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.**) P4216 (**номер группы**)

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

САНКТ-ПЕТЕРБУРГ
2024 г.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

LSP — language server protocol

IDE — integrated development invironment

AST — abstract syntax tree

ПРЕДИСЛОВИЕ

Данный отчет содержит результаты аналитической работы проведенной в течении третьего семестра. Так как в это время мною активно изучалась различная научная литература по вопросу (в первую очередь по теории типов) формат и структура метода сильно менялись. Это в свою очередь вызвало некоторую смену внимания к проблемам разработки метода, поэтому данный отчет является в сущности одним из обзорных шагов в области межъязыкового анализа. В связи с этим, программная реализация метода проведена не была, поэтому эпоха 4 описывает предполагаемую структуру анализатора которая хорошо подходит для инструментальных средств. При этом, в соответствующих разделах отчета я оставляю дальнейшие направления развития исследования.

1 ЭПОХА

1.1 Прочитать протокол LSP, LSIF а также VSCode Extension API

1.1.1 LSP

`LSP` [1] является протоколом, обобщающим языковые концепции для получения общих методов взаимодействия с программой в контексте инструментальных средств разработки (в первую очередь IDE)

Один из недостатков LSP — его применимость сильно ограничена в контексте полного анализа проекта (так как в любом случае требуется обработка всех AST всех фрагментов кода). Соответственно, само использование языкового протокола для данной задачи выглядит нецелесообразным. Однако, его рассмотрение все же может показать полезные сценарии использования.

В область функциональных особенностей LSP входит (обобщенно):

1. найти объявление или определение идентификатора (типа или терма);
2. определить иерархию вызовов;
3. найти все ссылки на идентификатор;
4. построить иерархию вызовов/наследования;
5. подсветить идентификатор под курсором;
6. автодополнение идентификатора под курсором;
7. дать подсказку по проблеме под курсором (Inlay hint);
8. остальные функции связанные с редактированием (подсветка, форматирование).

Данную функциональность можно разделить на составляющие, представленные в таблице 1.1

Таблица 1.1 – Функциональность LSP разбитая на составляющие

Функциональность	Назначение	Составляющие
Поиск объявления или определения	Навигация по коду	Информация о областях видимости; информация о типах
Определение иерархии вызовов	Навигация по коду	Информация о областях видимости; информация о типах
Поиск ссылок на идентификатор	Навигация, редактирование	Информация о областях видимости; информация о типах
Подсветка символа	Редактирование	Информация о областях видимости
Автодополнение символа	Редактирование	Информация о областях видимости; информация о типах
Подсказка по проблеме	Редактирование, корректность	Специфическая для языка информация

Как видно из таблицы, даже при таком грубом разделении функциональности на назначение и составляющие возникают четкие требования к информации которая требуется. В подавляющем большинстве случаев требуется информация о областях видимости. В меньшем числе случаев требуется информация о типах.

Таким образом, можно более четко определить источник информации для работы метода. Изначальной идеей было использование языкового `parse tree` как источника информации о идентификаторах. Однако, более подхо-

дящим вариантом представляется изначальный анализ другого представления информации — типизированного AST.

Типизированным AST может называться AST, некоторые узлы которого (в первую очередь идентификаторы) имеют связанные типы. Но получение и анализ такой информации затруднителен по как минимум двум причинам:

1. такое дерево будет иметь типы, специфичные для конкретного языка;
2. многие современные языки не имеют явных типовых аннотаций, поэтому получения такого дерева будет недостаточно.

Одно можно сказать наверняка — AST содержит достаточное количество информации для моделирования областей видимости в языке. Обычно, области видимости являются лексическими и представляют собой вложенную структуру узлов, обозначающих начало области. Остальные правила по построению областей могут быть языкоспецифичными и могут определяться на уровне языка.

В итоге, определяется первая часть анализатора (что можно назвать также языкоспецифической частью или фронтендом) — это отображение, способное преобразовывать изначальное программное AST в два (совмещенных или раздельных) представления информации об областях видимости и типах.

1.1.2 LSIF

`LSIF` (language server index format) [2] — относительно молодой протокол спецификации информации о среде разработки (англ. code workspace). На данный момент он представляет малый интерес по нескольким причинам:

- основной сценарий его использования — индексирование, чего недостаточно для поставленной задачи;
- он слабо развит на момент проведения исследования;
- является в сущности механизмом кеширования, поэтому может использоваться только в связке с LSP.

Однако, идея унифицированного представления информации об окружении проекта является очень мощной в контексте межъязыкового анализа и заслуживает отдельного изучения в дальнейшем.

1.1.3 VSCode Extension API

`VSCode Extension API` это формат и программное API, определяющее формат данных и аргументы эндпоинтов `Visual Studio Code` соответственно. Изучение API оказалось полезным в контексте дальнейшей интеграции анализатора в IDE. Таким образом, анализатор должен предоставлять результаты анализа в совместимом с VSCode формате, вероятно через использование LSP.

1.2 Выявить особенности внедрения различных языковых серверов для организации межъязыкового анализа

Одной из изначальных идей было рассмотрение способов получения AST из программного проекта. Эта задача представляется нетривиальной в общем случае и сильно зависит от следующих факторов:

- сценарий использования метода;
- конфигурация метода сборки и версионирования кода;
- внешнее системное окружение (ОС, библиотеки, пакеты и прочее).

Таким образом, в данном случае интеграция различных языковых серверов для решения данной задачи является промежуточным решением и в общем случае недостаточным. Следует рассмотреть более общие варианты способов извлечения AST которые будут в первую очередь обладать корректностью и полнотой. Данная работа выходит за рамки исследования проведенного в этом семестре и должна быть продолжена в следующем. Однако, на данный момент практичные методы извлечения такой информации (например спецификация файлов для анализа вручную) выглядят достаточными для продолжения текущего исследования.

2 ЭПОХА

2.1 Изучить известные существующие системы типов лямбда исчисления

В данном разделе я буду руководствоваться книгой Б. Пирса ”Типы и языки программирования” [3]. Я выбрал эту книгу по нескольким причинам:

- пройденный мною курс ”Системы типизации лямбда исчисления” [4] имел в источниках ссылку на эту книгу, и также она была рекомендована лектором как хорошее и более полное изложение материала;
- эта книга представлена в том числе на русском языке, что упрощает терминологию;
- фундаментальные исследования в области теории типов были проведены в 60-80х годах и даже такая относительно старая книга всё ещё остается актуальной в качестве базового пособия по теме.

2.1.1 Простое типизированное лямбда исчисление

Простое типизированное лямбда исчисление представляет собой простейшую систему типов из возможных — в языке поддерживающим такую систему типизируются только функции. Так как сигнатура конструктора функции это `* -> *` в таком языке должен быть хотя бы один тип от которого можно строить другие типы. Таким типом можно выбрать, например, `unit` — тип, содержащий лишь одно значение, обозначаемое `unit` или `()`.

В целом, простое типизированное лямбда исчисление легко расширяется следующими типами без существенного усложнения теорем типизации:

- базовые типы (Bool, Int, Float...) т.е. такие, конструктор которых имеет арность 0;
- пары, кортежи, записи (также называемые типами-произведениями);
- суммы, варианты, перечисления (также называемые типами-суммами).

Однако ясно, что такая система очень ограничена в выразительности и не типизирует все возможные термы. Например, затруднительна типизация комбинатора неподвижной точки — его можно задать только в формате синтаксической формы (что не является частью системы типов). Соответственно, в такой системе рекурсивные функции нетипизируемы.

2.1.2 Простое типизированное лямбда исчисление с ссылками

Если взять простое типизированное лямбда исчисление и добавить в него конструктор вида `Ref :: *`, то можно получить исчисление с поддержкой ссылок. Определение операций над ссылками прямолинейно и обычно включает:

- присваивание: Γ, t_1: Ref T, t_2 : T, $\Sigma \vdash (t_1 := t_2)$: Unit;
- разыменование: Γ, t: Ref T, $\Sigma \vdash$!t:T;
- конструирование: Γ, t: T, $\Sigma \vdash$ ref T: Ref T.

