

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

«На правах рукопису»
УДК 004.02

До захисту допущено:

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«___» _____ 2023 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Системне програмування та
спеціалізовані комп'ютерні системи»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Модифікований метод статичного аналізу коду для рішення задачі
згортки рядкових констант»

Виконав:

студент II курсу, групи КВ-11мн,
Касянчук Дмитро Павлович _____

Науковий керівник:

доцент кафедри СПіСКС, к.т.н.,
Марченко Олександр Іванович _____

Консультант з нормоконтролю:

Рецензент:

Засвідчую, що у цій магістерській дисертації
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____

Київ – 2023 року

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-наукова програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Віталій РОМАНКЕВИЧ

«__» _____ 2023 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту**

Касянчуку Дмитру Павловичу

1. Тема дисертації «Модифікований метод статичного аналізу коду для рішення задачі згортки рядкових констант», науковий керівник дисертації Марченко Олександр Іванович, доцент кафедри СПіСКС, к.т.н., затверджені наказом по університету від «30» березня 2023 р. №1359-с.
2. Термін подання студентом дисертації: 18.05.2023.
3. Об'єкт дослідження: процес згортки рядкових констант програми.
4. Вихідні дані: розроблений метод статичного аналізу коду для рішення задачі згортки рядкових констант; програмна реалізація системи статичного аналізу коду, що базується на положеннях розробленого методу.
5. Перелік завдань, які потрібно розробити:
 - опис предметної області досліджень та аналіз існуючих методів аналізу рядкових констант;
 - розробка модифікованого методу для рішення задачі згортки рядкових констант;
 - програмна реалізація розробленого методу для рішення задачі згортки констант у мові Java;

- оцінка ефективності розробленого методу, а також порівняльний аналіз з існуючими;

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: презентація.

7. Орієнтовний перелік публікацій:

- Касянчук Д.П., Марченко О.І. Модифікований метод статичного аналізу коду для рішення задачі згортки рядкових констант. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2023. № 50. С. 57–64. URL: <https://doi.org/10.36910/6775-2524-0560-2023-50-08>.
- Касянчук Д. П., Марченко О. І. Порівняльний аналіз ефективності методів статичного аналізу коду для рішення задачі згортки рядкових констант. Scientific Collection «InterConf». 2023. № 152. С. 568–573. URL: <https://archive.interconf.center/index.php/conference-proceeding/article/view/3218>.

8. Дата видачі завдання: 07.10.2020.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення літератури за тематикою роботи	10.10.2020	
2	Аналіз існуючих методів аналізу рядкових констант	22.10.2020	
3	Визначення структури магістерської дисертації, а також змісту пояснювальної записки	05.02.2021	
4	Аналіз підходів статичного аналізу програмного коду, робота над другим розділом	09.05.2021	
5	Розробка модифікованого методу статичного аналізу коду для рішення задачі згортки рядкових констант, робота над третім розділом	03.09.2021	
6	Програмна реалізація запропонованого методу для рішення задачі згортки рядкових констант у мові Java, робота над четвертим розділом	10.05.2022	
7	Оцінка ефективності розробленого методу, робота над п'ятим розділом	13.09.2022	
8	Оформлення текстової і графічної частини магістерської дисертації	15.04.2023	
9	Попередній розгляд магістерської дисертації на кафедрі	04.05.2023	

Студент
Науковий керівник

Дмитро КАСЯНЧУК
Олександр МАРЧЕНКО

РЕФЕРАТ

Актуальність теми. Рядки широко використовуються в сучасних мовах програмування, зокрема у рядкових літералах для надання інформації користувачу, у програмах обробки текстів, у програмах, що використовують рефлексію тощо. Тому розробка спеціальних методів статичного аналізу для задачі згортки рядкових констант є актуальною і важливою задачею, як з наукової, так і з практичної точки зору.

Об'єктом дослідження є процес згортки рядкових констант програми.

Предметом дослідження є методи статичного аналізу для рішення задачі згортки рядкових констант.

Мета роботи: прискорення процесу згортки рядкових констант програми; розробка більш швидкого методу чи модифікованого методу згортки рядкових констант програми.

Наукова новизна полягає у наступному.

1. Вперше запропоновано модифікований метод згортки рядкових констант, який полягає у анотуванні пропагованих значень додатковою інформацією, аналіз якої дозволяє не здійснювати конкатенацію значень, які не можуть існувати разом під час реального виконання програми, відрізняється від існуючих використанням вкладеної хеш-таблиці в якості значень основної хеш-таблиці замість структури даних типу «множина» та модифікованої трансферної функції і дозволяє зменшити розмір проміжних результатів аналізу потоку даних, завдяки чому зменшується складність алгоритмів, що використовуються, тим самим покращуючи швидкість і точність згортки рядкових констант програми.
2. Вперше виконано порівняльний аналіз розробленого модифікованого методу із стандартним методом і показано ефективність запропонованого модифікованого методу.

Практична цінність отриманих в роботі результатів полягає в тому, що запропонований модифікований метод дає змогу використати його у більшій кількості застосунків, ніж стандартний метод, в яких треба визначити повну множину рядків, що використовуються в програмі, але наявні методи не можуть бути ефективно використані через їх незадовільну швидкодію.

Апробація роботи. Основні положення роботи були представлені у випуску №50 2023 року наукового фахового журналу «Комп'ютерно-інтегровані технології: освіта, наука, виробництво». Порівняльний аналіз розробленого методу з існуючими був представлений у випуску №152 2023 року науково дослідницького центру «InterConf».

Структура та обсяг роботи. Магістерська дисертація складається з вступу, п'яти розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень.

У першому розділі розглянуто існуючі методи для аналізу рядкових констант, їхні особливості, недоліки та переваги.

У другому розділі розглянуто основні моделі статичного аналізу коду на яких базується запропонований метод.

У третьому розділі розглянуто основні положення запропонованого методу.

У четвертому розділі наведена програмна реалізація системи статичного аналізу коду, що базується на моделях описаних у другому розділі, а також реалізація аналізу згортки рядкових констант у мові Java на основі розробленого методу.

У п'ятому розділі наведено опис системи для оцінки ефективності запропонованого методу на основі його програмної реалізації, а також здійснено порівняльний аналіз з існуючими методами аналізу рядкових констант.

У висновках представлені результати проведеної роботи.

Робота представлена на 91 аркушах, містить 3 додатки та посилання на список використаних літературних джерел.

Ключові слова: статичний аналіз; згортка констант; граф потоку виконання; аналіз потоку даних.

ABSTRACT

Actuality of theme. Strings are widely used in modern programming languages, particularly in string literals to provide information to the user, in text processing programs, in programs that use reflection, and so on. Therefore, the development of special methods of static analysis for the convolution problem of string constants is an urgent and important task, both from a scientific and a practical point of view.

The object of research is the process of convolution of string constants of the program.

The subject of the research is static analysis methods for solving the problem of folding of string constants.

The purpose of the work: speed-up of the process of folding of string constants of the program; development of a faster method or a modified method of convolution of string constants of the program.

The scientific novelty.

1. For the first time, a modified method of folding of string constants is proposed, which consists in annotating the propagated values with additional information, the analysis of which allows not to carry out concatenation of values that cannot exist together during the actual execution of the program, differs from existing methods by using a nested hash table as the values of the main hash table instead of a set data structure and a modified transfer function and allows to reduce the size of the intermediate results of the data flow analysis, which reduces the complexity of the algorithms used, thereby improving the speed and accuracy of folding of the string constants of the program.
2. For the first time, a comparative analysis of the developed modified method with the standard method was performed and the effectiveness of the proposed modified method was shown.

The practical value of the results obtained in the work is that the proposed modified method makes possible to use it in a larger number of applications than the

standard method, in which it is necessary to determine the complete set of lines used in the program, but the existing methods cannot be effectively used due to their unsatisfactory performance.

Approbation of work. The main state of the work are presented in issue No. 50 of 2023 of the scientific journal "Computer-integrated technologies: education, science, production". A comparative analysis of the developed method with the existing ones is presented in issue No. 152 of 2023 of the scientific research center "InterConf".

Structure and scope of work. The master's work includes of an introduction, five chapters and conclusions.

In the introduction, a general description of the work is presented, an assessment of the current state of the problem is made, the relevance of the research direction is substantiated, and the goal and tasks of the research are formulated.

In the first chapter, the existing methods for the analysis of string constants, their features, disadvantages and advantages are considered.

In the second chapter, the main stages of static code analysis, on which the proposed method is based, are considered.

In the third chapter, the main provisions of the proposed method are considered.

The fourth chapter presents the software implementation of the static code analysis system, which consists of the stages described in the second chapter, as well as the implementation of the folding analysis of string constants in the Java language based on the developed method.

The fifth chapter describes the system for checking the effectiveness of the proposed method based on its software implementation, as well as a comparative analysis with existing methods of analyzing string constants.

The results of the work are presented in the conclusions.

The work is presented on 91 sheets, contains 3 appendices and links to the list of used literary sources.

Keywords: static analysis; constants folding; control flow graph; data flow analysis.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ	4
ВСТУП	5
1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБГРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ	6
1.1 Задача статичного аналізу коду	6
1.2 Аналіз існуючих рішень	8
Висновки	16
2. ОСНОВИ СТАТИЧНОГО АНАЛІЗУ ПРОГРАМНОГО КОДУ	17
2.1 Синтаксичне дерево	17
2.2 Граф потоку виконання	20
2.3 Аналіз потоку даних	24
Висновки	27
3. МОДИФІКОВАНИЙ МЕТОД СТАТИЧНОГО АНАЛІЗУ КОДУ ДЛЯ РІШЕННЯ ЗАДАЧІ ЗГОРТКИ РЯДКОВИХ КОНСТАНТ	28
3.1 Основні положення стандартних методів	28
3.2 Опис модифікованого методу	35
Висновки	42
4. ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНОГО МЕТОДУ	43
4.1 Загальний опис системи	43
4.2 Модуль генерації інструментів синтаксичного аналізу	45
4.3 Модуль синтаксичного аналізу програмного коду	49
4.4 Модуль побудови графу потоку виконання	61
4.5 Модуль аналізу потоку даних	70
Висновки	79
5. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЗАПРОПОНОВАНОГО МЕТОДУ	80
5.1 Опис інструментів для оцінки ефективності	80
5.2 Порівняльна характеристика отриманих результатів	85
Висновки	88
ВИСНОВКИ	89

	3
СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ	92
ДОДАТОК 1. Презентація.....	93
ДОДАТОК 2. Лістинг програми	93
ДОДАТОК 3. Копії публікацій	93

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

АСД – абстрактне синтаксичне дерево

БНФ – Бекусова нормальна форма.

РБНФ – розширена Бекусова нормальна форма.

CFG (Control Flow Graph). – граф потоку виконання.

DFA (Data Flow Analysis). – аналіз потоку даних.

MFP (Maximal Fixed Point) – максимальна фіксована точка.

MOP (Meet Over All Paths) – об'єднання всіх шляхів.

Трансферна функція – операція DFA, мета якої модифікація вхідних властивостей вузла графу після аналізу інструкцій у ньому.

Join-функція – операція DFA, мета якої обробка декількох можливих результатів і отримання на їх основі єдиного.

Equal-функція - операція DFA, мета якої порівняння отриманого значення після чергової ітерації з попереднім.

CSV (Comma Separated Values) — файловий формат для представлення табличних даних, у якому поля відокремлюються символом коми та переходу на новий рядок.

ВСТУП

Рядки широко використовуються в сучасних мовах програмування, зокрема у рядкових літералах для надання інформації користувачу, у програмах обробки текстів, у програмах, що використовують рефлексію тощо. Наприклад, у мові PHP рядки є способом обміну між програмами, тоді як у JAVA вони широко використовуються у якості SQL запитів, або для отримання доступу до інформації про класи через рефлексію. Під час виконання інструкції `str.substring(str.indexOf('a'))` виникне помилка у разі, якщо рядок не містить такого символу. Тобто, у цьому випадку було б корисно мати можливість відслідковувати множину можливих символів, які належать конкретному рядку. Іншим же прикладом є робота з SQL запитами. Наприклад, після виконання такого запиту – `"delete from some_tbl where id = " + id`, де `id` – `"10 OR TRUE"` – весь зміст таблиці буде втрачений без можливості відновлення. Таким чином, неправильні маніпуляції з рядками можуть призвести не тільки до виняткових, а також до драматичних і невідновлюваних ситуацій.

З вищесказаного можна зробити висновок, що важливість методів, які автоматично аналізують і виявляють помилки, що можуть виникнути при роботі з рядками, є дуже високою. Але існуючі методи такого типу все ще не повністю задовольняють сучасним вимогам: методи, які базуються на використанні автоматів та регулярних виразах є точними, але повільними, і, головне їх швидкодію важко покращити; багато інших методів є доволі швидкими, але вони є досить спеціалізованими, оскільки орієнтовані на певні властивості або класи програм. Тому задача розробки нових методів, які не були б вузько спеціалізованими і достатньо швидкими, є актуальною.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ТА ОБГРУНТУВАННЯ ТЕМИ МАГІСТЕРСЬКОЇ ДИСЕРТАЦІЇ

1.1 Задача статичного аналізу коду

Основна задача статичного аналізу – це дотримання стандартів кодування, а також перевірка можливих помилок, які можуть з’явитися під час майбутньої роботи програми. У даному підрозділі я опишу статичний аналіз, а також переваги використання автоматизованих інструментів для виконання такого типу аналізу.

Статичний аналіз дозволяє усунути недоліки вихідного коду, які можуть призвести до вразливостей. Звичайно, цього можна досягти за допомогою перегляду кодової бази програмістом, але використання автоматизованих засобів є набагато ефективнішим.

Зазвичай, статичний аналіз використовується для дотримання певних рекомендацій щодо кодування, наприклад, MISRA, або для відповідності галузевим стандартам, таким як ISO 26262.

Розглянемо більш детально різницю між статичним і динамічним аналізом. Обидва виявляють різні дефекти, але велика різниця полягає у тому, в якому циклі розробки вони це роблять. Статичний аналіз визначає дефекти перед запуском програми (наприклад, між написанням коду і модульним тестуванням). Динамічний аналіз робить це після запуску програми (наприклад, під час модульного тестування). Отже, статичний аналіз може виявляти дефекти, які динамічний може вже не визначити.

Статичний аналіз коду використовується для певних цілей на конкретній фазі розробки. Зазначимо деякі обмеження даного інструменту аналізу коду.

1. Він може виявити можливе переповнення в арифметичному виразі, але не зможе визначити, що функція принципово не робить те, що очікується.
2. Деякі правила кодування залежать від зовнішньої документації, або відкриті для суб’єктивного тлумачення.

3. У фрагменті коду (рис. 1.1), через те що поведінка функції `foo` невідома, значення `x`, яке буде отримане в останньому операторі залишається невизначеним. Це означає, що інструменти можуть повідомляти про дефекти, яких насправді не існує(хибно позитивні). Або вони можуть не повідомити про реальні дефекти(хибно негативні).

```
int divide(void)
{
    int x;
    if(foo())
    {
        x = 0;
    }
    else
    {
        x = 5;
    }
    return (10/x);
}
```

Рисунок 1.1 – Приклад можливого дефекту

Інструменти статичного аналізу коду мають кілька переваг, особливо якщо потрібно відповідати галузевим стандартам. Перелічимо властивості найкращих інструментів статичного аналізу коду.

1. Швидкість. Розробникам потрібен деякий час для того, щоб зробити якісне рецензування програмного коду. Автоматизовані інструменти роблять це набагато швидше. Статична перевірка усуває можливі проблеми на ранньому етапі, а також точно вказує на помилки у коді. Основними плюсами цього є раннє виправлення помилок, що, зазвичай, обходиться дешевше.
2. Обсяг. Тестування не може охопити всі можливі шляхи виконання коду, на відміну від статичного аналізу. Він перевіряє код під час його «збирання». У результаті ви отримуєте поглиблений аналіз того, де можуть виникнути потенційні проблеми у вашій кодовій

базі на основі заданих правил. На рисунку 1.2 зображено приклад поглибленого аналізу коду в Helix QAC.

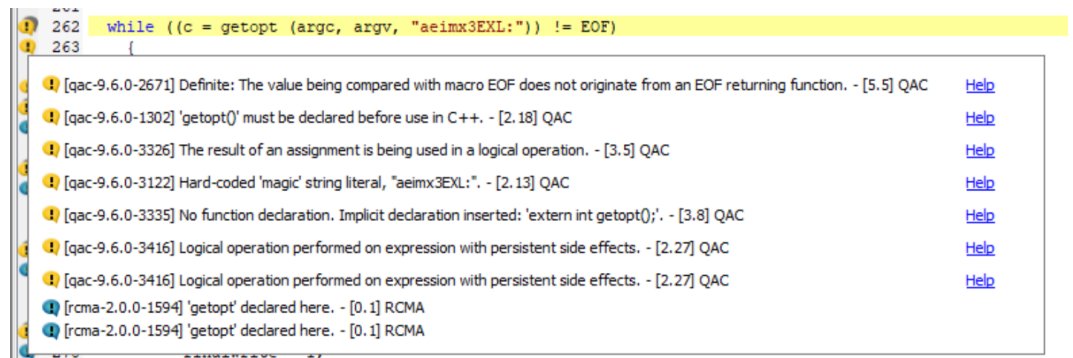


Рисунок 1.2 – Поглиблений аналіз коду

3. Точність. Автоматизовані інструменти не мають схильності до помилок. Вони сканують кожен рядок коду, щоб визначити потенційні проблеми, і це допомагає переконатися, що програмне забезпечення найвищої якості — ще до початку тестування. Зрештою, коли ви дотримуєтеся певних стандартів кодування, якість має вирішальне значення.

1.2 Аналіз існуючих рішень

У цьому підрозділі описані методи аналізу рядків, які базуються на абстрактній інтерпретації і їх реалізацію за допомогою доменів, які відстежують різні типи інформації. Таким чином в залежності від ситуації може використовуватися більш швидкий, але менш точний аналіз, або повільніший, але більш точний.

Мета абстрактної інтерпретації програми полягає у записі обчислень в деякому всесвіті об'єктів для опису обчислень у іншому всесвіті абстрактних об'єктів, так що результат виконання абстрактних обчислень дасть певну інформацію про реальні обчислення.

Абстрактне значення - це апроксимація конкретного значення, яка використовується в межах абстрактної інтерпретації. Нехай, C - це конкретне

значення, A – множина абстрактних значень. Трансформація конкретних значень до абстрактного виконується за допомогою функції абстракції:

$$\alpha : C \rightarrow A$$

Протилежною є функція конкретизації, яка трансформує абстрактне значення у множину конкретних:

$$\gamma : A \rightarrow C$$

Програма на рисунку 1.3 конструює SQL запит за допомогою конкатенації рядків, в якій нас цікавить чи правильно сформований SQL запит.

```

1  var query = "SELECT '$\$$' ||
2    (RETAIL/100) FROM INVENTORY WHERE ";
3  if (1 != null)
4    query = query+"WHOLESALE > "+1+" AND ";
5
6  var per = "SELECT TYPECODE, TYPEDESC FROM
7    TYPES WHERE NAME = 'fish' OR NAME = 'meat'";
8  query = query+"TYPE IN (" + per + ")";
9  return query;

```

Рисунок 1.3 – Тестова програма

Введемо певні псевдоніми (див. таблицю 1.1) для рядкових констант.

Таблиця 1.1 – Скорочення для рядкових констант

Ім'я	Рядкова константа
s_1	“SELECT ' \$ ' (RETAIL/100) FROM INVENTORY WHERE ”
s_2	“WHOLESALE > ”
s_3	“ AND ”
s_4	“SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = ' fish' OR NAME = ' meat' ”
s_5	“TYPE IN (”
s_6	“); ”

Впродовж опису наявних доменів я буду посилатися на ці програми.

Кожна мова програмування містить велику множину операторів для роботи з рядковими константами. Немає сенсу розглядати кожен з цих операторів окремо, адже більша частина з них виконують подібні дії, приймаючи різні дані на вхід. Тому обмежимо список операторами, які використовуються найчастіше у реальних програмах або за допомогою яких можна представити більш складні оператори. Кожен оператор має свою семантику, а також різні методи для її апроксимації, які будуть описані в подальшому.

Операції, які будуть використовуватися в подальшому:

- 1) створення нового рядку - $new\ String(str)$, де str – послідовність символів;
- 2) конкатенація рядків - $concat(s_1, s_2)$, де s_1, s_2 – рядки;
- 3) читання вхідного рядку від користувача – $readLine$;
- 4) взяття підрядка - $substring_b^e(s)$, де s – рядок, b і e – натуральні числа;

Кожен конкретний домен складається з рядків. На основі алфавіту K , який являється скінченною послідовністю символів, визначається послідовність (яка може буде нескінченною) рядків. Задача полягає в ефективній апроксимації таких послідовностей.

