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

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.415.2

«До захисту допущено»
Науковий керівник кафедри
_____ І.А. Дичка
«___»_____2019 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 121 Інженерія програмного забезпечення

на тему: «Спосіб оптимізації хвостових викликів функцій для Node.js на основі CPS-форми внутрішнього представлення»

Виконав:
студент ІІ курсу, групи КП-81мп
Стилик Роман Григорович _____

Керівник:
Доцент кафедри СПіСКС, к.т.н., доцент,
Марченко О.І. _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай М.В. _____

Рецензент:
С.н.с. кафедри ОТ, д.т.н., проф.
Сергієнко А.М. _____

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

Київ – 2019 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення» («Програмне забезпечення комп'ютерних та інформаційно-пошукових систем»)

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ І.А. Дичка

«___»_____ 2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Стилику Роману Григоровичу

1. Тема дисертації «Спосіб оптимізації хвостових викликів функцій для Node.js на основі CPS-форми внутрішнього представлення», науковий керівник дисертації доцент кафедри СПіСКС, к.т.н., Марченко Олександр Іванович затверджені наказом по університету від «13» листопада 2019 р. № 3895-с

2. Термін подання студентом дисертації «18» грудня 2019 р.

3. Об'єкт дослідження: процес оптимізації коду програм для платформи Node.js.

4. Предмет дослідження: способи оптимізації хвостових викликів програм для платформи Node.js.

5. Перелік завдань, які потрібно розробити:
   - провести аналіз методів оптимізації хвостових викликів для платформи Node.js;
   - дослідити роботу існуючих методів та алгоритмів побудови CPS-форми внутрішнього представлення;
   - запропонувати та обґрунтувати шляхи модифікації способу оптимізації хвостових викликів;
   - розробити метод оптимізації хвостових викликів функцій для платформи Node.js з використанням CPS-форми внутрішнього представлення;
   - виконати програмну реалізацію розробленого методу;
   - виконати порівняння запропонованого методу із існуючими аналогами та надати оцінку отриманих результатів.

6. Орієнтовний перелік публікацій:
   - XII наукова конференція магістрантів та аспірантів «Прикладна математика та комп'ютинг» (ПМК-2019);

7. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., к.т.н., доцент		

8. Дата видачі завдання «15» грудня 2018 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	17.02.2019	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.03.2019	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	16.04.2019	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення	14.05.2019	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження	17.09.2019	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації; підготовка матеріалів доповіді на конференції ПМК-2019.	02.10.2019	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу;	28.10.2019	
8.	Оформлення текстової і графічної частини магістерської дисертації	10.11.2019	

Студент Р.Г. Стилик

Науковий керівник дисертації О.І. Марченко

РЕФЕРАТ

Актуальність теми. Зростаюча популярність функціонального стилю програмування під час розробки програмного забезпечення мовою JavaScript обумовила необхідність використання рекурсивних функцій. Через особливості еволюції мови JavaScript, а саме її поступового перетворення із дуже обмеженої суто скриптової мови web-браузера на мову програмування загального використання, тобто таку, що може використовуватися для написання багатоплатформеного програмного забезпечення будь-якої складності, стандарт мови ECMAScript створювався здебільшого шляхом суперечок представників провідних компаній-розробників, що використовували JavaScript. Станом на сьогодні це призвело до наступних проблем:

1. В стандарті закріплені ранні помилки дизайну мови, які неможливо виправити через проблеми зворотної сумісності;
2. Розробники рушіїв ставляться до стандарту як до набору рекомендацій і не виконують всіх його вимог;
3. Рушії та середовища виконання пропонують різноманітний функціонал, що взагалі не описаний в стандарті.

Наявність цих проблем впливає на використання рекурсивних функцій наступним чином: в додатках для різних браузерів, мобільних платформ та серверної частини, що, здавалося б, використовують мову одного стандарту, хвостові виклики функцій працюють по-різному, що значно ускладнює розробку таких додатків. Так, наприклад, в єдиній серверній платформі мови JavaScript Node.js взагалі відсутня будь-яка оптимізація хвостових викликів, що призводить до того, що розробники іноді взагалі змушені відмовлятись від рекурсивного підходу на користь ітеративного. Виходячи з цього, створення способу оптимізації хвостових викликів функцій для Node.js є актуальною задачею як з наукової, так і практичної точки зору.

Об’єктом дослідження є процес оптимізації коду програм для платформи Node.js.

Предметом дослідження є способи оптимізації хвостових викликів програм для платформи Node.js.

Мета роботи: створення алгоритму побудови CPS-форми внутрішнього представлення для вхідного коду мовою JavaScript та застосування даної форми для розробки способу оптимізації хвостових викликів програм для платформи Node.js.

Методи дослідження. В роботі використовуються методи статичного аналізу коду, методи трансляції коду, методи оптимізації.

Наукова новизна роботи полягає в наступному:

1. Запропоновано алгоритм побудови CPS-форми внутрішнього представлення для програм мови JavaScript стандарту ES6, який відрізняється від наявних реалізацій побудови CPS-форми можливістю як міжпроцедурної, так і часткової побудови, а також незалежністю від цільової платформи вхідного JavaScript коду.
2. Запропоновано спосіб оптимізації хвостових викликів функцій на основі побудованої CPS-форми, який відрізняється від існуючих збереженням контролю над стеком виконання, що в свою чергу дозволяє використання точок зупинки і покрокового виконання оптимізованої програми.
3. Виконано порівняльний аналіз запропонованого способу з існуючими для вхідного коду мови JavaScript і наведено приклади використання, за яких отриманий спосіб демонструє розширені можливості роботи з отриманим кодом. Проаналізовано вплив проведеної оптимізації на швидкодію та використання ресурсів результуючої програми.

Практична цінність отриманих в роботі результатів полягає в тому, що розроблений спосіб оптимізації хвостових викликів, окрім безпосереднього вирішення проблеми виснаження стеку викликів, дозволяє подальше покрокове налагодження отриманої програми стандартними засобами платформи Node.js. Важливою є можливість застосування перетворення як для окремих функцій мови, так і для модулів, що спрощує подальше використання і інтеграцію отриманого

коду у вже існуючі програмні продукти. Крім того, існує можливість окремого використання побудованої CPS-форми без застосування оптимізації хвостових викликів у вигляді програмного коду для певних цілей розробника.

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

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень та показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі наведено загальний огляд алгоритмів отримання форм внутрішнього представлення вхідного коду та методів оптимізації, виконано оцінку переваг та недоліків існуючих способів оптимізації хвостових викликів.

У другому розділі наведено теоретичні засади способів побудови CPS-форми внутрішнього представлення та проведення оптимізації хвостових викликів функцій з урахуванням необхідності подальшої роботи зі стеком викликів. Запропоновано спосіб оптимізації хвостових викликів на основі CPS-форми внутрішнього представлення.

У третьому розділі наведено особливості реалізації розробленої системи.

У четвертому розділі представлено підходи до порівняльного аналізу отриманого коду та тестування системи, а також виконаний власне порівняльний аналіз розробленої системи на основі запропонованого способу з існуючими.

У п’ятому розділі представлено розроблений на основі запропонованого способу стартап-проект, наведено бізнес модель.

У висновках проаналізовано отримані результати роботи.

Робота виконана на 80 аркушах, містить 2 додатки та посилання на список використаних літературних джерел з 17 найменувань. У роботі наведено 10 рисунків і 5 таблиць.

Ключові слова: хвостові виклики, рекурсія, метод оптимізації, трансляція, JavaScript, Node.js.

ABSTRACT

Topicality. The increasing popularity of functional programming style in JavaScript software development necessitated the use of recursive functions. Due to the peculiarities of the evolution of JavaScript, namely its gradual transformation from a very limited purely web-based scripting language to a general-purpose programming language, that is, which can be used to write multi-platform software of any complexity, the ECMAScript standard has been created largely through disputes of leading JavaScript companies' developers. As of today, this has led to the following problems:

1. The standard enshrines early design errors that cannot be corrected because of backward compatibility issues;
2. Engine developers refer to the standard as a set of recommendations and do not meet all its requirements;
3. Engines and runtime offer a variety of functionality not described in the standard at all.

The presence of these problems affects the use of recursive functions in the following way: in applications for different browsers, mobile platforms and server side, which seemingly use the same standard language, tail functions function differently, which significantly complicates the development of such applications. For example, Node.js' single-server JavaScript platform does not have any tail-call optimization at all, which means that developers sometimes have to abandon the recursive approach in favor of an iterative one. With this in mind, creating a way to optimize the tail of function calls for Node.js is a pressing task, both scientifically and practically.

