Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ГЕНЕРАЦИЯ KOTLIN КОДА С ПОМОЩЬЮ НЕЙРОННОЙ СЕТИ АРХИТЕКТУРЫ TRANSFORMER ДЛЯ ФАЗЗИНГА КОМПИЛЯТОРА

Автор: Тихонов Виталий Андреевич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Фильченков А.А., канд. физ.-мат. наук _____

Санкт-Петербург, 2021 г.

Обучающийся Тихонов Виталий Андреевич
Группа М3436 Факультет ИТиП

Направленность (профиль), специализация
Математические модели и алгоритмы в разработке программного обеспечения

Консультанты:

а) Петухов В.А., без степени, без звания _____

ВКР принята «_____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «15» июня 2021 г.

Секретарь ГЭК Павлова О.Н. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

УТВЕРЖДАЮ

Руководитель ОП

проф., д.т.н. Парфенов В.Г. _____

«____» _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Тихонов Виталий Андреевич

Группа M3436 **Факультет** ИТиП

Квалификация: Бакалавр

Направление подготовки: 01.03.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы: Математические модели и алгоритмы в разработке программного обеспечения

Тема ВКР: Генерация Kotlin кода с помощью нейронной сети архитектуры Transformer для фаззинга компилятора

Руководитель Фильченков А.А., канд. физ.-мат. наук, доцент, научный сотрудник Университета ИТМО

2 Срок сдачи студентом законченной работы до: «31» мая 2021 г.

3 Техническое задание и исходные данные к работе

Требуется попробовать применить нейронные сети, построенные на архитектуре Transformer для генерации кода на языке Kotlin. В архитектуру сети должна быть заложена генерация кода не только в соответствии с грамматикой языка (правила синтаксиса), но и в соответствие с некоторыми правилами семантики.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

а) Провести обзор существующих архитектур нейронных сетей Transformer и сделать аргументированный выбор трансформера для генерации кода.

б) Изучить существующие способы делать code embedding и выбрать наиболее подходящий для решаемой задачи.

в) Обучить сеть на наборе тестов компилятора и/или фрагментов кода в багтрекере.

г) Применить построенный генератор кода для поиска проблем в компиляторе Kotlin: выбрасываемых исключений и проблем с производительностью.

5 Перечень графического материала (с указанием обязательного материала)

Графические материалы и чертежи работой не предусмотрены

6 Исходные материалы и пособия

Исходные материалы и пособия работой не предусмотрены

7 Дата выдачи задания «01» сентября 2020 г.

Руководитель ВКР _____

Задание принял к исполнению _____ «01» сентября 2020 г.

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Тихонов Виталий Андреевич

Наименование темы ВКР: Генерация Kotlin кода с помощью нейронной сети архитектуры Transformer для фаззинга компилятора

Наименование организации, в которой выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработать модель поддерживающую обучение элементам семантики языка для более качественной генерации кода

2 Задачи, решаемые в ВКР:

а) Реализовать модель для генерации кода

б) Раширить модель, поддержав обработку некоторой семантичекой информации

в) Оценить пользу от добавления семантичекой информации

г) Использовать модель для генерации кода и выявления проблем компилятора

3 Число источников, использованных при составлении обзора: 0

4 Полное число источников, использованных в работе: 5

5 В том числе источников по годам:

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	0	0	5	0	0

6 Использование информационных ресурсов Internet: да, число ресурсов: 5

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	**Раздел работы**
IntelliJ IDEA	2, 3
PyCharm	2, 3
Google Colab	3

8 Краткая характеристика полученных результатов

На данный момент результатов получено не было

9 Гранты, полученные при выполнении работы

нет

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

нет

Обучающийся Тихонов В.А. _____

Руководитель ВКР Фильченков А.А. _____

«____» _____ 20___ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1. Обзор существующих решений 7
1.1. Способы обработки кода 8
1.1.1. Тривиальные подходы 8
1.1.2. Графовые нейронные сети 9
1.1.3. Дерево разбора как список правил 10
1.1.4. Дерево разбора как набор путей 11
1.1.5. Выбор модели 11
1.2. Подходы к извлечению семантической информации 11
1.2.1. Предсказание типов 11
1.2.2. Семантические ребра 11
1.3. Улучшение модели 12
Выводы по главе 1 12
2. Обработка семантики 13
2.1. Добавление семантики 13
2.1.1. Типы 13
2.1.2. Семантические токены 13
Выводы по главе 2 13
3. Реализация 14
3.1. Имплементация модели 14
3.2. Извлечение путей 14
3.3. Архитектура генератора 14
3.4. Интеграция сематники 14
3.5. Результаты генерации 14
Выводы по главе 3 14
ЗАКЛЮЧЕНИЕ 15
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 16