С точки зрения теории, сложность такого исчисления заключается в том, что все типовые предпосылки должны включать новый контекст — Σ. Этот контекст захватывает семантику линейной памяти и взаимодействие с ней.

Основная проблема заключается в данном случае в том, что при расширении исчисления каким-либо типом полиморфизма (даже если полиморфизм достигается через синтаксические конструкции) возникают проблемы типизации. Подобное, например, представлено в `ML` [5]. Это несколько усложняет вывод типов и их проверку, а также может конфликтовать с другими аспектами системы типов, делая её некорректной.

С точки зрения практики, такая система вводит дополнительные сложности с `aliasing`. Это явление заключается в том, под несколькими идентификаторами может находится одно и то же значение в памяти. Это приводит к большим сложностям при вычислении и к неприятным краевым случаям при анализе программ.

Рассмотрим следующий код (на языках `C` и `JavaScript`):

```
// File: script.js
```

```
let lib = loadLib('lib.so')
let x = lib.x
console.log(x)

// File: lib1.c
static void* x;
int main(int argc, char** argv) {
    x = malloc(max(sizeof(int), sizeof(float)));
    int* p1 = (int*)x;
    float* p2 = (float*)x;
    *p1 = 10;
    *p2 = 20.5;
}

// File: lib2.c
static void* x;
int main(int argc, char** argv) {
    x = malloc(max(sizeof(int), sizeof(float)));
    int* p1 = (int*)x;
    float* p2 = (float*)x;
    *p2 = 20.5;
    *p1 = 10;
}
```

В данном сценарии переменной `x` нельзя присвоить какой-либо конкретный тип, так как он в сущности зависит от исполнения программы. Информация о том, откуда появилась `lib.so` может содержаться в системном окружении или параметрах сборки проекта и в зависимости от этого анализатором могут выдаваться разные результаты. Но консервативный анализ не позволит получить какого-либо типа только из типовых сигнатур в принципе, поэтому при связывании слабо типизированных идентификаторов между собой будет сильно страдать полнота анализа.

2.1.3 Лямбда-исчисление с подтипами $\lambda_{<:}$

Подтипы являются очень важным признаком богатой системы типов и сравнимы с простыми системами типов аналогично тому как отличаются структурные языки от языков ассемблера. Способность иерархического структурирования типов позволяет задавать очень сложные конструкции, иначе не типизируемые в простых системах.

Одним из главных введений в системе с подтипами является определение операции $<:$ над типами, т.е. суждение $S <: T$ значит, что тип S является подтипом типа T и значения этого типа могут использоваться в том же контексте что и значения типа T.

Через определение отношения подтипизации между составными типами (функциями, записями, суммами и т.д.) можно серьезно расширить полноту анализа. Особенно популярны подтипы в ООП языках, где на их основе реализуются принципы переиспользования кода, подклассы и полиморфизм подклассов.

Основной сложностью в данной системе является то, что некоторые проверки типовой информации переносятся во время исполнения программы. Однако, следует заметить, что такие проверки выносятся во время исполнения только при наличии в языке возможности нисходящего приведения типов.

$S <: T \vdash x: S \Rightarrow cast(T, x) : T$

К сожалению, без нисходящего приведения типов многие ООП языки теряют большую часть своей функциональности. Это касается многих популярных языков, таких как `Java`, `C#` или `C++`.

Рассмотрим следующий код (на языке `Java`):

```
package Program;
class Program {
    public static Object foo() {
        return new Int(10);
    }
    public static void main(String args[]){
        Object x = foo();
        System.out.println((Int)x);
    }
```

```
}
```

Здесь, переменная `x` имеет тип `Int`, но это известно только во время исполнения программы, так как для того чтобы это узнать необходима интерпретация кода функции `foo`. В частности, все системы типов с подтипами имеют определенные динамические аспекты. При этом, ситуация ухудшается для динамических языков типа `JavaScript` или `Python`, так как подтипизация сопровождается отсутствием явных аннотаций типов, что полностью лишает анализ какой-либо полноты.

В общем случае анализ хоть и является неполным (в том отношении что не предоставляет реального типа сущности во время исполнения программы), все же позволяет выявить большее число правильно типизированных термов.

Несмотря на перечисленные минусы, для нужд анализа такая система является очень хорошим дополнением, существенно расширяющим простую систему типов. Также, допустимы некоторые расширения данной системы, например типы объединения и типы пересечения (или их более слабая версия, типы-уточнения). Такие расширения могут сгладить вышеперечисленные недостатки.

Следует отметить, что рассмотренные варианты систем типов (простое типизированное лямбда исчисление с ссылками и лямбда-исчисление с подтипами) страдают от похожих проблем — часть информации о типах содержится в сущностях времени исполнения и не может быть статически определена путем простого анализа кода без интерпретации. Следует рассмотреть более сложные системы типов которые могли бы выразить семантику последовательного исполнения.

2.1.4 Система F (а также её расширения)

Основная отличительная особенность системы F от других предложенных систем заключается в том, что путем расширения просто типизированного лямбда исчисления универсальными квантификаторами можно получить

существенно более мощное исчисление, в котором существенно повышается количество типизируемых термов.

Стоит сразу отметить, что такая система хорошо подходит для анализаторов с практической точки зрения, так как имеет под собой четко сформулированную и хорошо изученную теорию как самого исчисления, так и принципов реконструкции типов (англ. type inference).

Экзистенциальные типы как дополнение системы F, описанное в книге [3], может являться хорошим способом моделирования ”непрозрачных” или ”полупрозрачных” типов. Это особенно полезно для ООП языков или языков с модулями, где некоторые номинальные типы могут быть смоделированы как экзистенциальные с некоторым подлежащим типом-представлением (который может быть неизвестен на момент анализа).

Рассмотрим следующий код (на языке `Golang`):

```
package main
import ”fmt”

type Numeric interface {
    toInteger() int
}

type number struct {int}
func (n number) toInteger() int {
    return n.int
}

func main() {
    var n Numeric = number{10}
    fmt.Println(n.toInteger())
}
```

Здесь объявляется тип `Numeric` который является интерфейсом с методом `toInteger`. Затем, объявляется структурный тип `number` который является типом, реализующим такой интерфейс (в Go реализация интерфейсов

не производится явно, в отличие от таких языков как Java). В функции `main` происходит вывод значения структуры `number` благодаря методу интерфейса.

В данном примере тип `Numeric` может быть смоделирован как экзистенциальный тип вида `{∃ Numeric, { toInteger: Unit -> Int } }`. Затем, так как структурный тип `number` реализует этот интерфейс (неявно), этот тип может представлять из себя такую конструкцию: `{ *number, { int: Int, toInteger: Unit -> Int } } as {∃ Numeric, { toInteger: Unit -> Int } }`. Следует отметить, что реализация нескольких интерфейсов одним типом происходит аналогично, достаточно использования вложенных экзистенциальных пакетов.

Существует расширение системы F — $F_{<:}$ (читается как F-sub). Это расширение позволяет использовать всю полноту и мощность системы F, дополняя её подтипами и возможностью спецификации ограниченной квантификации (т.е. допустима запись вида `∀X <: Int` что обозначает любой тип, являющийся подтипом `Int`).

По книге [3] (глава 28.5) полная $F_{<:}$ является неразрешимой системой — не существует алгоритма проверки (и вывода) типов, завершающегося на всех входных данныъ. Это, в свою очередь, означает что такую систему не стоит использовать при разработке анализатора (по крайней мере с самого начала) так как она неоптимальна на данном этапе. Однако, существует вариант $F_{<:}$, называемый ядерной $F_{<:}$, в котором такая проблема отсутствует и при этом не происходит сильных потерь в полноте анализа. Также, ядерная система $F_{<:}$ расширяется экзистенциальными типами и типами-пересечениями/объединениями с сохранением своих первоначальных свойств без серьезных трудностей.

