**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”**

**Факультет** Программной инженерии и компьютерной техники

**Образовательная программа** Системное и прикладное программное обеспечение

**Направление подготовки (специальность)** 09.04.04 Программная инженерия

# ОТЧЕТ

## о научно-исследовательской работе

Тема задания: «Выработка методов к анализу мультиязыковых текстов программ»

Обучающийся: Орловский М.Ю (**Фамилия И.О.**) Р4116 (**номер группы**)

Руководитель практики от университета: Маркина Т.А, доцент факультета ПИиКТ

Санкт-Петербург
2023 г.

# 1 Эпоха

## 1.1 Исследовать возможность обобщения предлагаемого метода на сочетание различных языков программирования.

В ходе подготовки статьи на конференцию «КМУ XII» была проведена небольшая ревизия метода анализа, для его большей формализации. Было выяснено, что структуру семантического узла можно обобщить, если немного изменить свойства его компонентов. В данный момент, структура узла представляет информацию, представленную в таблице 1.

Таблица 1 – Обновленная структура семантического узла

| Имя поля | Тип поля | Семантика | Что обновлено |
|---|---|---|---|
| Узел | Идентификатор AST узла | Указывает позицию анализируемой синтаксической конструкции | Было решено, что будет разумнее рассматривать позицию как конкретный AST узел из исходного дерева кода |
| Атрибуты | Гетерогенный список | Содержит список атрибутов узла | В отличие от предыдущего варианта хранения атрибутов (строковый ключ – строковое значение), гетерогенный список является более гибким механизмом, который проще обрабатывать программно |
| Вид | Перечисление | Является способом типизации конкретного семантического узла, исходя из ЯП, который он описывает | Решено было удалить этот атрибут, т. к. необходимая информация уже содержится в поле Узел |
| Семантика | Перечисление | Используется для отражения семантики, представляемой узлом | Решено было удалить этот атрибут, по причинам описанным ниже |

Решение об удалении поля «Семантика» связано в первую очередь с распределением сложности анализа. Предполагается, что у узлов будет единая семантика связи (семантика ребра в сети) – это семантика «Зависимость». Такая семантика была выбрана по нескольким причинам:

1. Многие семантические зависимости узлов являются бесполезными на этапе анализа – по ним сложно делать какие-либо выводы о структуре сети в связи с неупорядоченностью узлов при их связывании;
2. Семантические зависимости разного рода усложняют анализатор, делая его более специфичным и снижая гибкость;
3. Предполагается, что все семантические отношения между узлами можно будет вывести из готовой сети, при практическом применении этой сети в конкретном инструменте.

Таким образом, предполагается, что данная структура узла будет более гибкой в отношении способов хранения информации о семантике конкретных синтаксических конструкций.

Что касается алгоритма связывания, то он не претерпел явных изменений. Более формальное описание алгоритма следует в другой главе, но на высоком уровне алгоритм производит связывание узлов исходя из их типа и набора атрибутов. Алгоритм при этом имеет сложность $O(n^2)$.

## 1.2 Исследовать наиболее часто используемые парадигмы программирования (процедурное, ОО, декларативное) в контексте мультиязыкового анализа

Прежде чем рассматривать мультиязыковой анализ в контексте обрабатываемых в нем языков, было бы полезно обобщить и категорировать эти языки. В целях работы предлагается использовать общепринятое разделение языков по парадигмам [1]. Данное разделение представлено в таблице 2.

Таблица 2 – Разделение языков по парадигмам

| Парадигма | Основные особенности | Представители |
|---|---|---|
| Императивная | Переменные как способ манипуляции памятью;<br>Мутабельность как основной способ проведения вычислений;<br>Описание вычислений как конкретного набора инструкций | C, Pascal, Java, Golang, Bash, JavaScript и многие другие |
| Декларативная | Описание результата и его свойств, как способ его получения;<br>Описание вычислений как высокоуровневых объявлений; | ML, Lisp, Haskell, SQL, Make, HTML и также многие другие |

Несмотря на то, что такого разделения достаточно для категоризации языков, оно будет расширено и детализировано, так как в рамках данной работы предстоит рассматривать различные языки очень разной природы. Такое расширенное и уточненное разделение представлено в таблице 3.

Таблица 3 – Расширенное разделение языков по парадигмам

| Парадигма | Основные особенности | Представители |
|---|---|---|
| Процедурная и структурная | Инструкции сгруппированы в процедуры/подпрограммы; переменные как способ взаимодействия с памятью | C, Pascal, Golang, Bash, и многие другие |
| Объектно-ориентированная | Данные и код инкапсулированы в единое представление – объект; Объекты общаются посредством передачи сообщений друг-другу | Java (отчасти), C# (отчасти), JavaScript (отчасти), C++ (отчасти), Smalltalk, Erlang, Elixir, Self и другие |
| Функциональная | Функции, как способ выстроить процесс вычисления; Иммутабельность как фундамент формальных гарантий в коде | ML, Rust (отчасти), Haskell, Erlang, Elixir, Lisp |
| Макро | Процесс вычислений проводится путем лексических подстановок строк для достижения конечной строки; встречаются варианты, оперирующие на AST | Препроцессор C, Make, макросистема Rust, различные DSL в Web фреймворках |

Исходя из данной классификации, можно разделить анализируемые языки по категориям, в каждой из которых можно будет сформировать свою онтологию, описывающую основные особенности языков данной категории и их основные компоненты. Стоит учесть, что данное разделение сделано в целях упрощения и обобщения анализа и может не соответствовать общепринятому.

### 1.3 **Рассмотреть различные подходы к представлению семантической информации**

Основным способом представления информации (и самым универсальным) является семантическая сеть. В данной работе был избран именно такой подход, однако, семантические сети трудно структурировать и формализовывать. И хотя в дальнейшем будет использоваться именно этот подход к представлению семантической информации, полезно рассмотреть иные подходы, активно используемые в компиляторах, оптимизаторах и анализаторах. Такие подходы отражены в таблице 4.

Таблица 4 – Представление информации об исходном коде

| Название представления | Основные особенности | Плюсы (в рамках данной работы) | Минусы (в рамках данной работы) |
|---|---|---|---|
| AST | Представление иерархической структуры программы, посредством связывания её синтаксических конструкций в единое дерево | Легко анализируется, может быть использовано в анализах, где порядок анализа не важен; легко визуализируется; универсально для многих видов анализа | Сложно изменяется; не подходит для анализов, которые зависят от порядка; некоторые анализы неприменимы |
| CFG | Представление пути исполнения программы, моделируется связями (переходами) между узлами (операторами или выражениями) | Многие другие анализы легко реализуются через CFG; хорошо подходит для императивных языков | Плохо подходит для неимперативных языков, где путь исполнения программы задан неявно; сложнее реализовать чем AST |
| Семантическая сеть | Представление сущностей и их зависимостей в виде графа, узлы которого определяют сущности, а связи - отношения | Легко синтезируется; очень универсально; очень гибко и может быть использовано в различных анализах | Слабо структурировано; нет четкой схемы как связывать узлы между собой, по какому принципу выявлять сущности и отношения |
| Иерархия типов | Представление информации о связях сущностей в виде типов этих сущностей, формирующих определенную иерархию | Может быть простым вариантом представления большого объема информации в ОО и функциональных системах; имеет много формальных средств и методов для анализа | Сильно зависит от избранного языка; может не нести ценности, если язык имеет слабую типизацию; довольно сложна для анализа в общем случае |
| Онтология | Представление, основанное на формальных знаниях в конкретной предметной области | Однажды специфицированная онтология может быть использована в другом языке при наличии схожих семантик; универсальна и расширяема | Быстро становится сложной при расширении; обычно, предметные области (а в особенности ЯП) трудно формализуемы |

Стоит заметить, что существует еще большое множество иных представлений, которые слабо подходят для описания семантики языков программирования: фреймовые, логические и вероятностные модели, а также

нейронные сети. Хоть они и представляют ценность для реализации различных анализов, в рамках данной работы они либо слишком сложны для применения, либо не обеспечивают необходимой точности.