ВВЕДЕНИЕ

Процесс тестирования очень важен при разработке приложений. Причина в том, что продукт, работающий некачественно может доставлять пользователям дискомфорт, приводить к потере времени и денег, а это в свою очередь может стать поводом для отказа от продукта. Еще важнее тестировать приложения, не являющиеся конечным продуктом, ведь проблемы в них могут затронуть еще большее число пользователей. Одним из таких приложений являются компиляторы языков программирования.

Классический подход в разработке тестов - написание тестов программистами, сразу после обновления функционала программы. Очевидно, что таким способом сложно протестировать настолько большую программу как компилятор достаточно хорошо, поэтому необходимы и другие методы тестирования.

Примером такого метода может быть фаззинг. В процессе фаззинга для тестируемой программы генерируется большое количество экземпляров входных данных. Для каждого примера входных данных программа запускается независимо, и исследуются некоторые характеристики ее работы, такие, например, как затрачиваемые память и время. Входные данные на которых поведение программы аномально (например, фиксируется большое количество затраченной памяти или долгое время работы) выделяются для дальнейшего изучения программистом.

В данной работе в качестве тестируемого приложения выбран компилятор языка Kotlin. В случае с фаззингом компитлятора в качестве входных данных будут использоваться программы, написанные на соответствующем языке, а критериями аномальности могут служить время компиляции, затраченная память, выброшенные исключения, отличающееся поведение откомпилированных входных программ на разных платформах или разных версиях компилятора.

Получается, что в данном случае задача тестирования сводится к задаче генерации кода. Есть два основных аспекта, которые следует учитывать при генерации кода - это синтаксис и семантика языка. С синтаксисом языка программирования не должно возникать серьезных проблем - он описывается формальной грамматикой, и нет трудностей в генерации кода на ее основе. С семантикой все сложнее - у нее нет формального описания. В случае Kotlin’а она описывается спецификацией на естественном языке. Реализовать генера-

тор, полностью поддерживающий спецификацию - задача сопоставимая с разработкой самого компилятора. Более того, генератор будет иметь неформальную основу, а значит высоки шансы допустить ошибки при его разработке.

При этом опираться только на синтаксис нельзя - доля семантически некорректных, а значит некомпилируемых программ слишком высока. Поэтому возникает желание изучить семантику с помощью машинного обучения, вместо того чтобы формализовывать ее или писать сложный генератор, описывая всю спецификацию. Однако современные решения почти не предусматривают непосредственную передачу модели семантической информации в качестве входных данных.

Таким образом формулируется цель работы - разработать модель нейронной сети, обучаемую семантике языка Kotlin для генерации корректного кода для фаззинга компилятора.

В первой главе описываются некоторые существующие подходы к извлечению информации из исходного кода, выбирается базовый подход для дальнейшего улучшения. Во второй главе рассмативаются возможные улучшения базового подхода за счет добавления информации о семантике. Заключительная третья глава рассказывает о процессе реализации, проблемах, возникших на этом этапе и их устранении.

ГЛАВА 1. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Для начала необходимо разобраться разобраться как работать с такими сложными данными как исходный код. Модели машинного обучения отлично умеют работать с числами. Однако работать напрямую с текстовой информацией они не могут. Тут на помощь приходит механизм эмбедингов - данные разбиваются на структурные компоненты (например, текст можно разбить на слова, а исходный код на токены), и каждой такой компоненте сопостовляется вектор чисел, называемый эмбедингом. Таким образом каждый элемент кодируется точкой в некотором пространстве. Обработанные таким образом данные уже можно подавать на вход модели.

Существуют способы предлагать эти вектора по-умному, сохраняя некоторую семантическую информацию. Например, может получиться так, что вектор между точками, соответствующим словам ”Москва”и ”Россия”будет коллинеарен аналогичному вектору для слов ”Прага”и ”Чехия”. То есть такой вектор будет иметь смысл ”x является столицей y”.