Стоит заметить, что системы высших порядков (такие как F_{ω} и её расширения) являются на данный момент слишком сложными для анализа, поэтому их рассмотрение стоит продолжить в будущем. На данный момент, самым адекватным решением является выбор ядерной системы $F_{<:}$ как базиса для системы типов межъязыкового анализатора. Одним из достоинств такого подхода является возможность последовательного расширения системы типов, что позволит быстро разработать прототип анализатора с более простой системой (например

с просто типизированным лямбда исчислением) и постепенно развивать его в будущем для повышения полноты анализа.

При рассмотрении различных систем типов был опущен один из классов типов — рекурсивные типы. Такие типы обычно моделируются через упоминания идентификаторов (в номинальной системе типов) либо через оператор μ (в структурной системе типов). На данный момент неясно, стоит ли вводить рекурсивные типы в предложенную систему и насколько это будет эффективно с точки зрения полноты анализа. Необходимы дальнейшие исследования данного вопроса после более тщательной проработки сценариев использования метода.

2.2 Рассмотреть системы типов с эффектами

Системы типов с эффектами (и в дополнение с коэффектами) являются перспективным направлением развития теории типов. Основная идея эффектов в том, что они позволяют моделировать так называемые ”побочные эффекты” (англ. side-effect) функций на уровне типов.

Рассмотрим следующий код (на языке Koka [6]):

```
// declare an abstract operation: emit, how it emits is defined
   dynamically by a handler.
effect fun emit(msg : string) : ()

// emit a standard greeting.
fun hello() : emit ()
  emit(”hello world!”)

// emit a standard greeting to the console.
pub fun hello-console1() : console ()
  with handler
    fun emit(msg) println(msg)
  hello()
```

В данном фрагменте происходит следующее: объявляется эффект `emit`, который в свою очередь используется в сигнатуре функции `hello`. Семантика эффекта `emit` — возврат строки. Затем, в функции `hello-console1` объяв-

ляется обработчик этого эффекта, который выводит строку возвращенную из функции с таким эффектом на экран. Этот обработчик используется в связке с вызовом функции `hello`, что позволяет вывести строку ”Hello world!” на экран.

Семантика коэффектов немного сложнее — коэффект в сигнатуре функции означает, что функции для вызова (либо для другой операции над функцией как объектом) требуется определенный объект определенного типа. Эту семантику можно назвать ”вынесением” информации об окружении функции в её сигнатуру.

Семантики эффектов (и коэффектов) можно добиться, в том числе, при введении в язык специальных концептов класса монад, которые например активно используются в Haskell [7]. В связи с этим, а также в связи с тем, что системы эффектов и коэффектов сейчас активно развиваются, дальнейшее применение таких систем является неоптимальным. Однако, стоит заметить что в некоторых сценариях системы эффектов могли бы оказаться крайне полезными. Необходимо дальнейшее изучение пользовательских сценариев для анализа применимости систем с эффектами или коэфффектами.

2.3 Сформулировать способ отображения типовой информации конкретного языка в избранную систему типов

Так как типовая информация конкретного языка представима в виде синтаксического дерева, логичным является использование обычного синтаксического разбора на уровне дерева. При этом стоит заметить, что такой разбор эффективен только в том случае когда целевая система типов является структурно типизируемой и рекурсивные типы отсутствуют. Таким образом, целевая ядерная система $F_{<:}$ отлично сочетается с такого рода синтаксическим разбором. Предполагается использование разбора, описанного в ”Engineering a Compiler” [8] (стр. 198).

2.4 Описать семантику фрагментов и процесс их связывания

На данный момент высокоуровневое описание семантики фрагментов представлено на рисунке 2.1.

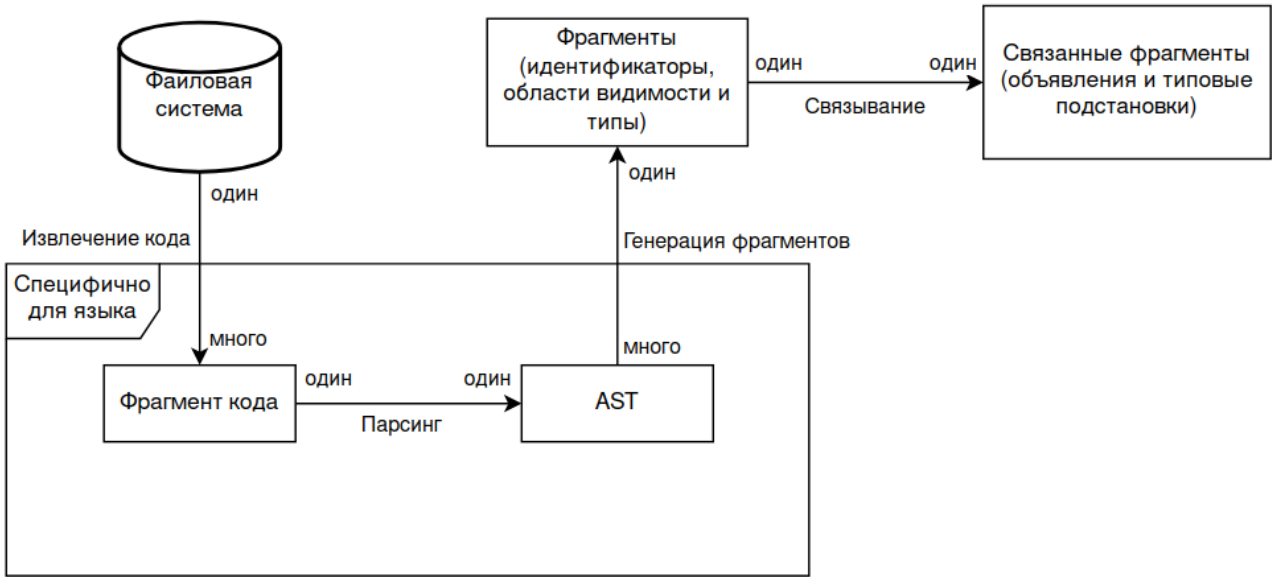

Рисунок 2.1 – Высокоуровневое описание семантики фрагментов

Общие шаги метода представлены на рисунке. Ниже приводится краткое описание шагов, их особенности и возможные трудности.

2.4.1 Извлечение кода

На данный момент эта часть метода является самой малоизученной по следующим причинам:

– специфика извлечения кода сильно зависит от окружения в котором он находится;
– существует бесконечно много различных конфигураций системного окружения, которые слабо формализуемы;
– виды фрагментов (по размеру или языку) сильно зависят от сценариев использования метода;
– на данный момент анализ отдельных фрагментов кода был достаточным для проработки метода.

Тем не менее, предполагается, что этот этап межъязыкового анализа является одним из самых важных и заслуживает внимания для дальнейшей

корректной проработки метода. Как уже было сказано, на данный момент возможно сведение этого этапа к получению списка файлов, описанного вручную.

2.4.2 Парсинг

О том, почему было решено использовать представление типа AST я писал в предыдущем отчете, во втором семестре. Теперь стоит описать принцип генерации такого AST.

В общем случае, для каждого отдельного языка требуется написание определенного парсера — отображения из текстового представления (строки) в структурированное дерево (возможно в определенном формате). Такое отображение должно при этом исполнятся относительно быстро. Для решения данной проблемы решено было использовать один из самых очевидных вариантов — генератор парсеров.

Используя генератор парсеров и грамматику определенного языка, можно получить парсер, способный распознавать язык и генерировать его `Parse tree`. Такое дерево, хоть и громоздко, содержит всю необходимую информацию для дальнейших этапов анализа. Как только парсер сгенерирован, при следующем вызове анализатора на конкретном фрагменте конкретного языка его исполняемый код можно переиспользовать, что позволяет проводить парсинг в реальном времени.