Перший домен базується на апроксимації рядка до символів, які, точно або можливо містяться в ньому [1]. Ця інформація є корисною для відстеження доступу до індексів через пошук певного символу у рядках. Рядок представлений парою множин: множина символів, які точно містяться в рядку - $\overline{C}$; множина символів, які можливо містяться в рядку - $\overline{MC}$:

Формула join операції:

$$\sqcup_{\overline{CI}} = \left((\overline{C}_1, \overline{MC}_1), (\overline{C}_2, \overline{MC}_2) \right) = (\overline{C}_1 \cap \overline{C}_2, \overline{MC}_1 \cup \overline{MC}_2)$$

Супремум визначається, як $\top_{\overline{CI}} = (\emptyset, K)$.

В таблиці 1.2 описані абстрактні семантики $\overline{S}_{CI}$ операцій над рядками, які я описав вище, для $\overline{CI}$.

Нехай π_i – взяття i -го елементу пари. Тоді функції абстракції:

$$\alpha_{\overline{CI}}(S_1) = \sqcup_{\overline{CI}, s \in S_1} \alpha'_{\overline{CI}}(s) = \left(\cap_{s \in S_1} \pi_1 \left(\alpha'_{\overline{CI}}(s) \right), \cup_{s \in S_1} \pi_2 \left(\alpha'_{\overline{CI}}(s) \right) \right),$$

$$\alpha'_{\overline{CI}}(s) = (char(s), char(s)),$$

Таблиця 1.2 – Абстрактні семантики $\overline{CI}$

$\overline{\mathbb{S}}_{CI} \llbracket new String(str) \rrbracket ()$	$(char(str), char(str))$
$\overline{\mathbb{S}}_{CI} \llbracket concat \rrbracket ((\overline{C}_1, \overline{MC}_1), (\overline{C}_2, \overline{MC}_2))$	$(\overline{C}_1 \cup \overline{C}_2, \overline{MC}_1 \cup \overline{MC}_2)$
$\overline{\mathbb{S}}_{CI} \llbracket readLine \rrbracket ()$	$(\emptyset, K)$
$\overline{\mathbb{S}}_{CI} \llbracket substring_b^e \rrbracket ((\overline{C}_1, \overline{MC}_1))$	$(\emptyset, \overline{MC}_1)$

В таблиці 1.3 представлено результат аналізу для тестової програми.

Таблиця 1.3 – Результат аналізу для першої програми

№ рядку	Змінна	$\overline{CI}$
1	query	$\alpha'_{\overline{CI}}(s_1)$
3	l	$(\emptyset, K)$
3	query	$(\pi_1(\alpha'_{\overline{CI}}(s_1)) \cup \pi_1(\alpha'_{\overline{CI}}(s_2)) \cup \pi_1(\alpha'_{\overline{CI}}(s_3)), K)$
4	query	$(\pi_1(\alpha'_{\overline{CI}}(s_1)), K)$
5	per	$\alpha'_{\overline{CI}}(s_4)$
6	query	$(\pi_{s_1}(\alpha'_{\overline{CI}}(s_1)) \cup \pi_1(\alpha'_{\overline{CI}}(s_4)) \cup \pi_1(\alpha'_{\overline{CI}}(s_5)) \cup \pi_1(\alpha'_{\overline{CI}}(s_6)), K)$

Наступний домен базується на апроксимації рядків до їх префіксів. Рядок представлений певною послідовністю символів на початку, і будь-якою послідовністю в кінці [1].

Операція join відповідає знаходженню найбільшого спільного префіксу рядків.

Функція абстракції представлена наступною формулою:

$$\alpha_{\overline{PR}}(S_1) = \sqcup_{\overline{PR}, s \in S_1} \alpha'_{\overline{PR}}(s)$$

$$\alpha'_{\overline{PR}}(s) = s$$

Супремум визначається, як ε (пустий префікс).

Також, можна апроксимувати рядки до суфіксів. Визначимо домен $\overline{SU}$, де рядок закінчується певною послідовністю символів [1].

Операція join відповідає знаходженню найбільшого спільного суфіксу рядків, а супремум визначається, як ε (пустий суфікс).

Функції абстракції:

$$\alpha_{\overline{SU}}(S_1) = \sqcup_{\overline{SU}, s \in S_1} \alpha'_{\overline{SU}}(s)$$

$$\alpha'_{\overline{SU}}(s) = s$$

Ці методи можуть бути використанні для перевірки деяких простих синтаксичних властивостей (наприклад, перевірка рядка, який використовується у якості SQL команди, на наявність «SELECT» команди на початку і «;» в кінці) при всіх можливих потоках виконання.

В таблицях 1.4 і 1.5 описані абстрактні семантики $\overline{\mathbb{S}}_{PR}$, $\overline{\mathbb{S}}_{SU}$.

Варто зазначити, що при знаходженні підрядка для $\overline{SU}$ семантика повертає найбільше значення, адже ми не знаємо скільки символів знаходяться до суфіксу. На відміну від нього, $\overline{PR}$ більш точний, якщо b і e менші, ніж довжина відомого нам префіксу.

Таблиця 1.4 – Абстрактні семантики $\overline{PR}$

$\overline{\mathbb{S}}_{PR}[\![new\ String(str)]\!]()$	str
$\overline{\mathbb{S}}_{PR}[\![concat](\overline{p}_1, \overline{p}_2)]$	$\overline{p}_1$
$\overline{\mathbb{S}}_{PR}[\![readLine]]()$	ε
$\overline{\mathbb{S}}_{PR}[\![substring_b^e](\overline{p})]$	$\begin{cases} \overline{p}[b \dots e - 1] & \text{if } e \leq len(\overline{p}) \\ \overline{p}[b \dots len(\overline{p}) - 1] & \text{if } e > len(\overline{p}) \wedge b < len(\overline{p}) \\ \varepsilon & \text{otherwise} \end{cases}$

Таблиця 1.5 – Абстрактні семантики $\overline{SU}$

$\overline{S}_{PR}[\![new\ String(str)]\!]()$	str
$\overline{S}_{PR}[\![concat]\!](\overline{s}_1, \overline{s}_2)$	$\overline{s}_1$
$\overline{S}_{PR}[\![readLine]\!]()$	ε
$\overline{S}_{PR}[\![substring_b^e]\!](\overline{s})$	ε

В таблиці 1.6 показано результат роботи аналізів для тестової програми. Відомо, що наприкінці першої програми, префіксом запиту є рядок - s_1 , суфіксом - s_6 , хоч і інформація між символами між ними втрачена.

Таблиця 1.6 – Результати аналізів $\overline{PR}$ і $\overline{SU}$ для тестової програми

№ рядку	Змінна	$\overline{PR}$	$\overline{SU}$
1	query	s_1	s_1
3	l	ε	ε
3	query	s_1	s_3
4	query	s_1	“ ”
5	per	s_4	s_4
6	query	s_1	s_6

В основі останнього домену $\overline{SG}$ лежить граф для представлення рядків. Така структура представляється триплетом $(\overline{N}, \overline{A}_F, \overline{A}_T)$, де $\overline{A}_F, \overline{A}_T$ – прямі і зворотні дуги. Кожен вузол $\overline{n} \in \overline{N}$ має мітку, яка позначається як $\overline{lb}(\overline{n})$, що описує вид терму, який представляє даний вузол. Для позначення і-ої дитини вузла будемо використовувати запису виду $\overline{n}/i$, а множини всіх дітей - $\{\overline{n}/1, \dots, \overline{n}/k\}$, де $k = \overline{outdegree}(\overline{n})$, а $\overline{outdegree}$ – це функція, яка повертає кількість дітей вхідного вузла [1].

Всі вузли поділяються на три класи.

1. Прості. Мітки яких належать множині - $\{max, \perp, \varepsilon\} \cup K$.

2. Функтори. Єдиний функтор, який використовується в даному домені – це *concat*, що описує конкатенацію рядків (мітка - *concat/k*).
3. OR.

На рисунку 1.4 зображено приклад графу. Коренем є OR вузол з двома дітьми: простий вузол з міткою "b" і функтор з двома дітьми. Друга дитина – це корінь графу, який з'єднаний зворотною дугою з даним. Така структура задає множину рядків: $\{b, ab, aab, aab, \dots\} = a * b$.

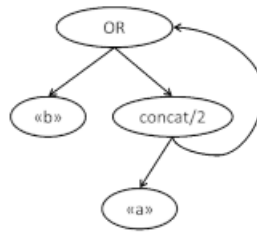


Рисунок 1.4 – Граф для представлення рядків

Такі графи чудово підходять для опису рядків, але декілька різних графів можуть представляти одну і ту саму множину рядків. Для зменшення кількості можливих графів застосовуються певні правила їх нормалізації.

$\overline{normStringGraph}$ - це функція, яка інкапсулює правила нормалізації.

Супремум представлений графом, який складається з *max*-вузла

Функція *join* над двома графами $\bar{T}_1$ і $\bar{T}_2$ еквівалентна створенню нового графу, де корінь – це OR-вузол з дітьми $\bar{T}_1$ і $\bar{T}_2$. Далі над результатом має бути викликана функція нормалізації. Кінцева формула описана наступним чином:

$$\sqcup_{\overline{SG}}(\bar{T}_1, \bar{T}_2) = \overline{normStringGraph}(OR(\bar{T}_1, \bar{T}_2))$$

Функції абстракції:

$$\alpha_{\overline{SG}}(S_1) = \sqcup_{\overline{SG}, s \in S_1} \alpha'_{\overline{SG}}(s) = \overline{normStringGraph}(OR\{\alpha'_{\overline{SG}}(s) : s \in S_1\})$$

$$\alpha'_{\overline{SG}}(s) = concat/k \{str[i] : i \in [0, k - 1]\},$$

де $k = len(s)$.

В таблиці 1.7 описані абстрактні семантики $\overline{\mathbb{S}}_{SG}$. Семантика рядка описана *concat* вузлом, який містить послідовність символів рядку. При конкатенації двох рядків повертається нормалізована версія *concat* вузла. Як зазвичай, семантика *readLine* повертає найбільше значення. Семантика *substring_b^e* повертає точне значення тільки у випадку, коли корінь – це *concat* вузол, який має як мінімум e символів де:

$$\overline{res} = concat / (e - b) \{ (\overline{root}(\bar{t}) / i) : i \in [b, e - 1] \}$$

Таблиця 1.7 – Абстрактні семантики $\overline{SG}$

$\overline{\mathbb{S}}_{SG}[\![new\ String(str)]\!](\emptyset)$	$concat / k \{ str[i] : i \in [0, k - 1] \}$
$\overline{\mathbb{S}}_{SG}[\![concat]\!](\bar{t}_1, \bar{t}_2)$	$\overline{normStringGraph}(concat / 2 \{ \bar{t}_1, \bar{t}_2 \})$
$\overline{\mathbb{S}}_{SG}[\![readLine]\!](\emptyset)$	$\top_{\overline{SG}}$
$\overline{\mathbb{S}}_{SG}[\![substring_b^e]\!](\bar{t})$	$\begin{cases} \overline{res} & \text{if } \overline{root}[\bar{t}] = concat / k \wedge \forall i \in [0, e - 1]: \\ & \overline{lb}(\overline{root}(\bar{t}) / i) \geq e \in K \\ \top_{\overline{SG}} & \text{otherwise} \end{cases}$

В таблиці 1.8 описаний результат роботи аналізу.

Таблиця 1.8 – Результат аналізу для тестової програми

№ рядку	Змінна	$\overline{SG}$
1	query	$concat[s_1]$
3	l	max
3	query	$concat[s_1 + s_2; max; s_3]$
4	query	$\overline{SG}_1 = OR[concat[s_1]; concat[s_1 + s_2; max; s_3]]$
5	per	$concat[s_4]$
6	query	$concat[\overline{SG}_1; concat[s_5 + s_4 + s_6]]$

Висновки

Описані домени відстежують різні типи інформації. Легко помітити, що домен $\overline{SG}$ є більш точним, ніж $\overline{CI}, \overline{PR}, \overline{SU}$ домени. За допомогою $\overline{CI}$ домену можна зробити висновок тільки про наявність певних символів, повністю ігноруючи їх порядок. $\overline{PR}$ і $\overline{SU}$ домени враховують і порядок, обмежуючись тільки початковою/кінцевою частиною рядка, так само збираючи тільки часткову інформація про наявність символів. $\overline{SG}$ домен враховує як наявність, так і порядок у рядку. Отже, $\overline{CI}, \overline{PR}, \overline{SU}$ домени мають не дуже велику точність, але зберігають лінійну складність, в той час як $\overline{SG}$ домен більш вимогливий (на практиці складність поліноміальна), але в той же час набагато точніший.

На основі вищенаведеного аналізу можна зробити висновок про доцільність створення методу, який буде таким же точним, що й метод на основі $\overline{SG}$ домену, але матиме меншу складність, що дозволить використовувати його у більшій кількості ситуацій, в яких потрібно визначати повну множину рядків, що використовуються в програмі, але наявні методи не можуть бути ефективно використані через їх незадовільну швидкодію.

2. ОСНОВИ СТАТИЧНОГО АНАЛІЗУ ПРОГРАМНОГО КОДУ

На рисунку 2.1 зображено типову схему статичного аналізу коду.

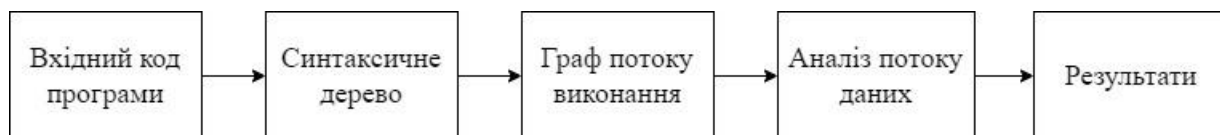


Рисунок 2.1 – Схема статичного аналізу коду

В цьому розділі розберемо більш детально кожний етап.

2.1 Синтаксичне дерево

Основою статичного аналізу є внутрішнє представлення вхідного програмного коду (зазвичай синтаксичне дерево або абстрактне синтаксичне дерево), яке є результатом синтаксичного аналізу, якому передуює лексичний аналіз коду, що перетворює текстове представлення у набір tokenів. Варто зазначити, що також існують підходи, які виконують лексичний і синтаксичний аналіз коду одночасно. Інструменти, які перетворюють вхідний код у синтаксичне дерево називають парсерами [2].

На рис. 2.2 зображено схему роботи парсеру.

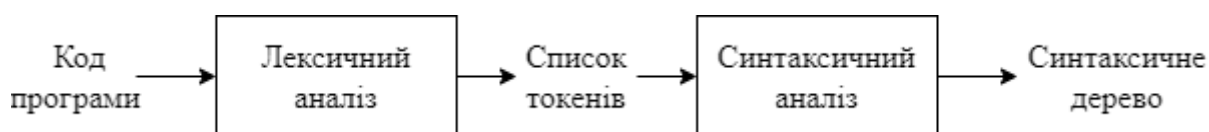


Рисунок 2.2 – Схема парсеру

Синтаксичне дерево – це структура даних, яка використовується для представлення програмного коду відповідно до контекстно-вільної граматики.

Абстрактне синтаксичне дерево – це представлення абстрактної структури тексту (зазвичай програмного коду) описаного формальною мовою. Кожен вузол дерева описує конструкцію, яка зустрічається у вхідному тексті. Абстрактна

приставка означає, що така структура містить тільки логічно важливі елементи і може опускати певні незначущі деталі. Наприклад, групувальні дужки неявно задані в структурі дерева, тому немає необхідності у їх наявності [2].

Основні переваги такого представлення програмного коду:

- 1) можна додавати додаткову інформацію, таку як властивості та анотації до вузлів, в той час коли над вхідним програмним кодом таке анотування являється неможливим, оскільки це б передбачало його зміну;
- 2) на відміну від текстового представлення, АСД не включає в себе знаки пунктуації і роздільники (дужки, крапки з комою і т.д.);
- 3) АСД зазвичай містить додаткову інформацію про програму, яка отримується через послідовні етапи аналізу (наприклад, позицію кожного елементу у вихідному коді, дозволяючи компілятору використовувати їх для формування змістовних повідомлень про помилки).

Загалом парсери можуть бути написані або вручну програмістом або отримані за допомогою автоматичних або напівавтоматичних генераторів. На вхід таких генераторів подається файл, який описує граматику заданої мови програмування. Вона, зазвичай, описана у БНФ або її розширеній версії РБНФ.

На рис 2.3 описане типове правило граматики у такій формі [2].

```
<symbol> ::= __expression__
```

Рисунок 2.3 – Правило граматики у БНФ

Де:

- 1) <symbol> - нетермінальна змінна;
- 2) __expression__ - послідовність, яка складається з термінальних і нетермінальних елементів;

- 3) ::= - означає, що символ з лівої частини має бути замінений виразом з правої частини.
- 4) можливі альтернативи послідовностей в правій частині відокремлюються вертикальною рисою - |.

На рис. 2.4 зображено типове правило граматики у РБНФ:

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
digit                = "0" | digit excluding zero ;
```

Рисунок 2.4 – Правило граматики у РБНФ

Будь-яка граMATика визначена у РБНФ може бути описана у БНФ, хоча представлення у останній формі, як правило, довші. Наприклад, повторення не можуть бути прямо виражені у БНФ і потребують створення проміжного правила. Також БНФ використовує символи - <, >, |, ::=, але не обрамлює термінальні рядки лапками. Такий підхід запобігає використанню даних символів у мовах, а також вимагає спеціального символу для порожнього рядка. У РБНФ термінали обов'язково обрамлюються лапками – “...” або ‘...’. Також, кутові дужки (<...>) навколо нетермінальних символів можуть не використовуватися. Правило БНФ можна представити лише в одному рядку, в той час як РБНФ має спеціальний символ закінчення - ;. Також ця форма містить механізми визначення кількості повторів, виключення, альтернатив, коментарів і т.п.

На виході генератора є код парсеру для аналізу текстів, які написані заданою мовою програмування.

На сьогодні створено багато потужних інструментів для генерації парсерів, такі як:

- 1) ANTLR.
- 2) Coco/R;
- 3) GNU Bison;
- 4) JavaCC.

Найпопулярнішим і найпотужнішим інструментом з вище зазначених є ANTLR. Перший реліз відбувся у 1989 році. На вхід приймає граматику описану у РБНФ. Він може генерувати лексери, парсери, їх комбінацію. Парсери можуть автоматично генерувати синтаксичні дерева. Понад 200 граматик, які реалізовані за допомогою ANLTR4 знаходяться у відкритому доступі на Github. 4 версія генератора дозволяє генерувати код парсера на таких мовах, як: PHP, C#, C++, Python, Java, Go, JavaScript і т.д [3].

За замовчуванням ANTLR аналізує граматику і генерує розпізнавач для мови, яка описана вхідною граматикою (тобто програму, яка читає вхідний текст і генерує помилку, якщо він не задовольняє описаний синтаксис). Якщо помилок немає, то згенерований інструмент завершує свою роботу без всіляких дій. Для виконання певної корисної роботи над мовою, дії можуть бути додані в елементи граматики. Вони описуються на мові згенерованого інструмента. Коли розпізнавач генерується, задані дії вбудовуються у його вихідний код у відповідних місцях. Також ANTLR використовується для створення аналізаторів дерев. Це засоби для обробки абстрактних синтаксичних дерев, які можуть бути автоматично згенеровані. Такі інструменти є унікальними для ANTLR, і допомагають у обробці дерев розбору [3].

2.2 Граф потоку виконання

Синтаксичні дерева забезпечують «природне» представлення граматичної структури коду, яке отримується за допомогою парсингу вхідного коду, але їх структура зовсім не підходить для представлення інших властивостей програм. Для моделювання даних аспектів поведінки програм, інструменти синтаксичного аналізу коду, а також компілятори, використовують більш загальні графи.

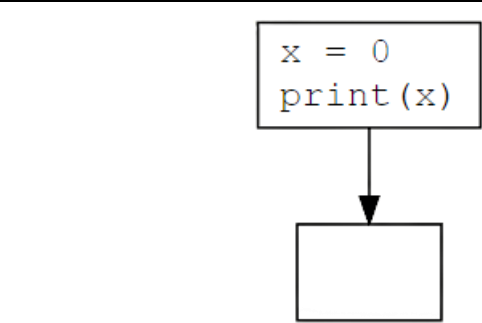
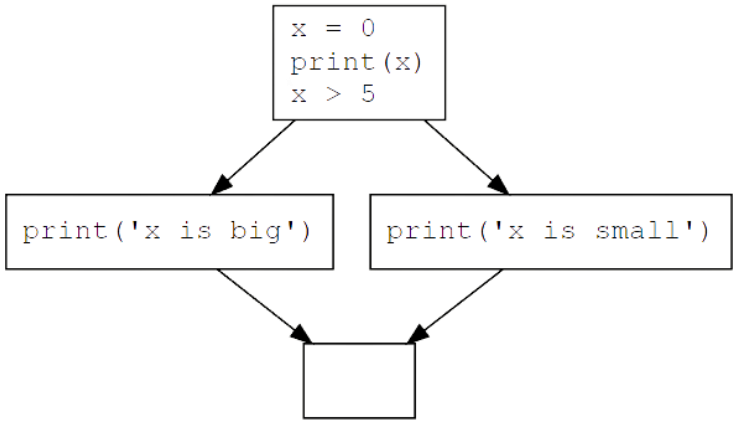
Одним з таких представлень є граф потоку виконання (CFG). Даний тип графів створений для представлення частин програми, а також шляхів їх виконання.

Одним з елементів CFG є, так званий, базовий блок. Він складається з послідовних операторів і виразів, які гарантовано будуть виконані разом, один за

одним. Тобто, він може закінчуватися операцією розгалуження, переходу або предикату.

CFG – це орієнтований граф, $G = (N, E)$, де $n \in N$ – базовий блок, (n_i, n_j) – ребро, яке описує можливий перехід керування від блоку n_i до блоку n_j [4]. В таблиці 2.1 зображено код програми, а також відповідний йому граф.

Таблиця 2.1 – Приклади графів потоку виконання

Код програми	Граф потоку виконання
<pre>x = 0 print(x)</pre>	
<pre>x = 0 print(x) if x > 5: print('x is big') else: print('x is small')</pre>	

Варто зазначити, що на другому графу умова $x > 5$ не знаходиться у власному блоці, натомість, вона об'єднана з попереднім оператором, оскільки завжди виконується після його виконання. Це наглядно ілюструє те, що я мав на увазі при описі терміну базового блоку.

Розглянемо також як на даних типах графах описуються цикли. На рис. 2.5 і 2.6 зображено код програми з циклом, а також його граф потоку виконання відповідно.