Текст имеет достаточно простую структуру - это по сути просто последовательность слов. Для работы с последовательностями существуют рекурентные нейронные сети, принимающие данные фермент за фрагментом. Исходный же код имеет более сложную древовидную структуру. Это приводит к некоторым интересным спецэффектам. Например, на уровне файла мы можем поменять местами определеня двух функций, и код от этого никак не изменится. Однако, если в тексте поменять местами два абзаца, то смысл может значительно поменяться. Другим примером являются многозначные слова. В случае с Java это может быть слово `final` или `static`, имеющие разные значения в зависимости от контекста, а конструкция `String s` может быть как объявлением переменной, так и параметром функции. В Kotlin’е есть ключевые слова `get`, `set`, `field`, которые в то же время могут использоваться как идентификаторы. Выходит, что рассматривать код как обычный текст - не лучшая затея, так как часть информации из входных данных просто не будет использоваться.

Другое хорошо изученное представление данных - изображения. Для работы с ними так есть целый класс нейронных сетей - сверточные. Изображения рассматриваются как двумерные объекты, полностью заполняя некоторый прямоугольник.

Графовое предствление данных находится где-то по середине, не умещаясь в одномерное пространство, при этом не заполняя полностью двумерное. Поэтому для графов как правило используются специальные подходы.

1.1. Способы обработки кода

Код можно рассматривать в некотором смысле как текст, а значит модели машинного обучения не смогут работать с ним напрямую. Поэтому необходимо разобраться в подходах, позволяющих представлять исходный код в виде векторов. Рассмотрим подробнее несколько подходов по представлению исходного кода и выберем наиболее подходящий для генерации кода.

1.1.1. Тривиальные подходы

Наиболее очевидным подходом к извлечению информации из кода являются разбение исходного кода программы на токены и обучение модели, основанной на рекуррентной сети на полученной последовательности. Как было сказано ранее, в таком случае часть информации утрачивается и не учитывается моделью.

Более разумным будет построение дерево разбора для исходного кода. Деревья разбора содержат гораздо больше полезной информации, чем код, представленный в виде текста. Например, если код из листинга 1 рассматривать последовательно, то мы поймем, что есть некоторый класс `A`, с которым ассоциируются поля `x: Int` и `y: Int`. При этом дерево разбора этого фрагмента, изображенное на рисунке 1 слева имеет более сложную структуру: тут явно указано, что `x: Int` относится к конструктору класса.

Листинг 1 – Пример исходного кода на Kotlin

```
class A(val x: Int) {
    val y: Int = 10
}
```

Чтобы модель могла работать с деревом предлагается обучить эмбединги для каждого типа вершин, а само дерево представять как последовательность вершин в порядке обхода в глубину. Полученную последовательность можно отдавать на вход модели на основе рекуррентной сети. Для графа, изображенного справа на рисунке 1 обход в глубину будет выгляеть следующим образом: P, A, B, D, E, F, C. Но не смотря на то что самом графе расстояние между

вершинами B и C равно двум, в полученной последовательности путь значительно вырос и равен четырем. Для решения такой проблемы можно добавить в путь стрелку вверх, если делается подъем, и стрелку вниз, если делается спуск. В таком случае результат обхода будет следующим: $P, \downarrow, A, \downarrow, B, \downarrow, D, \downarrow, E, \downarrow, F, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, C$. Стрелки - специальные символы, для которых также обучаются эмбединги.