The object of research is a code optimization process for the Node.js platform.

The subject of the study there are ways to optimize the tail of program calls for the Node.js platform.

The aim of the study creating an algorithm for constructing a CPS internal representation form for JavaScript input and using this form to develop a method for optimizing tailing applications for the Node.js platform.

Research methods. The methods of static code analysis, methods of code translation, optimization methods are used in the work.

The scientific novelty of the work is as follows:

1. An algorithm for constructing the CPS-form of internal representation for ES6 JavaScript programs is offered, which differs from the existing implementations of building the CPS-form by the possibility of both interprocedural and partial construction, as well as independence of the target platform of the input JavaScript code.
2. A method of optimization of tail calls of functions on the basis of the constructed CPS-form is offered, which differs from the existing ones maintaining control over the execution stack, which in turn allows the use of breakpoints and step-by-step execution of the optimized program.
3. A comparative analysis of the proposed method with the existing ones for JavaScript code input is given, and examples of use in which the obtained method demonstrates the enhanced capabilities of working with the obtained code. The influence of the conducted optimization on the performance and use of the resources of the resulting program is analyzed.

The practical value of the results obtained is that the developed method of optimization of tail calls, in addition to the direct solution of the problem of exhaustion of the call stack, allows further step-by-step debugging of the received program by standard means of the Node.js platform. Important is the ability to apply the conversion for both individual language functions and modules, which simplifies the subsequent use and integration of the resulting code into existing software products. In addition, it is possible to use the built-in CPS form individually without using tail-call optimization in the form of code for specific developer purposes.

Structure and scope of work. The master's thesis consists of an introduction, five sections, conclusions and appendices.

The introduction gives a general description of the work, assesses the current state of the problem, substantiates the relevance of the research direction, formulates the purpose and objectives of the research and shows the scientific novelty of the obtained results and practical value of the work.

The first section provides an overview of the algorithms for obtaining internal source code forms and optimization methods, evaluating the advantages and disadvantages of existing tail-call optimization methods.

The second section describes the theoretical foundations of how to build a CPS form of internal representation and optimize the tail function calls, taking into account the need for further work with the call stack. A method for optimizing tail calls based on the CPS form of the internal representation is proposed.

The third section describes the implementation features of the developed system.

The fourth section presents approaches to benchmarking the resulting code and testing the system, as well as performing a comparative analysis of the developed system on the basis of the proposed method with the existing ones.

The fifth section presents a startup project based on the proposed method, and presents a business model.

The results of the work are analyzed in the conclusions.

The work is done on 80 sheets, contains 2 appendices and references to the list of used literature sources, total 17. The work presents 10 figures and 5 tables.

Keywords: tail calls, recursion, optimization method, translation, JavaScript, Node.js.

РЕФЕРАТ

Актуальность темы. Растущая популярность функционального стиля программирования при разработке программного обеспечения на языке JavaScript обусловила необходимость использования рекурсивных функций. Из-за особенностей эволюции языка JavaScript, а именно его постепенного преобразования из очень ограниченного чисто скриптового языка web-браузера в язык программирования общего назначения, то есть такого, который может использоваться для написания кроссплатформенного программного обеспечения любой сложности, стандарт языка ECMAScript создавался в основном путем споров представителей ведущих компаний-разработчиков, использующих JavaScript. На сегодня это привело к следующим проблемам:

1. В стандарте закреплены ранние ошибки дизайна языка, которые невозможно исправить из-за проблем обратной совместимости;
2. Разработчики компиляторов относятся к стандарту как к набору рекомендаций и не выполняют всех его требований;
3. Интерпретаторы и среды выполнения предлагают широкий функционал, который не описан в стандарте.

Наличие этих проблем влияет на использование рекурсивных функций следующим образом: в приложениях для различных браузеров, мобильных платформ и серверной части, которые, казалось бы, используют язык одного стандарта, хвостовые вызовы функций работают по-разному, что значительно усложняет разработку таких приложений. Так, например, в единственной серверной платформе языка JavaScript Node.js вообще отсутствует какая-либо оптимизация хвостовых вызовов, что приводит к тому, что разработчики иногда вообще вынуждены отказываться от рекурсивного подхода в пользу итеративного. Исходя из этого, создание способа оптимизации хвостовых вызовов функций для Node.js является актуальной задачей как с научной, так и практической точки зрения.

Объектом исследования является процесс оптимизации кода программ для платформы Node.js.

Предметом исследования являются способы оптимизации хвостовых вызовов программ для платформы Node.js.

Цель работы: создание алгоритма построения CPS-формы внутреннего представления для исходного кода на языке JavaScript и применение данной формы для разработки способа оптимизации хвостовых вызовов программ для платформы Node.js.

Методы исследования. В работе используются методы статического анализа кода, методы трансляции кода, методы оптимизации.

Научная новизна работы заключается в следующем:

1. Предложен алгоритм построения CPS-формы внутреннего представления для программ языка JavaScript стандарта ES6, который отличается от имеющихся реализаций построения CPS-формы возможностью как межпроцедурного, так и частичного построения, а также независимостью от целевой платформы входящего JavaScript кода.
2. Предложен способ оптимизации хвостовых вызовов функций на основе построенной CPS-формы, который отличается от существующих сохранением контроля над стеком вызовов, что в свою очередь позволяет использование точек остановки и пошагового выполнения оптимизированной программы.
3. Выполнен сравнительный анализ предложенного способа с существующими для входного кода языка JavaScript и приведены примеры использования, при которых полученный способ демонстрирует расширенные возможности работы с полученным кодом. Проанализировано влияние проведенной оптимизации на быстродействие и использования ресурсов результирующей программы.

Практическая ценность полученных в работе результатов заключается в том, что разработанный способ оптимизации хвостовых вызовов, кроме непосредственного решения проблемы переполнения стека вызовов, позволяет дальнейшую пошаговую отладку полученной программы стандартными средствами платформы Node.js. Важной является возможность применения преобразования как для отдельных функций языка, так и для модулей, что упрощает дальнейшее использование и интеграцию полученного кода в уже существующие программные продукты. Кроме того, существует возможность отдельного использования построенной CPS-формы без применения оптимизации хвостовых вызовов в виде программного кода для определенных целей разработчика.

Структура и объём работы. Магистерская диссертация состоит из введения, пяти глав, заключения и приложений.

Во введении дана общая характеристика работы, выполнена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи исследований, показана научная новизна полученных результатов и практическая ценность работы.

В первой главе приведен общий обзор алгоритмов получения форм внутреннего представления входного кода и методов оптимизации, выполнена оценка преимуществ и недостатков существующих способов оптимизации хвостовых вызовов.

Во втором разделе приведены теоретические основы способов построения CPS-формы внутреннего представления и проведения оптимизации хвостовых вызовов функций с учетом необходимости дальнейшей работы со стеком вызовов. Предложен способ оптимизации хвостовых вызовов на основе CPS-формы внутреннего представления.

В третьем разделе приведены особенности реализации разработанной системы.

В четвертом разделе представлены подходы к сравнительному анализу полученного кода и тестирования системы, а также выполнен собственно

сравнительный анализ разработанной системы на основе предложенного способа с существующими.

В пятом разделе представлен разработанный на основе предложенного способа стартап-проект, приведена бизнес модель.

В выводах проанализированы полученные результаты работы.

Работа выполнена на 80 листах, содержит 2 приложения и ссылки на список использованных литературных источников из 17 наименований. В работе приведены 10 рисунков и 5 таблиц.

Ключевые слова: хвостовые вызовы, рекурсия, метод оптимизации, трансляция, JavaScript, Node.js.

ЗМІСТ

ВСТУП

Однією з найбільш широковживаних парадигм програмування на сьогодні є функціональне програмування, що засноване на базі математичної абстракції лямбда-обчислення.

Властивостями функціональних мов програмування є використання функцій в якості основних структурних компонентів побудови програм, підтримка функцій в якості об’єктів мови першого класу, наявність чистих функцій та незмінюваних структур даних. Найважливішою технікою для програмування у функціональному стилі є рекурсія, тобто виклики процедур чи функцій з них самих безпосередньо або за допомогою інших функцій. Використання рекурсії пов’язане з поняттям глибини рекурсії, тобто кількості вкладених викликів функції або процедури. Глибина рекурсії зазвичай обмежена розміром стеку викликів програми – структури даних, в яку додається інформація про функції, що викликаються. Рушії мов, що мають підтримку рекурсії, повинні також пропонувати певні способи оптимізації рекурсивних викликів, адже за її відсутності при певній глибині рекурсії може виникнути помилка переповнення стеку викликів.