2.4.3 Генерация фрагментов

До текущего момента всё предыдущее исследование было сосредоточено на генерации фрагментов и их связывании посредством использования типовой информации. Таким образом, синтаксис фрагментов описывается следующей EBNF:

```
# Fragment
# Implies ({ start: file_line_col, end: file_line_col }, lang:
   <ID from ontology>)
F ::= ID
```

```
# Alias for a type
A ::= ID '=' T
# Type of a fragment
T ::=
    | T '->' T                          # Function
    | T '*' T                           # Product
    | T '+' T                           # Sum
    | T '&' T                           # Intersection
    | T '|' T                           # Union
    | '{' (ID ':' T ',')* '}'          # Record
    | Unit | Any | Opaque               # Builtins
# Typed fragment (term)
t ::= '(' F ')' ':' T
# Collection of fragment terms
# In case of lhs of judgement ',' means conjunction
# In case of rhs of judgement ',' means multiple conclusions
ts ::= t (',' t)*
# Judgement
# ';' here means disjunction
J ::= ts (';' ts)* '|-' ts
# Block of judgements
# Judgements on the left of 'with' use judgements on the right
   for inference, but only one level deep
# Judgements within the block are not structured lexically,
   hence
# conclusions below can be used to infer judgments above
B ::= (J '\n')+ ('with' B)?
Start ::= (A '\n')* 'then' B
```

Следует заметить, что такая нотация скорее отражает идеи о составе фрагмента, нежели определяет их четкий формат. Таким образом, составляющие фрагмента (термы), состоят из идентификаторов и типов. Фрагменты собираются в суждения, которые состоят из дизъюнкции коньюнкций (ДНФ [9]). Левая сторона суждения означает предпосылку, а правая — логический вывод. Сами суждения собираются в структурированные лексически блоки, которые позволяют моделировать области видимости.

Рассмотрим следующий код (написан на языках Python, C и Sh):

```
# script.py
import ctypes
l = ctypes.CDLL('./liblib.so')
l.doTwoPlusTwo.argtypes = []
l.doTwoPlusTwo.restype = ctypes.c_int
print(l['doTwoPlusTwo']())

// lib.c
int doTwoPlusTwo() {
    return 2 + 2
}

# build.sh
rm lib.o 2> /dev/null || rm liblib.so 2> /dev/null
cc -c lib.c
cc -shared -o liblib.so lib.o
python3 script.py
```

Результат извлечения кода и парсинга вместе с генерацией фрагментов представлен ниже.

```
[]: Any |- [build.sh]: File
[]: Any |- [lib.c]: File
[ctypes]: File |- [script.py]: File
[lib.c]: File |- [doTwoPlusTwo]: Unit -> Int
with
    [./liblib.so]: String |- [ctypes]: { CDLL: String -> Any }
    [ctypes]: { CDLL: String -> Any } |- [l]: { doTwoPlusTwo: {
       argtypes: List Any, restype: Any } }
    [l]: { doTwoPlusTwo: { argtypes: List Any, restype: Any } }, [
       doTwoPlusTwo]: String |- [l]: String -> Unit -> Any
    [lib.c]: File |- [cc -c lib.c]: File -> File
    [lib.o]: File |- [cc -shared -o liblib.so lib.o]: File -> File
    [script.py]: File |- [python3 script.py]: File -> Any
    [build.sh]: File |- [liblib.so]: File
```

Здесь, фрагменты структурированы лексически, суждения выше могут использоваться для обнаружения связей ниже, но не наоборот. Суждение следует читать так — если верно (существует) то, что находится по левую часть от турникета, то верно и то, что находится по правую часть турникета.

2.4.4 Связывание

В повествовательных целях ниже приводится пример результата процесса связывания фрагментов, представленных в предыдущем разделе.

```
[ctypes]: File, |- script.py: File
[lib.c]: File |- [doTwoPlusTwo]: Unit -> Int
[lib.c]: File |- [cc -c lib.c]: File -> File,
[lib.o]: File |- [cc -shared -o liblib.so lib.o]: File -> File,
[Shell:0] script.py: File |- [python3 script.py]: File -> Any,
[python3 script.py]: File -> Any, |- [build.sh]: File
[build.sh]: File |- [liblib.so]: File
[Python:0] [script.py]: File |- [Shell:0] [script.py]: File
```

Путем рекурсивного сопоставления фактов выявляются соответствующие связи. Интерес здесь представляет транзитивная связь вида `[Shell:0] script.py: File |- [python3 script.py]: File -> Any |- [build.sh]: File |- [liblib.so]: File`. Таким образом, консервативный анализ выявляет связь между файлом `script.py` и файлом библиотеки `liblib.so`. Это свидетельствует о зависимости между этими сущностями, что позволяет, к примеру, вывести предупреждение в IDE разработчика, если библиотеки `liblib.so` нет в файловой системе.

В сущности, представленный процесс связывания является логическим выводом, основанным на фактах (суждениях), собираемых из фрагментов кода. Реализация такого вывода может быть проведена с использованием логического движка.

3 ЭПОХА

3.1 Проанализировать существующие решения для генерации парсеров в рамках платформы CLR

При выборе генераторов была произведена оценка по следующим характеристикам:

- тип грамматики, воспринимаемый результирующим парсером;
- частота обновлений генератора;
- скорость сгенерированного парсера;
- количество грамматик для различных популярных языков.

Было решено использовать `ANTLR` [10] как популярное решение, имеющее все выраженные вышеперечисленные качества. ANTLR использует `LL(*)` алгоритм для парсинга, что подходит большинству индустриальных языков программирования.

Так как выбор генератора влияет лишь на типы грамматик, используемые для создания парсера, возможна замена такого генератора на другое решение в дальнейшем.

3.2 Разработать стратегию извлечения или конвертации типовой информации используя Parse tree фрагмента кода

Как уже было описано, предполагается использовать синтаксическую трансляцию (разбор) для входного AST. В качестве входного формата AST предполагается использовать `S-выражения` как универсальный способ представления структурированной иерархически информации. Подробно эти выражения были описаны мною в отчете по практике второго семестра.

Пример такой трансляции представлен на рисунке 2.1. Фрагмент взят из книги [8].

	Production	Code Snippet
1	$Number \rightarrow Sign\ List$	$\$\$ \leftarrow \$1 \times \$2$
2	$Sign \rightarrow +$	$\$\$ \leftarrow 1$
3	$Sign \rightarrow -$	$\$\$ \leftarrow -1$
4	$List \rightarrow Bit$	$\$\$ \leftarrow \$1$
5	$List_0 \rightarrow List_1\ Bit$	$\$\$ \leftarrow 2 \times \$1 + \$2$
6	$Bit \rightarrow 0$	$\$\$ \leftarrow 0$
7	$Bit \rightarrow 1$	$\$\$ \leftarrow 1$

Рисунок 3.1 – Синтаксическая трансляция элементов двоичного числа

На данном фрагменте приведен пример синтаксической трансляции, переводящей двоичное число, записанное в формате строки в десятичную форму. Синтаксис идентификатора с использованием символа доллар определяет семантику взаимодействия со стеком.

В общем случае, левая часть синтаксической трансляции может содержать любую форму предполагаемого узла дерева (в том числе состоящего из других узлов), а правая часть любое вычисление. Это позволяет настроить транслятор для перевода определенного вида S-выражения в определенный тип, описанный в виде последовательного применения типовых конструкторов.

3.3 Описать онтологию как источник семантики для построения и связывания фрагментов

В ходе исследования было выработано понятие *Онтологии*. Онтология в данном контексте означает набор данных об окружении анализа — в первую очередь это касается языковых конструкций и типов.