Исходя из перечисленных в таблице 4 подходов к представлению семантической информации, можно заключить, что использование семантических сетей стоит совместить с иными способами представления информации. Предполагается, что использование онтологий упростит формирование семантических сетей, позволит их формализовать и повысит количество переиспользуемых техник анализа.

# 2 Эпоха

## 2.1 Провести анализ текущих решений в области мультиязыкового анализа на прикладном уровне (IDE и инструментальные средства)

На данный момент, в индустрии разработки ПО мало качественных и достаточных инструментов для мультиязыковой разработки, чего очень не хватает пользователям языков программирования [2]. Однако, существует ряд решений, направленных на обеспечение инструментальной поддержки разработки. Многие решения из этой области разработаны для специфичных предметных областей, в первую очередь для веб-разработки. В рамках данной работы были рассмотрены несколько инструментов, поддерживающие (либо имеющие возможность) мультиязыковой разработки для анализа межъязыковых связей.

### 2.1.1 JetBrains Rider

JetBrains Rider это кроссплатформенная IDE имеющая поддержку различных языков, в первую очередь .Net семейства (C#, Visual Basic, F# и прочие), а также ряда DSL которые используются в соответствующих фреймворках (Blazor, XAML и др.) [3]. Она использует ReSharper как анализатор и инструмент для обеспечения продуктивности разработчика. Вкупе с IntelliJ IDEA, которая служит базисом для IDE, Rider обеспечивает большое количество удобной для разработчика функциональности. Краткий перечень такой функциональности включает:

1. Обнаружение зависимостей между символами на мультиязыковом уровне;
2. Переименование символов по их зависимостям;

3. Контекстная информация о синтаксическом элементе;
4. Обнаружение несоответствий при межъязыковом взаимодействии;
5. Поиск определения или объявления символа в межъязыковом контексте;
6. Поиск всех использований определенного символа в межъязыковом контексте;
7. Автодополнение, доступное в межъязыковом контексте.

Несмотря на ряд продуктивных возможностей, можно выявить недостатки Rider, касающиеся межъязыкового анализа и рефакторингов. Одним из основных недостатков является узкий спектр поддерживаемых DSL – Rider хорошо взаимодействует с исходным кодом Blazor, но, к примеру файлы конфигурации или файлы проектов/решений им воспринимаются плохо. Ни одна (за исключением пункта 7) из возможностей перечисленных выше не применима к файлам JSON, YAML или SLN.

### 2.1.2 SonarQube

SonarQube это автоматический инструмент для проверки кода, предназначенный для увеличения качества кода, раннего устранения ошибок и повышения скорости внедрения. Основным инструментом, привлекающим внимание в рамках данного проекта, является SonarLint – универсальный статический анализатор, поддерживающий множество популярных языков программирования. SonarQube имеет возможность анализировать мультиязыковые проекты, хотя об анализе межъязыковых связей информации нет [4].

Возможности инструмента схожи с JetBrains ReSharper и также предоставляют результаты анализа в реальном времени, для возможности интеграции инструмента в IDE.

Одним из явных недостатков инструмента можно выявить высокую связность с инфраструктурой Sonar – его возможности ограничены, если не использовать полноценный SonarQube сервер и большое количество дополнительных инструментов, что может затруднить интеграцию SonarLint в небольшой проект.

### 2.1.3 Mulang

Mulang позиционируется как универсальный, мультиязыковой и мультипарадигменный статический анализатор, ориентированный на обнаружение ошибок и формирование предикатов (ожиданий) о коде [5].

Языки, поддерживаемые Mulang включают:

1. C;
2. Haskell;
3. Java;
4. JavaScript (ES6);
5. Python (2 и 3);
6. Prolog.

Также, имеется поддержка Ruby и PHP через специфические языковые инструменты.

Так как проект является открытым, есть возможность добавить поддержку определенных языков при необходимости.

Mulang является инструментом, позволяющим работать с кодом, как с базой знаний и, соответственно, не поддерживает полноценной интеграции с IDE. Однако, исходя из набора «ожиданий», сформированного разработчиками и из возможности определять свои «ожидания» можно заключить, что проект концептуально применим в IDE при разработке соответствующего плагина.

Небольшой набор «ожиданий» и ошибок, которые способен обнаруживать Mulang включает:

1. «Делает ли элемент вызов определенной функции?»;
2. «Присутствует ли переменная в выражении?»;
3. «Представлено ли вычисление как рекурсия?»;
4. «Используется ли оператор «если»?»;
5. Дупликация кода;
6. Ошибка в имени идентификатора;
7. Недостижимый код;
8. Длинный список параметров.

## 2.2 Провести анализ моделей представления семантической информации программ в контексте различных ЯП и технологических стеков

Несмотря на наличие инструментов для анализа мультиязыкового кода, большая часть из них является проприетарными решениями с закрытым исходным кодом. В связи с этим, большая часть инструментов работает как «черный ящик» и изучить модель представления семантической информации внутри таких инструментов не представляется возможным.

Однако, существуют анализаторы с открытым исходным кодом, представляющие интерес в данном вопросе.

### 2.2.1 Multilingual static analysis tool

В контексте MLSA используются различные модели представления семантической информации, а именно:

1. Граф вызовов функций;
2. CFG;
3. Граф присваиваний;
4. Граф зависимостей на уровне файлов и директорий.

Исходя из того, что большая часть представлений является внутренними структурами данных исходного языка, возможность расширения такого анализатора как MLSA является трудоемкой задачей. И хотя для каждого формата есть своё сериализованное унифицированное представление, зависимость проекта от специфической техники парсинга (Island grammar parsing) снижает его расширяемость.

Также, инструмент обязательно включает в себя парсер исходного языка, что затрудняет его интеграцию с существующими анализаторами.

### 2.2.2 Mulang

Mulang в свою очередь представляет особый интерес благодаря наличию унифицированного представления называемого авторами проекта «Abstract semantic tree».

Такое дерево состоит из узлов, попадающих в одну из 5 категорий:

1. Выражения;

2. Шаблоны;
3. Типы;
4. Равенства;
5. Генераторы.

Равенства можно описать как отображение из списка шаблонов (параметров функции или параметров match выражения) в тело (выражение либо набор частичных выражений). Они семантически представляют собой математическое отображение из входных параметров/шаблонов в одно или несколько выражений и, следовательно, подходят в первую очередь для функциональных языков. Генераторы семантически представляют собой выражения, имеющие ленивый (по надобности) тип вычислений.

На остальных трех категориях стоит остановится поподробнее.

#### 2.2.2.1 Выражения

Выражение является основной описания любого типа вычисления. Mulang AST моделирует через выражения иные конструкции, описывающие вычисления – в первую очередь операторы и объявления. Такой подход является гибким, хотя и размывает границу между чистыми вычислениями и вычислениями с сайд-эффектами.