Мова JavaScript, яка є однією з найпопулярніших на сьогодні мов програмування загального призначення, підтримує функціональний стиль розробки програм. Незважаючи на те, що підтримка таких елементів функціонального стилю, як лямбда-функцій, функцій вищого порядку та рекурсії, на разі жодне середовище виконання JavaScript не пропонує оптимізації хвостових викликів програм, через що можливості для використання рекурсивних функцій певним чином обмежуються.

Метою роботи є реалізація системи, що виконуватиме оптимізацію хвостових викликів функцій мови JavaScript на основі CPS-форми внутрішнього представлення коду.

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

CPS (Continuation Passing Style) – форма внутрішнього представлення вхідного коду, в якій передача контролю виконання проводиться за допомогою продовжень.

TCO (Tail Call Optimization) – загальна назва сімейства способів оптимізації хвостових викликів функцій.

Continuation (продовження) – представлення стану програми в певний момент часу, який є доступним для переходу в нього. В контексті теми доповіді продовження реалізуються за допомогою функцій вищого порядку.

Tail Call (хвостовий виклик) – виклик процедури/функції в якості останньої дії поточної процедури. Виклик тієї самої процедури називається хвостовою рекурсією.

1. АНАЛІЗ СПОСОБІВ ПОБУДОВИ ФОРМ ВНУТРІШНЬОГО ПРЕДСТАВЛЕННЯ ТА ОПТИМІЗАЦІЇ ХВОСТОВИХ ВИКЛИКІВ

1.1. Загальний огляд способів отримання форм внутрішнього представлення вхідного коду та компіляторних оптимізацій

Під час історичного розвитку інформаційних технологій і програмування як такого обсяги інформації, що оброблюється, невпинно зростали. Так, на початку комп'ютерної епохи електронні обчислювальні машини могли «розуміти» лише команди у вигляді нулів та одиниць, тобто послідовностей перепадів напруги. Програмістам того часу необхідно було писати програми у машинному коді, які одразу завантажувались в пам'ять і виконувались. Дуже швидко стало зрозуміло, що людині дуже важко взаємодіяти з комп'ютером в такому форматі, адже окрім виняткової уваги під час написання машинного коду було також необхідно дуже добре знати внутрішню структуру кожного обчислювального блоку. Налагоджувати такі програми або якось контролювати хід їхнього виконання було також дуже нетривіальною задачею. Крім того, такі програми були залежними від середовища їх виконання, тобто для кожної обчислювальної машини треба було писати свою програму.

Вирішенням цих проблем стала автоматизація формування машинного коду. Для написання програм було створено мнемонічну мову – мову асемблеру. Ця мова вже надавала певний рівень абстракції під час написання коду за допомогою використання символьних ідентифікаторів, які позначали двійкові коди, команди, адреси в пам'яті тощо. Однією з найважливіших абстракцій, що з'явилася в мові програмування, була змінна. Разом із цим асемблер вже не був повністю залежним від конкретної обчислювальної машини – він залежав і залежить від архітектури процесора, але навіть для

різних процесорних архітектур відповідні асемблери можуть різнитися лише певними наборами команд. Також асемблер дозволяв писати програми значно меншого розміру, що в ті часи було дуже важливим критерієм ефективності.

З часом обсяги оброблюваних даних ще стрімкіше зростали, і з'явилась потреба у великих, складних програмних комплексах, які було складно розроблювати лише на асемблері, що призвело до появи мов програмування високого рівня, які вже були машинно-незалежні, надавали потужні набори абстракцій та були відносно простими у використанні (наближені до натуральних мов). Важливою особливістю цих мов є орієнтування на алгоритми. Саме з такими мовами сьогодні працює переважна більшість програмістів.

З подальшим розвитком мов програмування рівень і складність абстракцій, що надаються програмістам, тільки зростав. З'являлись різні парадигми програмування, як-то функціональна, об'єктно-орієнтовна, процедурна, тощо. Дуже важливою стала багатоплатформність, тобто змога виконувати один і той самий високорівневий код на якомога більшій кількості різновидів апаратних платформ. Все це призвело до ускладнення процесу перетворення вхідного коду і появи великої кількості різноманітних трансляторів: компіляторів/декомпіляторів, інтерпретаторів, препроцесорів та ін.

Важливим компонентом будь-якого компілятора є оптимізатор проміжного коду. Потреба в такому модулі пояснюється тим, що через дуже високий рівень абстракції вхідного коду еквівалента програма в коді низького рівня може бути дуже неефективною з точки зору швидкодії та використання ресурсів, а написання такої програми вручну з використанням коду низького рівня є вкрай важкою задачею. Загалом, оптимізуючий транслятор приймає на вхід семантичне дерево програми і за допомогою певних способів роботи з

деревами/графами конструює певне нове внутрішнє представлення програми, оптимізоване за певним критерієм. Відповідно, використання одразу кількох різних оптимізаторів, кожні з яких засновані на різних формах внутрішнього представлення вхідного коду, дозволяє отримати на виході більш ефективну з точки зору використання ресурсів та швидкодії програму, ніж написану спочатку.

Реалізація більшості способів оптимізації вхідного коду може потребувати побудови певних форм внутрішнього представлення. Головними властивостями будь-якої форми внутрішнього представлення мають бути точність, тобто можливість репрезентувати вхідний код без втрат інформації, та незалежність від цільової мови компіляції або трансляції. Форми внутрішнього представлення можуть існувати в різному форматі, від цілком внутрішніх структур даних, як-то список, кортеж, граф, тощо, до цілком придатного навіть для людського сприйняття коду мови внутрішнього представлення (intermediate language). Типовими прикладами використання форм внутрішнього представлення можуть слугувати численні сучасні компілятори, в яких програмний код в лінійній текстовій формі перетворюється в проміжну графову структуру, яка дозволяє виконати аналіз потоку виконання та здійснити низку перетворень перед власне генерацією низькорівневого коду. Подальше використання таких форм проміжного представлення робить можливим створення систем компіляторів, які генерують код для певної множини різних цільових архітектур з вхідного коду різних мов високого рівня.

Отримання будь-якої форми внутрішнього представлення відбувається на чітко визначеному етапі компіляції, а саме після завершення роботи семантичного аналізатора. Побудова більшості з цих форм потребує також наявності символьної таблиці, яка містить інформацію про всі ідентифікатори,

наявні в програмі, та їх типи. Прикладами широковживаних форм внутрішнього представлення є наступні:

1. Дерево розбору. Дерево розбору є результатом роботи семантичного аналізатора та основою для подальшого отримання більш складних форм внутрішнього представлення. За суттю є впорядкованим деревом з одним коренем, яке репрезентує синтаксичну структуру певного рядка згідно певної контекстно-вільної граматики. Має наступні властивості, які полегшують подальші процеси оптимізації, трансляції та компіляції: по-перше, дерево розбору може бути змінене та доповнене за допомогою способів роботи з абстрактними деревами, що не є можливим для вхідного коду в текстовій формі; по-друге, в порівнянні з текстовим вхідним кодом, дерево розбору не містить надлишкові синтаксичні конструкції, як-то роздільники, дужки та інші; по-третє, дерево розбору дозволяє зберігання додаткової інформації про програму, як, наприклад, позиції елементів, що можна використовувати для подальшої генерації змістовних повідомлень про помилки у вхідному коді.

2. *S*-вирази. Є одночасно способом представлення вхідного коду, формою внутрішнього представлення та способом подання даних у мовах сімейства Lisp. За визначенням, кожний *s*-вираз може бути або атомарним символом, або виразом формату *(x . y)*, де *x* та *y* є *s*-виразами. Під час представлення вхідного коду дане визначення поєднується з префіксною нотацією, згідно першим елементом у виразі зазвичай є оператор або ім’я функції, а всі інші елементи є аргументами. Використання *S*-виразів призводить до гомоіконності, властивості мови програмування, за якої інструкції програми та дані не відрізняються синтаксично. Виконання цієї властивості означає можливість описувати код у вигляді структури даних, яка однозначно відповідатиме його дереву розбору, що в свою чергу значно спрощує синтаксичний аналіз, трансляцію та метапрограмування.