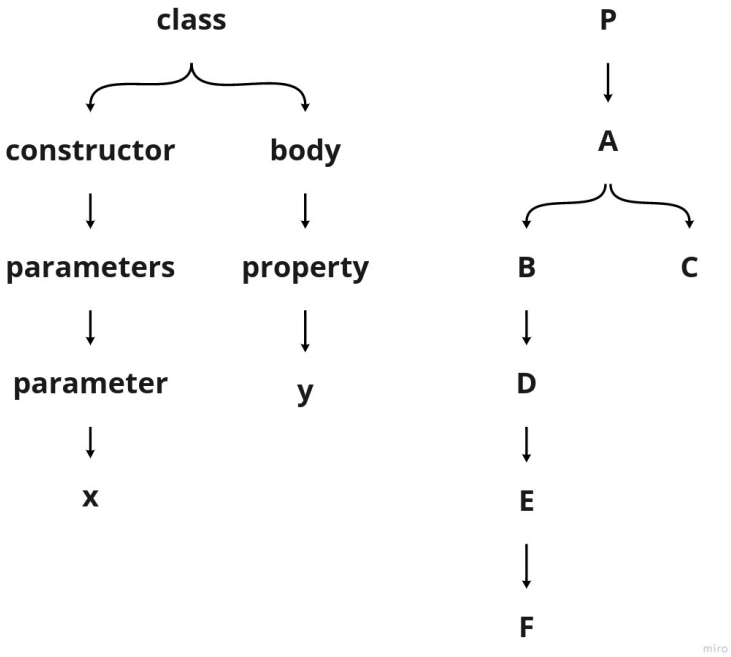

Рисунок 1 – Слева AST для листинга 1, справа - пример дерева

1.1.2. Графовые нейронные сети

Значительно более мощным является подход с использованием графовых нейронных сетей (Graph Neural Network - GNN). Каждой вершине сопостовляется вектор состояния, после чего происходит несколько этапов обмена информации между вершинами. Этап состоит из двух фаз: отправка сообщений и получение сообщений. Во время первой фазы для каждой вершины вычисляется сообщение для отправки. В качестве сообщения используется вектор состояния, преобразованный каким-либо образом. После того как все сообщения вычислины, вершины передают сообщения своим соседям, начинается вторая фаза. Во время фазы получения сообщений каждая вершина агрегирует получаемые от других вершин сообщения, например, в работе [2] сообщения взвешенно усреднются с использованием механизма внимания. Состояние вершины обновлятеся с помощью полученного только что агрегированно-

го сообщения. Это можно сделать, например, с помощью конкетенации этих векторов и линейного предобразования для уменьшения размерности конкатенированного вектора до исходной размерности состояния вершины.

Простой вариант - использовать AST в качестве графа. Однако в статье [1] предлагается дополнить этот граф ребрами data-flow графа. Например, можно соеденить вызов функции и возвращаемое значение из опредленение функции или аргументы вызова с параметрами функции. В таком случае информация по графовой нейронной сети в некоторых случаях будет передаваться также как и в исходном коде.

1.1.3. Дерево разбора как список правил

Дерево разбора можно представить в виде последовательности, но значительно хитрее, чем в описанном ранее случае, где дерево заменялось на список вершин в порядке обхода в глубину. Подход, используемый в статье [5] рассматривает синтаксическое дерево как последовательность приминения правил грамматики.

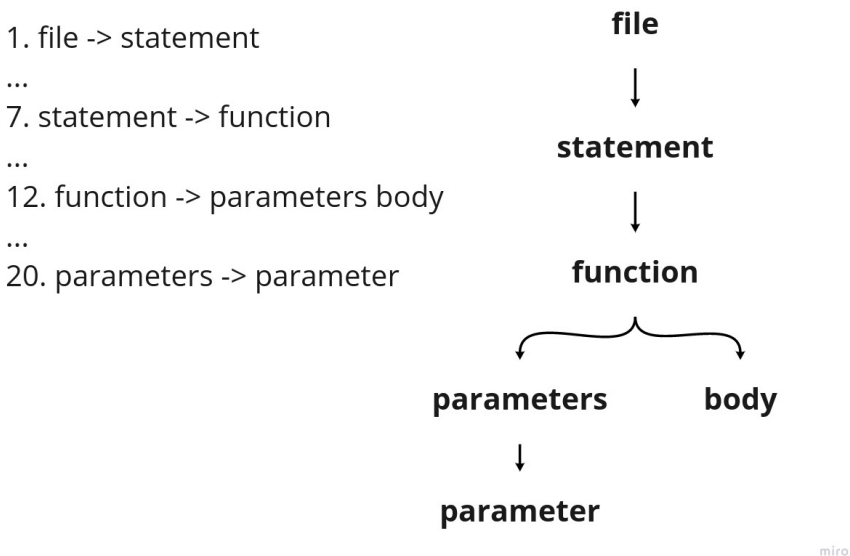

Рисунок 2 – Слева некоторые правила грамматики, справа - дерево, получающееся при последовательном применении правил 1, 7, 12, 20

Рассмотрим пример. Слева на рисунке [**treegen**] указаны некоторые правила грамматики. Будем считать, что `file` - это стартовый нетерминал, тогда после использования правил 1, 7, 12, 20 дерево примет вид изображенный справа на том же рисунке. В качестве текущего нетерминала для раскрытия выбирается первый в порядке обхода в глубину. То есть следующее правило

будет раскрывать нетерминал `parameter`. Такое решение описывает исходный код очень близко к грамматике.