```

x = 0
while x < 10:
    if x > 5:
        print(x)
    else:
        print('x less than 6')
    x += 1

```

Рисунок 2.5 – Програма з циклом

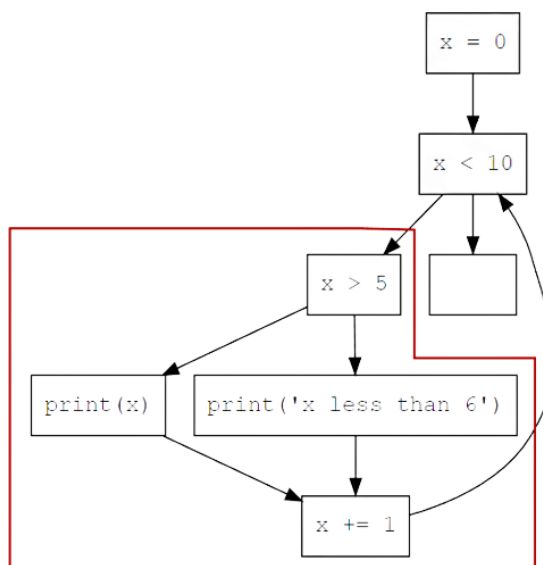


Рисунок 2.6 – Опис циклів у графі потоку виконання

Даний граф є дещо складнішим за попередні. Умова циклу винесена у власний базовий блок і має два різних блоки, які можуть слідувати після неї: тіло циклу, а також оператори після нього (в моєму прикладі немає таких, тому наступний блок після циклу являється кінцевим блоком програми). Тіло циклу представлено підграфом, який знаходиться у червоному прямокутнику. Варто зазначити, що після виконання кінцевого оператора тіла - $x += 1$, умова циклу виконується знову (даний перехід описаний ребром, яке з'єднує базовий блок, що описує останній оператор циклу, і умову циклу).

Наприкінці, розглянемо представлення операторів повернення у CFG. Як відомо, після виконання даних операторів функція негайно завершується. Тобто, коли базовий блок містить оператор повернення, з нього має виходити ребро, яке з'єднується зі спеціальною вершиною, що описує кінець програми. Оператор

повернення, а також його представлення у графах потоку виконання описані на рис. 2.7 і 2.8 відповідно.

```
def f(x: int) -> None:
    if x > 10:
        return
    else:
        y = x + 1
        print(x * y)
```

Рисунок 2.7 – Програма з оператором повернення

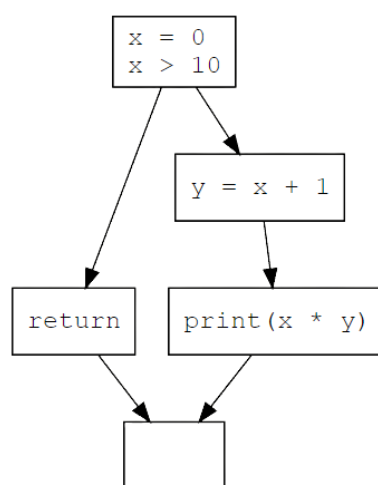


Рисунок 2.8 – Опис операторів повернення у графі потоку виконання

Варто зазначити про те, що існують і інші графові представлення [4], суть яких зазначена нижче.

1. Граф залежності даних (DDG) моделює потік значень від визначень до їх використання. Тобто, вузли представляють обчислення, а ребра потік значень від визначення до використання. Їх використовують в різних оптимізаціях, наприклад, трансформації, які змінюють порядок операторів в циклах для виявлення паралелізму, а також покращення використання пам'яті.
2. Граф викликів (Call graph) представляє зв'язки викликів між процедурами в програмі. Вузли описують процедури в програмі, а ребра – їх виклики.

2.3 Аналіз потоку даних

DFA передбачає «потік» інформації через елементи вхідної програми. Такий потік може бути вирахований, як в прямому, так і зворотному порядку, і часто використовує множини для зберігання значень, наприклад: арифметичні вирази, які були обраховані у попередніх операціях; змінні, що використовуються в наступних обчисленнях; змінні, які точно мають постійні значення (константи). Так як аналіз приблизно оцінює можливий потік значень, його результати не є точними [5].

Даний аналіз будується на основі CFG. Для полегшення аналізу базовий блок графу розбивається на декілька послідовних базових блоків. Кожний блок анотується певними властивостями, які зазвичай являються наборами, що мають семантичне значення. Набір властивостей в певному блоці може вказувати на змінні або вирази, значення яких передавалися вздовж шляху виконання до даного блоку [5].

Наприклад, аналіз доступних виразів вираховує, які арифметичні вирази вже були вираховані та готові до подальшого їх використання у певних місцях програми.

Інший аналіз передбачає, які змінні мають константні значення у певних місцях програми. Даний аналіз називається «згортка констант». Значення, якими анотуються базові блоки представляють собою множину виду:

$$\{(x_0, c_0), \dots, (x_n, c_n)\},$$

де x_i – унікальне ім'я змінної, а c_i – фіксоване (напр. 1 або 2) або невизначене значення (T), яке означає, що значення x_i визначене, але не може вважатися константним.

Як було сказано вище, існують два основних підходи до поширення значень через граф потоку виконання:

- 1) прямий (англ. – forward);
- 2) зворотній (англ. – backward).

У прямому аналізі інформація про потік даних поширюється від точки входу в програму до її виходу. Іншими словами, цей підхід генерує інформацію про потік даних у кожній точці програми, яка відображає вплив операторів, що були виконані

до цієї точки. Наприклад, аналіз в якому інформація про потік даних у точці програми представляє визначення змінних, які досягають цієї точки (reaching definitions).

У зворотному аналізі інформація поширюється від точки виходу програми до її входу. Тобто, інформація про потік даних відображає вплив операторів, які знаходяться після даної точки програми. Наприклад, аналіз, який генерує інформацію в точці програми про набір змінних, які використовуються після неї (live variables).

З міркувань про поведінку програми, існує два варіанти аналізу потоку виконання.

1. Must - це консервативний аналіз, що спрямований на виявлення поведінки програми, яка гарантовано відбудеться. Даний підхід обчислює набір властивостей, які виконуються в кожній точці програми незалежно від вхідних даних. Цей аналіз має бути більш точним, але, як правило, може пропускати деяку поведінку програми, виконання якої може не гарантувати.
2. May – аналіз, який спрямований на виявлення поведінки, що виникає за певних умов. Він обчислює набір властивостей, які можуть виконуватися в кожній точці програми, залежно від вхідних параметрів. Як правило, цей аналіз менш точний, але здатний охопити ширший діапазон можливих дій.

Зазвичай для анотації блоків графу використовується ітеративний аналіз. Починаючи з початкової конфігурації, яка представляє собою анотовані базові блоки пустими даними, алгоритм поширює значення вздовж всіх вузлів графу потоку виконання. Функція, яка описує зміну вхідних властивостей у певному вузлі називається трансферною функцією, і позначається як - $f_{l \rightarrow l'}$.

Існують два основних підходи реалізації ітеративного алгоритму.

MFP [5]. Інформація в блоці l' визначається наступним чином:

$$propertyAtNode(l') = \cup_{l \in pred(l')} f_{l \rightarrow l'} (propertyAtNode(l)),$$

де $pred(l')$ - це множина всіх вузлів графу потоку виконання з яких можна потрапити у вузол l' ,

$f_{l \rightarrow u}$ - трансферна функція.

На початку, вузли графу мають певне початкове значення, наприклад - пусту множину. Аналіз продовжується, поки значення у вузлах не стабілізуються (тобто перестануть змінюватися після чергової ітерації) [5].

На рис. 2.9 зображено псевдокод MFP алгоритму для прямого поширення потоку даних (forward analysis).

```

Вхідні дані:
- Граф потоку виконання (CFG)
- Трансферна функція
- Початкові дані для кожного вузла
Вихідні дані:
- Інформація про потік даних для кожного вузла

// Крок 1: Ініціалізація
for each basic block B in CFG {
    in[B] = початкове значення
    out[B] = початкове значення
}
worklist = множина всіх вузлів графу

// Крок 2: Поширення значень
while worklist is not empty {
    B = обраний вузол з worklist
    for each predecessor of B {
        in[B] = in[B] ∪ out[P] // об'єднання властивостей з батьківських вузлів
    }
    out[B] = transfer(B, in[B]) // обробка вхідних властивостей трансферною функцією
    if out[B] has changed {
        for each successor of B {
            add S to worklist
        }
    }
    remove B from worklist
}

```

Рисунок 2.9 – Псевдокод MFP алгоритму

При обчисленні зворотного поширення потоку даних (backward analysis) відбувається взаємозаміна сутностей predecessor (предок) на successor (нащадок).

МОР [5]. Даний тип аналізу вираховує значення у вузлах на кожному можливому шляху виконання, які потім об'єднуються наприкінці аналізу. Оскільки програма може мати нескінчену кількість шляхів, то може відбуватися певне їх стиснення (згортка циклів і/або викликів процедур).

Висновки

Підсумовуючи, типовий статичний аналіз коду складається з 3-ьох етапів:

- 1) перетворення вхідного коду в синтаксичне дерево згідно граматики мови, якою була написана ця програма (парсинг програмного коду);
- 2) перетворення синтаксичного дерева у структуру, яка описує поведінку програми;
- 3) обчислення інформації про потік даних на основі структури, яка була обрана на попередньому етапі.

Перед кожним етапом виконується підготовчі дії для вибору інструментів і/або підходів для ефективного виконання.

В даній магістерській дисертації для виконання задачі розбору вхідного коду було обрано інструмент ANTLR, який створений для генерації парсерів, а також має широкий спектр інструментів для обробки синтаксичних дерев. В якості поведінкової структури було обрано граф потоку виконання, адже саме він, на мою думку, являється найбільш підходящим для задачі «згортки констант» через його зручність і гнучкість поширення значень. В якості алгоритму аналізу потоку даних був обраний ітеративний підхід MFP.

3. МОДИФІКОВАНИЙ МЕТОД СТАТИЧНОГО АНАЛІЗУ КОДУ ДЛЯ РІШЕННЯ ЗАДАЧІ ЗГОРТКИ РЯДКОВИХ КОНСТАНТ

3.1 Основні положення стандартних методів

Розглянемо простий, але в той же час наочний, приклад типової програми, псевдокод якої показаний на рис.3.1. В даній програмі відбувається формування SQL запиту шляхом об'єднання значень змінних `query`, `where` і `order`. Зазначимо, що змінні `where` і `order` відповідають SQL операторам фільтрації і вибірки відповідно, і формуються по-різному, в залежності від параметру `flag`. Цей приклад показує одну із причин збільшення розміру множини рядків, які можуть бути присвоєні певній змінній базового блоку у вершині графу, при використанні трансферної функції, яка базується на декартовому добутку, навіть для невеликої програми [6].

```
query = "SELECT id1, id2 FROM tbl ";

if (flag == 1) {
    where = "WHERE id1 IS NOT NULL ";
    order = "ORDER BY id1";
} else if (flag == 2) {
    where = "WHERE id2 IS NOT NULL ";
    order = "ORDER BY id2";
} else {
    where = "";
    order = "";
}

query = query + where + order;
```

Рисунок 3.1 – Приклад програми з декількома розгалуженнями

Варто зазначити, що в методі, який базується на $\overline{SG}$ домені, що був описаний у першому розділі даної дисертації, при перетворенні представлення значень змінної у вигляді графу у кінцеву множину рядків, операція декартового добутку відповідає функтор вузлам (тобто, операції об'єднання рядків), адже саме при цій операції відбувається пошук всіх комбінацій множин рядків, що отримуються при

перетворенні нащадків даного вузла. Наприклад, графу який зображений на рис.3.2 відповідають комбінації, що зображені у табл. 3.1.

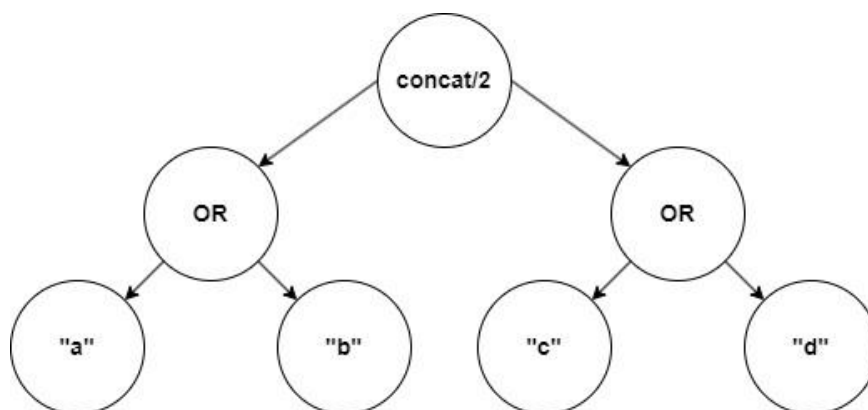


Рисунок 3.2 – Представлення рядкових змінних у вигляді графу

Таблиця 3.1 – Можливі комбінації рядків

Значення з першого підграфу	Значення з другого підграфу
"a"	"c"
"a"	"d"
"b"	"c"
"b"	"d"

Як правило, для вирішення задачі згортки констант (рядкових і числових), з метою зберігання різних властивостей базових блоків у вузлах графу, використовується хеш-таблиця, де ключем хешування є змінна, а значенням – множина можливих констант, які ця змінна може набувати до моменту виконання того базового блоку, що представляє даний вузол [6].

При використанні такої структури даних для зберігання значень, join-функція базується на функції злиття декількох хеш-таблиць в одну. Опишемо кроки найпростішого алгоритму.

Крок 1. З двох вхідних таблиць, вихідною вважається та, яка має найбільше інформації (тобто, містить більше ключів).

Крок 2. Виконується обхід по ключам з відповідними значеннями в меншій таблиці. Якщо у вихідній таблиці присутнє значення для поточного ключа, то його значення об'єднується з існуючим за допомогою функції об'єднання. У іншому випадку дане відношення переноситься у вихідну таблицю.

Псевдокод функції злиття зображено на рис. 3.3.

```

Вхідні дані:
- Перша хеш-таблиця (t1)
- Друга хеш-таблиця (t2)
- Функція об'єднання значень (mergeValues)
Вихідні дані:
- Нова хеш-таблиця (res)
// Крок 1: Пошук більшої таблиці
if (t1.count > t2.count) {
    res = t1; other = t2
}
else {
    res = t2; other = t1
}
// Крок 2: Злиття таблиць
for key, value of other {
    // якщо існує значення для ключа, то вони об'єднуються
    if (res.has(key)) {
        res[key] = mergeValues(res[key], value)
    }
    // у іншому випадку додається нове відношення
    else {
        res[key] = value
    }
}

```

Рисунок 3.3 – Псевдокод функції злиття хеш-таблиць

Суть функції об'єднання значень в join-функції стандартного методу полягає у використанні операції об'єднання над множинами.

Опишемо кроки алгоритму трансферної функції, що використовується в методах, які базуються на декартовому добутку [6].

Крок 1. Аналізується інструкція, яка описана даним вузлом графу. У випадку, якщо вона представляє присвоювання в рядкову змінну, то додатково аналізується права частина (тобто, її нове значення). Вона складається зі списку сутностей, що послідовно об'єднуються в один

рядок. Сутність, в даному випадку, може бути або змінною або рядковим літералом.

Крок 2. У випадку, якщо сутність – це змінна, то визначається множина можливих значень, які вона може набувати. Дана множина міститься в таблиці стану змінних, яка була отримана при аналізі попередніх вузлів. Якщо сутність представлена рядковим літералом, то множина складається тільки з цього літералу.

Крок 3. Обчислюється декартів добуток множин, які були отримані на попередньому кроці.

Крок 4. Складові елементів декартового добутку об'єднуються між собою.

Крок 5. Новий стан змінної, який представлений множиною, що була отримана на попередніх кроках, передається у вихідні властивості вузла (тобто на вхід наступним вузлам).

Псевдокод трансферної функції, кроки якої описані вище, зображено на рис.3.4.

```

Вхідні дані:
- Вузол CFG (node)
- Таблиця значень змінних (in)
Вихідні дані:
- Оновлена таблиця значень змінних (out)
// Обробка операції присвоювання
if (node.type == 'assign') {
    to = ім'я змінної
    operands = список операндів
    values = [] // список множин відповідних операндів
    for operand of operands {
        if (isVariable(operand)){
            // якщо операнд описує змінну, то отримується множина її значень
            add in[operand] to values
        } else {
            // у випадку літералу, множина складається з цього літералу
            add {operand} to values
        }
    }
    result = {} // нова множина значень змінної
    for elements of cartesian(values) {
        // отримання нового рядку, шляхом конкатенації елементу декартового добутку
        add String.join(elements) to result
    }
    // оновлення значень змінної
    in[to] = result
}

```

Рисунок 3.4 – Псевдокод трансферної функції стандартного методу

Також важливою частиною аналізу є порівняння обчислених властивостей у вузлах графу для знаходження точки зупинки.

Таке порівняння реалізується, так званою, equal-функцією. Дана функція базується на функції порівняння хеш-таблиць, найпростіший алгоритм якої складається із зазначених нижче двох кроків.

Крок 1. Обчислюється список всіх можливих ключів.

Крок 2. Відбувається обхід списку, який був обчислений на попередньому кроці. У випадку, якщо значення відсутнє хоча б в одній таблиці або присутні значення в обох таблицях відрізняються відповідно до функції порівняння, таблиці вважають різними.

Псевдокод функції порівняння хеш-таблиць зображено на рис. 3.5.