В общем случае, онтология включает:

– Правила взаимодействия языков (видимость одного языка из другого, название связи такой комбинации)
– Предметно-ориентированные типы и их конструкторы, а также отношения подтипизации
– Ссылки на пару (Грамматика, Парсер) для каждого вовлеченного языка

Таким образом, построение связей между языками производится только исходя из заранее заданной семантики их взаимодействия. На данный момент такая семантика описывается обычной строкой, но также в дальнейшем могут рассматриваться варианты полноценной типизации такой семантики.

3.4 Интегрировать разработанную стратегию в систему анализа фрагментов

На рисунке 3.2 представлена дополненная схема фронтенда анализатора, вовлекающая разработанную стратегию извлечения и парсинга кода.

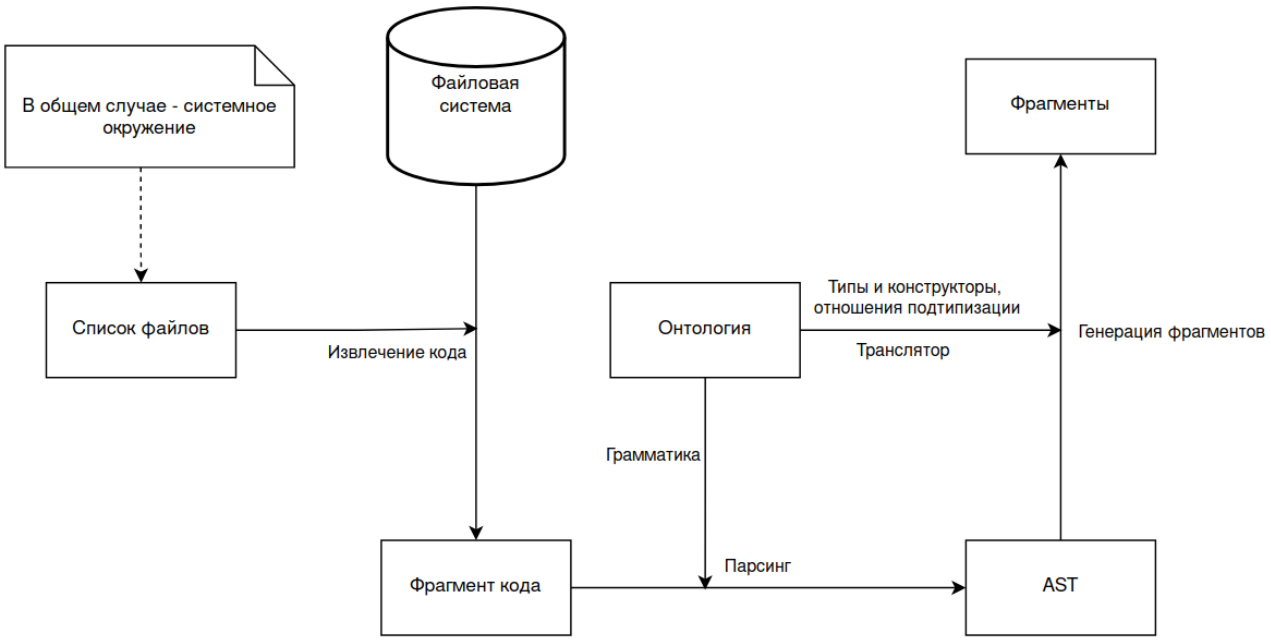

Рисунок 3.2 – Фронтенд анализатора

4 ЭПОХА

4.1 Разработать первоначальную архитектуру межъязыкового анализатора

В этой эпохе исследование предпринимает неожиданный для меня оборот. Дело в том, что в ходе смежной к исследованию деятельности я наткнулся на такой формализм как `графы областей видимости` (англ. Scope Graphs) [11]. В дальнейшем будем называть такой формализм графом областей.

4.1.1 Графы областей

Аннотация статьи гласит, что авторами описывается языконезависимая теория для отражения правил привязки имен (англ. name-binding) и обнаружения областей или определений (англ. scope resolution). Авторы достигают этой задачи путем создания графа, узлы которого представляют различные конструкции для обнаружения определений: определения (англ. declarations), обращения (англ. references) и области видимости (англ. scopes). Затем, вводится исчисление разрешения (англ. resolution calculus) что позволяет производить логический вывод на таком графе и получать пары (обращение, определение) и соответствующие пути (заданные как набор переходов от одного узла к другому).

Путем построения графа областей авторами моделируются различные языковые конструкции — в первую очередь модули и их импорты/включения, а также лексические области видимости и let-связывания различных видов (обычные, рекурсивные, параллельные). Этот подход позволяет описать семантику связывания имен многих языков программирования, которая при этом будет обладать корректностью и полнотой (что доказывается авторами статьи).

При более глубоком рассмотрении идей авторов, стало понятно, что они решают именно ту проблему которая возникла у меня при межъязыковом

анализе, а именно — правильное структурирование сущностей между собой в контексте областей видимости. Именно эта идея лежала в основе адаптированных мною модулей, которые я называл *фрагментами*. Однако, при дальнейшем развитии этого механизма, стало ясно что он не удовлетворяет всем сценариям использования.

На данный момент граф областей является более полным и гораздо более корректным (так как приводится формальное доказательство) решением поставленной мною задачи. Однако, вторая часть поставленной мною проблемы восходит к сложности типизации фрагментов. В данном случае необходимо рассмотреть следующую статью по графам областей — попытку совмещения графов и типовых ограничений.

4.1.2 Графы областей в сочетании с типовыми ограничениями

Следующая статья от тех же авторов [12] посвящена развитию идеи графов областей в контексте типизации.

Графы областей получили развитие в виде новой конструкции — типизированной записи (англ. record). Такая конструкция в сущности моделирует любую структуру иерархически структурированных данных, но в отличие от модулей, запись является типом, который в свою очередь инстанцируется в конкретное значение и связывается с конкретным идентификатором (тип которого без вывода типов или таблицы символов определить невозможно). Таким образом, авторами показывается проблема навигации по графу областей в контексте типовых ограничений — процесс навигации по графу будет в общем случае смешиваться с процессом вывода типов для конкретных идентификаторов и оба этих процесса могут происходить в любом порядке.

Типовые ограничения вводимые авторами вдохновлены алгоритмом унификации W [13]. Однако, вводимые ограничения немного упрощены и не включают, допустим, абстракций по типовыми переменным (т.е. универсальные квантификаторы). Список вводимых типовых ограничений включает:

– ограничение на тип объявления $D : T$;

– ограничение на равенство типов $T \equiv T$, где типом может быть известный тип или типовая переменная;
– ограничение на подтип $T \leq T$;
– ограничение на наименьший общий тип (англ. least upper bound) $TisT_1 \sqcup T_2$.

Как видно из ограничений, такая система соответствует $\lambda_{<:}$ (с учетом существования монотипов и конструктора функции). Конечно, эта система менее полна чем $F_{<:}$, но из практических соображений её вполне достаточно для использования в анализаторе.

Опуская несколько трудностей с которыми столкнулись авторы (ассоциированные области и прямые импорты), алгоритм работает следующим образом:

1. на вход подаются языкоспецифичные типы, их сигнатуры и отношения подтипизации;
2. затем, происходит этап извлечения ограничений из программы путем однопроходной синтаксической трансляции термов из AST в набор (конъюнкцию) ограничений (на граф и на типы).
3. эти данные, называемые истинными фактами (англ. ground facts), интерпретируются в контексте $(G, \leq, \psi)$, где G это граф областей, $\leq$ это порядок над типами, а ψ это контекст типизации;
4. затем, цель алгоритма это соблюдение правдивости следующего утверждения $\mid \phi(F_p), \psi \models \phi(C_p)$, т.е. поиск таких ϕ (подстановки разрешения обращений и областей) и ψ (контекста типизации);

Таким образом, полученные структуры содержат достаточную информацию о типах и разрешениях обращений для использования такой информации по сценариям перечисленным в **??**.