В Mulang представлен весь стандартный, несколько упрощенный набор семантических конструкций, описывающих выражения, встречающийся в многих популярных императивных и функциональных языках. Из-за высокого уровня абстракции можно выявить некоторые недоработки онтологии.

Она выражает общие языковые конструкции, но делает это неоднородно, к примеру цикл языка C описан как отдельная конструкция ForLoop, хотя семантически она может являться общим случаем итерации, так как через неё выражаема конструкция For (являющаяся генератором). Также, онтология не покрывает семантику некоторых языков, хотя причин для этого не имеет. К примеру, конструкция Class имеет лишь одно поле для обозначения базового класса, однако к C++ это в общем случае не подходит.

#### 2.2.2.2 Шаблоны

Шаблоны в терминах Mulang AST обозначают параметры или, собственно, шаблоны, предназначенные для обозначения хода вычислений. Они не несут значений сами по себе и являются абстракцией над многими управляющими и структурирующими конструкциями языков. К примеру, они могут быть использованы для обозначения параметров функций, параметров в выражениях match и switch, а также для обозначения литеральных конструкций (числа, строки, списки, кортежи и т. д.).

При этом многие шаблонные конструкции удаляют изначальную семантическую информацию о моделируемых семантических концепциях и служат, скорее, как Ad-hoc решение, направленное на выражение минимальной информации о конструкции.

#### 2.2.2.3 Типы

Типы в Mulang AST представляют собой довольно слабый, но тем не менее, достаточный набор семантик для описания операций над типами. Типы обозначают операции, которые могу проводиться над заданными типами как над множествами или функциями над множествами. К примеру, есть возможность задать параметризированный тип или задать ограничения на тип. Также есть возможность явного обозначения типа у определенного идентификатора.

Стоит заметить, что все конструкции оперируют над строками – таким образом, большая часть специфичной семантической информации остается необработанной и, следовательно, неиспользованной.

Если подытожить, то основные характеристики данной онтологии применимо к данной работе включают:

1. Онтология достаточно полная для многих языков (особенно высокоуровневых);
2. Некоторые конструкции выражений могут быть доработаны или расширены, в зависимости от задачи;
3. Конструкции шаблонов и равенств позволяют поверхностно описать необходимую семантику управляющих и структурных конструкций языка;

4. Базовое описание типов является малопригодным в общем случае, т. к. чаще проще обозначать операции над типами как стандартные выражения;
5. Онтология из-за специфики решаемой инструментом задачи не задает явным образом контекст времени связывания сущностей и идентификаторов, что очень важно для полного описания семантики языков.

## 2.3 Исследование влияния прикладных областей языков на возможность обобщения метода мультиязыкового анализа

Согласно исследованию [6], проведенному для выяснения важных аспектов мультиязыковой разработки, было выяснено что большая часть разработчиков, использующих несколько языков в проекте, занимается веб-разработкой и клиент-серверными решениями. Также, было выяснено, что наиболее частым сочетанием языков была пара GPL/DSL, что объясняется большим количеством DSL существующих в индустрии веб-разработки.

Согласно этому исследованию, наиболее популярные возможности, предоставляемые инструментами разработчика, включали в себя (от частых к редким по инструментальной поддержке):

1. Подсветка синтаксиса;
2. Переименование схожих семантически идентификаторов;
3. Навигация по коду;
4. Анализ ошибок и нарушений консистентности.

Исходя из исследования [7], основные прикладные области, имеющие явную гетерогенную структуру в отношении языков программирования, включают:

1. Энтерпрайз разработку (J2EE, .NET Platform);
2. Веб-разработку (современные JavaScript фреймворки, Electron);
3. Встроенные системы.

Также, стоит учесть, что и в иных областях нередки ситуации использования различных языков (в первую очередь это касается файлов конфигурации или сборки).

Основные сложности, которые существуют при анализе мультиязыковых текстов программ, применимо к разным предметным областям, представлены в таблице 5.

Таблица 5 – Сложности мультиязыкового анализа

| Предметная область | Проблема | Пояснение |
|---|---|---|
| Энтерпрайз, Веб-разработка | Большое количество различных языков, созданных для разных целей | Использование разных языков для фронтенд и бекенд разработки, а также для операций с БД является очень частым явлением и вкупе с другими языками для конфигурации и сборки может очень усложнить проект |
| Энтерпрайз | Широко распределенная система зависимостей | Зависимости между языками могут существовать не только на привычном уровне исходного кода, но также и на уровне файлов или даже целых отдельных API (например HTTP) |
| Встроенные системы | Большое разнообразие аппаратных средств | Из-за высокого разнообразия аппаратных средств, такие универсальные языки как C или ассемблер не имеют достаточной выразительной силы, в связи с чем возникает необходимость в дополнительных средствах (например, в кодогенерации) |
| Встроенные системы | Специфическая семантика в зависимости от аппаратных средств | Многие компиляторы для встроенных систем (C, C++, Python) не всегда следуют стандартам в угоду машинно-специфичным возможностям, из-за чего семантика исходного языка может быть нарушена, что затрудняет статический анализ |

Таким образом, обобщение мультиязыкового анализатора выглядит как достаточно сложная задача. Однако, благодаря использованию фиксированной формальной онтологии и соответствующих семантик представляется возможным реализация поверхностного анализа, отвечающего требованиям целевой платформы.

# 3 Эпоха

Следует описать предлагаемый метод языкового анализа, разделив описание на три секции:

1. Входная информация для анализа;
2. Формальная структура анализатора, как реализации метода анализа;
3. Формат результатов анализа и входной метаинформации.

## 3.1 Рассмотреть природу входной метаинформации и способы её обработки и хранения

Метод, описываемый в данной работе, является агрегатором различных специфичных анализаторов, каждый из который проводит свой собственный анализ для конкретного языка программирования.

Таковой анализ проводится над AST, полученных каким-либо образом для избранных программ, написанных на избранных языках программирования. Способ получения AST, специфический анализ и язык программирования, на котором реализован анализ не имеют для метода анализа, рассматриваемого в данной работе, особого значения. Анализатор вправе использовать любые методы обхода AST и извлечения информации из него. Также, допустимо использование других структур данных, например CFG или DFG до тех пор, пока они используются для получения конечного результата.

Однако, имеются накладываемые ограничения на природу данных, используемых анализатором и формат данных результата анализа.

Для реализации конкретного анализа конкретного языка предполагается использование определенной онтологии – набора формально определенных семантических конструкций, которые будут использоваться для описания семантики избранного языка. Онтология является унифицированным языком, что позволяет совместить результаты работы разных специфических анализаторов для дальнейшего использования в мультиязыковом контексте. Таким образом, каждый специфический анализатор, помимо использования AST конкретного языка будет также использовать определенную онтологию для кодирования результатов анализа.

В отношении формата выходных данных такого анализатора, его результатом работы должен быть узел семантической сети, имеющий структуру, представленную в таблице 1. Формат представления таких узлов может быть любым, до тех пор, пока он совместим с анализатором-агрегатором.

### 3.2 **Сформулировать формально метод языкового анализа**

После получения семантических узлов от всех специфических анализаторов запускается анализатор-агрегатор или компоновщик. Его основной задачей является связывание узлов по их атрибутам. Для этого вводится функция равенства пары узлов, принимающая их атрибуты и возвращающая булевый результат.