```
Вхідні дані:
- Перша хеш-таблиця (t1)
- Друга хеш-таблиця (t2)
- Функція порівняння значень (compareValues)
Вихідні дані:
- Булеве значення
// Крок 1: Пошук ключів, які присутні в обох таблицях
keys = distinct(concat(t1.keys, t2.keys))
// Крок 2: Порівняння значень
for key of keys {
    // якщо хоча б одна таблиця не має значень для ключа
    // або ці значення не рівні, то таблиці вважаються різними
    if (!(t1.has(key) && t2.has(key) && compareValues(t1[key], t2[key]))) {
        return false
    }
}
// У випадку рівності всіх ключів, таблиці вважаються також рівними
return true
```

Рисунок 3.5 – Псевдокод функції порівняння хеш-таблиць

Суть функції порівняння значень в equal-функції стандартного методу полягає у використанні операції порівняння множин.

На рис.3.6 показано результат DFA, обчислений стандартним методом. Наведемо опис елементів позначень на цьому графі [6].

1. Еліпси, що нарисовані суцільними лініями – вузли графу.
2. Ребра, що нарисовані суцільними лініями - передача управління.

3. Ребра, що нарисовані пунктиром, з'єднують вузли графу і блоки з пунктирними рамками. Ці блоки описують стан змінних, який буде отриманий після аналізу інструкції в даному блоці за допомогою трансферної функції.
4. Також граф містить спеціальний вузол JOIN, який показаний пунктирною рамкою. Він був доданий для позначення місця виконання join-функції.

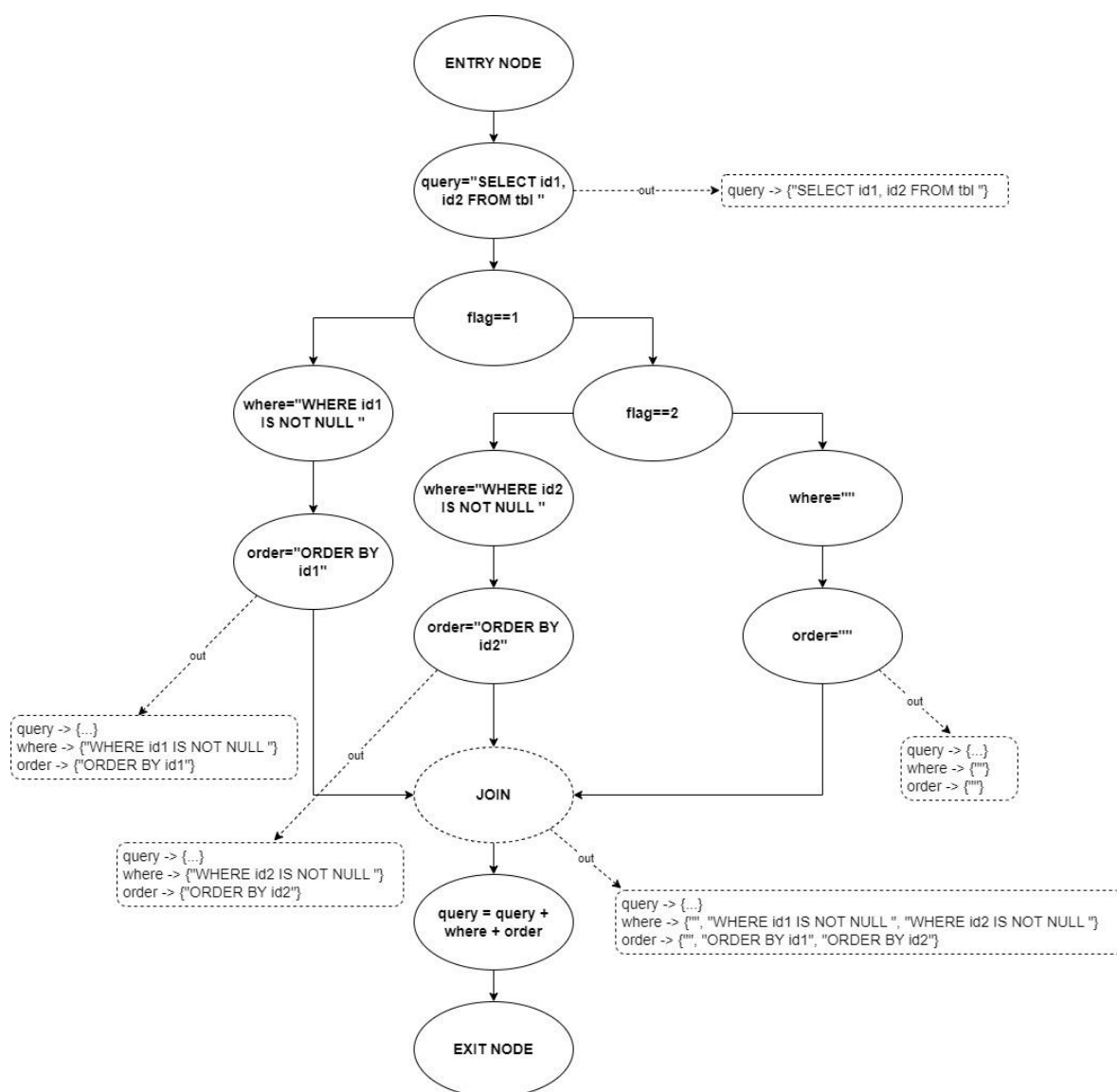


Рисунок 3.6 - Результат DFA, обчислений стандартним методом

Розглянемо роботу трансферної функції для вузла графу, який передусе закінченню програми. Даний вузол описує операцію об'єднання трьох значень

змінних: query, where і order. Вхідна таблиця стану змінних містить наступні значення (таблиця 3.2).

Таблиця 3.2. – Вхідна таблиця стану змінних

query	where	order
“SELECT id1, id2 FROM tbl ”	“”	“”
	“WHERE id1 IS NOT NULL ”	“ORDER BY id1”
	“WHERE id2 IS NOT NULL ”	“ORDER BY id2”

Після обчислення декартового добутку множин можливих значень змінних, отримуємо наступні комбінації рядків, які зображено в таблиці 3.3.

Таблиця 3.3. – Комбінація значень змінних

query	where	order
“SELECT id1, id2 FROM tbl ”	“”	“”
“SELECT id1, id2 FROM tbl ”	“”	“ORDER BY id1”
“SELECT id1, id2 FROM tbl ”	“”	“ORDER BY id2”
“SELECT id1, id2 FROM tbl ”	“WHERE id1 IS NOT NULL ”	“”
“SELECT id1, id2 FROM tbl ”	“WHERE id1 IS NOT NULL ”	“ORDER BY id1”
“SELECT id1, id2 FROM tbl ”	“WHERE id1 IS NOT NULL ”	“ORDER BY id2”
“SELECT id1, id2 FROM tbl ”	“WHERE id2 IS NOT NULL ”	“”
“SELECT id1, id2 FROM tbl ”	“WHERE id2 IS NOT NULL ”	“ORDER BY id1”
“SELECT id1, id2 FROM tbl ”	“WHERE id2 IS NOT NULL ”	“ORDER BY id2”

Шляхом конкатенації всіх комбінацій рядків, які зображено вище, отримуємо наступну множину рядків для змінної query:

- 1) “SELECT id1, id2 FROM tbl ”;
- 2) “SELECT id1, id2 FROM tbl ORDER BY id1”;
- 3) “SELECT id1, id2 FROM tbl ORDER BY id2”;
- 4) “SELECT id1, id2 FROM tbl WHERE id1 IS NOT NULL ”;

- 5) “SELECT id1, id2 FROM tbl WHERE id1 IS NOT NULL ORDER BY id1”;
- 6) “SELECT id1, id2 FROM tbl WHERE id1 IS NOT NULL ORDER BY id2”;
- 7) “SELECT id1, id2 FROM tbl WHERE id2 IS NOT NULL ”;
- 8) “SELECT id1, id2 FROM tbl WHERE id2 IS NOT NULL ORDER BY id1”;
- 9) “SELECT id1, id2 FROM tbl WHERE id2 IS NOT NULL ORDER BY id2”;

Аналізуючи вхідний код програми, можна визначити, що запити під номерами 2, 3, 4, 6, 7, 8 ніколи не будуть створені під час реального виконання інструкцій цієї програми. Інакше кажучи, наявність надлишкових рядків призводить до двох проблем:

- 1) зменшення точності;
- 2) комбінаторного вибуху кількості проміжних станів програми, що призводить до значного погіршення швидкодії алгоритму.

3.2 Опис модифікованого методу

На основі аналізу вищеописаних проблем було вирішено розробити модифікований метод статичного аналізу коду для рішення задачі згортки рядкових констант на основі MFP алгоритму, шляхом анотування рядків, які містяться в множині можливих значень змінної, списками компонентів. В даному випадку, компонент — це впорядкована послідовність вузлів графу, що відповідають інструкціям створення рядків, з яких складається значення, що анотується цим компонентом, а список компонентів відповідає операції альтернативи, тобто, якщо список складається з декількох компонентів, то це означає, що такий рядок може бути отриманий декількома шляхами [6].

Для зберігання значень змінних використовується хеш-таблиця, де ключем хешування є змінна, а значенням є ще одна хеш-таблиця. В цій вкладеній хеш-таблиці ключем хешування є рядок, а значенням — множина компонентів. Використання вкладеної хеш-таблиці в якості значень замість структури даних типу «множина», що використовується у стандартному алгоритмі, є відмінною рисою запропонованого методу, яка дозволяє досягти поставленої мети дослідження [6].

Так як структура даних для зберігання значень змінних відрізняється від стандартного алгоритму, то і equal-функція модифікованого методу має інший вигляд. Відміністю є функція порівняння значень, яка використовується при порівнянні хеш-таблиць. В модифікованому методі її суть полягає у порівнянні вкладених хеш-таблиць за допомогою раніше описаної функції (рис.3.5), де в якості інструменту для порівняння значень виступає порівняння множин компонентів.

Досить очевидно, що зміна структури даних також вимагає модифікацію і join-функції. Відміністю є функція об'єднання значень, яка використовується при злитті хеш-таблиць в стандартному методі. В модифікованому методі її суть полягає у злитті вкладених хеш-таблиць за допомогою раніше описаної функції (рис. 3.3), де суть функції об'єднання значень полягає у поєднанні множин компонентів.

На рис. 3.7 зображено псевдокод трансферної функції модифікованого методу.

```

Вхідні дані:
- Вузол CFG (node)
- Таблиця значень змінних (in)
Вихідні дані:
- Оновлена таблиця значень змінних (out)
// Обробка операції присвоювання
if (node.type == 'assign') {
    to = ім'я змінної
    operands = список операндів
    // список, який складається зі списків пар відповідних операндів
    values = []
    for operand of operands {
        if (isVariable(operand)){
            // у випадку змінної додається список пар
            add in[operand].keyValuePairs() to values
        } else {
            // у випадку літералу, список пар створюється вручну
            add [[operand, new Components(node)]] to values
        }
    }
    result = [] // новий список пар значень змінної
    for elements of cartesian(values) {
        compatiblePair = можливе нове значення, що отримується після аналізу елементів
        if (compatiblePair) add compatiblePair to result
    }
    // оновлення значень змінної перетвореним у таблицю списком
    in[to] = Map.fromKeyValuePairs(result)
}

```

Рисунок 3.7 – Трансферна функція модифікованого методу

Опишемо кроки знаходження нового стану змінної [6].

Крок 1. У випадку, якщо сутність - це змінна, то отримується таблиця можливих значень з вхідної таблиці станів змінних, яка трансформується у список пар «рядок – множина компонентів». Якщо сутність представлена літералом, стає очевидним, що інструкція створення рядка описана поточним вузлом. Тому, список пар складається з одного елементу, який містить даний літерал і множина з одного компонента, що складається з поточного вузла.

Крок 2. Обчислюється декартів добуток списків пар, які були отримані на попередньому кроці.

Крок 3. Далі розглянемо аналіз, який виконується для кожного елементу добутку, отриманого на попередньому кроці (тобто, для кожної комбінації пар). На початку, обчислюється декартів добуток множин компонентів, що є другими елементами пар. Після цього, кожна комбінація компонентів перетворюється на єдину компоненту, шляхом поєднання її складових. Наступним кроком є видалення тих компонентів, існування яких неможливе. Іншими словами, існують складові (тобто, вузли) які є несумісними між собою. Псевдокод алгоритму цього пошуку показаний на рис.3.8. Вузол компонента вважається сумісним, якщо він пов'язаний з кожним вузлом даного компонента (тобто, існує шлях між ними у CFG). Алгоритм перевірки компонентів на сумісність показаний на рис.3.9. Така перевірка базується на використанні множини попередніх вузлів, яка, як правило, збирається за допомогою додаткового DFA і міститься у класі аналізу згортки констант в якості окремого поля. У випадку, якщо не знайдено жодної сумісної комбінації компонентів, ця послідовність пар ігнорується.

Крок 4. Новий стан змінної, який представлений списком пар «рядок – множина компонентів» перетвореним у хеш-таблицю, передається у вихідні властивості вузла (тобто на вхід наступним вузлам).

```

Вхідні дані:
- Елемент декартового добутку (pairs)
Вихідні дані:
- Нова пара або null (newPair)
// Крок 1: Отримати списки рядків і множин компонентів
newPair = null
listOfComponents = []
listOfStrs = []
for str, components of pairs {
    add components to listOfComponents
    add str to listOfStrs
}
// Крок 2: Знайти сумісні комбінації компонентів
newComponents = {}
for combination of cartesian(listOfComponents) {
    newComponent = Components.join(combination)
    // якщо всі вузли компоненти сумісні між собою, то
    // вона додається у результуючий список
    if (CanComponentExist(component)){
        add newComponent to newComponents
    }
}
// Крок 3: Об'єднати рядки, якщо існує хоч одна сумісна комбінація
if (newComponents.count > 0) {
    newPair = [String.join(listOfStrs), newComponents]
}

```

Рисунок 3.8 - Псевдокод алгоритм пошуку сумісних пар

```

Вхідні дані:
- Компонент (component)
- Таблиця попередних вузлів (previousNodes)
Вихідні дані:
- Булеве значення, яке показує сумісність

for i from 0 to component.count {
    for j from i + 1 to component.count {
        // якщо вузли не пов'язані у графі потоку виконання
        // його виникнення вважається неможливим
        if (!previousNodes.get(node1).contains(node2) &&
            !previousNodes.get(node2).contains(node1)) {
            return false
        }
    }
}

```

Рисунок 3.9 - Псевдокод алгоритму перевірки складових компонента на сумісність

Для порівняння запропонованого методу зі стандартним, покажемо відмінності результату DFA, обчисленого запропонованим методом (рис. 3.10) від стандартного методу (рис. 6) для однієї й тієї ж програми (рис. 3.1). Для наочності, вузли графу на рис. 3.10 пронумеровані. На цьому графі списки компонентів

зображені квадратними дужками, які містять компоненти, що позначаються або числом (наприклад [1]), де число – це номер вузла, в якому знаходиться інструкція створення рядку, або переліком номерів вузлів, поєднаних амперсандами (наприклад [2&5&6]), якщо рядок даного вузлу був створений шляхом конкатенації інших рядків [6].



Рисунок 3.10 - Результат DFA, обчислений запропонованим методом

Розглянемо роботу трансферної функції для вузла графу, який передуює закінченню програми. Даний вузол описує операцію об'єднання трьох значень змінних: query, where і order. Вхідна таблиця стану змінних містить наступні значення (таблиця 3.3).

Таблиця 3.3. – Вхідна таблиця стану змінних

query	where	order
“SELECT id1, id2 FROM tbl ” [1]	“” [9]	“” [10]
	“WHERE id1 IS NOT NULL ” [5]	“ORDER BY id1” [6]
	“WHERE id2 IS NOT NULL ” [7]	“ORDER BY id2” [8]

Комбінації значень, аналогічні таблиці 3.2. Після знаходження всіх можливих комбінацій рядків, обчислюється декартів добуток множин компонентів, результат якого зображений в таблиці 3.4.

Таблиця 3.4. – Комбінація множин компонентів

query	where	order
1	9	10
1	9	6
1	9	8
1	5	10
1	5	6
1	5	8
1	7	10
1	7	6
1	7	8

Аналізуючи граф потоку виконання (рис.3.10) видно, що наступні вузли не пов’язані з собою (тобто, не існує порядку в якому інструкції, які описані даними вузлами, виконуються під час реального виконання програми): 9 – 6; 9 – 8; 5 – 10; 5 – 8; 7 – 10; 7 – 6. Отже, рядки які були проанотовані відповідними компонентами

не можуть бути об'єднаними. Результат трансферної функції для змінної query зображено в таблиці 3.5.

Таблиця 3.4. – Анотована множина значень змінної query

Рядкове значення	Множина компонентів
“SELECT id1, id2 FROM tbl ”	[1&9&10]
“SELECT id1, id2 FROM tbl WHERE id1 IS NOT NULL ORDER BY id1”	[1&5&6]
“SELECT id1, id2 FROM tbl WHERE id2 IS NOT NULL ORDER BY id2”	[1&7&8]

Висновки

В даному розділі був розроблений модифікований метод для рішення задачі згортки рядкових констант, який полягає у анотуванні пропагованих значень додатковою інформацією та має меншу алгоритмічну складність і вищу точність, ніж існуючі методи. Запропонований метод відрізняється від існуючих використанням вкладеної хеш-таблиці в якості значень основної хеш-таблиці замість структури даних типу «множина» та модифікованої трансферної функції і дозволяє зменшити складність алгоритмів, що використовуються, завдяки ігноруванню тих конкатенацій рядків, які не будуть створені під час реального виконання програми. Ця дія дозволяє зменшити кількість проміжних станів програми у DFA, завдяки чому вдається уникнути комбінаторного вибуху алгоритмічної складності, характерної для існуючих методів статичного аналізу для цієї задачі. Крім того, даний модифікований метод також дозволяє збільшити точність звуження теоретичної множини рядків, що можуть бути створені згідно CFG заданої програми, до множини рядків, які фактично будуть створені та використані при виконанні цієї програми. Все це призводить до прискорення і покращення точності процесу згортки рядкових констант програми

4. ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНОГО МЕТОДУ

4.1 Загальний опис системи

В якості програмної реалізації методу було вирішено створити систему статичного аналізу програмного коду для рішення задачі згортки рядкових констант. Мова програмування розробленої системи – Clojure. Варто зазначити, що через те, що програми написані мовою Clojure компілюються в байткод JVM, вони з легкістю можуть взаємодіяти з іншими мовами, що також компілюються в цей байткод (напр. - Java). Структуру розробленої системи, зображено на рис. 4.1.

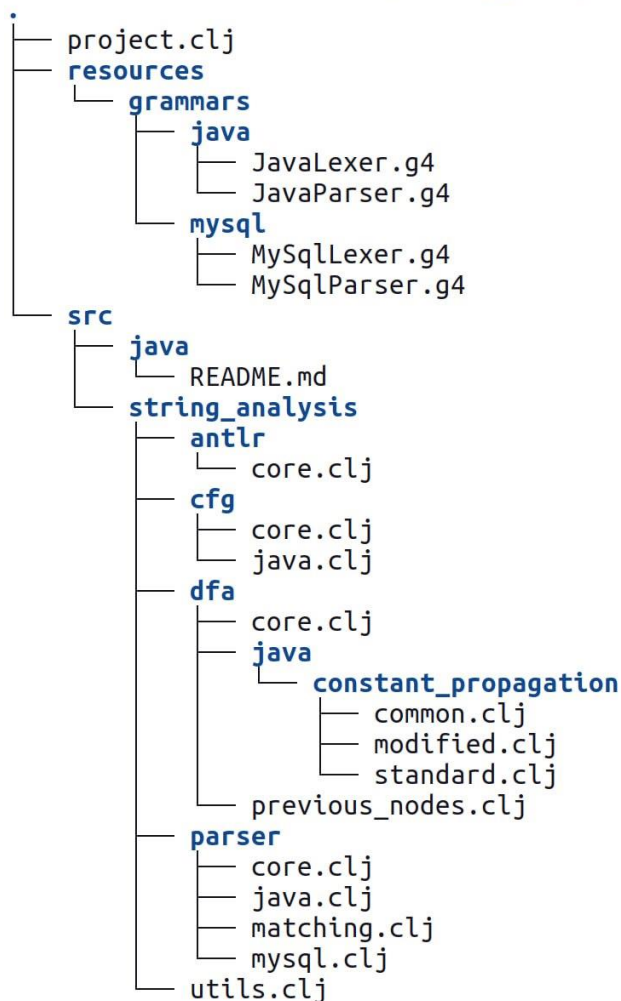


Рисунок 4.1 – Структура розробленої системи

project.clj – це головний файл системи, який містить різноманітні конфігурації, а також залежності від інших інструментів (напр. - antlr).

На рис. 4.2 зображено вміст даного файлу.