3. Абстрактний семантичний граф, або терм-граф. Є більш абстрактною формою подання вхідного коду, заснованою на дереві розбору. Отримується шляхом ускладнення дерева розбору, наприклад, за допомогою додання зворотних посилань, ваг ребер, циклів тощо. Семантичні графи відрізняються від дерев розбору можливістю зберігати спільні підвирази, тобто абстрактні вузли, що можуть мати кілька різних батьків, і через це часто використовуються в якості проміжного значення для зберігання результатів оптимізації мінімізації підвиразів, виконаної для дерев розбору.

4. Static Single Assignment. Абстрактна форма внутрішнього представлення, яка відповідає наступним умовам: кожній змінній в програмі значення має привласнюватись лише один раз; кожна змінна має бути оголошена, перш ніж використана. Отримується з попередньої форми внутрішнього подання шляхом розбиття існуючих змінних на версії, де кожна нова версія з'являється в момент використання оператора присвоєння на попередню. Використання цієї форми робить явними DU- та UD-ланцюги, інформація про які є необхідною для подальших компіляторних оптимізацій, як-то пропагація констант, видалення мертвого коду, мінімізація спільних підвиразів тощо.

5. Continuation Passing Style. Головною властивістю даної абстрактної форми є передача контролю над потоком виконання за допомогою продовжень. Продовження є представленням стану програми в певний момент часу, яке містить всю необхідну для подальшого продовження роботи програми з цього моменту інформацію. На практиці під продовженнями зазвичай маються на увазі функції мови, які приймають один аргумент – результат обчислень попередньої функції. Продовження в такому вигляді зазвичай передаються до цільових функцій у вигляді додаткового аргументу, а всі операції повернення значення замінюються викликом функції-

продовження. Використання даної форми внутрішнього представлення робить явними наступні важливі для подальшого аналізу властивості вхідного коду: повернення процедур, проміжкові значення, порядок обчислення аргументів функцій та хвостові виклики.

Отримані форми внутрішнього представлення в подальшому використовуються під час створення оптимізуючих компіляторів, тобто компіляторів, що намагаються мінімізувати або максимізувати певні атрибути оброблюваної програми. Найбільш розповсюдженими цілями оптимізуючого транслятора є зменшення часу виконання програми, зменшення обсягів необхідної пам’яті та зниження витрат електроенергії (останнє є найбільш актуальним для мобільних пристроїв). Компіляторна оптимізація зазвичай реалізується у вигляді послідовності оптимізуючих трансформацій, тобто способів, які приймають на вхід початкову програму і продукують семантично еквівалентну програму, яка використовує меншу кількість ресурсів або виконується швидше. Більшість з існуючих компіляторних оптимізацій є NP-повними або навіть алгоритмічно нерозв’язними. Крім цього, на фактичну реалізацію способів компіляторних оптимізацій накладаються додаткові обмеження часу виконання та обсягів пам’яті, що використовується, оскільки зазвичай процес оптимізації є дуже ресурсовитратним для процесора та оперативної пам’яті. Зважаючи на ці фактори, компіляторні оптимізації на практиці дуже рідко продукують оптимальний в будь-якому сенсі результат, а в деяких випадках можуть навіть погіршувати ефективність роботи програми. Через це в реалізаціях таких оптимізацій дуже поширене використання евристичних способів, а використання їх рекомендоване лише для певних підмножин вхідних програм.

Способи, які використовуються в оптимізації, можуть бути поділені на декілька категорій, які розрізняються можливістю впливати на структурні

одиниці вхідного коду. Ці категорії можуть охоплювати як одиничні вирази вхідної програми, так і всю програму повністю. Загалом, способи локального застосування простіші для реалізації, ніж глобальні, їх результати легше спрогнозувати, проте вони можуть призводити до менш значущого результату. Прикладами таких категорій способів оптимізацій є наступні:

1. Peephole-оптимізації.

Зазвичай виконуються на пізніх етапах компіляції, інколи навіть після генерації машинного коду. Такі оптимізації розглядають суміжні інструкції програми, через що і отримали свою назву: вхідний код розглядається ніби «крізь щілину». Метою таких оптимізацій є визначення можливості заміни послідовностей суміжних інструкцій на одну інструкцію чи коротшу послідовність інструкцій. Наприклад, множення значення на 2 може бути замінене на семантично еквіваленте зміщення лівого значення або додавання значення до самого себе, що результує у зниженні вартості виконуваної операції та підвищенні ефективності роботи програми.

2. Локальні оптимізації.

Об’єктом таких оптимізацій виступають базові блоки вхідної програми, тобто лінійні послідовності вхідного коду, що мають лише одну точку входу і одну точку виходу і не містять галужень. Оскільки базові блоки не мають потоку керування, необхідний для цих оптимізацій аналіз є дуже простим і невитратним з точку зору ресурсів, оскільки розглядає лише лінійні послідовності виразів мови. Прикладами локальних оптимізацій є локальна мінімізація підвиразів, локальна пропагація констант та локальне знищення мертвого коду.

3. Функціональні оптимізації.

Вони також називаються внутрішньопроцедурними оптимізаціями і працюють з цілими функціями. З одного боку, це дозволяє під час аналізу

розглядати галуження та різні точки входу-виходу і використовувати графові способи аналізу потоку керування, а з іншого ускладнює аналіз і призводить до більших витрат ресурсів. Найбільш складні для аналізу функції, в яких містяться виклики інших функцій та взагалі посилання на змінні зовнішнього контексту, оскільки інформація про них в даному типі аналізу обмежена.

4. Оптимізації циклів.

Об’єктом оптимізації виступають циклічні конструкції вхідної програми. Дозволяють значно підвищити ефективність використання кешу та проведення паралельних обчислень. Трансформації над циклами потребують додаткового аналізу їх коректності, оскільки на етапі компіляції часто неможливо підрахувати точну кількість повторюваних інструкцій, і, відповідно, точно визначити вплив трансформації на семантику ітеративного процесу. Таким чином, доцільним є використання евристики, оскільки в разі невеликої кількості ітерацій в початковій програмі проведення цієї оптимізації може погіршити ефективність отриманого в результаті трансформацій коду. Виходячи з цього, окремою важливою задачею під час розробки оптимізацій циклів є прогнозування потенційного приросту ефективності оптимізованого коду.

5. Міжпроцедурні оптимізації.

Об’єктом оптимізації є повний вхідний код програми. Оперують максимальною кількістю інформації про вхідну програму і можуть застосовувати найбільш комплексні перетворення, які матимуть найбільш значущий вплив на ефективність результуючої програми, але є також найбільш складними та ресурсовитратними. Дозволяють виконання трансформацій, які є принципово неможливі для більш локальних способів, наприклад, inline expansion. Окрім означених вище категорій, компіляторні оптимізації розділяють також на:

1. Залежні та незалежні від вхідної мови програмування.

Більшість мов високого рівня поділяють загальні синтаксичні конструкції та програмні абстракції: оператори умовного переходу або рішення (*if, switch* та інші), цикли (*for, while, repeat* тощо) та засоби інкапсуляції (структури, об'єкти). Таким чином, способи оптимізації, що працюють з даними абстракціями в теорії можна використовувати для різних мов програмування. Однак деякі особливості реальних мов програмування можуть ускладнювати розробку таких універсальних способів оптимізації. Наприклад, наявність покажчиків у мовах C і C++ значно ускладнює способи оптимізації, що працюють з масивами мови, наприклад, *alias*-аналіз. В той же час такі мови, як PL/1, яка також пропонує підтримку покажчиків, все ж мають реалізовані оптимізуючі компілятори, які використовують специфічні для цільової мови техніки та мають доведену ефективність. Таким чином, деякі особливості мови можуть спрощувати певні оптимізації, але робити неможливим їх застосування для інших мов. Прикладом такої особливості є заборона побічних ефектів функцій у певних функціональних мовах програмування: якщо програма робить кілька викликів до однієї і тієї ж функції з однаковими аргументами, компілятор може негайно зробити висновок про те, що результат функції потрібно обчислити лише один раз. У мовах, де функціям дозволяється мати побічні ефекти, можлива інша стратегія. Оптимізатор може визначити, яка функція не має побічних ефектів, і обмежити такі оптимізації функціями, що не мають побічних ефектів. Так чи інакше, використання подібних технік оптимізації можливе лише тоді, коли оптимізуючий компілятор має доступ до викликаної функції.

2. Залежні та незалежні від цільової платформи.