Для получения такой функции для конкретной пары узлов происходит несколько действий:

1. У узлов берутся их типы (которые де-факто являются типами исходных AST исходных языков) и формируется пара таких типов;
2. Для заданной пары типов ищется конкретная функция сравнения узлов.

По результатам сравнения узлов происходит решение связывать ли узлы или нет – если узлы «равны», то их стоит связать, в противном случае – нет.

Ниже в псевдокоде описан алгоритм связывания узлов.

```
function link(nodes) {
    let links: array(pair(node, node))
    for i = 0; i < nodes.length(); i++ {
        for j = i; j < nodes.length(); j++ {
            lhs = nodes[i]; rhs = nodes[j]
            equal = ontology.getEqualFunction(lhs.type, rhs.type)
            if equal(lhs, rhs) {
                links.add(lhs, rhs)
            }
        }
    }
    return links
}
```

Результат работы анализатора в виде пар связей узлов в дальнейшем предполагается использовать в конкретном инструменте. Формат представления

таких данных также не имеет особого значения и может являться любым, что поддерживает конкретный инструмент.

### 3.3 **Выбрать оптимальные структуры данных для представления метаинформации и хранения результатов анализа**

#### 3.3.1 Структура данных для представления метаинформации

Для представления метаинформации (семантической информации о конкретных сущностях конкретной программы) используется поле «Атрибуты» в семантическом узле. Это поле является гетерогенным списком.

Гетерогенный список – список, состоящий из сущностей разных типов. Таким списком является, например, список или массив в языках с динамической типизацией (Python, JavaScript, Scheme, Clojure). Так как на типы элементов нет ограничений, список тоже может являться элементом такого списка, что задает рекурсивную вложенность и позволяет формировать дерево.

Предполагается реализация такого списка в формате s-выражений [8]. Такой формат представления используется в языках семейства Lisp, что позволяет реализовывать мощные алгоритмы обработки кода как данных. В рамках данной работы данный формат избран по следующим причинам:

1. Единственная операция над атрибутами узла (помимо формирования таковых) — это сравнение и формат древовидного представления данных позволяет реализовать достаточно гибкую функцию сравнения, без необходимости дополнительной обработки атрибутов;
2. У гетерогенного списка три базовые операции – cons (добавить элемент в начало), car (взять голову списка), cdr (взять список без головы) и это позволяет существенно упростить формирование списка и работу с ним;
3. Во многих языках программирования (в том числе статически типизированных) реализация гетерогенного списка тривиальна, в некоторых есть готовые библиотеки – это повышает гибкость анализатора в отношении технологий, которые будут использоваться для реализации.

### 3.3.2 Структура данных результатов анализа

Для хранения результатов анализа, а именно пар связей узлов, возможно использование бинарного формата данных либо же уже упомянутых s-выражений. Одним из преимуществ s-выражений в данном случае будет выступать их простота и наглядность при преобразовании в текстовое представление.

Такой формат очень легко обрабатывается инструментом и, так как структура данных заранее известна и фиксирована, не требует сложных манипуляций с данными. Это упрощает интеграцию анализатора.

# 4 Эпоха

## 4.1 Провести разработку прототипа для тестирования метода в различных сценариях

В ходе изучения предметной области, была прочитана статья [9] описывающая формализм для структурирования программ, используя такую сущность как модуль. Модуль по природе своей является инструментом разделения функциональности программы на части, способные взаимодействовать между собой.

В статье описывается, предположительно, первая формально доказанная, в отношении корректности, система модулей для языка программирования. Формально также вводятся такие понятия как окружение, сигнатура, привязка, связывание. Также вводится такое понятие как набор связей (англ. linkset) представляющее собой отображение модуля после процесса связывания.

Статья представляет интерес в первую очередь по причине описания формально обоснованного фреймворка модульной системы, которая может быть напрямую применена в данной работе. Действительно, в отношении мультиязыкового анализа на этапе межъязыкового анализа можно утверждать, что взаимодействующие фрагменты кода на разных языках есть не что иное как программные модули, и их связывание позволяет получить описывающую их взаимодействие семантическую сеть.

Таким образом, решено было использовать терминологию, описанную в данной статье и адаптировать предлагаемую систему модулей под задачу мультиязыкового анализа. Стоит сразу заметить, что предполагается частичная адаптация терминологии и концепций, но для неё необходимы частичные изменения, что приводит к потере формально доказанной корректности такой системы.

## 4.2 **Разработать избранную модель представления семантической информации для использования в прототипе**

### 4.2.1 Система модулей

Перед разработкой необходимо определить основную терминологию и используемые структуры данных.

Модуль состоит из набора утверждений. Окружением называется утверждение, используемое для объявления зависимости модуля от другого модуля. Сигнатурой называется утверждение, используемое для обозначения возможности создания зависимости от данного модуля. Каждое окружение должно быть связано только с одной сигнатурой, однако сигнатура может быть использована во многих окружениях.

Каждое утверждение имеет тип – являющийся, в своем роде, уникальным идентификатором утверждения и в то же время задающий правила связывания модулей через связывание сигнатур с соответствующими окружениями. Так как проблема связывания модулей стоит на межъязыковом уровне, вводится простая и очень обобщенная система типов, базирующаяся на типизированном лямбда-исчислении первого порядка.

Описание языка, используемого для описания утверждений, выдвигаемых модулями:

```
A, B ::= P | A -> B | A x B | A + B | Any
a, b ::= a | \(a : A)b | b(a)
```

Таким образом, каждый модуль может иметь в окружении (импортировать) или иметь в сигнатуре (экспортировать) типизированный терм, описывающий семантику такого утверждения.

В качестве базового типа P решено было выбрать обобщенный знаковый целочисленный тип, но система типов может быть расширена и другими типами, однако при этом могут возникнуть различные усложнения. Тип Any отвечает за тип, населенный значениями всех возможных типов и поэтому всегда включает в себя любой другой тип. Он введен в первую очередь для обозначения отсутствия информации об описываемой утверждением сущности. В избранных сценариях

использования также были использованы конструкторы типов (например List, обозначающий список элементов). Стоит заметить, что последнее выражаемо через сумму произведений избранных типов, поэтому такой конструктор и подобные ему служат лишь для упрощения записи.

Утверждение также имеет ссылку на избранный узел AST дерева для обеспечения его связи с исходным кодом по результатам анализа.

При внутриязыковом анализе возникает возможность связывания утверждений, составляющих модули, во внутреннюю сеть. При этом семантика утверждения (импорт или экспорт) не играет роли, т. к. задается на уровне внутри языка и обозначает совершенно другую форму зависимости. Такая зависимость может отражать поток исполнения, цепочку вызовов или поток данных и являться частным случаем одного из представлений, описанных в таблице 4.

Вследствие широкой семантической природы связей между модулями может возникнуть ситуация, при которой необходимо задать порядок связывания модулей между собой. Предполагается, что такой порядок необходимо выстраивать только в рамках внутриязыкового анализа (т. е. только модули, относящиеся к одному языку, будут иметь определённый порядок). Таким образом вводится порядок (как целочисленный номер), назначаемый каждому модулю на этапе внутриязыкового анализа и обеспечивающий правильное связывание.

## 4.3 **Провести тестирование и собрать соответствующую статистику**

### 4.3.1 Описание сценариев использования

При разработке анализатора были рассмотрены сценарии использования, отраженные в таблице 6.