```
(defproject string_analysis "0.1.0-SNAPSHOT"
  :description "Program realization of modified method of static code
               analysis for solving problem of folding of string constants"
  :url "https://github.com/dkasianchuk/string_analysis"
  :license {:name "EPL-2.0 OR GPL-2.0-or-later WITH Classpath-exception-2.0"
           :url "https://www.eclipse.org/legal/epl-2.0/"}
  :dependencies [[org.clojure/clojure "1.10.0"]
                [org.antlr/antlr4 "4.9.2"]
                [org.antlr/antlr4-runtime "4.9.2"]
                [me.raynes/fs "1.4.6"]
                [org.clojure/math.combinatorics "0.2.0"]
                [org.clojure/data.csv "1.0.1"]]
  :java-source-paths ["src/java"]
  :target-path "target/%s"
  :profiles {:antlr
             {:main string-analysis.antlr.core
              :java-source-paths ^:replace []}
             :uberjar {:aot :all}}}
  :aliases {"antlr" ["with-profile" "antlr" "run"]})
```

Рисунок 4.2 – Вміст головного файлу системи

Перший рядок містить ім'я системи, а також її поточну версію. Наступні рядки мають наступний синтаксис - «:ключове_слово параметри». Кожне ключове слово реалізує певну задачу. Розглянемо ключові слова, які були використані в розробленій системі.

1. :description – короткий опис задач, які вирішує система.
2. :url – посилання за яким знаходиться програмна реалізація.
3. :license – ліцензія, яку має дана система.
4. :dependencies – містить список залежностей. Кожна залежність складається з пар, де перший елемент – це ім'я бібліотеки, а другий – це або версія або код коміту або ім'я тегу коміту.
5. :java-source-paths – список директорій, в якій знаходиться вихідний код написаний мовою Java. Всі файли з розширенням java, що лежать в

даних директоріях компілюються в class файли, і відповідні класи можуть бути використані під час розробки мовою Clojure.

6. `:target-path` – директорія, яка містить результати компіляції вихідного коду системи, а також її залежностей.
7. `:profiles` – описує режими в яких може працювати система.
8. `:aliases` – описує команди виклик яких робить підстановку відповідної послідовності слів.

Розроблена система має наступні залежності:

- 1) `org.clojure/clojure` – необхідна для задання версії Clojure;
- 2) `org.antlr4/antlr4` – містить інструменти для автоматичної генерації інструментів синтаксичного розбору програмного коду;
- 3) `org.antlr4/antlr4-runtime` – містить засоби для взаємодії зі згенерованими інструментами (див. п.2);
- 4) `me.raynes/fs` – містить утиліти для роботи з файловою системою;
- 5) `org.clojure/math.combinatorics` – містить реалізацію основних комбінаторних функцій;
- 6) `org.clojure/data.csv` – містить інструменти для запису і читання CSV файлів.

4.2 Модуль генерації інструментів синтаксичного аналізу

Розроблена система має спеціальний режим (`antlr`), що був створений для генерації парсерів відповідно до файлів граматик, які знаходяться в директорії з наступним шаблоном для певної мови програмування - `resources/grammars/<ім'я мови>`. Конфігурація даного режим містить пустий список директорій з вихідним кодом `java`, адже його основна мета - це їх генерація.

Основною програмною одиницею проєкту є простір імен. На рис. 4.3 зображено оголошення основного простору імен `antlr` режиму.

Опишемо ключові слова, що можуть використовуватися в оголошеннях просторів імен.

1. `:require` – необхідне для завантаження бібліотек від яких залежить даний простір імен. Бібліотека описана списком, де перший елемент –

це її назва, а інші елементи описують опції. Наприклад, мета опції, що задається ключовим словом «as» - це створення псевдоніму для відповідної бібліотеки. Також можуть бути використані інші ключові слова, які будуть розглянуті далі.

2. `:import` – створене для завантаження скомпільованих класів у даний простір імен. Далі ці класи можуть бути використанні використовуючи лише їх ім'я (тобто, без задання пакету в якому вони знаходяться).
3. `:gen-class` – мета якого є генерація класу для даного простору імен. Головною функцією (тобто, `main`) вважається та, яка має ім'я – «-main».

```
(ns string-analysis antlr.core
  (:require [clojure.java.io :as io]
            [clojure.string :as str]
            [me.raynes.fs :as fs])
  (:import [java.io File]
            [org.antlr.v4 Tool])
  (:gen-class))

(def grammars-dir (-> "grammars" io/resource .getPath File))
(def java-dir (-> "src/java/" fs/file .getAbsolutePath))
(def grammar-exts #{" .g4" ".g"})
```

Рисунок 4.3 – Інтерфейс основного простору імен antlr режиму, а також константи

Інтерфейс головної функції зображено на рис.4.4. В якості вхідних аргументів вона приймає список мов для яких мають згенеруватися парсери. Список мов конвертується у відповідний список директорій з їх граматиками. У випадку, якщо не існує директорії для деякої мови програмування, виконання генерації зупиняється і видається відповідна помилка. Якщо не передано жодного імені мови програмування, то генерація відбувається для усіх мов для яких існують директорії з граматиками у `resources/grammars`.

```

(defn- grammars->paths [grammars]
  (if (seq grammars)
    (for [grammar grammars]
      :let [g (str grammar)
            path (File. grammars-dir g)]]
      (do (assert (.exists path) (format "Cannot find grammar: %s" g)) path))
    (-> grammars-dir fs/list-dir (filter fs/directory?))))

(defn -main [& grammars]
  (doseq [path (grammars->paths grammars)]
    (gen-parser path)))

```

Рисунок 4.4 – Головна функція antlr режиму

Після визначення директорій з файлами граматики викликається функція генерації для кожної директорії окремо.

Алгоритм даної функції складається з кроків, що описані нижче.

Крок 1. Визначається дві групи файлів, що містяться у вхідній директорії.

Перша група – це безпосередньо файли, що описують граматику у РБНФ. Головною ознакою цієї групи файлів є розширення .g або .g4, що описані константною змінною – grammar-exts (див. рис.4.3). Всі інші файли відносяться до другої групи. Необхідність виділення даної групи з'явилася після детального вивчення граматик ANTLR, що знаходяться у вільному доступі (в деяких випадках згенеровані інструменти потребують додаткових файлів для коректної роботи синтаксичного аналізатора). Варто зазначити, що генерація не відбувається у випадку, якщо перша група не містить жодного файлу.

Крок 2. Визначається кінцева директорія, що буде містити згенеровані файли. Дана директорія має наступний шлях – src/java/parser/generated/<ім'я мови>.

Крок 3. Визначається пакет згенерованих класів, який має наступний вигляд – parser.generated.<ім'я мови>.

Крок 4. Попередньо, видаляються всі файли і директорії, що містяться у кінцевій директорії.

Крок 5. Створюється об'єкт ANTLR інструменту в який передаються наступні параметри: шляхи файлів грамматики, шлях кінцевої директорії, а також ім'я пакету. Після цього викликається метод, який відповідає за процес генерації. Якщо процес закінчився успішно (тобто, кількість помилок не перевищує 0), копіюються файли і/або директорії, що знаходяться в другій групі, в кінцеву директорію. У випадку помилки видається відповідне повідомлення і функція завершує своє виконання.

Код алгоритму, який описаний вище, зображено на рис.4.5.

```
(defn- gen-parser [dir-path]
  (let [{grammar-files true other-files false}
        (group-by (comp (partial contains? grammar-exts) fs/extension)
                  (fs/list-dir dir-path))
        grammar (.getName dir-path)
        dir-tree ["parser" "generated" grammar]
        package (str/join \. dir-tree)
        target-path (apply fs/file java-dir dir-tree)]
    (if (empty? grammar-files)
      (printf "Cannot find %s grammar files (with either .g or .g4 extension)\n" grammar)
      (let [antlr-options ["-package" package "-o" (.getPath target-path)]
            antlr-tool (->>
                        (map #(.getPath %) grammar-files)
                        (concat antlr-options)
                        (into-array String)
                        Tool.)]
          (when (fs/exists? target-path) (fs/delete-dir target-path))
          (.processGrammarsOnCommandLine antlr-tool)
          (let [errors-num (-> antlr-tool .errMgr .getNumErrors)]
            (if (> errors-num 0)
              (printf "%s parser hasn't been generated. Number of errors - %s.\n" grammar errors-num)
              (do (doseq [file other-files]
                    (if (fs/directory? file)
                      (fs/copy-dir file target-path)
                      (->> file fs/base-name (fs/file target-path) (fs/copy file))))
                  (printf "%s parser has been successfully generated.\n" grammar)))))))))
```

Рисунок 4.5 – Функція генерації синтаксичного аналізатора

Після генерації синтаксичних аналізаторів система працює у звичайному режимі (іншими словами, режим статичного аналізу), реалізація яких знаходиться в модулях, які будуть описані далі у розділі.

4.3 Модуль синтаксичного аналізу програмного коду

Мета даного модулю – це перетворення дерев розбору, які були отримані за допомогою згенерованих інструментів, у внутрішнє представлення, що описане структурами даних мови Clojure.

Даний модуль складається з декількох просторів імен:

1. `core.clj` – містить засоби для роботи з даними, які повертаються згенерованими інструментами;
2. `matching.clj` – містить засоби для пошуку певних шаблонів (напр. операцій присвоювання) у дереві розборі;
3. `java.clj` – містить засоби перетворення програмного коду, що написаний мовою Java, у внутрішнє представлення, а також функцій, що додають додаткову інформацію до певних вузлів дерева, яка необхідна для статичного аналізу;
4. `mysql.clj` – містить функцію-обгортку для синтаксичного розбору SQL запитів. Вона буде використана для оцінки ефективності розробленого методу (див. розділ 5).

Розглянемо кожен простір імен більш детально.

На рис. 4.6 зображено оголошення першого простору імен. Як можна помітити, у оголошенні необхідних бібліотек з'явилося нове ключове слово - `:refer`. Воно необхідне для задання символів, що треба завантажити у поточний простір імен. В даному випадку `:all` означає, що всі символи мають бути завантажені.

```
(ns string-analysis.parser.core
  (:require [string-analysis.utils :refer :all])
  (:import [org.antlr.v4.runtime CharStreams CommonTokenStream]
           [org.antlr.v4.runtime.tree TerminalNode]
           [org.antlr.v4.runtime.tree.pattern RuleTagToken TokenTagToken]))
```

Рисунок 4.6 – Інтерфейс простору імен `core`

Засоби синтаксичного розбору, що були згенеровані інструментами з ANTLR інструменту, повертають корінь дерева. Вузли дерева діляться на 2 види: нетермінальний та термінальний відповідно до правил грамматики.

На рис. 4.7 зображено тіло функції розбору програмного коду. Вона приймає чотири параметри: клас лексичного аналізатору; клас синтаксичного аналізатору; початкове правило розбору; код програми, що може бути представлений або файлом або рядком.

```
(defn parse [lexer-class parser-class rule source]
  (let [input (if (instance? java.io.File source)
                 (CharStreams/fromFileName (.getPath source))
                 (CharStreams/fromString source))
        lexer (new-instance lexer-class input)
        tokens (CommonTokenStream. lexer)
        parser (new-instance parser-class tokens)]
    (java-tree->clj-term (invoke-method parser (name rule)) parser)))
```

Рисунок 4.7 – Функція розбору програмного коду

Її суть полягає у послідовному перетворенні сутностей (рис.4.8).

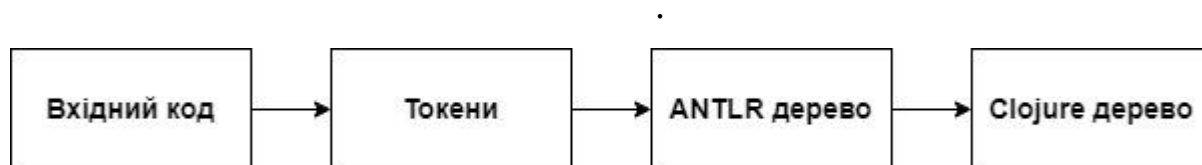


Рисунок 4.8 – Послідовність перетворення сутностей

На рис. 4.9 зображено тіло функції, що необхідна для розбору програмного коду в якому присутні шаблони. Відповідно до документації ANTLR, шаблони задаються наступним чином - <мітка : правило> або <правило> (у цьому випадку мітка має ім'я аналогічне назві правила). Наприклад, шаблон операції присвоювання може бути заданий наступним чином: <identifier> = <expression>. Під час низхідного розбору програмного коду, синтаксичний аналізатор зупиняє розбір на правилах, що відповідають шаблонам і створює спеціальні шаблонні

токени, які показують, що на їх місці можуть бути присутні дерева з відповідними коренями, або термінали певного типу.

```
(defn parse-pattern [parser rule pattern-str]
  (java-tree->clj-term
    (->> (.getRuleIndex parser (name rule))
          (.compileParseTreePattern parser pattern-str)
          (.getPatternTree))
    parser))
```

Рисунок 4.9 – Функція розбору програмного коду з шаблонами

Дана функція приймає наступні параметри: об'єкт синтаксичного аналізатору; початкове правило розбору; рядок, що представляє програмний код;

Послідовність кроків алгоритму перетворення ANTLR дерева у Clojure дерево зазначено нижче.

Крок 1. Якщо поточний вузол є термінальним, то повертається об'єкт токenu.

Крок 2. Якщо поточний вузол є нетермінальним, то отримується список перетворених дочірніх дерев шляхом рекурсивного виклику функції над кожним нащадком даного вузла. Кінцеве дерево представляється списком, де голова описує ім'я граматичного правила, якому відповідає поточний вузол, а хвіст - перетворений список нащадків. В якості додаткової інформації в метадані отриманого списку додається об'єкт синтаксичного аналізатору

На рис 4.10 зображено код функції перетворення.

```
(defn java-tree->clj-term [tree parser]
  (letfn [(iter [tree]
            (if (instance? TerminalNode tree)
                (.getSymbol tree)
                (-> (rule-name parser tree)
                    (cons (map1 iter (get-children tree))
                          (with-meta {:parser parser}))))]
    (iter tree)))
```

Рисунок 4.10 – Функція перетворення ANTLR дерева у Clojure

В подальшому терміну «дерево розбору» буде відповідати термін Clojure дерева.

Код функцій для отримання списку нащадків, а також імені правила зображено на рис. 4.11.

```
(defn- get-children [tree]
  (->> tree .getChildCount range (map #(.getChild tree %))))

(defn- rule-name [parser tree]
  (->> tree .getRuleIndex (get (.getRuleNames parser) keyword)))
```

Рисунок 4.11 – Допоміжні функції обробки вузла дерева

Також, даний простір імен містить функцію для заміни піддерев у вхідному дереві розбору, код якої зображений на рис. 4.12.

```
(defn replace-subtree [term replace-map]
  (or (get replace-map term)
      (if (coll? term)
          (->> (rest term)
                (mapl #(replace-subtree % replace-map))
                (cons (first term))
                (copy-meta term))
          term))))
```

Рисунок 4.12 – Функція для заміни піддерев

Одним з найважливіших засобів є зворотне перетворення дерева розбору у програмний код. Так як інструменти згенеровані новими версіями ANTLR зберігають всі «непробільні» токени (включно з комами, дужками і т.п.), то суть зворотного перетворення полягає у послідовному поєднанні текстів токенів через пробіл. Варто зазначити, що в такому випадку кінцевий код програми не буде точною відповідністю реальному тексту (це пов'язано з ігноруванням всіх пробільних символів під час процесу синтаксичного розбору), але він має точну відповідність його структурі.

Код функції, що містить дану логіку зображено на рис 4.13.

```
(defn term->string [tree]
  (let [out (StringBuilder.)
        iter
        (fn iter [tree]
          (if (coll? tree)
              (doseq [child (rest tree)] (iter child))
              (.append out (format "%s " (.getText tree)))))]
    (iter tree)
    (str out)))
```

Рисунок 4.13 – Функція перетворення дерева розбору у програмний код

Оголошення наступного простору імен (matching), який містить засоби для зіставлення дерев розбору з шаблонами зображено на рис. 4.14.

```
(ns string-analysis.parser.matching
  (:require [string-analysis.parser.core :refer :all]
            [string-analysis.utils :refer :all])
  (:import [org.antlr.v4.runtime.tree.pattern RuleTagToken TokenTagToken]))
```

Рисунок 4.14 – Інтерфейс простору імен matching

Всі засоби зіставлення базуються на макросі `forthis`, який приймає параметри, суть яких описана нижче.

1. Дерево розбору.
2. Змінну, яка представляє хеш-таблицю, де ключем хешування є ім'я мітки, а значенням дерево розбору, що зіставилося з шаблоном токеном з даною міткою.
3. Списки шаблонів. Голова кожного списку описує шаблони і може бути представлена або парою (<ім'я правила> <код з шаблонами>) або списком, де головою є ключове слово `:or`, а хвостом - список пар (<ім'я правила> <код з шаблонами>). Хвостом списку є послідовність форм,

що виконуються у разі успішного зіставлення з даним шаблоном (або шаблонами у випадку присутності ключового слова `:or`).

Код даного макросу (з допоміжною функцією) зображений на рис. 4.15.

```
(defn- forthis-aux [rule pattern-str term]
  (let [parser (term-parser term)]
    (assert parser (format "Broken tree: %s" term))
    (match term (parse-pattern parser rule pattern-str))))

(defmacro forthis
  { :style/indent :defn }
  [term spec & bindings]
  (letfn [(expand [bindings]
            (when-first [[patterns & body] bindings]
              (if (= patterns true)
                `(let [~'self ~term] ~@body)
                `(if-let [~spec
                          (or ~@(for [[rule pattern-str]
                                       (if (= (first patterns) :or)
                                           (rest patterns)
                                           [patterns]))]
                                `(#'forthis-aux '~rule ~pattern-str ~term)))]
                  (let [~'self ~term] (do ~@body))
                  ~(expand (rest bindings))))))
          (expand bindings))])
```

Рисунок 4.15 – Макрос зіставлення дерев розбору

Приклад використання даного макросу для перевірки дерева розбору на приналежність операції присвоювання зображено на рис.4.16.

```
(forthis term spec
  ((statement "<identifier> = <expression> ;")
   (printf "Here's target: %s" (term->string (:identifier spec)))
   (printf "Here's assign value: %s" (term->string (:expression spec)))))
```

Рисунок 4.16 – Приклад використання макросу `forthis`

Також, варто зазначити що у випадку успішного зіставлення з одним із шаблонів, інші шаблони не перевіряються.

Алгоритм зіставлення дерева розбору з шаблонним складається з описаних нижче кроків.

Крок 1. У випадку, якщо обидва дерева являються нетермінальними (якщо ні то див. крок 5), то порівнюються їх перші елементи (тобто імена правил), і у випадку їх нерівності, зіставлення зупиняється.

Крок 2. У іншому випадку перевіряється перший нащадок шаблонного дерева, і, якщо, він представляє спеціальний токен для опису нетермінальних шаблонів то в кінцеву хеш-таблицю для мітки даного токена додається значення дерева розбору.

Крок 3. Якщо перший елемент не є шаблонним токеном, то порівнюються довжини списків нащадків дерев, і у випадку їх нерівності зіставлення зупиняється.

Крок 4. У іншому випадку, для відповідних нащадків повторюється послідовність дій (див. крок 1).

Крок 5. Якщо одне з дерев являється нетермінальним, то зіставлення зупиняється.

Крок 6. Якщо ж обидва дерева є терміналами, то спочатку аналізується значення шаблонного токена. У випадку, якщо він являється спеціальним токеном для опису термінальних шаблонів (якщо ні - див. крок 7), то перевіряється відповідність типів токенів, і якщо вони однакові, то у хеш-таблицю для мітки шаблонного токена додається значення токена з дерева розбору (у іншому випадку зіставлення зупиняється).

Крок 7. Перевіряється текст токенів і у випадку їх нерівності зіставлення зупиняється.

Код алгоритму, який описаний вище, зображено на рис. 4.17

```

(defn- match [term pattern-term]
  (letfn [(bind [term pattern-term]
            (cond
              (and (coll? term) (coll? pattern-term))
              (let [[rule1 & children1] term
                    [rule2 & children2] pattern-term]
                (if (= rule1 rule2)
                  (let [maybe-pattern (first children2)]
                    (cond
                     (instance? RuleTagToken maybe-pattern)
                     [[(pattern-label maybe-pattern) term]]
                     (= (count children1) (count children2))
                     (lazy-mapcat bind children1 children2)
                     :else [:not-matched]))
                  [:not-matched]))
              (or (coll? term) (coll? pattern-term))
              [:not-matched]
              (instance? TokenTagToken pattern-term)
              (if (= (.getType term) (.getType pattern-term))
                [[(pattern-label pattern-term) term]]
                [:not-matched])
              (not= (.getText term) (.getText pattern-term))
              [:not-matched]))])
    (reduce
     (fn [acc spec]
       (cond
         (= spec :not-matched) (reduced nil)
         (some? spec) (assoc acc (keyword (first spec)) (second spec))
         :else acc))
     {}
     (bind term pattern-term))))

```

Рисунок 4.17 – Функція зіставлення дерева розбору з шаблоном деревом

Ще одним інструментом зіставлення є рекурсивна версія макросу `forthis`, який називається `foreach`, що призначений для рекурсивного переписування дерев розбору.

Даний макрос приймає параметри, суть яких описана нижче.

1. Дерево розбору.
2. Хеш-таблиця зі значенням міток.
3. Списки шаблонів. Відмінністю є можлива наявність спеціального ключового слова перед шаблонами, яке регулює порядок зіставлення. Списки шаблонів розділяються на дві групи на основі яких створюється відповідні `forthis`-и, що повертають модифіковане дерево розбору.

Перша група містить ті списки, шаблони яких обрамлені ключовим словом `:pre`, і представляє зіставлення, що має відбуватися на спуску. Друга група містить ті списки, шаблони яких обрамлені ключовим словом `:post` або які не мають жодного ключового слова порядку (іншими словами, `:post` задається за замовчуванням), і представляє зіставлення, що має відбуватися на підйомі. Також до кожного згенерованого `forthis`-у додається гілка, яка повертає спеціальне ключове слово для ідентифікації неуспішності переписування.

Код даного макросу зображений на рис. 4.18.