Багато способів оптимізації, які засновані на аналізі абстрактних програмних концепцій (циклі, об’єкти, структури), не залежать від цільової

платформи, на яку спрямований компілятор, але найбільш ефективними оптимізації є ті, які найкраще використовують особливості цільової платформи. До таких особливостей можна віднести наступне: кількість доступних регістрів процесора, використання RISC або CISC наборів інструкцій, наявність конвеєра інструкцій, кількість функціональних одиниць процесора, обсяги наявної кеш-пам'яті та інші. Прикладами таких оптимізацій є згортка ланцюжків процесорних інструкцій та аналіз гілок виконання інструкцій, що містять оператори умовного переходу.

На початку історії компіляторів компіляторні оптимізації значно поступалися за ефективністю рукописним. По мірі вдосконалення технологій компіляції більшість популярних компіляторів, як правило, генерують код достатньо ефективний для того, щоб повністю відмовитись від ручного написання оптимізованого коду на мові асемблеру (за винятком кількох спеціальних випадків).

Для RISC-архітектур процесора, а тим більше для апаратного забезпечення VLIW, використання компіляторних оптимізацій є ключовою умовою отримання ефективного коду, оскільки набори інструкцій RISC настільки компактні, що людині важко вручну планувати або комбінувати невеликі інструкції для отримання ефективних результатів. В подальшому архітектури даного типу стали розроблятись одразу з урахуванням подальшого застосування компіляторних оптимізацій для отримання ефективного коду.

Незважаючи на це, жоден з наявних способів компіляторної оптимізації не є ідеальним. Використання компіляторних оптимізацій в жодному випадку не може гарантувати, що для всіх вхідних програм буде отримано максимально ефективний еквівалент наданої програми. Більше того, існування такого компілятору в принципі не є можливим, оскільки він міг вирішити проблему зупинки.

Це може бути доведено на прикладі виклику деякої абстрактної функції *foo*. Ця функція нічого не повертає і не має побічних ефектів (не містить операцій вводу-виводу, не змінює глобальні змінні та структури даних контексту виконання тощо). Найшвидшою еквівалентною програмою була б програма, в якій виклик цієї функції взагалі усунено. Однак, якщо функція *foo* насправді не повертає значення, програма з викликом *foo* відрізнятиметься від програми без виклику і оптимізуючий компілятор повинен буде визначити це шляхом вирішення проблеми зупинки, що є неможливим.

Крім того, існує ряд більш конкретних питань щодо практичного використання оптимізуючих компіляторів:

- Компіляторні оптимізації здебільшого фокусуються на відносно неглибоких константних поліпшеннях і, як правило, не покращують алгоритмічну складність програм. Наприклад, компілятор не зможе замінити використання алгоритму сортування бульбашкою на швидке сортування.
- Зазвичай компілятори повинні забезпечувати виконання умов, що певним чином конфліктують між собою, наприклад, вартість реалізації, швидкість компіляції та якість генерованого коду.
- Компілятор, як правило, опрацьовує лише частину програми в певний момент часу, наприклад, код, що міститься в одному файлі або модулі. Це призводить до того, що під час аналізу втрачається інформація про контекст, яка може бути отримана лише за допомогою аналізу інших файлів.
- Витрати ресурсів на власне роботу компілятора. Процес компіляції зазвичай сам по собі вимагає значних витрат пам’яті та процесорного часу, коли до нього додається повнопрограмна оптимізація, витрати ресурсів стають критичною проблемою.

- Часто ускладнена взаємодія між різними фазами оптимізації ускладнює пошук оптимальної послідовності виконання цих фаз.

Більше того, способи оптимізації є складними, і, особливо за використання під час компіляції комплексних мов програмування високого рівня, вони можуть мати помилки, які вносять помилки в компільований код або викликають внутрішні помилки під час компіляції. Помилки компілятора будь-якого виду є неприємним для користувача явища, але особливо в цьому випадку, оскільки, можливо, не ясно, що винна оптимізаційна логіка. У випадку внутрішніх помилок проблема може бути частково компенсована спеціальною fail-safe технікою програмування, в якій логіка оптимізації в компіляторі реалізується таким чином, що потенційна помилка відловлюється, видається попереджувальне повідомлення та інша частина компіляції продовжується до успішного завершення.

Робота з удосконалення технологій оптимізації триває. Один із підходів – використання так званих оптимізаторів пост-проходу. Такі оптимізатори приймають на вхід готовий до виконання результат звичайних оптимізаторів і намагаються ще більше вдосконалити його. Оптимізатори даного типу зазвичай працюють на рівні асемблеру або машинного коду (на відміну від компіляторів, що оптимізують форми проміжного представлення програм). Продуктивність таких оптимізаторів обмежена тим, що значна частина інформації, наявної в оригінальному вихідному коді, не завжди доступна для них.

Оскільки продуктивність процесора продовжує вдосконалюватися швидкими темпами, ніж пропускна здатність пам’яті, оптимізації, які зменшують потреби в пропускній здатності пам’яті (навіть за рахунок підвищення кількості виконуваних процесором інструкцій), стають все більш

корисними. Приклади цього включають оптимізацію вкладених циклів та рематеріалізацію.

1.2. Аналіз способів оптимізації хвостових викликів

За визначенням хвостовий виклик є виклик підпроцедури, виконаний в якості останньої інструкції процедури. Явище, коли цей виклик може в майбутньому призвести до виклику цієї самої підпроцедури, має назву хвостової рекурсії, яка зазвичай рекомендована для використання на користь інших видів рекурсії через певні властивості, які спрощують подальший аналіз програми. Головною властивістю хвостової рекурсії є те, що під час розробки компіляторів та рушіїв мови хвостові виклики можуть бути реалізовані без додавання додаткових фреймів до стеку виконання. Більшість унікальних фреймів процедур, що використовують хвостові виклики, можуть бути замінені на фрейм хвостового виклику, що може призвести до значної економії обчислювальних ресурсів та часу, а також вирішить проблему виснаження стеку викликів. Таким чином, головною метою способів оптимізації хвостових викликів є мінімізація кількості елементів, що під час виконання програми мають бути додані до стеку викликів. Крім цього, окрім основної проблеми обмеженої глибини стеку викликів існують також проблеми, пов’язані з подальшим використанням модифікованої програми.

1.2.1. *Проблема обмеженої глибини стеку викликів*

Однією з найважливіших абстракцій будь-якої сучасної мови програмування є функція, тобто фрагмент програмного коду, який репрезентує певну послідовність команд і до якого можуть звертатись інші компоненти програми. З появою і подальшим розвитком функціональної парадигми програмування широкого поширення набули функції вищого порядку та рекурсія.

В контексті програмування під рекурсією зазвичай мають на увазі виклик функції з неї ж самої. Існують два різновиди рекурсії з точки зору складності:

1. Проста або пряма рекурсія. Отримується за допомогою однієї функції, яка викликає сама себе. Найпоширеніший випадок рекурсії, який порівняно нескладно аналізувати. Прикладами прямої рекурсії можуть слугувати «наївні» реалізації математичних функцій факторіалу або чисел Фібоначчі, функції для обходу дерев та графів, функції графічного відображення фракталів тощо.
2. Складна або, відповідно, непряма рекурсія. Отримується з використанням кількох функцій.

```
function foo () {
    /**
     * do some smart stuff
     */
    if (/**some condition*/)
        return bar()
    else
        return;
}

function bar () {
    /**
     * do some another smart stuff
     */
    if (/**another condition */)
        return;
    else
        return foo()
}
```

В даному прикладі маємо дві функції, foo та bar, кожна з яких містить власну логіку роботи з подальшим рекурсивним крос-викликом одна одної. Також кожна з цих функцій містить умову припинення рекурсії, про необхідність наявності якої буде написано далі.

Кількість вкладених викликів функції називається глибиною рекурсії. Загалом, використання рекурсії дозволяє описати повторювані або навіть

нескінченні обчислення, уникаючи при цьому дублювання програмного коду, використання операторів безумовного переходу та циклів.

З точки зору візуалізації ходу виконання програм рекурсивна функція на верхньому рівні завжди містить команду розгалуження, тобто вибір однієї з двох або більше альтернатив подальшого потоку виконання в залежності від певного набору умов. Такі умови в даному випадку доречно назвати «умовами припинення рекурсії». В подальшому програма має дві або більше альтернативні гілки, з яких хоча б одна обов'язково є рекурсивною і хоча б одна – термінальною. Рекурсивна гілка виконується, коли умова припинення рекурсії не виконується, а програма містить хоча б один рекурсивний виклик – прямий або непрямий. Відповідно, термінальна гілка виконується, коли умова припинення рекурсії виконується, після чого відбувається «згортання» рекурсії і повернення функцією певного значення. Належним чином реалізована рекурсивна функція повинна гарантувати, що через кінцеве число рекурсивних викликів буде досягнуто виконання умови припинення рекурсії, в результаті чого ланцюжок послідовних рекурсивних викликів перерветься і відбудеться вихід з рекурсії з опціональним збиранням результату роботи на рекурсивному підйомі.