Следует заметить, что авторы столкнулись с техническими сложностями при реализации самого алгоритма и им потребовалось ввести новую форму графа областей такую как *неполный граф областей*, что позволяет строить суждения инкрементально. Такой алгоритм работает как недетерминированная система переписывания кортежей вида $(C, G, F^{<:}, \psi)$, которая может создавать

различные ветви вывода в случае неопределенностей (например, одно обращение разрешается в несколько определений).

Чтобы подытожить, я хотел бы заметить что всё вышеописанное было частично решено в рамках моей работы по межъязыковому анализу, многие конструкции являются очень схожими по смыслу (например Онтология в моей работе и набор фактов о типах и связях в графе). Поэтому, я решаю что разумным решением будет интеграция этого формализма в рамках моей работы по развитию межъязыкового анализа. Из изученных мною работ этих авторов (и тех, кто цитировал их фундаментальное исследование) я не обнаружил схожей с моей работы. Все рассмотренные авторы концентрируются на изучении моноязыковой теории.

В заключение обобщенная схема фронтенда с внедренным формализмом графов областей представлена на рисунке 4.1

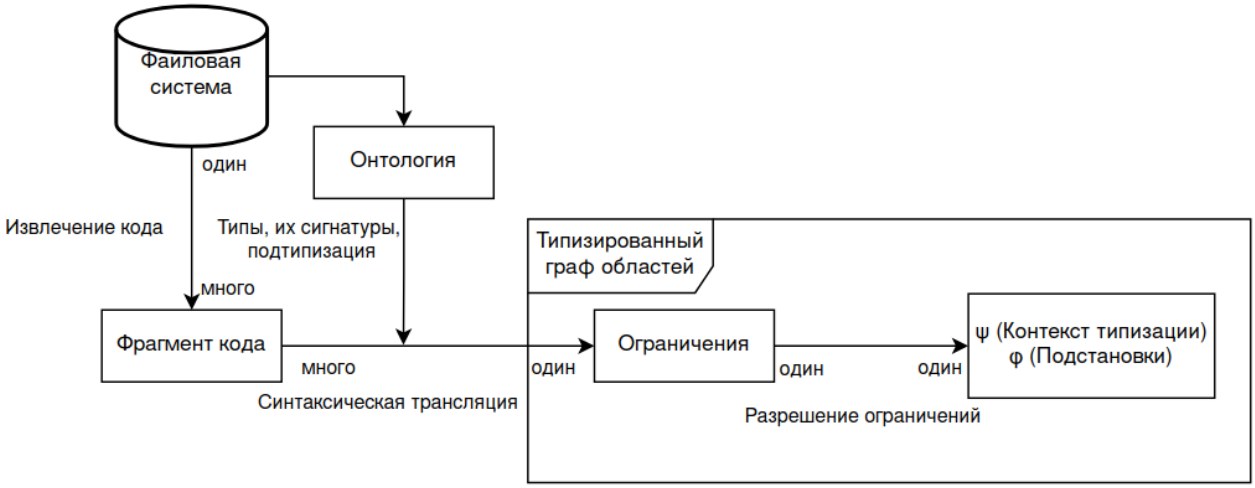

Рисунок 4.1 – Общая схема метода с внедренным формализмом графа областей

4.2 Проработать структуру компонентов анализатора

4.2.1 Общая схема анализатора

Следующая секция определяет структуру анализатора в виде коллекции модулей с соответствующими сигнатурами. Псевдокод приведен на языке схожем с Haskell.

```
type Code = Text
type URL = Text
type ProgramList = [URL]
type Extractor = Filesystem -> ProgramList -> [Code]

module System {
    extractCode :: Extractor -> [Code]
    getOntology :: Ontology

    -- state is a filesystem, immutable
}

type Lang = String
type TB = [b | kind b == 1] -- base types such as Int, Float,
   Text...
type TC = [t | kind t > 1] -- Type constructors such as (->), (
   x), (+)...
type TS = TB || TC
data Relation S T =
    Subtype S T
    ForAllTypesSuper T
    ForAllTypesSubtype S

type Grammar = Text
data AST = forall a. Nil | Cons a AST AST -- heterogenous AST (
   probably with strings)
type Parser = Code -> AST

type Constraints = F && C -- conjunction of facts and proper
   constraints

module Ontology {
    langs :: Set Lang -- for example Java, C#, Python...
```

```
        langLinks :: [{ from :: Lang, to :: Lang, semantic :: Text
           }]
        types :: Set TS
        subtyping :: Set Relation
        grammar :: Lang -> Grammar
        parser :: Lang -> Grammar -> Parser
        translator :: Lang -> AST -> Constraints
    }

    module Generator {
        -- This function is basically all the algorithm of
           constraint generation
        constraints :: [Code] -> Ontology -> Constraints
    }

    type Resolution = (Phi, Psi) -- Multi-sorted substitution and
       typing environment

    module Solver {
        solve :: Constraints -> Set Resolution -- all inferred
           branches that produced valid resolutions
    }

    main = do
        let code <- System.extractCode (\fs, programs -> read fs
           programs)
        let ontology <- System.getOntology
        let constraints <- Generator.constraints code ontology
        let resolutions <- Solver.solve constraints
        forM_ resolutions $ \resolution -> print resolution
```

Такая нотация хоть и поверхностна, но позволяет отразить общую структуру анализатора с его функциональными компонентами. Стоит обратить внимание на особенности синтаксического транслятора — ему не обязательно необходимо быть полным (в понимании типов узлов которые анализируются). В отличие от оригинальной статьи, подход с межъязыковым анализом под-

разумевает анализ только ”внешних” сущностей (т.е. тех, что видны между файлами). Однако, для полноты анализа всё же полезно проводить полную синтаксическую трансляцию, так как в ином случае может теряться транзитивная связь между узлами.

4.2.2 Особенности представления информации о системе

В ходе проработки примеров анализа я столкнулся с проблемой вовлечения информации о системе в процесс анализа. Очевидно, что часто многие компоненты системы в проекте используются неявно или недостаточно явно. В таблице 4.1 описаны компоненты системы и способы обращения к ним в программном коде проекта.

Таблица 4.1 – Различные компоненты системы

Компонент	Способ обращения в проекте
Путь к файлу	Строка
Системная переменная	Идентификатор
Обращение к сети	URL
Различная конфигурация	Структурированные форматы (JSON, XML, TOML ...)

Рассмотрим сначала заключительный пункт таблицы, так как это самый простой вариант конфигурации системы. В данном случае мы имеем файл, который хоть и не является исполняемым кодом, всё же является источником ценной информации о проекте. С такого вида источником всё просто — достаточно использования того же метода, что описан выше.

В случае остальных пунктов, однако, ситуация сложнее. Такая информация не содержится в файле напрямую, а может содержаться, например, в бинарном виде где-то в файловой системе. Решение, которое позволит вовлечь в анализ такого вида информацию достаточно прямолинейно — нужно отобразить эту информацию в том же виде, в каком находится любая другая структури-

рованная информация. Например, для файловой системы можно построить индекс всех (необходимых) файлов в файловой системе в виде, допустим, JSON. Путем анализа такого индекса как обычных структурированных данных (с извлечением графа областей и типовых ограничений) можно получить доступ к различным системным переменным или путям, что позволит анализировать неявные обращения. К таким обращениям можно отнести упоминания файла в скрипте сборки или упоминание URL адреса в коде сервера приложения.

Этот метод отражения информации о системе фактически *опускает* информацию об окружении из внешнего системного контекста на уровень анализатора. Таким образом предположительно можно отразить любую информацию об окружении, но дальнейшее изучение метода отражения такой информации всё же необходимо.

4.2.3 Нечеткий вывод

Предыдущий подпункт вносит в анализатор полноту, однако, корректность такого анализа отсутствует в общем случае. Как можно было убедится из таблицы 4.1 многая семантически важная информация кодируется как строка. Это является очень сильным ограничением так как во многих случаях мы ожидаем сущность являющуюся чем-то определенным, например идентификатором переменной или путем к файлу.