Таблица 6 – Рассматриваемые сценарии использования

| Название сценария | Описание сценария | Рассматриваемые особенности |
|---|---|---|
| Клиент-серверное взаимодействие по HTTP | Используются два языка: C# и JavaScript. На первом реализован сервер с использованием ASP.NET Core, на втором клиентское веб-приложение, делающее запросу к этому серверу. Очень популярный сценарий использования в веб-разработке | Обобщенный вызов – моделирование зависимостей как вызовов функций; Рассмотрение взаимодействия нетипизированного и типизированного языков; |

| | | |
|---|---|---|
| Вызов C функции из Python | Используются три языка: C, Python и Shell. На первом реализована библиотека для быстрых вычислений, на втором инфраструктура для манипуляции числовыми данными, на третьем – скрипт для компиляции библиотеки и запуска вычислений. Очень популярный сценарий использования в научных вычислениях | Связь посредством неявной зависимости – файловой системы;<br>Зависимость модулей на различных уровнях одновременно (файловая система и код);<br>Тернарная зависимость языков; |
| Условная компиляция C кода в зависимости от команд компилятора | Используются два языка: C и Shell. На первом реализована программа, а на втором скрипт для её сборки. В зависимости от поставляемых компилятору флагов результирующая программа отличается. Очень популярный сценарий в кроссплатформенной разработке и встраиваемых системах | Зависимость GPL от DSL;<br>Частный вариант зависимости программного кода от кода системы сборки или конфигурации; |

Программный код всех сценариев представлен в Приложении 1.

Ниже представлен и проанализирован результат анализа данных сценариев и интерпретация полученных результатов.

### 4.3.2 Сценарий 1

На рисунке 1 отображена схема взаимодействия фрагментов кода.

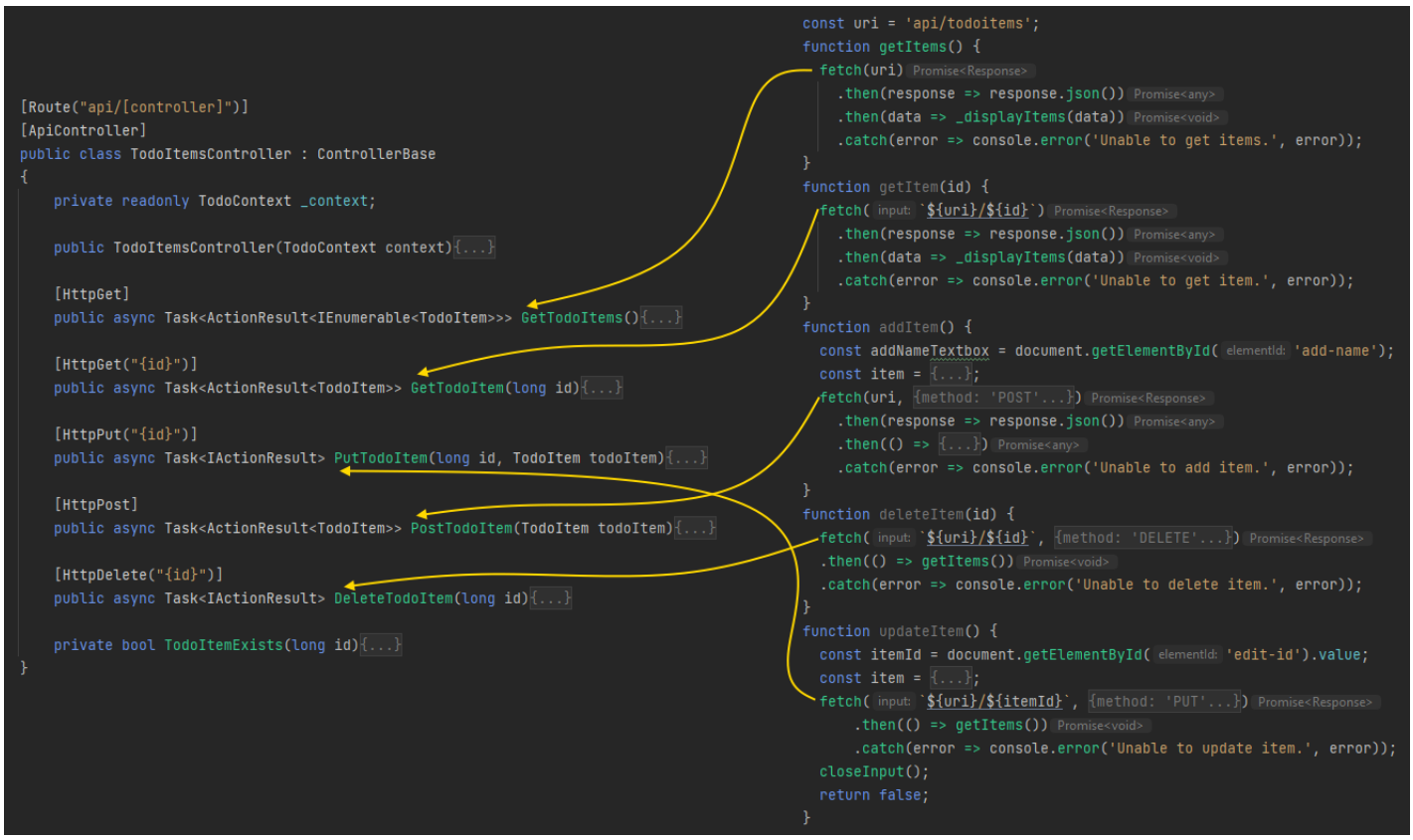


Рисунок 1 – Схема взаимодействий сценария использования 1

Здесь и далее для обозначения межмодульных (межъязыковых) связей используется направленная стрелка желтого цвета.

Стрелка отражает семантику «зависит», и более подробная семантика зависимости задается в онтологии, используемой при проведении анализа. Таким образом, такая онтология может быть использована в том числе для интерпретации результатов анализа.

На рисунке 2 приведено текстовое представление результатов анализа.