Крім розглянутого випадку, коли рекурсивна гілка виконує лише один рекурсивний виклик, існує також так звана «паралельна рекурсія», за якої одна рекурсивна гілка потоку виконання може містити більше, ніж один рекурсивний виклик. Саме такий вид рекурсії найчастіше використовується для роботи зі складними структурами даних, наприклад, деревом, кучею, графом та ін. Крім того, прикладом паралельної рекурсії може вже згадана функція обчислення ряду Фібоначчі, в якій для отримання кожного n-ного числа необхідно обчислити (n–1)-й і (n–2)-й елементи, що можна виконувати

паралельно. У випадку обробки дерев паралельними викликами будуть відповідно виклики функції обробки для піддерев.

Реалізація рекурсивних викликів функцій у переважній більшості сучасних мовах програмування високого рівня і середовищах їх виконання, як правило, заснована на алгоритмі стеку викликів.

Стек викликів – це LIFO-стек, що містить інформацію про адреси викликаних функцій та адреси місць в програмі, в які повертається потік керування після завершення роботи функцій. Стек викликів зазвичай зберігається у оперативній пам’яті, тому його розмір є доволі обмеженим. Кожний виклик функції додає елемент до стеку, після завершення роботи функції цей елемент вилучається зі стеку. Дані про рекурсивні виклики, тобто виклики функцією самої себе, також додаються до стеку і зберігаються в ньому аж до завершення рекурсії. Оскільки стек має обмежену місткість, а глибина рекурсії під час обробки великих об’ємів даних може бути дуже великою, з’являється проблема виснаження стеку викликів. Різні мови програмування по-різному вирішують цю проблему, більше того, в рамках однієї мови можуть одночасно існувати кілька способів її вирішення.

Незважаючи на це, питання про доречність використання рекурсивних функцій в програмуванні є неоднозначним: з одного боку, рекурсивна форма може бути більш наочною простою для розуміння, особливо у випадках, коли алгоритм, який реалізується теж є рекурсивним. Крім цього, в деяких декларативних або чистих функціональних мовах (наприклад, Prolog або Haskell) просто не існує синтаксичних засобів для організації циклів, і рекурсія в них – єдиний доступний механізм реалізації повторюваних обчислень. Через це такі мови повинні забезпечувати дуже потужний набір оптимізацій для ефективного управління рекурсивними викликами, інакше практичне використання таких мов. Проте в інших, більш популярних та широко

використаних в індустрії комерційної розробки програмного забезпечення таких оптимізації можуть бути не реалізовані, тому дуже часто можна побачити рекомендації щодо уникнення використання рекурсії у місцях програми, які в деяких умовах можуть призводити до дуже великої глибині рекурсії. Так, вже згадуваний та широко поширений в навчальній літературі приклад рекурсивного обчислення факторіала є, скоріше, прикладом того, як не треба застосовувати рекурсію, оскільки така реалізація для достатньо великих вхідних чисел гарантовано призведе до виснаження стеку виклику або буде працювати дуже повільно у випадку наявності певних оптимізацій стеку виклику. В той же час цей алгоритм має очевидну ефективну реалізацію з використанням звичайного циклу.

Існує також спеціальний тип рекурсії, що дістав назву «хвостової рекурсії». Структура такого рекурсивного алгоритму наступна: рекурсивний виклик є останньою виконуваної операцією в функції, а його результат безпосередньо повертається в якості результату функції. Інтерпретатори і компілятори більшості функціональних мов програмування, що підтримують оптимізацію коду (вихідного або виконуваного), автоматично перетворять хвостову рекурсію до ітерації, завдяки чому забезпечується виконання способів з хвостової рекурсією в обмеженому обсязі пам'яті. Такі рекурсивні обчислення, навіть якщо вони формально нескінченні (наприклад, коли за допомогою рекурсії організовується робота командного інтерпретатора, що приймає команди користувача), ніколи не призводять до виснаження пам'яті. Однак далеко не завжди стандарти мов програмування чітко визначають, яким саме умовам повинна задовольняти рекурсивна функція, щоб транслятор гарантовано перетворив її в ітерацію.

Одним з таких випадків є мова JavaScript, стандарт якої містить вимоги до оптимізації рекурсивних викликів, але транслятори якої не пропонують

достатніх засобів для їх реалізації. Формальний опис цих вимог, а також проблеми, які ускладнюють їх виконання під час створення оптимізуючих трансляторів потребують окремого детального аналізу, наведеного в наступному розділі.

1.2.2. *Проблема збереження контролю над стеком викликів*

Під час розробки програм з використанням будь-якої парадигми, функціональної, об’єктно-орієнтовної, процедурної та інших, дуже важливою є можливість подальшого налагодження розробленої програми. Одним з найважливіших елементів налагодження програми є робота зі стеком викликів програми, а саме можливість вільного переходу між різними фреймами та доступ до даних цих фреймів. В загальному випадку застосування оптимізації хвостових викликів певним чином модифікує стек викликів, що призводить до ситуації, коли під час подальшого налагодження певна інформація, необхідна для виправлення помилок, втрачається, або не відповідає дійсності.

У програмах, що містять використання рекурсивних функцій будь-якого вигляду, або навіть просто великої кількості вкладених викликів функцій, у стек викликів за умови звичайного виконання програми будуть додаватись всі вкладені виклики функцій, жоден з них не буде прибраний зі стеку до тих пір, поки не завершить своє виконання найглибший, останній виклик. Всі проміжкові дані, наприклад, значення аргументів функцій, стан залежних змінних з контексту виконання тощо також будуть зберігатись для кожного елементу стеку під час виконання, та, відповідно, налагодження програми, що і використовують розробники інструментів налагодження. Різні підходи до оптимізації вкладених та хвостових викликів можуть по-різному впливати на кількість інформації, що зберігається, і якщо в теорії такі оптимізації мають мінімізувати кількість елементів стеку викликів та збережених проміжних

даних, на практиці важливим є збереження достатньої кількості інформації для подальшої коректної роботи інструментів налагодження.

Найважливішим елементом, який необхідно зберігати, є стек-трейс, тобто знімок стану стеку викликів у певний момент часу, який дозволяє відслідкувати ланцюжок викликів функцій і в разі виникнення помилки під час виконання програми знайти локалізувати цю помилку. Оскільки способи оптимізації хвостових викликів передбачають прибирання «зайвих» елементів стеку виклику, їх використання може значно ускладнити або навіть зробити неможливим коректне налагодження програми.

1.2.3. *Порівняння способів оптимізації хвостових викликів*

Загалом, існують кілька підходів до оптимізації хвостових викликів, які можуть бути застосовані в залежності від певних особливостей мови, як-то наявність операторів безумовного переходу, наявність функцій вищого порядку, анонімних функцій та інших. Існуючі способи оптимізації можуть бути поділені на наступні групи, що принципово відрізняються між собою:

1. Правильні хвостові виклики (Proper tail calls). Ця техніка оптимізації реалізовується на рівні рушія мови і не передбачає використання форм проміжного представлення або змін синтаксису мови. Натомість пропонується реалізація механізму повторного використання вже існуючих елементів стеку викликів: хвостові виклики замість виділення пам’яті під новий елемент стеку використовують адресний простір поточного елементу, який відповідає функції, що містить хвостовий виклик. Цей спосіб вирішує проблему виснаження стеку викликів, проте накладає ряд обмежень на програми, до яких його можна застосовувати, оскільки він працює лише для функцій з виключно хвостовими викликами, і жодним чином не вирішує проблем, пов’язаних з подальшим налагодженням отриманих програм.

2. Syntactic Tail Calls. Цей спосіб передбачає передачу контролю над механізмами переходу між хвостовими викликами розробнику програми шляхом надання йому особливих синтаксичних конструкцій, які дозволять йому в явному вигляді позначити перехід між функцією та її хвостовим викликом. Є найпростішим з точки зору реалізації, оскільки потребує, по суті, лише реалізації оператора безумовного переходу goto, але фактично не є автоматизованим способом оптимізації, оскільки вимагає від розробника внесення ручних змін до його функцій. Крім цього, цьому способу притаманні також всі проблеми, пов’язані з використанням операторів безумовного переходу, до яких відноситься значно ускладнений аналіз потоку керування та певні вразливості безпеки адресного простору програм з такими операторами.