1.1.4. Дерево разбора как набор путей

Еще один подход для представления дерева разбора кода рассмотрен в статье [3]. При таком способе AST представляется в виде набора путей между всеми парами вершин в дереве. Далее эбмединги путей проходят через блок трансформера, механизм внимания позволяет выделить наиболее важные пути, после чего они взвешенно суммируются. Таким образом вычисляется векторное представление примера кода.

Логику этого подхода продолжает статья [4]. Там ставится задача предсказания вершины по контексту (code completion). В качестве входных данных на очередном шаге используются пути от всех листьев до родителя предсказываемой вершины, а также путь от корня дерева. пример, в том числе с несколькими предсказаниями

1.1.5. Выбор модели

Для задачи генерации кода графовые нейронные сети не самый лучший вариант. Во-первых, не приминялись ??. Во-вторых, они тяжело учатся ??.

Подход со списком правил выглядит неплохим вариантом, но что?))) Наверное можно сказать что он постарее, чем SLM и нигде??? больше не при Тем не менее его можно рассмотреть в дальнейших исследованиях.

В данной работе в качестве основы выбран подход из статьи Structural language models of code. В целом подход себя зарекомендовал для решения таких задач как предсказание имени метода и возможно для чего-то еще. Более того, конкретно в этой статье подход используется для генерации (пусть и небольших фрагментов) кода.

надо чета сказать про трансформеры

1.2. Подходы к извлечению семантической информации

налить воды

1.2.1. Предсказание типов

LambdaNet?

1.2.2. Семантические ребра

семантические ребра из ggnn

1.3. Улучшение модели

Практически во всех указанных работах модели не используют напрямую семантическую информацию. Исключением является статья GGNN, где внедряют информацию о типах каких именно, пытаясь обучить решетку на типах. Такой подход позволяет частично сохранить иерархию типов, даже для новых, не видимых на этапе обучения типов. Однако типы определяются не только иерархией. Кроме нее, есть как минимум поля и методы, которые можно вызвать на экземпляре класса.

Поэтому ставится задача внедрить механизмы работы с семантикой в модель

Выводы по главе 1

вфвф

ГЛАВА 2. ОБРАБОТКА СЕМАНТИКИ

Во второй главе исследовательской работы описывается: Предлагаемое теоретическое решение (подход/метод/алгоритм/схема) Обоснование, почему оно удовлетворяет требованиям, сформулированным в первой главе. Теоретическое сравнение с существующими решениями

2.1. Добавление семантики

2.1.1. Типы

Подход для эмбедингов типов

2.1.2. Семантические токены

обрезка путей на невидимой информации?? (private)

Выводы по главе 2

В конце каждой главы желательно делать выводы. Вывод по данной главе — нумерация работает корректно, ура!

ГЛАВА 3. РЕАЛИЗАЦИЯ

3.1. Имплементация модели

Имплементация модели

3.2. Извлечение путей

Фильтрация датасета, afterLastTree, выбор глубины

3.3. Архитектура генератора

Архитектура генератора (в том числе ограничение на длину и ширину)

3.4. Интеграция сематники

Интеграция сематники

3.5. Результаты генерации

Результаты генерации

Выводы по главе 3

фывфв

ЗАКЛЮЧЕНИЕ

В данном разделе размещается заключение.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Miltiadis Allamanis Marc Brockschmidt M. K.* Learning to Represent Programs with Graphs [Электронный ресурс]. — 2018. — URL: https://arxiv.org/abs/1711.00740.

2 *имена не на английском*. Graph Attention Networks [Электронный ресурс]. — 2018. — URL: https://arxiv.org/abs/1710.10903.

3 *Uri Alon Meital Zilberstein O. L.* code2vec: Learning Distributed Representations of Code [Электронный ресурс]. — 2019. — URL: https://arxiv.org/abs/1803.09473.

4 *Uri Alon Roy Sadaka O. L.* Structural Language Models of Code [Электронный ресурс]. — 2020. — URL: https://arxiv.org/abs/1910.00577.

5 *Zeyu Sun Qihao Zhu Y. X.* TreeGen: A Tree-Based Transformer Architecture for Code Generation [Электронный ресурс]. — 2019. — URL: https://arxiv.org/abs/1911.09983.