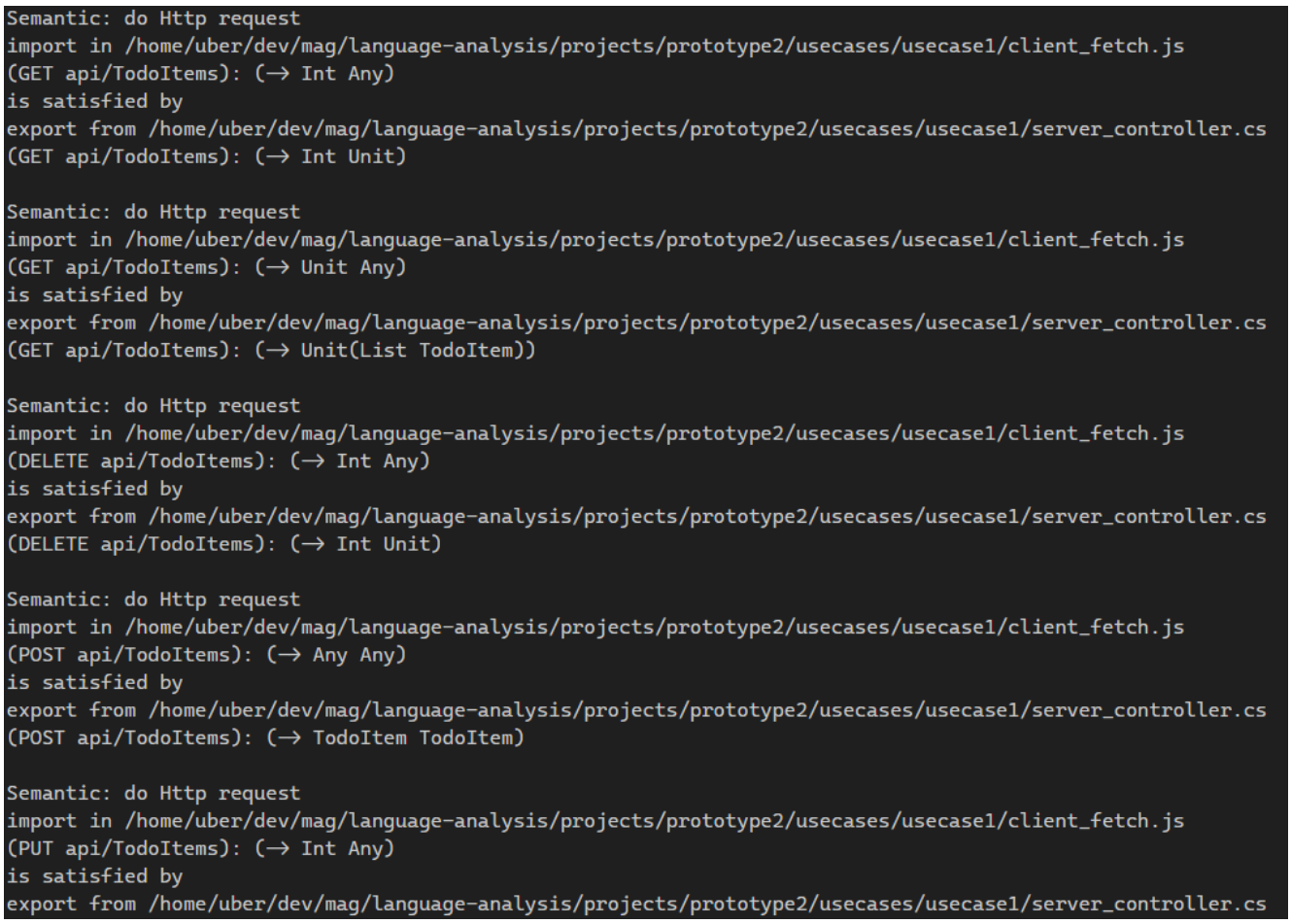


Рисунок 2 – Текстовое представление анализа сценария использования 1

Каждый абзац отвечает за одну межмодульную связь. Первый абзац можно прочитать следующим образом: «Представлена семантика HTTP запроса, импорт в файле client_fetch.js требующий функцию “GET api/TodoItems” с типом “Int -> Any” удовлетворяется экспортом в файле server_controller.cs предоставляющим функцию “GET api/TodoItems” с типом “Int -> Unit”».

Как можно увидеть из результатов анализа, все связи успешно выявлены и обозначены. Каждое утверждение связано с представлением исходного кода в виде AST, поэтому результаты этого анализа просто интегрировать в инструментальные средства.

Благодаря грамотной обработке аннотаций в коде C# контроллера и знанию о семантике fetch в коде JavaScript на этапе внутримодульного (внутриязыкового) анализа удается моделировать семантику этих конструкций как обычных функций, что позволяет достичь простоты и корректности межмодульного анализа;

Полноту межмодульного анализа можно обеспечить сложностью внутримодульного анализа – как видно из связи, отвечающей за PUT запрос, при уточнении типа функции, импортируемой в модуль client_fetch.js можно достичь более точного анализа и избежать появления ложноположительных результатов. На данном этапе импортируемая функция имеет тип Int -> Any, потому что без интенсивного анализа fetch и последующих методов then невозможно определить, какой эффект такой вызов будет иметь для всей системы. В данном случае, в методе then происходит неявная модификация глобальной переменной todoItems (не отражена на рисунке 1), что эффективно меняет состояние всего движка исполнения JavaScript. Поэтому, для отражения такого эффекта используется возвращаемый тип Any у импортируемой в код функции.

### 4.3.3 Сценарий 2

На рисунке 3 отображена схема взаимодействия фрагментов кода.

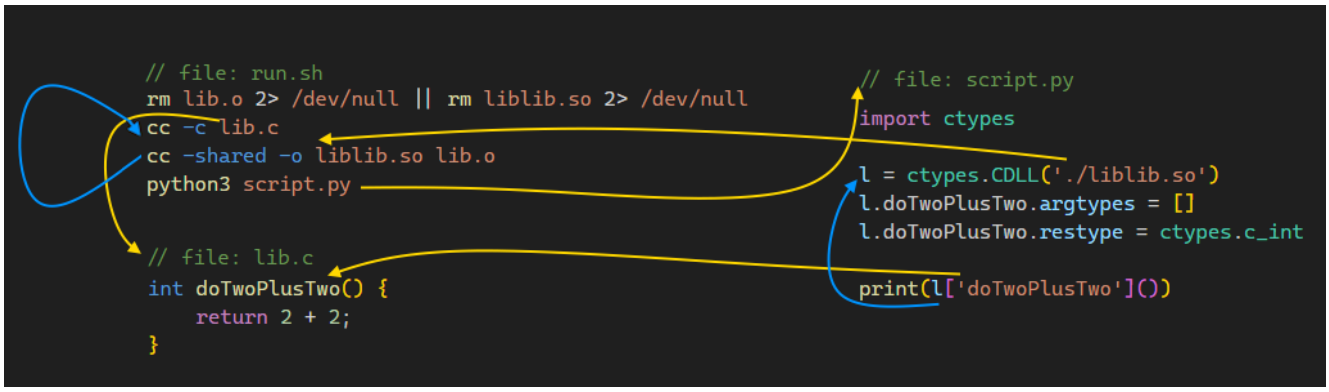


Рисунок 1 – Схема взаимодействий сценария использования 2

В данном случае желтые стрелки всё также обозначают межмодульные связи, а вот синие – внутримодульные связи. Решение отразить внутримодульные связи в данном сценарии продиктовано обозначением полноты межязыкового анализа – как видно из рисунка, такой анализ не позволяет составить полноценную сеть зависимостей, что достигается только с введением внутримодульных связей.

На рисунке 4 приведено текстовое представление результатов анализа.