```
(defmacro foreach
  {style/indent :defn}
  [term spec & bindings]
  (let [{:keys [pre post]}
    (->>
      (for [[pre-or-post pattern] & body :as binding] bindings)
      (case pre-or-post
        (:pre :post) [pre-or-post (cons pattern body)]
        [:post binding]))
    (group-by first)
    (map-vals (partial map second))))]
  `( #'match-all ~term
      (fn [term#] (forthis term# ~spec ~@pre (true ::not-matched)))
      (fn [term#] (forthis term# ~spec ~@post (true ::not-matched))))))
```

Рисунок 4.18 – Макрос рекурсивного зіставлення дерев розбору

Кроки алгоритму рекурсивної функції зіставлення описані нижче.

Крок 1. Робиться спроба переписування «на спуску» над поточним деревом розбору і у випадку її успішності, переписування нащадків не відбувається. Повертається отримане дерево розбору зі скопійованими метаданими з вхідного дерева.

Крок 2. Якщо спроба переписування була невдалою, то формується нове дерево, корінь (тобто, голова списку) якого є аналогічним вхідному дереву, а список нащадків формується шляхом рекурсивного виклику даної функції над кожним нащадком вхідного дерева (тобто, відбувається переписування нащадків). Відбувається спроба

модифікації нового дерева «на підйомі» і у випадку її успішності, повертається отримане дерево зі скопійованими метаданими з вхідного.

Крок 3. У випадку неуспішності спроби переписування повертається дерево з модифікованими нащадками.

На рис 4.19 зображено код функції в якій реалізовано даний алгоритм.

```
(defn- match-all [term pre-match-fn post-match-fn]
  (if (coll? term)
    (let [pre-term (pre-match-fn term)]
      (if (is-matched? term pre-term)
        (copy-meta term pre-term)
        (let [new-term (->> (rest term)
                              (mapl #(match-all % pre-match-fn post-match-fn)
                                     (cons (first term)
                                           (copy-meta term)))
                              post-term (post-match-fn new-term))]
          (if (is-matched? term post-term)
            (copy-meta term post-term)
            new-term))))
    term))
```

Рисунок 4.19 - Функція рекурсивного зіставлення дерева розбору з шаблонним деревом

В якості мови програмування над якою виконується статичний аналіз коду було обрано мову Java. Наступний простір імен (java), оголошення якого зображено на рис. 4.20, містить засоби обробки програмного коду написаного цією мовою.

```
(ns string-analysis.parser.java
  (:require [string-analysis.parser
             [core :as parser]
             [matching :refer :all]]
            [string-analysis.utils :refer :all])
  (:import [parser.generated.java JavaLexer JavaParser]))
```

Рисунок 4.20 – Інтерфейс простору імен java

Даний простір імен містить декілька функцій-обгорток (рис. 4.21) синтаксичного розбору коду, мета яких полягає у інкапсуляції згенерованих класів лексичного і синтаксичного аналізаторів.

```
(defn parse [rule source]
  (parser/parse JavaLexer JavaParser rule source))

(defn parse-with-convert [rule source]
  (let [id-table (java.util.HashMap.)]
    [(-> (parse rule source) (convert-identifiers id-table))
     (into {} id-table)]))
```

Рисунок 4.21 – Функції-обгортки синтаксичного аналізу

Додаткову увагу варто приділити другій функції, мета якої полягає у створенні унікальних числових ідентифікаторів для імен змінних. Необхідність створення даної функції пов'язана з тим, що ім'я не являється показником унікальності для змінної, адже у мові Java область видимості змінної - це блок у якому вона створюється. Так як основна мета згортки констант – це знаходження множини можливих значень для змінних програми, то використання їх імен у якості ключів хеш-таблиці, що зберігає значення, є неправильним.

Функція перетворення приймає наступні параметри: дерево розбору; інформаційна хеш-таблиця, де ключем хешування є числовий ідентифікатор змінної, а значенням інформаційний об'єкт; список контекстів, де кожен контекст описаний хеш-таблицею, де ключем є ім'я змінної, а значенням – числовий ідентифікатор.

Її алгоритм полягає у переписуванні зазначених нижче шаблонів.

1. Дерево, що відповідає конструкції блоку. У разі його знаходження, новий контекст ініціалізується пустою хеш-таблицею. Рекурсивно викликається функція перетворення з оновленим списком контекстів над нащадками зіставленого дерева. На основі перетворених нащадків формується нове дерево розбору.

2. Дерево, що представляє операції оголошення змінних. У разі його знаходження викликається функції обробки оголошення змінних, суть якої полягає у послідовному аналізі кожної операції оголошення шляхом генерації ідентифікаторів для імен змінних, і додаванні їх у перший контекст, а також створення інформаційних об'єктів з подальшим додаванням у інформаційну хеш-таблицю. Після цього у метадані дерева розбору, що представляє ідентифікатор, додається згенерований числовий ідентифікатор. Якщо змінна при оголошенні має початкове значення, то відбувається переписування шаблонів описаних у п.3, у дереві, яке представляє початкове значення.
3. Дерево розбору, що представляє використання змінної у різноманітних виразах. У разі його знаходження відбувається пошук числового ідентифікатора у вхідних контекстах (у разі невдачі відбувається генерація числового ідентифікатора і інформаційного об'єкту, що представляє невідому змінну, під час аналізу даного правила), який додається у метадані дерева розбору, що представляє ідентифікатор.

Код функції перетворення змінних зображено на рис. 4.22.

```
(defn convert-identifiers
  ([term id-table]
   (convert-identifiers term id-table (list (java.util.HashMap))))
  ([term id-table contexts]
   (foreach term {:keys [identifier t]}
    ((:pre (block "<block>"))
     (let [contexts (cons (java.util.HashMap.) contexts)]
      (->>
       (rest self)
       (mapl #(convert-identifiers % id-table contexts))
       (cons (first self))))))
   ((:pre (localVariableDeclaration "<t:typeType> <variableDeclarators>"))
    (add-declarations self {:type t} id-table contexts))
   ((expression "<identifier>")
    (let [id-name (identifier-name identifier)
          id (or (find-in-contexts contexts id-name)
                 (:id (add-var id-table contexts {:type :unknown :name id-name}))))]
     (parser/replace-subtree self {identifier (vary-meta identifier merge {:id id})}))))
```

Рисунок 4.22 – Функція перетворення змінних

Код функції обробки оголошення змінних зображено на рис. 4.23.

```
(defn- add-declarations [term info id-table contexts]
  (foreach term {:keys [identifier var-init]}
    ((:pre (variableDeclarator "<identifier>"))
      (let [{:keys [id]}
            (add-var id-table contexts (assoc info :name (identifier-name identifier)))]
        (parser/replace-subtree self
                                {identifier (vary-meta identifier merge {:id id})})))
    ((:pre (variableDeclarator "<identifier> = <var-init:variableInitializer>"))
      (let [{:keys [id]}
            (add-var id-table contexts (assoc info :name (identifier-name identifier)))]
        (parser/replace-subtree
          self
          {identifier (vary-meta identifier merge {:id id})
            var-init (convert-identifiers var-init id-table contexts)}}))))
```

Рисунок 4.23 – Функція обробки оголошення змінних

Останній простір імен, оголошення якого зображено на рис. 4.24, аналогічно до простору імен java містить функцію-обгортку синтаксичного розбору коду для інкапсуляції згенерованих класів лексичного і синтаксичного аналізаторів. Але, на відміну від простору імен java, дані функції будуть використані в подальшому для оцінки ефективності розробленого методу в різних умовах.

```
(ns string-analysis.parser.mysql
  (:require [string-analysis.parser.core :as parser])
  (:import [parser.generated.mysql MySqlLexer MySqlParser]))

(defn parse [rule source]
  (parser/parse MySqlLexer MySqlParser rule source))
```

Рисунок 4.24 – Простір імен mysql

4.4 Модуль побудови графу потоку виконання

Мета даного модулю – це створення графів потоку виконання на основі дерев розбору отриманих за допомогою засобів реалізованих в попередньому модулі.

Даний модуль складається з наступних просторів імен:

- 1) `core.clj` – містить структури даних для представлення графів потоку виконання, а також засоби для роботи з ними;
- 2) `java.clj` - містить засоби для створення графу потоку виконання на основі дерев розбору отриманих на основі синтаксичного розбору коду програм, що написані мовою Java.

Інтерфейс простору імен `core` зображений на рис. 4.25.

```
(ns string-analysis.cfg.core
  (:require [string-analysis.utils :refer :all]
            [clojure.java.io :as io]
            [string-analysis.parser.core :refer [term->string]]))
```

Рисунок 4.25 – Інтерфейс простору імен `core`

Технічно, абстрактний граф потоку виконання реалізований на основі структури даних – протокол, інтерфейс якого зображений на рис. 4.26.

```
(defprotocol CFG
  (set-bounds [this] "Initialize start and end points of control flow")
  (get-bounds [this] "Get start and end nodes points of control flow")
  (add-edge [this node1 node2] "Add edge between two nodes")
  (node-children [this node] "Get children of node")
  (node-parents [this node] "Get parents of node"))
```

Рисунок 4.26 – Інтерфейс CFG протоколу

Даний протокол містить наступні методи (без конкретної реалізації):

- 1) `set-bounds` – метод для ініціалізації початкового і кінцевого вузла графу;
- 2) `get-bounds` – метод для отримання початкового і кінцевого вузла графу;
- 3) `add-edge` – метод для додавання зв'язку (ребра) між двома вузлами;
- 4) `node-children` – метод для отримання вузлів в які можна потрапити з вхідного вузла напрямку;

- 5) `node-parents` – метод для отримання вузлів з яких можна потрапити у вхідний вузол напряму.

На рис. 4.27 зображено оголошення класу з реалізаціями методів, що були описані вище, а також конструктору.

```
(deftype SimpleCFG [^:volatile-mutable entry-node
                    ^:volatile-mutable exit-node
                    ^java.util.HashMap node-to-parents
                    ^java.util.HashMap node-to-children]

  CFG
  (set-bounds [this]
    (set! (.entry-node this) (->EntryNode))
    (set! (.exit-node this) (->ExitNode)))
  (get-bounds [this]
    (vector entry-node exit-node))
  (add-edge [this node1 node2]
    (put-update (partial cons node2) node1 node-to-children)
    (put-update (partial cons node1) node2 node-to-parents))
  (node-children [this node]
    (.get node-to-children node))
  (node-parents [this node]
    (.get node-to-parents node)))

(defn make-simple-cfg []
  (doto (->SimpleCFG nil nil (java.util.HashMap.) (java.util.HashMap.))
    (set-bounds)))
```

Рисунок 4.27 – Клас графу потоку виконання

Даний клас має наступні поля:

- 1) `entry-node` – поле для зберігання початкового вузла графу;
- 2) `exit-node` – поле для зберігання кінцевого вузла графу;
- 3) `node-to-parents` – хеш-таблиця, де ключем хешування є об'єкт вузла графу, а значенням – список прямих предків;
- 4) `node-to-children` – хеш-таблиця, де ключем хешування є об'єкт вузла графу, а значенням – список прямих нащадків.

Також містяться оголошення протоколів для створення на їх основі класів вузлів для узагальнення певних понять (напр. – вузол, що представляє кінцеву

точку програми) або ті, які базуються на деревах розборі на основі яких виконуються різноманітні аналізи. На рис. 4.28 зображено протоколів, а також класів, що представляють кінцевий і початкових вузол графу.

```
(defprotocol TagNode
  (node-tag [this]))

(defprotocol TermNode
  (node-term [this]))

(deftype EntryNode []
  TagNode
  (node-tag [this] :entry))

(deftype ExitNode []
  TagNode
  (node-tag [this] :exit))
```

Рисунок 4.28 – Оголошення протоколів вузлів, а також класів для представлення початкового і кінцевого вузла графу

На рис.4.29 зображено код функції для обходу графу потоку виконання

```
(defn traverse-cfg
  ([f cfg]
   (traverse-cfg f cfg (first (get-bounds cfg))))
  ([f cfg entry-node]
   (let [visited (java.util.HashSet.)
         traverse-aux
         (fn traverse-aux [node]
            (when-not (.contains visited node)
              (.add visited node)
              (f node)
              (doseq [child-node (node-children cfg node)]
                (traverse-aux child-node)))]
             (traverse-aux entry-node)))]
```

Рисунок 4.29 – Функція обходу графу потоку виконання

Дана функція робить pre-order обхід (тобто, спочатку корінь, а потім нащадки) і викликає вхідну функцію обробки, що приймає об'єкт вузла в якості аргументів.

Також, даний простір імен, містить функцію запису графу у dot форматі (це текстова мова опису різноманітних графів), код якої зображений на рис. 4.30.

```
(defn cfg->dot
  ([cfg]
   (let [w (java.io.StringWriter.)]
     (cfg->dot w cfg)
     (str w)))
  ([writer cfg]
   (with-open [w (io/writer writer)]
     (write-str w "digraph G {\n")
     (traverse-cfg
      (fn [node]
        (write-str w "\t%s [label=\"%s\"] ;\n" (hash node) (node-text node))
        (doseq [child-node (node-children cfg node)]
          (write-str w "\t%s -> %s;\n" (hash node) (hash child-node))))
      cfg)
     (write-str w "}")
     )))
```

Рисунок 4.30 – Функція перетворення графу потоку виконання у dot формат

Наступний простір імен java (рис. 4.31) містить логіку для створення графу потоку виконання на основі дерев розбору.

```
(ns string-analysis.cfg.java
  (:require [string-analysis.parser
             [matching :refer :all]
             [java :refer :all]]
            [string-analysis.cfg.core :refer :all]))
```

Рисунок 4.31 – Інтерфейс простору імен java

В кінцевому графу потоку виконання можуть бути присутні наступні типи вузлів:

- 1) BasicNode – представляє будь-яку інструкцію, що не має специфічної семантики;
- 2) MethodExitNode – представляє кінцеву точку виконання методу;
- 3) ConditionNode – представляє інструкцію, що має декілька можливих шляхів виконання в залежності від умови, яку він описує;
- 4) ReturnNode – представляє інструкцію повернення з методу;
- 5) MethodInitNode – представляє початкову точку виконання методу, і містить дерево розбору, що описує формальні параметри.

На рис. 4.32 зображено код класів, що були описані вище.

```
(deftype BasicNode [term]
  TermNode
  (node-term [this] term))

(deftype MethodExitNode []
  TagNode
  (node-tag [this] :method-exit))

(deftype ConditionNode [term]
  TermNode
  (node-term [this] term))

(deftype ReturnNode [term]
  TermNode
  (node-term [this] term))

(deftype MethodInitNode [term]
  TermNode
  (node-term [this] term))
```

Рисунок 4.32 – Вузли графу потоку виконання

На рис. 4.33 зображено головна функція побудови графу потоку виконання. Її суть полягає у створенні об'єкту класу, що представляє граф (див. попередній простір імен), замикає його у глобальну змінну (для збільшення читабельності коду), а також виклику функції перетворення дерева розбору у вузол графу. Дана функція приймає вузол, що представляє дію яка, можливо, буде наступною (в

початковій конфігурації ця дія представлена кінцевим вузлом графу) і дерево розбору, на основі якого створюється і повертається вузол графу, як результат даної функції.

```
(defn build-cfg [term]
  (with-cfg (make-simple-cfg)
    (let [[entry-node exit-node] (get-bounds *cfg*)]
      (add-edge *cfg* entry-node (term->node exit-node term))
      *cfg*)))
```

Рисунок 4.33 – Функція побудови графу потоку виконання

Функція перетворення базується на пошуку певних шаблонів і створення зв'язків (іншими словами, можливих шляхів виконання) на їх основі. Розглянемо більш детально кожен шаблон.

На рис. 4.34 зображено обробка дерева, що представляє оголошення методу.

```
((methodDeclaration "<typeTypeOrVoid> <identifier> <formalParameters> <block>")
 (let [init-node (->MethodInitNode (:formalParameters spec))
       exit-node (->MethodExitNode)]
   (add-edge *cfg* init-node (term->node exit-node block))
   (add-edge *cfg* exit-node next-node)
   (traverse-cfg
    (fn [node]
      (when (instance? ReturnNode node)
        (add-edge *cfg* node exit-node)))
    *cfg*
    init-node)
   init-node))
```

Рисунок 4.34 – Перетворення оголошення методу у вузол графу

Його суть полягає у створенні початкового (зі зберіганням дерева розбору, що описує формальні параметри) і кінцевого вузла методу (даний вузол передається у рекурсивний виклик функції, що приймає його в якості наступного, і дерево розбору, що представляє тіло методу). Також додається ребро між кінцевим вузлом і наступним, що прийшов на вхід разом з деревом розбору для аналізу, де кінцевий вузол являється предком. Також відбувається обхід

побудованого графу для пошуку вузлів повернення для зв'язування їх з кінцевим вузлом, де вузол повернення являється нащадком. В якості результату повертається початковий вузол методу.

На рис. 4.35 зображено обробку блок конструкції, суть якої полягає у зворотному обході інструкцій, що належать даному блоку, так що вузол, що повертається при обробці інструкції передається у якості наступного вузла при обробці попередньої інструкції (для останньої інструкції наступним вузлом буде вхідний). В якості результату повертається вузол, що представляє першу інструкцію.

```
((:or (block "<block>") (statement "<block>"))
  (let [[root & children] block]
    (->> children rest reverse rest (reduce term->node next-node))))
```

Рисунок 4.35 – Перетворення блок конструкції у вузол графу

На рис 4.36 зображено обробку умовного оператора. Створюється вузол для збереження умовного виразу. Даний вузол (в якості предка) пов'язується з вузлами, що були отримані при перетворенні інструкцій істини та хиби, що приймають вхідний вузол у якості наступного. У разі якщо гілка хиби відсутня, то замість нього вузол умови пов'язується з вхідним вузлом. У якості результату повертається вузол умови.

```
((statement "if ( <expression> ) <stmt:statement>")
  (let [expr-node (->ConditionNode expression)]
    (add-edge *cfg* expr-node (term->node next-node stmt))
    (add-edge *cfg* expr-node next-node)
    expr-node))
(statement "if ( <expression> ) <stmt:statement> else <stmt2:statement>")
(let [expr-node (->ConditionNode expression)]
  (add-edge *cfg* expr-node (term->node next-node stmt))
  (add-edge *cfg* expr-node (term->node next-node stmt2))
  expr-node))
```

Рисунок 4.36 – Перетворення умовного оператора у вузол графу

На рис. 4.37 зображено обробку циклу. Її відмінність від обробки умовного оператора полягає у тому, що перетворення тіла циклу у якості наступного вузла приймає вузол умови, який також пов'язується з вхідним вузлом.

```
((statement "while ( <expression> ) <stmt:statement>")
 (let [expr-node (->ConditionNode expression)]
  (add-edge *cfg* expr-node (term->node expr-node stmt))
  (add-edge *cfg* expr-node next-node)
  expr-node))
```

Рисунок 4.37 - Перетворення циклу у вузол графу

Як було сказано вище, інструкція повернення має спеціальне семантичне значення, тому при її обробці створюється спеціальний вузол (рис. 4.38).

```
((:or (statement "return ;")
      (statement "return <expression> ;"))
 (->ReturnNode term))
```

Рисунок 4.38 – Перетворення інструкції повернення у вузол графу

На рис. 4.39 зображено обробку: нащадка дерева для представлення інструкції блоку; дерева, що описує оголошення змінних; всіх інших типів дерев, що не мають специфічної семантики, що обробляється шаблонами, які описані вище.

```
((blockStatement "<stmt:statement>")
 (term->node next-node stmt))
(blockStatement "<var-decl:localVariableDeclaration> ;"
 (let [node (->BasicNode var-decl)]
  (add-edge *cfg* node next-node)
  node))
(true
 (let [node (->BasicNode term)]
  (add-edge *cfg* node next-node)
  node))))
```

Рисунок 4.39 – Перетворення інших типів інструкцій у вузол графу

4.5 Модуль аналізу потоку даних

Даний модуль містить реалізацію MFP алгоритму описаного у другому розділі, а також статичного аналізу java коду для вирішення задачі згортки рядкових констант.

Інтерфейс оголошення core неймпейсу зображено на рис. 4.40.

