PROLOG-ANALYZER ОПИСАНИЕ ПРОГРАММЫ

Листов 11

Новосибирск 2024

101

АННОТАЦИЯ

В данном документе приведено описание статического анализатора фрагментированных программ на языке LuNA prolog-analyzer. Пользователь может использовать prolog-analyzer для автоматизированного обнаружения семантических ошибок в программах на языке LuNA в процессе разработки.

Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

СОДЕРЖАНИЕ

1 Общие сведения	104
1.1 Обозначение и наименование программы	104
1.2 Необходимое программное обеспечение	104
1.3 Языки программирования	104
2 Функциональное назначение	105
2.1 Назначение программы	105
2.2 Сведения о функциональных ограничениях на применение	105
3 Описание логической структуры	106
3.1 Алгоритм программы	106
3.2 Используемые методы	106
3.3 Структура программы	106
3.5 Связи программы с другими программами	106
4 Используемые технические средства	107
5 Вызов и загрузка	108
6 Входные данные	109
7 Выходные данные	110
8 Лист регистрации изменений	111

1 Общие сведения

1.1 Обозначение и наименование программы

Полное наименование программы — статический анализатор `prolog-analyzer`.

1.2 Необходимое программное обеспечение

Для функционирования `prolog-analyzer` необходимы система LuNA, SWI-Prolog, интерпретатор Python версии 3.10 или выше, а также модуль `click`.

1.3 Языки программирования

`Prolog-analyzer` написан на Python и Prolog. Средство разработки — интегрированная среда разработки PyCharm от компании JetBrains.

2 Функциональное назначение

2.1 Назначение программы

Разработанная программа предназначена для анализа фрагментированных программ, написанных на языке LuNA, с целью обнаружения различных типов семантических ошибок.

2.2 Сведения о функциональных ограничениях на применение

Программа не предназначена для анализа программ на языке LuNA, содержащих оператор `while`, тернарный условный оператор, операторы приведения типов, а также строковые и вещественные константы. Программа не предназначена для работы на ОС семейства Windows, так как для данной платформы не существует реализации системы LuNA.

3 Описание логической структуры

3.1 Алгоритм программы

Программа вызывает компилятор LuNA. Затем, используя сгенерированные компилятором файлы, программа осуществляет генерацию логического представления анализируемых программ на языке LuNA.

Если предыдущий этап завершился успехом, выполняется анализ сгенерированного логического представления, после чего создается или дополняется существующий файл с информацией о найденных ошибках.

3.2 Используемые методы

В работе используется подход логического метапрограммирования. Анализируемая программа представляется в виде набора фактов на языке Prolog, а для анализа был реализован набор предикатов, выполняющих поиск различных семантических ошибок. Процесс анализа заключается в проверке истинности этих предикатов после загрузки логического представления программы.

3.3 Структура программы

Prolog-analyzer состоит из генератора логического представления программы и программы-анализатора. Связь между ними осуществляется посредством использования программой-анализатором логического представления анализируемой программы на языке LuNA, созданного генератором.

3.5 Связи программы с другими программами

Программа использует компилятор LuNA, являющийся частью системы LuNA. Также для работы программы необходимы интерпретатор Python версии 3.10 или выше, командный интерпретатор bash и SWI-Prolog.

4 Используемые технические средства

Prolog-analyzer предоставляет интерфейс командной строки. Устройство, на котором запускается программа, должно иметь:

- Оперативная память объемом не менее 4 Гб.
- Жёсткий диск объёмом не менее 64 Гб.
- Центральный процессор, имеющий два или более ядер и тактовую частоту не менее 1.5 ГГц.
- Монитор.
- Клавиатура.

5 Вызов и загрузка

Запуск программы осуществляется вызовом команды `prolog-analyzer` в командном интерпретаторе `bash`.

6 Входные данные

Пользователь должен указать значение обязательного параметра “--project-dir” — имени директории, содержащей файлы исходного кода на языке LuNA, а также передать набор имен файлов исходного кода на языке LuNA.

Пользователь может указать следующие опциональные параметры:

- “--build-dir” — имя директории, которая должна быть использована компилятором LuNA при сборке программы.
- “--output-dir” — имя директории, в которой должны быть размещены созданные в процессе анализа файлы.
- “--errors-file” — имя файла, в который должна быть записана информация о найденных ошибках. Если файл существует, новая информация будет записана в конец, иначе файл будет создан.
- “--no-cleanup” — не удалять созданные в процессе анализа файлы, не являющиеся файлом с информацией о найденных ошибках.

7 Выходные данные

По завершении работы программы будет создан или дополнен существующий файл в формате JSON с информацией о найденных ошибках.

8 Лист регистрации изменений

Таблица В.1 — Лист регистрации изменений руководства оператора

[illegible]