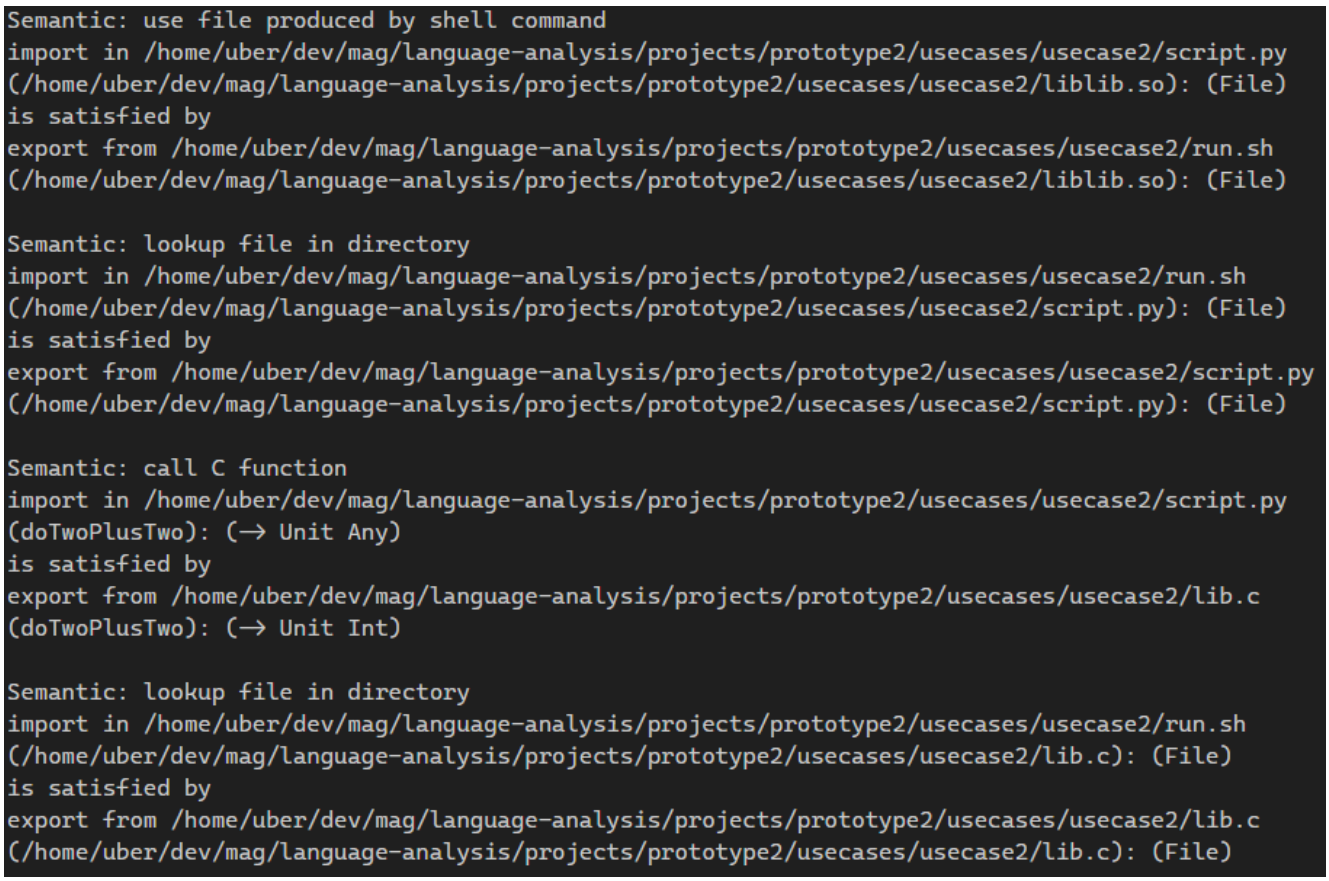


Рисунок 4 – Текстовое представление анализа сценария использования 2

Стоит заметить, что текстовое представление отражает только межмодульные связи, в связи с этим является менее полным. Это связано с тем, что внутримодульных связей достаточно много и многие из них неинформативны. Поэтому, было решено отразить наиболее важные из таких связей на рисунке.

Явным отличием от сценария 1 является введение новой семантики – семантики файлов в файловой системе. Такая семантика настолько распространена, что её описание предоставляет очень полезную информацию при межъязыковом анализе. Также, в данном примере используется язык оболочки Shell, основным предметом обработки которой являются файлы.

Возможность связи файлов между собой обеспечивается в том числе фактом того, что при анализе очередного файла можно по умолчанию добавить экспорт этого файла в формируемый анализом модуль. Это возможно, так как файл своим существованием обеспечивает наличие такой информации и дальнейшее её использование при формировании связей.

Как видно из рисунка 3, наблюдается не только зависимость на уровне функций (за счет функции «doTwoPlusTwo»), но и на уровне файлов, что позволяет учесть также и необходимость наличия скомпилированной библиотеки из файла lib.c для корректной работы скрипта script.py. Это возможно в том числе из-за наличия внутримодульной транзитивной зависимости lib.c -> lib.o -> liblib.so, находящейся в run.sh.

### 4.3.4 Сценарий 3

На рисунке 5 отображена схема взаимодействия фрагментов кода.

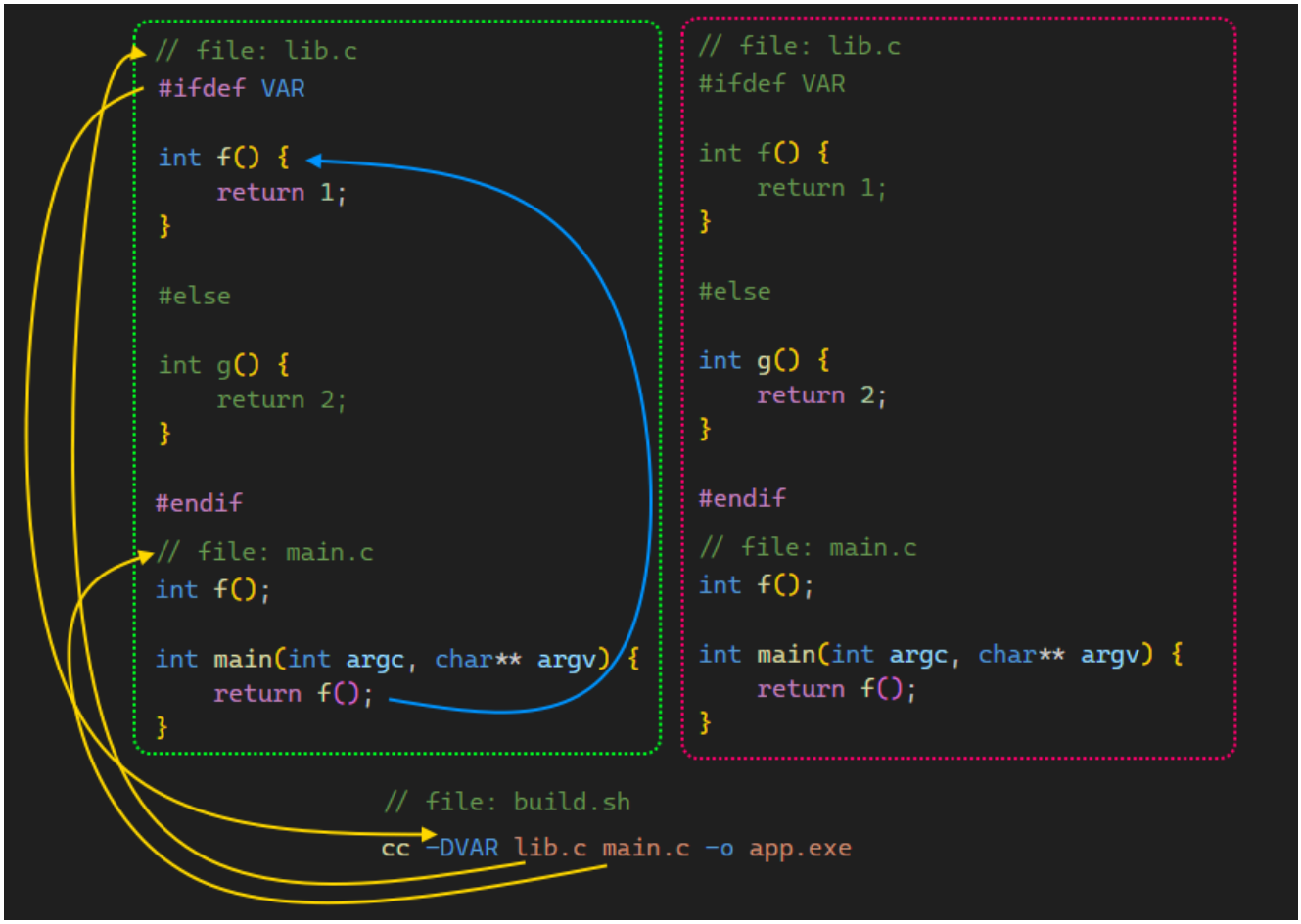


Рисунок 5 – Схема взаимодействий сценария использования 3

Предыдущие обозначения сохраняют силу и к ним добавлено новое обозначение – пунктирная рамка обозначает границы модуля. Это сделано в первую очередь для иллюстративности понятия «модуль». На данном рисунке модуль включает в себя два файла – файл lib.c и файл main.c. Таким образом, модуль это в первую очередь логическая группировка сущностей. Это же подтверждается тем, что для разрешения проблемы условной компиляции