3. TCO/TCE. Ця категорія передбачає виконання певних перетворень вхідної програми з метою отримання семантично еквівалентної програми, яка буде ефективніше використовувати стек викликів. Оптимізована програма може отримуватись кількома способами, наприклад, заміною рекурсивних викликів на безумовні переходи, перетворення рекурсивних способів в ітеративні, заміну вкладених викликів на послідовності викликів тощо. Саме цей спосіб в тому чи іншому вигляді використовується в більшості компіляторів різних мов програмування, але він жодним чином не вирішує проблеми збереження інформації для налагодження оптимізованих програм.

1.3. Оптимізація хвостових викликів в мові JavaScript

Мова JavaScript із самого початку свого розвитку пропонувала широкий інструментарій для написання коду у функціональному стилі, тому рекурсія набула дуже широкого використання у дуже великій кількості програмних продуктів. Незважаючи на це, жодної оптимізації рекурсивних викликів спочатку передбачено не було. Деякі рушії мови пропонували власні реалізації TCO та інших оптимізацій, проте ці реалізації залежали від конкретного рушія,

не були належним чином документовані та часто не підтримувались, навіть якщо коректно працювали. Використання цих реалізацій в програмному коді завжди супроводжувалося нестабільністю роботи програми, погіршенням швидкодії та значними витратами на подальшу підтримку програмних продуктів. Лише в рамках останнього стандарту мови, ECMAScript 6, було визначено вимоги до оптимізації хвостових рекурсивних викликів і на їх основі розроблено специфікацію, яку відтепер зобов'язані реалізовувати рушії, що відповідають цьому стандарту.

Згідно стандарту, функції, що написані з використанням директиви препроцесора «strict mode», за жодних умов не мають призводити до переповнення стеку викликів. Декомпозиція цієї вимоги дозволяла виокремити наступні пункти, досягнення яких робить можливим її виконання:

1. Реалізація абстрактної операції IsInTailPosition. Результатом виконання цієї операції має бути інформація про те, чи знаходиться певна конструкція вхідного коду в позиції хвостового виклика. Операція має приймати в якості вхідного параметра нетермінал мови і виконувати наступні кроки:
    1. Переконатись, що переданий аргумент є розпізнаною продукцією граматики.
    2. Якщо вхідний код, що відповідає аргументу, не помічено директивою «strict mode», повернути false.
    3. Якщо аргумент не інкапсулює продукції типів FunctionBody або ConciseBody, повернути false.
    4. Якщо тіло аргументу має тип GeneratorBody, повернути false.
    5. Повернути результат операції аналізу позиції продукції HasProductionInTailPosition для тіла аргументу.

2. Реалізація абстрактної операції PrepareForTailCall. Виконання даної операції має гарантувати вивільнення будь-яких метаданих, що пов’язані з контекстом виконання поточної функції перед викликом цільової функції або забезпечити повторне використання цих ресурсів цільовою функцією за допомогою прямої передачі ресурсів контексту виконання цільової функції. Операція виконує наступні кроки:
    1. Зберегти у змінну leafContext поточний контекст виконання.
    2. Виконати операцію suspend над leafContext.
    3. Виконати операцію pop для стеку контексту виконання. Після цього контекст виконання на вершині стеку стає поточним контекстом виконання.
    4. Переконатись, що поточний контекст виконання більше ніколи не буде перетинатись зі вмістом змінної leafContext.

Тим не менш, станом на сьогодні TCO підтримується лише одиницями рушіїв, і навіть в цих рушіях ця функція зазвичай не рекомендується для використання в production-коді.

Розглянемо проблеми, які з’являються під час реалізації і подальшого використання Tail Call Optimization.

1. Швидкодія. Загалом, окрім вирішення проблеми переповнення стеку виконання TCO є оптимізуючою стратегією, що передбачає подальше покращення швидкодії оптимізованого коду. Проте дослідження команди розробників рушія V8 свідчать як мінімум про нейтральний вплив алгоритму на швидкодію отриманого коду, а команда ChakraEngine під час реалізації даного алгоритму в своєму рушії була змушена обирати між значним погіршенням швидкодії отриманого коду та невідповідністю роботи алгоритму специфікаціям. Оскільки TCO орієнтується переважно на код web-застосувань, які інтенсивно

використовують рекурсивні виклики, погіршення швидкодії матиме в якості прямого наслідку збільшення часу відповідей під час взаємодії з користувачами, що в загальному є неприпустимим.

2. Обробка помилок виконання під час налагодження програми. Найважливішим інструментом, який використовують всі розробники для налагодження, є власне стек викликів. Алгоритм TCO безпосередньо оперує зі стеком викликів, а результатом роботи алгоритму стає також те, що більшість елементів цього стеку зникає через оптимізацію. Це значно ускладнює, а в певних випадках навіть унеможливлює проведення будь-якого аналізу стеку викликів, а отже є неприпустимим. Для вирішення цієї проблеми можна зберігати інформацію про видалені елементи стеку викликів у зовнішній структурі даних, яка буде використовуватися під час налагодження. Це зі свого боку призводить до додаткових витрат пам’яті та потреби в додаткових ітераціях Garbage Collector, що також необхідно враховувати під час розробки.

Таким чином, в загальному випадку оптимізація рекурсивних викликів вирішує проблему виснаження стеку викликів, але може призвести до погіршення швидкодії оптимізованого коду, може застосовуватись лише для дуже обмеженої множини вхідних програм, а також значно ускладнює механізми обробки помилок виконання програми під час налагодження. Запропонований модифікований спосіб оптимізації ставить на меті вирішення проблем універсальності та збереження інформації, необхідної для подальшого налагодження програм.

2. СПОСІБ ОПТИМІЗАЦІЇ ХВОСТОВИХ ВИКЛИКІВ НА ОСНОВІ CPS-ФОРМИ

2.1. Побудова CPS-форми внутрішнього представлення. Аналіз ключових компонентів AST для побудови CPS-форми

В загальному розумінні під CPS, стилем передачі продовжень, мається на увазі стиль програмування, під час якого контроль виконання передається в явному вигляді за допомогою продовжень.

Функція, написана у стилі передачі продовжень, має в якості додаткового аргументу продовження в явному вигляді, яке являє собою лямбда-функцію. Коли функція завершила свої обчислення і готова до повернення результату, вона замість використання інструкції повернення значення викликає продовження з цим значенням в якості вхідного аргументу. Це означає, що при виклику функції, написаної в CPS-формі, від цієї функції вимагається надання процедури подальшої обробки обчислюваного значення, тобто продовження. Побудова вхідного коду в цій формі робить явним ряд властивостей програми, які не є явними в інших формах представлення. До них відносяться: операції повернення процедури, які стають очевидними у вигляді викликів продовжень; проміжні значення, якими в даному випадку є всі іменовані значення; порядок обчислення аргументів, який стає явним; і хвостові виклики, які є просто викликають процедуру з тим самим немодифікованим продовженням. Розглянемо переваги CPS-форми на прикладі функції обчислення факторіалу. Наївна реалізація такої функції виглядає наступним чином:

```
function fact1(n) {
      if (n === 0)
            return 1
      else
            return n * fact(n-1)
}
```

Варіант реалізації цієї функції з використанням хвостової рекурсії:

```
function fact2(n) {
      return tail_fact (n, 1)
}

function tail_fact (n, a) {
      if (n === 0)
            return a
      else
            return tail_fact (n-1, n*a)
}
```

Зведені вручну до CPS-форми версії цих функцій виглядають наступним чином:

```
function fact1 (n, ret) {
      if (n === 0)
            ret (1)
      else
            fact (n-1, (t0) => ret (n * t0))
}

function fact2 (n, ret) {
      tail_fact (n,1, ret)
}
function tail_fact (n, a, ret) {
      if (n == 0)
            ret(a)
      else
            tail_fact (n-1, n*a, ret)
}
```

Очевидним є те, що незалежно від типу рекурсії, в якому був написаний початковий код, отриманий код в CPS-формі в будь-якому випадку містить виключно хвостові виклики функцій, а отже, є повністю придатним до застосування способів оптимізації хвостових викликів.

Існує детермінований алгоритм перетворення програм з прямого стилю в CPS. Під час компіляції вхідного коду, програмні конструкції високого рівня, від ко-рутин і обробників виключних ситуацій до циклів і операторів переходу поступово зводяться до послідовностей комбінацій двох структур: викликів функцій та викликів продовжень. Після цього, під час CPS-трансформації, виклики продовжень зводяться до викликів функцій, що призводить до того, що залишається лише одна керуюча структура: лямбда-функція.