Решением такой проблемы можно предложить продолжение идей из [12], а именно — нечеткого вывода. Каждому утверждению можно присвоить степень истинности (вероятность истинности). В базовом случае истинные утверждения будут иметь степень истинности равную 1, а ложные равные 0. При этом, появится класс утверждений, степень истинности которых лежит в диапазоне от 0 до 1. Соответственно, логический вывод начинает подчиняться законам нечеткой логики [14]. Таким образом, помимо различных ветвей вывода (в случае неопределенностей), каждая ветвь вывода будет иметь определенный показатель истинности. Инструменты, использующие результат такого

анализа, смогут использовать эти показатели как степень уверенности, в различных задачах (например при автодополнении или при поиске объявления).

4.3 Разработать первую модель исполняемой архитектуры анализатора

Как было сказано в предисловии, на текущую эпоху программного прототипа разработано не было ввиду резкой смены принципов анализа (в том числе из-за моего ознакомления с [11]).

Однако, за время исследования было разобрано несколько примеров кода для анализа, который может встречаться в реальном проекте. Ниже представлены такие примеры с процессом их предполагаемого анализа.

4.3.1 Пример анализа вызова функции C из Python

Код проекта приведен ниже. Здесь, каждому идентификатору или строке присвоено число — в настоящем анализаторе это число будет представляться уникальным местоположением символа (например тройкой: файл, строка, колонка).

```
# script.py
import ctypes1
l2 = ctypes3.CDLL4('./liblib.so'5)
l6.doTwoPlusTwo7.argtypes8 = []
l9.doTwoPlusTwo10.restype11 = ctypes12.c_int13
print14(l15['doTwoPlusTwo'16]())



// lib.c
int doTwoPlusTwo17() {
    return 2 + 2
}

# build.sh
rm18 lib.o19 2> /dev/null || rm20 liblib.so21 2> /dev/null
cc22 -c lib.c23
```

```
    cc24 -shared -o liblib.so25 lib.o26
    python327 script.py28
```

Допустим, онтология содержит информацию, представленную ниже.

```
Ontology =
    langs = {”C”, ”Python”, ”Shell”}
    langLinks = [
        { from = ”Python”, to = ”C”, semantic = ”FFI Call” },
        { from = ”C”, to = ”Python”, semantic = ”...” },
        { from = ”Shell”, to = ”C”, semantic = ”Configure compile
           command” },
        { from = ”C”, to = ”Shell”, semantic = ”...” },
        { from = ”Python”, to = ”Shell”, semantic = ”...” },
        { from = ”Shell”, to = ”Python”, semantic = ”Execute script
           ” },
    ]
    types = {
        Top :: *, Bot :: *,
        Unit :: *, Int :: *, String :: *, URL :: *,
        -> :: * -> *, * :: * -> *, + :: * -> *,
        Rec :: * -> *
     }
    subtyping = {
        ForAllTypesSuper Top,
        ForAllTypesSubtype Bot,
        Subtype URL String
    }
    grammar = {
        C = ”/path/to/grammar/C”
        Python = ”/path/to/grammar/Python”
        Shell = ”/path/to/grammar/Shell”
    }
    parser = {
        C = ”/path/to/parser/executable/C”
        Python = ”/path/to/parser/executable/Python”
        Shell = ”/path/to/parser/executable/Shell”
    }
```

```
translator = {
    C = ”/path/to/syntactic/translator/executable/C”
    Python = ”/path/to/syntactic/translator/executable/Python”
    Shell = ”/path/to/syntactic/translator/executable/Shell”
}
```

Тогда, ограничения, извлеченные из кода, будут выглядеть следующим образом. Для обозначений различных узлов AST, используются числовые идентификаторы, введенные в листинге проекта выше. Граф областей видимости представлен на рисунке 4.2.

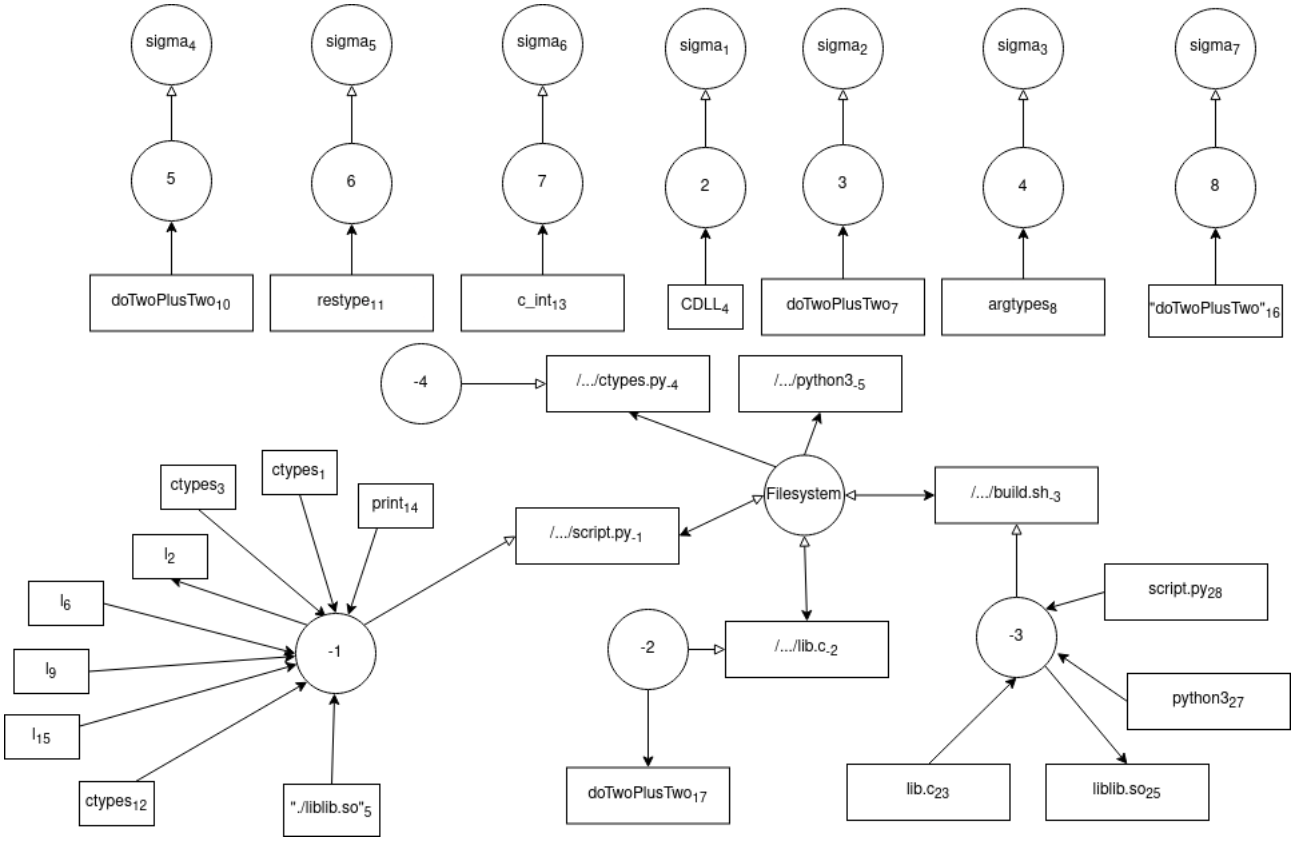

Рисунок 4.2 – Граф областей видимости 1

Здесь, согласно процессу описанному в [12], задается граф видимости символов, в котором:

– файлы представляются как именованные модули, по-умолчанию экспортируемые в системное окружение;
– сущности в файлах представляются как определения или обращения, в зависимости от использованной языковой синтаксической трансляции;

– файловая система задана как одна большая область видимости, в которой видны все файлы-модули и другое окружение (например интерпретатор Python);
– обращения с ассоциированными областями заданы как графовые переменные вида σ_i.