(изменение исходного кода в зависимости от среды сборки) решено было отразить суперпозицию такой компиляции – два модуля. Первый модуль отражает структуру исходного кода в случае, когда переменная препроцессора VAR определена. Второй модуль отражает обратную ситуацию.

На рисунке 6 приведено текстовое представление результатов анализа.

```
Semantic: use file produced by shell command
import in /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/lib.c(2:6);main.c
(VAR): (String)
is satisfied by
export from /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/build.sh
(VAR): (String)

Semantic: lookup file in directory
import in /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/build.sh
(/home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/lib.c): (File)
is satisfied by
export from /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/lib.c(2:6);main.c
(/home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/lib.c): (File)

Semantic: lookup file in directory
import in /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/build.sh
(/home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/main.c): (File)
is satisfied by
export from /home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/lib.c(2:6);main.c
(/home/uber/dev/mag/language-analysis/projects/prototype2/usecases/usecase3/main.c): (File)
```

Рисунок 6 – Текстовое представление анализа сценария использования 3

Следует обратить внимание на информацию о пути к модулю (идущую после утверждений import и export). В данном случае путь представляет собой «составной файл» – модуль состоит из части файла lib.c (строки 2–6) и файла main.c.

На рисунке 5 видно, что модуль, обозначенный красной рамкой, не имеет никаких связей, рисунок 6 в том числе не имеет его упоминания. Это связано с тем, что для обеспечения правильного связывания были использован определенный порядок. Такой порядок задается на этапе внутримодульного анализа. В данном случае, модуль, обозначенный зеленой рамкой первым, был связан с необходимыми утверждениями из модуля на языке Shell, что препятствовало дальнейшему связыванию другого модуля, поскольку, как только import утверждение удовлетворено дальнейшее связывание с ним невозможно.

### 4.4 **Исследовать ограничения предлагаемого метода**

Данный метод, хоть и является довольно универсальным и, отчасти, формализованным, имеет ряд недостатков.

Основным его недостатком является предположение, что при внутримодульном анализе семантика многих конструкций языка будет рассмотрена с точки зрения т. н. «внешнего мира». Т. е. анализатор кода на C# должен также предоставлять информацию о том, что публичные методы контроллера, обозначенные соответствующими аннотациями, представляют собой семантическую информацию об экспортируемых функциях. Такой недостаток может быть исправлен разработкой расширения для анализатора избранного языка программирования, что позволит формировать такую информацию.

Вторым ограничением метода является зависимость от результатов внутримодульного анализа для сопровождения достаточной информации. Например, в случае больших фрагментов кода, имеющих несколько межмодульных зависимостей, число внутримодульных зависимостей будет на порядки больше и, соответственно, основная семантическая информация будет недоступна. Это ограничение также может быть исправлено при разработке расширения для избранного языка программирования с учетом адаптации семантической информации о внутримодульном анализе для дальнейшего её использования.

Третьим ограничением метода является система типов, введенная для формирования связей между модулями. В данный момент она слишком примитивна, несмотря на наличие тип сумм и типов произведений. Также, многие номинальные типы (в сущности, все, за исключением Any и None) являются несовместимы между собой, хотя семантически это не всегда так. Примером является тип int в Си и тип Int в Java, семантика которых одинакова (за исключением области возможных значений). На данный момент решетка типов представляет собой плоскую иерархию, где все Any олицетворяет все типы, а None обозначает единственный ненаселенный тип. Также, для отражения сайд-эффектов требуется гораздо более сложная система типов (такая, которая позволяет описывать монадические вычисления), но её реализация требует серьезной теоретической и практической проработки для обеспечения практической полезности.

Заключительным, четвертым, недостатком метода является обеспечение подхода к условной компиляции. Формирование явного порядка связывания методов усложняет анализ и не покрывает все возможные случаи условной компиляции. Решение проблемы условной компиляции может заключаться в выполнении одного из предложенных условий:

1. Анализируемый код не будет претерпевать никаких синтаксических и семантических изменений в ходе сборки системы;
2. Семантическая информация о таких изменениях будет отражена в структуре модуля, возможно, понадобится введение параметризованных модулей [10].

### 4.5 **Рассмотреть иные варианты получения информации для работы метода**

Как видно из рассмотренных сценариев использования, большая часть семантической информации о межъязыковых связях находится в соответствующий DSL – в первую очередь это касается языков сборки и развертывания.

Таким образом, имеет смысл рассмотрения определенной области для анализа как среды программирования, имеющей свои уровни и особенности. Наиболее ярко это отражается во введении в онтологию типа File для обеспечения правильного анализа сценариев 2 и 3. При мультиязыковом анализе следует рассматривать всю среду программирования – от возможностей, предоставляемых операционной системой до специфических семантических особенностей различных сопровождающих процесс фреймворков: систем сборки, развертывания, конфигурации и кодогенерации.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. https://homes.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html
2. Pfeiffer, RH., Wąsowski, A. (2012). Cross-Language Support Mechanisms Significantly Aid Software Development. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds) Model Driven Engineering Languages and Systems. MODELS 2012. Lecture Notes in Computer Science, vol 7590. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33666-9_12
3. https://www.jetbrains.com/rider/
4. https://www.sonarsource.com/products/sonarlint/
5. https://mumuki.github.io/mulang/
6. Mayer, P., Kirsch, M. & Le, M.A. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *J Softw Eng Res Dev* **5**, 1 (2017). https://doi.org/10.1186/s40411-017-0035-z
7. Z. Mushtaq and G. Rasool, "Multilingual source code analysis: State of the art and challenges," *2015 International Conference on Open Source Systems & Technologies (ICOSST)*, Lahore, Pakistan, 2015, pp. 170-175, doi: 10.1109/ICOSST.2015.7396422.
8. Davis, M. (1968). John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. Communications of the Association for Computing Machinery, vol. 3 (1960), pp. 184–195. *The Journal of Symbolic Logic*, *33*(1), 117-117. doi:10.2307/2270078
9. Luca Cardelli. 1997. Program fragments, linking, and modularization. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '97). Association for Computing Machinery, New York, NY, USA, 266–277. https://doi.org/10.1145/263699.263735
10. Andrew W. Appel and David B. MacQueen. 1994. Separate compilation for Standard ML. SIGPLAN Not. 29, 6 (June 1994), 13–23. https://doi.org/10.1145/773473.178245