```
(ns string-analysis.dfa.core
  (:require [string-analysis.cfg.core :refer :all]))
```

Рисунок 4.40 – Інтерфейс простору імен core

З псевдокоду MFP алгоритму було виділено важливу логіку на якому базується даний алгоритм, в наступні методи, кожен з яких приймає в якості першого аргументу об'єкт, що описує тип аналізу, який виконується. Дані методи реалізуються окремо для кожного типу аналізу і їх суть полягає у наступному:

- 1) dfa-bound – метод, що повертає пусте значення, яке необхідне для початкової ініціалізації;
- 2) dfa-transfer – даний метод реалізує логіку трансферної функції;
- 3) dfa-equal? – необхідний для перевірки нових значень на наявність змін після певної ітерації аналізу (equal-функція);
- 4) dfa-join – метод, що використовується для об'єднання декількох можливих значень у єдине (join-функція).

На рисунку 4.41 зображено код оголошення даних методів (без конкретної реалізації), а також допоміжну функції необхідну для пошуку підходящого методу.

```
(defn- dfa-class [dfa & args] (class dfa))

(defmulti dfa-bound {:arglists '[[dfa]]} dfa-class)

(defmulti dfa-transfer {:arglists '[[dfa in-value node]]} dfa-class)

(defmulti dfa-equal? {:arglists '[[dfa value1 value2]]} dfa-class)

(defmulti dfa-join {:arglists '[[dfa value1 value2]]} dfa-class)
```

Рисунок 4.41 – Оголошення методів аналізу потоку даних

На рис. 4.42 зображено код функції, суть якої полягає у початковій ініціалізації описаних нижче значень.

1. Множина вузлів, які треба проаналізувати (за допомогою трансферної функції). Вона представлена такою структурою даних, як `LinkedHashSet`, що представляє множину, елементи якої відсортовані в порядку додавання. На початку дана множина містить всі вузли графу, що були зібрані за допомогою функції обходу графу виконання, яка була описана в минулому підрозділі.
2. Вхідна і вихідна хеш-таблиця значень, що на початку містить для кожного вузла значення отримані за допомогою `dfa-bound` методу.

```
(defn- initialize-dfa [dfa cfg]
  (let [worklist (java.util.LinkedHashSet.)
        in-table (java.util.HashMap.)
        out-table (java.util.HashMap.)
        init (dfa-bound dfa)]
    (traverse-cfg
      (fn [node]
        (.add worklist node)
        (.put in-table node init)
        (.put out-table node init))
      cfg)
    [worklist in-table out-table]))
```

Рисунок 4.42 – Функція ініціалізації

Також було створено клас, що містить хеш-таблиці (вхідну і вихідну) значень, об'єкт якого повертається після виконаного аналізу. Оголошення даного класу зображено на рис. 4.43.

```
(deftype DFAData [in-table out-table])
```

Рисунок 4.43 – Клас для представлення результату аналізу

Для представлення невідомих значень (тобто, якщо під час аналізу неможливо визначити, яке значення продукує певна операція) було створено спеціальну змінну, а також відповідний предикат (рис. 4.44).

```
(def dfa-unknown-value :dfa-unknown-value)

(defn is-unknown? [value]
  (= value dfa-unknown-value))
```

Рисунок 4.44 – Невідоме значення аналізу

На рис. 4.45 зображено реалізацію функції обчислення аналізу потоку даних, що базується на основних положеннях MFP алгоритму. В якості вхідних параметрів вона приймає: об'єкт для опису аналізу (він необхідний для пошуку конкретних методів, що були описані вище); граф потоку виконання.

```
(defn compute-dfa [dfa cfg]
  (let [[worklist in-table out-table] (initialize-dfa dfa cfg)]
    (loop []
      (if (.isEmpty worklist)
        (->DFADData (into {} in-table) (into {} out-table))
        (let [node (first worklist)
              prev-values (->> (node-parents cfg node) (map #(.get out-table %)))
              in-value (if (seq prev-values)
                          (reduce (partial dfa-join dfa) prev-values)
                          (.get in-table node))
              new-out-value (dfa-transfer dfa in-value node)]
          (.put in-table node in-value)
          (.remove worklist node)
          (when-not (dfa-equal? dfa (.get out-table node) new-out-value)
            (some->> (node-children cfg node) (.addAll worklist))
            (.put out-table node new-out-value))
          (recur))))))
```

Рисунок 4.45 – Функція обчислення аналізу потоку даних

Мета першого аналізу, простір імен якого зображений на рис. 4.46, полягає у збиранні множини вузлів з яких можна потрапити у конкретний вузол.

```
(ns string-analysis.dfa.previous-nodes
  (:require [string-analysis.dfa.core :refer :all]))
```

Рисунок 4.46 – Інтерфейс простору імен previous-nodes

На рис. 4.47 зображено оголошення класу даного аналізу, а також реалізацію методів.

```

(deftype PreviousNodesDFA [])

(defmethod dfa-bound PreviousNodesDFA [_]
  #{})

(defmethod dfa-transfer PreviousNodesDFA [_ in-value node]
  (conj in-value node))

(defmethod dfa-equal? PreviousNodesDFA [_ value1 value2]
  (= value1 value2))

(defmethod dfa-join PreviousNodesDFA [_ value1 value2]
  (clojure.set/union value1 value2))

```

Рисунок 4.47 – Клас і методи аналізу попередніх вузлів

Наступний простір імен, інтерфейс якого зображений на рис. 4.48, містить засоби необхідні для реалізації аналізу згортки констант.

```

(ns string-analysis.dfa.java.constant-propagation.common
  (:require [string-analysis.parser.matching :refer :all]
    [string-analysis.dfa.core :refer :all]))

```

Рисунок 4.48 – Інтерфейс простору імен common

На рис. 4.49 зображено код функції для розбору операцій присвоювання.

```

(defn extract-assignments
  ([term]
    (extract-assignments term operation-map))
  ([term lookup-operation-fn]
    (forthis term {:keys [identifier expression var-decls]}
      ((variableDeclarator "<identifier>")
        [[identifier]])
      ((:or (statement "<identifier> = <expression> ;")
            (variableDeclarator "<identifier> = <expression>"))
        [(cons identifier (expr->lambda expression lookup-operation-fn))])
      ((statement "<identifier> += <expression> ;")
        [(cons identifier (binop->lambda "+" lookup-operation-fn identifier expression))])
      ((localVariableDeclaration "<typeType> <var-decls:variableDeclarators>")
        (->> (rest var-decls)
          (partition-all 2)
          (mapcat
            (fn [[term _]]
              (extract-assignments term lookup-operation-fn)))))))

```

Рисунок 4.49 – Функція розбору операцій присвоювання

Дана функція приймає наступні параметри: дерево розбору; функція, що приймає операцію у вигляді рядку з дерева розбору, і повертає відповідну clojure функцію. В якості результату вона повертає список, де кожен елемент складається з трьох елементів: дерево розбору, що представляє змінну, яка отримує нове значення; clojure функцію, що приймає хеш-таблицю, де ключем є сутність (дерево розбору, що представляє або змінну або літерал), а значенням - відповідний clojure літерал; множина можливих сутностей. Для розбору виразів (тобто, виділення операцій з операндами) була створена допоміжна функція, код якої зображений на рис. 4.50. Вона повертає clojure функцію, що є аналогом виразу, який розбирається, і множину операндів (дерева розбору). На даний момент підтримується тільки операція додавання (тобто, об'єднання рядків), але в подальшому дану функцію можна з легкістю доповнити необхідними операціями.

```
(defn expr->lambda [expr lookup-operation-fn]
  (forthis expr {:keys [expr1 expr2 leaf] :as spec}
    ((:or (expression "<leaf:identifier>")
          (expression "<leaf:literal>")
          (identifier "<leaf:identifier>")
          (literal "<leaf:literal>")))
    [(fn [spec] (get spec leaf)) #{leaf}])
    ((expression "( <expr1:expression> )")
     (expr->lambda expr1 lookup-operation-fn))
    ((expression "<expr1:expression> + <expr2:expression>"
     (binop->lambda "+" lookup-operation-fn expr1 expr2))))
```

Рисунок 4.50 – Допоміжна функція для розбору виразів

На рис. 4.51 зображено код допоміжної функції, що призначена для створення функцій-відповідників. Вона має наступні параметри: операція з дерева розбору; функція, яка повертає відповідник операції; перший і другий аргументи операції з дерева розбору. Дана функція рекурсивно викликає функцію обробки виразів над першим і другим аргументом. У випадку, якщо їх розбір пройшов успішно, тобто, були отримані відповідники, то створюється новий відповідник, що приймає хеш-табличку, інтерфейс якої був описаний вище (у іншому випадку нічого не повертається). Після цього викликаються функції-відповідники

аргументів (в якості вхідних параметрів їм передається хеш-таблиця), і у разі якщо жодне з отриманих значень не являється невідомим, виконується операція, що була отримана за допомогою функції, що повертає відповідники (у іншому випадку повертається невідоме значення).

```
(defn binop->lambda [op lookup-operation-fn expr1 expr2]
  (when-let [op (lookup-operation-fn op)]
    (let [[f1 operands1] (expr->lambda expr1 lookup-operation-fn)
          [f2 operands2] (expr->lambda expr2 lookup-operation-fn)]
      (when (and f1 f2)
        [(fn [spec]
            (let [v1 (f1 spec) v2 (f2 spec)]
              (if (or (is-unknown? v1) (is-unknown? v2))
                  dfa-unknown-value
                  (op v1 v2))))
          (clojure.set/union operands1 operands2)]))))
```

Рисунок 4.51 – Функція для створення відповідників

Наступний простір імен, оголошення якого зображено на рис. 4.52, містить реалізацію запропонованого в даній дисертації методу.

```
(ns string-analysis.dfa.java.constant-propagation.modified
  (:require [string-analysis.dfa.core :refer :all]
            [string-analysis.cfg.core :refer :all]
            [string-analysis.parser.matching :refer :all]
            [string-analysis.dfa.java.constant-propagation.common :refer :all]
            [clojure.math.combinatorics :as math]))
```

Рисунок 4.52 – Інтерфейс простору імен modified

На рис. 4.53 зображено оголошення класу аналізу. Як можна помітити, він містить наступні поля: результат аналізу попередніх вузлів, адже на його основі робиться перевірка існування шляху між вузлами графу; інформаційна таблиця змінних, яка необхідна для визначення типу змінних при розборі операції присвоювання (цікавими для аналізу є тільки рядкові змінні).

```
(deftype ConstantPropagationDFA [prev-nodes-dfa id-table])
```

Рисунок 4.53 – Оголошення класу аналізу

На рис. 4.54 зображено реалізацію методів отримання початкового значення, порівняння і об'єднання результатів аналізу.

```
(defmethod dfa-bound ConstantPropagationDFA [_]
  {})

(defmethod dfa-equal? ConstantPropagationDFA [_ value1 value2]
  (= value1 value2))

(defmethod dfa-join ConstantPropagationDFA [_ value1 value2]
  (merge-with (partial merge-with clojure.set/union) value1 value2))
```

Рисунок 4.54 – Реалізація певних методів аналізу

На рис. 4.55 зображено код функції для отримання можливих значень операндів (сутностей). Компонент і їх список представлені множинами.

```
(defn- operand-values [operand node in-value]
  (forthis operand spec
    ((identifier "<identifier>")
      (get in-value (-> self meta :id) [[dfa-unknown-value #{#{node}}]]))
    ((:or (literal "<s:STRING_LITERAL>")
          (literal "<s:CHAR_LITERAL>"))
      (let [s (-> spec :s .getText)]
        [[(->> s count dec (subs s 1)) #{#{node}}]]))
    (true [[dfa-unknown-value #{#{node}}]])))
```

Рисунок 4.55 – Функція отримання значень операндів

На рис. 4.56 зображено код функції перевірки списку вузлів на сумісність.

```
(defn- are-nodes-compatible? [prev-nodes-table nodes]
  (loop [[curr & other] nodes]
    (or (empty? other)
        (and
          (every?
            (fn [node]
              (or (get-in prev-nodes-table [node curr])
                  (get-in prev-nodes-table [curr node])))
            other)
          (recur other)))))
```

Рисунок 4.56 – Функція перевірки вузлів на сумісність

Реалізація методу, що відповідає трансферній функції зображено на рис. 4.57.

```
(defmethod dfa-transfer ConstantPropagationDFA [dfa in-value node]
  (let [prev-nodes-table (-> dfa .prev-nodes-dfa .in-table)]
    (->
      (if (satisfies? TermNode node) (extract-assignments (node-term node)) [])
      (into in-value
        (map
          (fn [[target func operands]]
            (let [id (-> target meta :id)]
              (->
                (if (-> (.id-table dfa) (get id) :type is-string-type? (and func))
                  (-> (map #(operand-values % node in-value) operands)
                    (apply math/cartesian-product)
                    (keep (partial try-get-new-pair func operands prev-nodes-table))
                    (into {})))
                  {dfa-unknown-value #{#{node}}}))
                (vector id))))))))))
```

Рисунок 4.57 – Реалізація трансферної функції

На рис. 4.58 зображено код функції для отримання нової пари.

```
(defn- try-get-new-pair [func operands prev-nodes-table pairs]
  (when-let [components
    (-> (map second pairs)
      (apply math/cartesian-product)
      (map (partial apply clojure.set/union))
      (filter (partial are-nodes-compatible? prev-nodes-table))
      (seq))]
    [(func (zipmap operands (map first pairs))) (set components)]))
```

Рисунок 4.58 – Функція отримання нової пари

На рисунку 4.59 зображено оголошення простору імен, що містить реалізацію стандартного методу.

```
(ns string-analysis.dfa.java.constant-propagation.standard
  (:require [string-analysis.dfa.core :refer :all]
    [string-analysis.cfg.core :refer :all]
    [string-analysis.parser.matching :refer :all]
    [clojure.math.combinatorics :as math]
    [string-analysis.dfa.java.constant-propagation.common :refer :all]))
```

Рисунок 4.59 – Інтерфейс простору імен standard

На рис. 4.60 зображено оголошення класу аналізу (на відміну від класу модифікованого методу, він містить тільки інформаційну таблицю), а також частини методів аналізу (всіх окрім трансферної функції).

```
(deftype ConstantPropagationDFA [id-table])

(defmethod dfa-bound ConstantPropagationDFA [_]
  {})

(defmethod dfa-equal? ConstantPropagationDFA [_ value1 value2]
  (= value1 value2))

(defmethod dfa-join ConstantPropagationDFA [_ value1 value2]
  (merge-with clojure.set/union value1 value2))
```

Рисунок 4.60 – Клас аналізу з реалізацією частини методів

На рис. 4.61 зображено реалізацію методу, що відповідає трансферній функції, а також код допоміжної функції для отримання множини можливих значень операндів (сутностей).

```
(defn- operand-values [operand in-value]
  (forthis operand spec
    ((identifier "<identifier>")
      (get in-value (-> self meta :id) #{dfa-unknown-value}))
    ((:or (literal "<s:STRING_LITERAL>")
          (literal "<s:CHAR_LITERAL>"))
      (let [s (-> spec :s .getText)] #{(-> s count dec (subs s 1))}))
    (true #{dfa-unknown-value})))

(defmethod dfa-transfer ConstantPropagationDFA [dfa in-value node]
  (-> (if (satisfies? TermNode node) (extract-assignments (node-term node)) [])
    (into in-value
      (map
        (fn [[target func operands]]
          (let [id (-> target meta :id)]
            (->
              (if (-> dfa .id-table (get id) :type is-string-type? (and func))
                (-> (map #(operand-values % in-value) operands)
                  (apply math/cartesian-product)
                  (map (comp func (partial zipmap operands)))
                  (into #{}))
                #{dfa-unknown-value}))
              (vector id))))))))
```

Рисунок 4.61 – Реалізація трансферної функції

Висновки

В якості демонстрації роботи запропонованого методу було створено систему, що складається з наступних модулів:

- 1) antlr – модуль для автоматичної генерації інструментів лексичного і синтаксичного розбору програмного коду;
- 2) parser – модуль, що містить засоби для взаємодії зі згенерованими інструментами, а також інструменти зіставлення дерев розбору для пошуку певних шаблонів;
- 3) cfg – модуль побудови графу потоку виконання на основі дерева розбору;
- 4) dfa – модуль, що містить засоби для обчислення аналізу потоку даних, а також реалізацію стандартного і модифікованого методів статичного аналізу коду для рішення задачі згортки рядкових констант програм, що написані мовою Java.

Кожен модуль реалізує відповідний етап статичного аналізу коду, які були описані у другому розділі даної магістерської дисертації.

Варто зазначити, що в даній системі реалізовано низку інструментів, що дозволяють з легкістю додати підтримку нових мов програмування.

5. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЗАПРОПОНОВАНОГО МЕТОДУ

5.1 Опис інструментів для оцінки ефективності

Для оцінки ефективності запропонованого методу було створено простір імен, оголошення якого зображено на рис. 5.1. Даний простір імен містить засоби для генерації тестових програм з заданою кількістю розгалужень, а також інструменти для виміру часу і обробки результатів аналізу.

```
(ns string-analysis.constant-propagation
  (:require [me.raynes.fs :as fs]
            [clojure.java.io :as io]
            [clojure.data.csv :as csv]
            [string-analysis.parser.mysql :as mysql]
            [string-analysis.parser.java :as java]
            [string-analysis.cfg.core :refer :all]
            [string-analysis.cfg.java :refer :all]
            [string-analysis.dfa.previous-nodes :refer :all]
            [string-analysis.dfa.java.constant-propagation.modified :as cp-mod]
            [string-analysis.dfa.java.constant-propagation.standard :as cp-std]
            [string-analysis.dfa.core :refer :all]
            [string-analysis.utils :refer :all]))
```

Рисунок 5.1 – Інтерфейс простору імен засобів тестування

На рис. 5.2 зображено оголошення змінної, що містить шлях до директорії з тестовими програмами, а також функції, яка приймає кількість розгалужень у якості параметрів і повертає шлях до тестової програми.

```
(defonce programs-dir (->> (fs/temp-dir "programs") delay))

(defn program-path [branches]
  (java.io.File. @programs-dir (format "sql%s.txt" (str branches))))
```

Рисунок 5.2 - Директорія з файлами і функція для отримання шляху

На рис. 5.3 зображено оголошення змінної, що містить шаблон тестової програми суть якої полягає у формуванні різних SQL запитів в залежності від вхідного параметру b.

Дана програма містить оголошення трьох змінних:

- 1) query – містить кінцевий SQL запит, який, в якості початкового значення, представляє вибірку всіх рядків з певної таблиці;
- 2) orderBy – містить значення оператора сортування вибірки;
- 3) where – містить значення оператора фільтрації вибірки.

Наприкінці, значення змінних об'єднуються між собою, тим самим формуючи кінцевий запит. Місце вставки операторів, що відповідають зміні частин SQL запиту, що описані вище, рядок – “%s”.

```
(def program-template
  (clojure.string/join
    \newline
    ["String getQuery (int b) {"
     "String query = \"SELECT * FROM TBL \" , orderBy, where;"
     "%s"
     "query += where + orderBy + ';' ;"
     "return query; }"])))
```

Рисунок 5.3 – Шаблон тестової програми

На рис 5.4 зображено макрос для виміру часу виконання. Він повертає вектор, де перший елемент – результат виконання форм, а другий – час його отримання.

```
(defmacro exec-time [& body]
  `(let [start# (. System (nanoTime))]
    [(do ~@body) (/ (double (- (. System (nanoTime)) start#)) 1000000.0)]))
```

Рисунок 5.4 – Макрос для виміру часу

На рис. 5.5 зображено допоміжну функцію для створення тестових програм, що приймає функцію для генерації рядку для вставки у шаблон програми, а також максимальну кількість розгалужень.


```
(defn create-programs-aux [gen-branches-fn max-branches]
  (dotimes [i max-branches]
    (with-open [w (io/writer (program-path i))]
      (->> (range (inc i))
            (gen-branches-fn)
            (format program-template)
            (.write w))))))
```

Рисунок 5.5 – Допоміжна функція створення тестової програми

Функція генерації приймає на вхід послідовність чисел від 0 до i , де i змінюється у діапазоні від 0 до $\text{max-branches} - 1$, де max-branches – це максимальна кількість розгалужень.

На рис. 5.6 зображено функцію генерації програм першого типу, суть яких полягає у присвоюванні нових значень змінним, що відповідають операторам сортування і фільтрації, у різних гілках одного умовного оператора.