Следует заметить, что граф является сильно упрощенной версией того, что может поступить анализатору в угоду демонстративных целей. Ниже в листинге показаны типовые ограничения и ограничения обращений, извлеченные из кода проекта.

```
Declarations:
    l_2 : τ_1
    doTwoPlusTwo_17 : τ_2
    liblib.so_25 : τ_3
    /.../script.py_-1 : URL
    /.../lib.c_-2 : URL
    /.../build.sh_-3 : URL
    /.../ctypes.py_-4 : URL
    /.../python3_-5 : URL

Reference constraints:
    CDLL_4 -> δ_1                          δ_1 : String -> τ_1
    "./liblib.so"_5 -> δ_2                 δ_2 : String
    l_6 -> δ_3                             δ_3 : Rec(δ_4), δ_4 → σ_2
    doTwoPlusTwo_7 -> δ_5                  δ_5 : Rec(δ_6), δ_6 → σ_3
    ...
    l_15 -> δ_7                            δ_7 : Rec(δ_8), δ_8 → σ_7
    "doTwoPlusTwo"_16 -> δ_9               δ_9 : Unit -> Top
    "lib.c"_23 -> δ_10                     δ_10 : String
    "python3"_27 -> δ_11                   δ_11 : String
    "script.py"_28 -> δ_12                 δ_12 : String

Type constraints:
    τ_1 ≡ Top
    τ_2 ≡ Unit -> Int
    τ_3 ≡ String
```

Решение заданных ограничений отражено в листинге ниже.

```
Solution:
    δ1 = /.../ctypes.py_4                    δ2 = liblib.so25
    δ3 = l2                                  δ4 = liblib.so25
    δ5 = ⊥                                   δ6 = ⊥
    δ7 = l2                                  δ8 = liblib.so25
    δ9 = doTwoPlusTwo17                      δ10 = /.../lib.c_2
    δ11 = /.../python3_5 : URL               δ12 = /.../script.py_1
    τ1 = Top
    τ2 = Unit -> Int
    τ3 = String
    σ2 = -4
    σ3 = ⊥
    σ7 = ⊥
```

Таким образом, видно что произошло связывание различных символов с вовлечением типов. Например, файл lib.c, упомянутый в Shell скрипте был связан с файлом /.../lib.c находящимся в файловой системе. А символ doTwoPlusTwo так как имел функциональный тип был связан с соответствующим определением в файле lib.c.

Такая информация может быть использована в различных инструментальных средствах (например IDE), согласно сценариям использования рассмотренным выше. На рисунке 4.3 отражен один из сценариев использования, а именно автодополнение идентификатора.

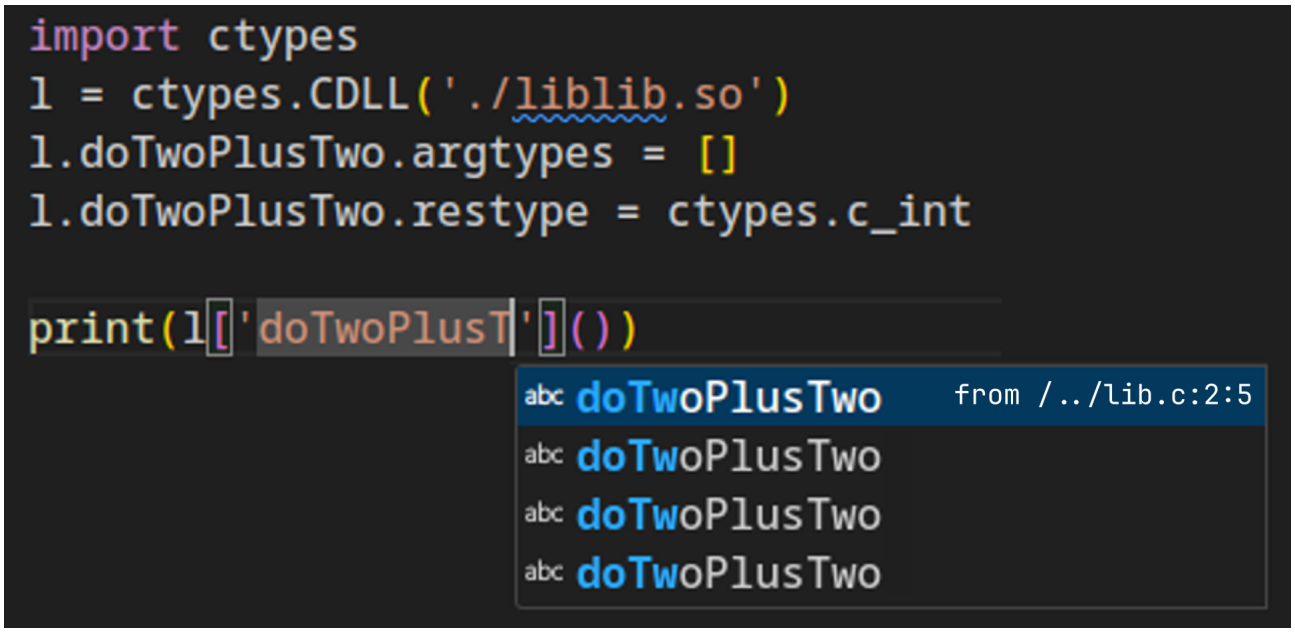

Рисунок 4.3 – Пример использования анализа в IDE для автодополнения

Здесь, за счет транзитивной связи liblib.so с файлом lib.c, обнаруживается определение возможного символа doTwoPlusTwo которое может быть предложено программисту с соответствующим источником.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Language Server Protocol [Электронный ресурс] // Microsoft URL: https://microsoft.github.io/language-server-protocol/ (Дата обращения: 08.09.2023)

2. Language Server Index Format [Электронный ресурс] // Sourcegraph URL: https://code.visualstudio.com/blogs/2019/02/19/lsif (Дата обращения: 30.11.2023)

3. Benjamin C. Pierce. 2002. Types and Programming Languages (1st. ed.). The MIT Press.

4. Системы типизации лямбда исчисления [Электронный ресурс] // Lektorium URL: https://www.lektorium.tv/course/22797 (Дата обращения: 12.07.2023)

5. Слабые типовые переменные в ML [Электронный ресурс] // Ocamlverse URL: https://ocamlverse.net/content/weak_type_variables.html (Дата обращения: 05.01.2024)

6. Язык программирования Koka [Электронный ресурс] // Koka-lang URL: https://koka-lang.github.io/koka/doc/index.html (Дата обращения: 05.01.2024)

7. Всё о монадах [Электронный ресурс] // Haskell-wiki URL: https://wiki.haskell.org/All_About_Monads (Дата обращения: 10.01.2024)

8. Cooper, Keith & Torczon, Linda. (2011). Engineering a compiler: Second edition. 1-800.

9. Поздняков С.Н., Рыбин С.В. Дискретная математика. — С. 303.

10. ANTLR official documentation [Электронный ресурс] // ANTLR URL: https://github.com/antlr/antlr4/blob/master/doc/index.md (Дата обращения: 10.08.2023)

11. Neron, P., Tolmach, A., Visser, E., Wachsmuth, G. 2015. A Theory of Name Resolution. In: Vitek, J. eds Programming Languages and Systems. ESOP 2015. Lecture Notes in Computer Science, vol 9032. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-46669-8_9

12. Hendrik van Antwerpen et al, ”Language-Independent Type-Dependent Name Resolution,” Delft University of Technology, Software Engineering Research Group, Tech. Rep., 2015 https://researchr.org/publication/TUD-SERG-2015-006.

13. M. Wand. A simple algorithm and proof for type inference. Fundamenta Infomaticae, 10:115–122, 1987

14. Novák, V.; Perfilieva, I.; Močkoř, J. 1999. Mathematical principles of fuzzy logic. Dordrecht: Kluwer Academic. ISBN 978-0-7923-8595-0.