```
(defn create-programs-1 [max-branches]
  (create-programs-aux
    (fn [idx-coll]
      (->> idx-coll
            (mapv
              (fn [i]
                (format "if (b == %s) {
                        orderBy = \"ORDER BY ID%s\";
                        where = \"WHERE ID%s IS NOT NULL \"; }
                        else\n" i i i)))
              (#(conj % "{ orderBy = \"\"; where = \"\"; }")))
              (clojure.string/join "")))
    max-branches))
```

Рисунок 5.6 – Функція генерації програм першого типу

На рис. 5.7 зображено функцію генерації програм другого типу, суть яких полягає у присвоюванні нових значень змінним у різних умовних операторах.

```

(defn create-programs-2 [max-branches]
  (create-programs-aux
    (fn [idx-coll]
      (->> (mapcat
        (fn [i]
          [(format "if (b == %s) {"
            orderBy = \"ORDER BY ID%s\"; }\\n\" i i)
          (format "if (b == %s) {"
            where = \"WHERE ID%s IS NOT NULL \"; }\\n\" i i)])
        idx-coll)
      (clojure.string/join "")))
    max-branches))

```

Рисунок 5.7 – Функція генерації програм другого типу

На рис. 5.8 зображено функцію для виконання аналізу, що базується на положеннях стандартного методу.

```

(defn compute-standard-dfa [path]
  (let [[term id-table]
        (java/parse-with-convert :methodDeclaration path)
        cfg (build-cfg term)]
    (->> (compute-dfa (cp-std/->ConstantPropagationDFA id-table) cfg)
      (query-values id-table cfg))))

```

Рисунок 5.8 – Стандартний аналіз потоку даних

На рис. 5.9 зображено функцію для виконання аналізу, що базується на положеннях запропонованого модифікованого методу.

```

(defn compute-modified-dfa [path]
  (let [[term id-table]
        (java/parse-with-convert :methodDeclaration path)
        cfg (build-cfg term)
        prev-nodes-dfa (compute-dfa (->PreviousNodesDFA) cfg)]
    (->> (compute-dfa (cp-mod/->ConstantPropagationDFA prev-nodes-dfa id-table) cfg)
      (query-values id-table cfg)
      (keys))))

```

Рисунок 5.9 – Модифікований аналіз потоку даних

В якості результату, функції повертають значення змінної `query` в кінці виконання тестової програми (тобто, яке міститься у вхідній таблиці кінцевого вузла графу потоку виконання). Вміст даної функції зображено на рис. 5.10.

```
(defn query-values [id-table cfg dfa]
  (->> (get-bounds cfg)
        (second)
        (get (.in-table dfa))
        (some
         (fn [[id values]]
           (when (= (get-in id-table [id :name]) "query")
                 values))))))
```

Рисунок 5.10 – Функція отримання значень змінної `query`

На рис. 5.11 зображено функцію для виміру часу виконання аналізу, яка записує результат у csv файл, де перша колонка – кількість розгалужень, друга – відповідний час виконання аналізу, третя – результат аналізу (напр. кількість рядків). Вона приймає наступні параметри: шлях до csv файлу; функція аналізу, що приймає шлях тестової програми з певною кількістю розгалужень, максимальне значення яких відповідає останньому параметру функції виміру часу.

```
(defn measure-dfa-time [csv-path f branches]
  (with-open [w (io/writer csv-path)]
    (->> (range branches)
          (map
           (fn [i]
             (let [[res time] (->> (program-path i) (f) (exec-time))]
               (vector i time res))))
          (csv/write-csv w))))
```

Рисунок 5.11 – Функція виміру часу аналізу

На рис. 5.12 зображено функція суть якої полягає у розборі SQL запитів, що містять у множині можливих значень аналізу потоку даних. Попередньо з даної множини видаляються невідомі значення аналізу.

```
(defn parse-queries [data]
  (->> data
    (remove is-unknown?)
    (mapl #(mysql/parse :sqlStatement %))))
```

Рисунок 5.12 – Функція синтаксичного розбору кінцевих SQL запитів

5.2 Порівняльна характеристика отриманих результатів

На рис. 5.13 зображено графік, який показує залежність кількості рядків, що можуть бути отримані для змінної query перед закінченням програми, від кількості розгалужень для програм першого типу. Цей графік показує, що така залежність для запропонованого модифікованого методу є лінійною, в той час як для стандартної реалізації методу ця залежність є квадратичною. Такий результат вдалося досягти шляхом ігнорування тих комбінацій рядків, які складаються зі значень, що були отримані у різних гілках умовного оператора програми, але які ніколи не з'являться при реальному виконанні програми [7].

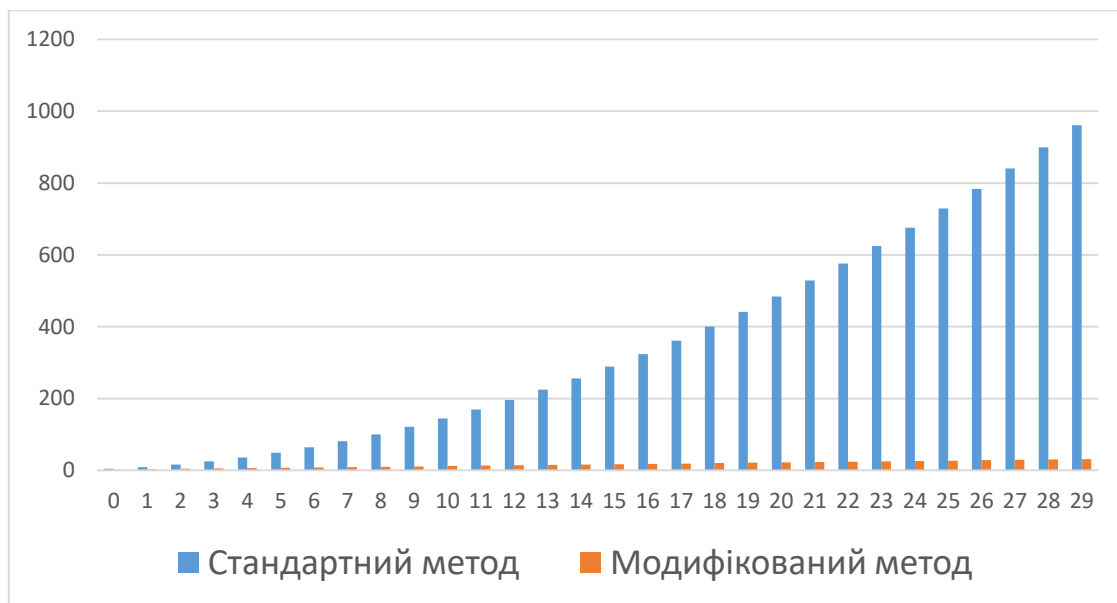


Рисунок 5.13 – Залежність кількості рядків від кількості розгалужень

Графік залежності часу роботи (в мілісекундах) процесу аналізу програм першого типу від кількості розгалужень зображено на рис.5.14. Неважко помітити, що залежність як для стандартного, так і для модифікованого методу є лінійною.

Отже, можна зробити висновок, що додатковий час, який витрачається на пошук сумісних компонентів, компенсується зменшенням кількості операцій об'єднання рядків за рахунок зменшення розміру множин можливих значень операндів [7].

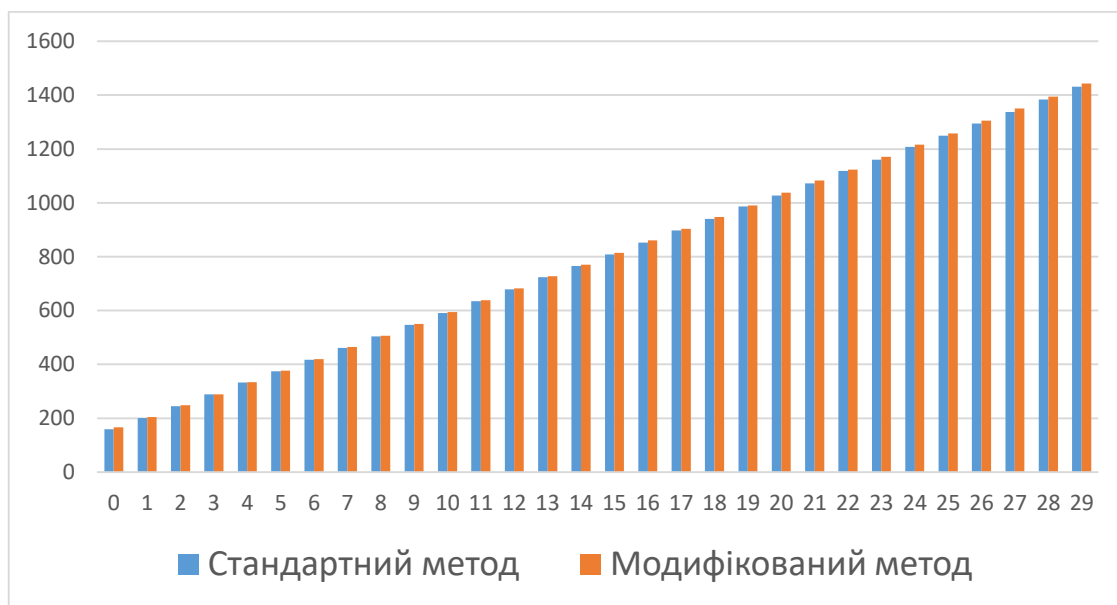


Рисунок 5.14 – Залежність часу виконання (мс) процесу аналізу від кількості розгалужень

На рис. 5.15 показано графік залежності часу роботи (в мілісекундах) процесу аналізу програм другого типу від кількості розгалужень. Особливістю цих програм є те, що присвоювання різних частин SQL запиту відбувається в різних умовних операторах. Через дану особливість, всі значення змінних `where` і `orderBy` об'єднуються між собою, адже компоненти якими вони були анотовані, вважаються сумісними через те, що існує шлях у графу потоку виконання, бо вузли в яких ці значення були отримані, не знаходяться у контексті спільного умовного оператора, як у програм першого типу. Наслідком цього є те, що не відбувається зменшення розміру множини значень модифіковано методу і додатково до цього відбувається додаткова перевірка компонентів на сумісність, що викликає збільшення часу виконання аналізу на 5-6%.

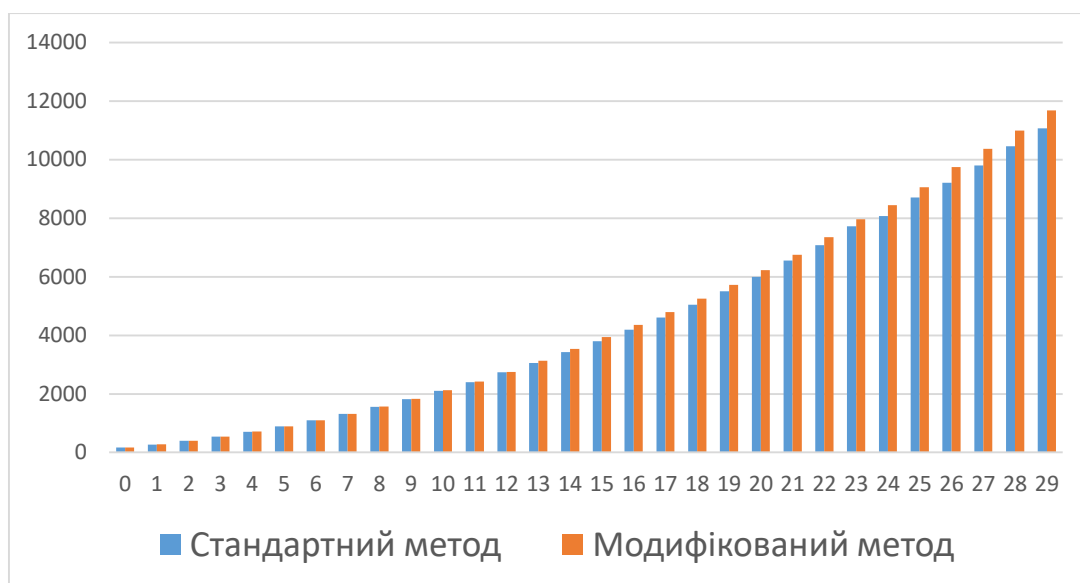


Рисунок 5.15 – Залежність часу виконання (мс) аналізу від кількості розгалужень

На рис.5.16 зображено графік залежності часу роботи (в мілісекундах) аналізу програм першого типу з обробкою його результатів від кількості розгалужень (рис.5.16). Тобто, за рахунок зменшення розміру кінцевої множини SQL запитів, вдалося зекономити час, що витрачався на розбір тих рядків, виникнення яких неможливе під час реального виконання програми [7].

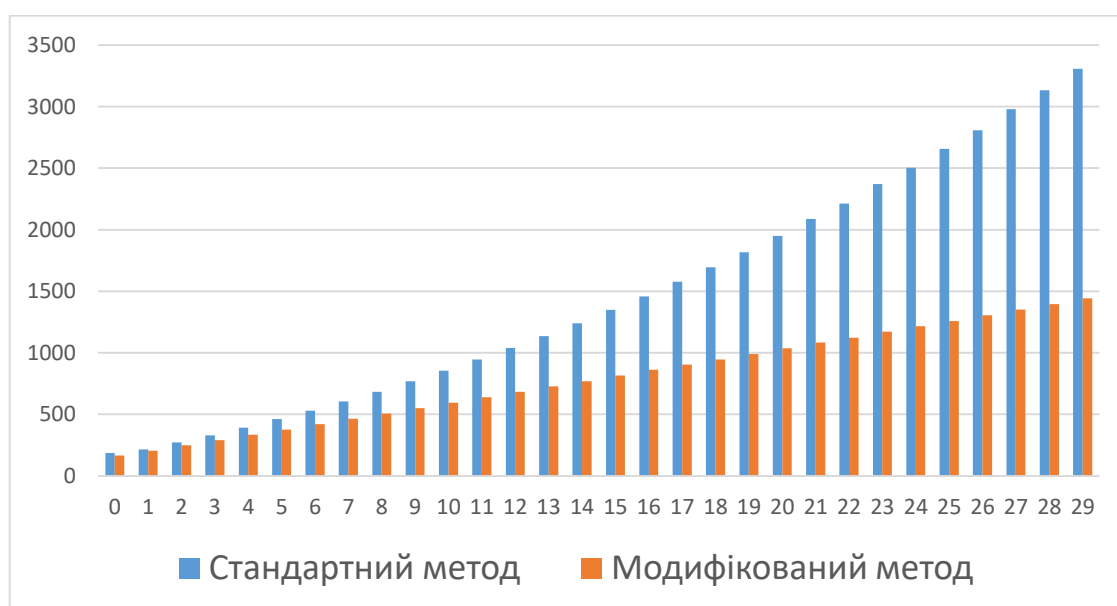


Рисунок 5.16 – Залежність часу виконання (мс) аналізу з обробкою його результату від кількості розгалужень

Висновки

Підсумовуючи, можна сказати, що запропонований метод для рішення задачі згортки рядкових констант показав свою ефективність за рахунок зменшення складності існуючих алгоритмів, яке було отримане завдяки ігноруванню операцій об'єднання тих значень операндів, що не будуть створенні під час реального виконання програми. Додаткова логіка запропонованого методу для пошуку сумісних компонентів не погіршила швидкодію порівняно з існуючими методами. Це було отримано за рахунок економії часу, який витрачається існуючими методами на велику кількість операцій об'єднання. Також, зменшення розміру кінцевої множини рядків дозволяє зекономити час, який витрачається на подальшу обробку результатів аналізу.

Додатково до цього було проаналізовано ефективність на програмах в яких не відбувається зменшення проміжних результатів (іншими словами, поведінка модифікованого методу аналогічна стандартному). В даному випадку швидкодія алгоритму, що базується на положеннях запропонованого методу приблизно на 5-6% гірша в порівнянні зі стандартним.

ВИСНОВКИ

В даній магістерській дисертації було здійснено дослідження процесу згортки рядкових констант у програмі.

Метою дослідження було розроблення більш швидкого методу статичного аналізу коду для вирішення задачі згортки рядкових констант.

В ході виконання роботи було виконано аналіз існуючих рішень і зроблено висновок, що існуючі методи аналізу рядків мають наступні недоліки:

- 1) направленість на вузькі домени;
- 2) можливий комбінаторний вибух складності аналізу при великій кількості розгалужень.

Наступним етапом був аналіз моделей статичного аналізу коду і був обраний підхід, який складається з наступних етапів:

- 1) перетворення вхідного коду в синтаксичне дерево згідно граматики мови, якою була написана ця програма (парсинг програмного коду), за допомогою автоматично згенерованих інструментів синтаксичного розбору;
- 2) створення графу потоку виконання на основі синтаксичного дерева;
- 3) обчислення інформації про потік даних на основі графу виконання використовуючи ітеративний підхід MFP.

В результаті було створено модифікований метод суть якого полягає у анотуванні пропагованих значень додатковою інформацією, аналіз якої дозволяє не здійснювати конкатенацію значень, які не можуть існувати разом під час реального виконання програми. Він відрізняється від існуючих використанням вкладеної хеш-таблиці в якості значень основної хеш-таблиці замість структури даних типу «множина» та модифікованої трансферної функції і дозволяє зменшити розмір проміжних результатів аналізу потоку даних, завдяки чому зменшується складність алгоритмів, що використовуються, тим самим покращуючи швидкість і точність згортки рядкових констант програми.

Наступним етапом було створено систему статичного аналізу коду. Вона була реалізована за допомогою мови Clojure і містить:

- 1) інструменти для обробки дерев розбору, що були отримані за допомогою згенерованих засобів ANTLR інструментом;
- 2) засоби для пошуку шаблонів у деревах розбору;
- 3) структуру даних графу потоку виконання, а також методи для взаємодії з ним;
- 4) реалізацію розробленого методу для рішення задачі згортки рядкових констант у програмах, що написані мовою Java.

На основі програмної реалізації було здійснено порівняльний аналіз розробленого методу з існуючими. Запропонований метод показав свою ефективність за рахунок зменшення складності існуючих алгоритмів, яке було отримане завдяки ігноруванню операцій об'єднання тих значень операндів, що не будуть створенні під час реального виконання програми. Додаткова логіка запропонованого методу для пошуку сумісних компонентів не погіршила швидкодію порівняно з існуючими методами. Це було отримано за рахунок економії часу, який витрачається існуючими методами на велику кількість операцій об'єднання. Також, зменшення розміру кінцевої множини рядків дозволяє зекономити час, який витрачається на подальшу обробку результатів аналізу.

Додатково до цього було проаналізовано ефективність на програмах в яких не відбувається зменшення проміжних результатів (іншими словами, поведінка модифікованого методу аналогічна стандартному). В даному випадку швидкодія алгоритму, що базується на положеннях запропонованого методу приблизно на 5-6% гірша в порівнянні зі стандартним.

Варто зазначити, що реалізовані інструменти дозволяють з легкістю додати підтримку нових мов програмування у розроблену систему статичного аналізу.

Напрямами подальших досліджень можуть бути:

- 1) розширення області застосування запропонованого методу для рішення задачі згортки не тільки рядкових, а також числових та інших видів констант;

- 2) вдосконалення логіки перевірки значень на сумісність, наприклад, застосуванням аналізу предикатів

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Costantini G., Ferrara P., Cortesi A. Static Analysis of String Values. 13th International Conference on Formal Engineering Methods, м. Дарем, 26–28 жовт. 2011 р. С. 505–521. URL: https://doi.org/10.1007/978-3-642-24559-6_34 (дата звернення: 20.11.2020).
2. Марченко О.І. Конспект лекцій з дисципліни «Інженерія програмного забезпечення-1. Основи проектування трансляторів», Київ 2013 (дата звернення: 16.05.2021)
3. ANTLR. URL: <https://www.antlr.org/> (дата звернення: 02.06.2021)
4. Cooper K. D., Torczon L. Engineering a Compiler. 3-тє вид. Elsevier, 2023. 820 с. URL: <https://doi.org/10.1016/c2014-0-01395-0> (дата звернення: 02.06.2021)
5. Beyer D., Gulwani S., Schmidt D. A. Combining Model Checking and Data-Flow Analysis. Handbook of Model Checking. Cham, 2018. С. 493–540. URL: https://doi.org/10.1007/978-3-319-10575-8_16 (дата звернення: 03.07.2021).
6. Касянчук Д.П., Марченко О.І. Модифікований метод статичного аналізу коду для рішення задачі згортки рядкових констант. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2023. № 50. С. 57–64. URL: <https://doi.org/10.36910/6775-2524-0560-2023-50-08>
7. Касянчук Д. П., Марченко О. І. Порівняльний аналіз ефективності методів статичного аналізу коду для рішення задачі згортки рядкових констант. Scientific Collection «InterConf». 2023. № 152. С. 568–573. URL: <https://archive.interconf.center/index.php/conference-proceeding/article/view/3218>

ДОДАТОК 1. Презентація

ДОДАТОК 2. Лістинг програми

ДОДАТОК 3. Копії публікацій