Для опису алгоритму зведення абстрактного синтаксичного дерева мови JavaScript ключовими продукціями граматики мови є анонімні функції, вирази та змінні. Зазначені продукції граматики під час роботи з деревом розбору описуються наступним чином:

- Function. Має наступну структуру:

```
Function <: Node {
    id: Identifier | null;
    params: [ Pattern ];
    body: FunctionBody;
}
```

Є абстрактним інтерфейсом будь-якої функції і не є самостійною продукцією граматики, оскільки може існувати лише в якості обгортки інших функціональних продукцій: FunctionExpression, ArrowFunctionExpression, FunctionDeclaration.

- FunctionExpression. Має наступну структуру:

```
FunctionExpression <: Function, Expression {
    type: "FunctionExpression";
}
```

Вираз, що містить оголошення іменованої або анонімної функції. Зазвичай функціональні вирази містять анонімні функції, але є також можливість використання іменованих функцій, що може бути корисним в разі наявності рекурсивних викликів або для ідентифікації функції у стеках викликів під час подальшого налагодження програми. Функціональні об’єкти у складі продукції FunctionExpression набуває всіх властивостей звичайного виразу мови, тобто може передаватись в якості параметрів іншим функціям, бути значенням змінної, тощо. Крім того, лише FunctionExpression може бути значенням властивості value продукції MethodDefinition, тобто бути оголошенням методу класу.

- ArrowFunctionExpression. Має наступну структуру:

```
ArrowFunctionExpression <: Function, Expression {
    type: "ArrowFunctionExpression";
    body: FunctionBody | Expression;
    expression: boolean;
}
```

Вираз, що містить стрілкову функцію.

- FunctionDeclaration. Має наступну структуру:

```
FunctionDeclaration <: Function, Declaration {
    type: "FunctionDeclaration";
    id: Identifier;
}
```

Вираз мови, що містить оголошення іменованої функції, тобто ідентифікатор якої не може набувати значення null.

Як вже було зазначено, під час виконання CPS-перетворення всі функціональні об’єкти будуть перетворюватись в лямбда-функції, які у випадку мови JavaScript являють собою стрілочні функції одного аргументу. Стрілочні функції є частиною стандарту ES6 і мають ряд відмінностей від звичних функцій, оголошених за допомогою ключового слова function. В контексті побудови CPS-форми важливо знати про деякі особливості таких функцій. По-перше, тілом стрілочної функції може виступати не тільки блок виразів, а і звичайний одиночний вираз мови. В цьому випадку функція буде повертати значення цього виразу. Якщо така функція має повертати вираз, оголошення якого синтаксично ідентичне BlockStatement, тобто виконується за допомогою фігурних дужок, наприклад, єдиний літерал об’єкта, то такий вираз має бути обгорнутий у додаткові круглі дужки:

() => {some: object} //синтаксична помилка

() => ({some: object}) //правильний код

Таким чином, під час побудови CPS-форми в разі застосування перетворення до стрілочних функцій необхідно окремо оброблювати функції, що містять єдиний вираз, оскільки фактично вони містять також неявну інструкцію повернення значення, використання якої не дозволене в CPS-формі.

Іншою важливою відмінністю стрілочних функцій від звичайних є механізм роботи посилань на зовнішній контекст виконання за допомогою ключового слова this. Функції, оголошені за допомогою function, визначають власне значення this, яке залежить від способу виклику функції:

- Звичайний виклик, відсутня директива ‘use strict’. В цьому випадку this буде посилатись на глобальний об’єкт, тобто window у випадку браузера або global у випадку Node.js.
- Звичайний виклик, директива ‘use strict’. В цьому випадку значення this буде таким, як в оточуючому контексті в момент входу в контекст виконання функції.
- Виклик за допомогою методів call та apply. У випадку використання цих методів значення першого параметра буде взято в якості значення this.

Якщо до виклику call або apply було передано значення, що не є об’єктом, то воно буде неявно приведене до об’єкта за допомогою метода toObject. У випадку примітивів буде викликано відповідний конструктор, наприклад:

foo.call(5)

Буде викликано конструктор Number, значенням this буде об’єкт типу Number зі значенням властивості value 5.

foo.apply(‘str’)

Аналогічно, буде викликано конструктор String, значенням this буде об’єкт типу String зі значенням властивості value «str».

- Виклик функції, отриманої за допомогою метода bind. У стандарті ES5 було введено метод bind, який дозволяє отримати копію функції зі значенням this, що відповідає значенню параметра, переданого у виклик bind.

Для стрілочних функцій в загальному випадку значення this буде відповідати значенню this зовнішнього лексичного контексту, що є ідентичним використанню директиви ‘use strict’ для звичайних функцій. Проте в разі виклику стрілочної функції за допомогою методів call та apply, а також виклику

функції, отриманої зі стрілочної за допомогою bind, значення першого параметра буде проігнороване, і значення this не буде змінене.

Таким чином, під час застосування CPS-перетворень основним елементом дерева розбору є стрілочна функція, до якої зводяться всі інші функціональні об’єкти, наявні у вхідному коді, і до якої буде застосовуватись більшість подальших перетворень. Іншим важливим елементом для перетворень є набори аргументів функцій, до яких буде додаватись генероване функціональне продовження. Останніми ключовими елементами для отримання CPS-форми є інструкції повернення значень в кінцях гілок виконання функціональних блоків, на значенні яких буде викликатись додане до аргументів продовження, що матиме в якості результату отримання програми з явною передачею контролю виконання.

2.1.1. *Перетворення вхідного коду для отримання CPS-форми*

Зведення всіх функцій вхідної програми до стрілочних з урахуванням означених особливостей є тривіальним процесом заміни елементів дерева розбору еквівалентними. Після виконання цієї трансформації подальші перетворення функціональних об’єктів у CPS-форму розглядають синтаксичні конструкції вхідної мови в якості правил абстрактної граматики наступного вигляду:

$$\langle expr \rangle ::= \left(\lambda \ \left(\langle var \rangle \right) \langle expr \rangle \right)$$
$$| \langle var \rangle$$
$$| \left(\langle expr \rangle \langle expr \rangle \right)$$

CPS-форма для цього правила виглядає наступним чином:

$$\langle aexp \rangle ::= \left(\lambda \ \left(\langle var \rangle^{*} \right) \langle cexp \rangle \right)$$
$$| \langle var \rangle$$
$$\langle cexp \rangle ::= \left(\langle aexp \rangle \langle aexp \rangle^{*} \right)$$

Для отримання цієї форми було введено дві нові граматичні субпродукції: атомарні та комплексні вирази, $\langle aexp \rangle$ та $\langle cexp \rangle$ відповідно. Атомарними виразами називаються вирази, які завжди продукують значення та ніколи не містять побічних ефектів. Комплексні вирази в свою чергу можуть не завершувати своє виконання, не продукувати значення та можуть мати побічні ефекти. Першим етапом в отриманні CPS-форми є виконання т. зв. наївного перетворення для означених продукцій граматики, яка складається з двох абстрактних функцій *M* та *T*:

- функція $M: expr \to aexp$ переводить термінальний вираз мови, тобто змінну або лямбда-функцію в атомарний CPS-вираз;
- функція $T: expr \times aexp \to cexp$ приймає на вхід вираз та синтаксичне продовження та викликає продовження на результат конвертованого в CPS-форму.

Вираз вигляду $T\ (expr\ cont)$ означає наступне: трансформувати *expr* в CPS-форму таким чином, щоб *cont* було викликано на результаті, отриманому з *expr*. Функція M в свою чергу застосовується лише для лямбда-виразів: для кожного оголошення лямбда-функції f вона додає вхідний параметр продовження *k*, після чого трансформує тіло функції за допомогою виклику функції *T* з аргументами *f* та *k* таким чином, щоби воно викликало *k* на результаті *f*. Змінні при цьому залишаються без змін.

Функції M та T визначаються наступним чином:

$$. \quad : M \to expr \to aexp$$

$$[\![\lambda (var)(expr)]\!] = \left[\!\!\left[\begin{array}{l} (gensym() \leftarrow \$k) \\ \lambda (var\ \$k)[\![T\ \ (expr)\ \ \$k]\!] \end{array} \right]\!\!\right]$$

$$[\![(?\ symbol\ ?)]\!] = [\![(expr)]\!]$$

$$. \quad : T \to expr \to cont \to cexp$$

$$[\![\lambda(\_)]\!] = [\![cont\ M\ (expr)]\!]$$

$$[\![(?\ symbol\ ?)]\!] = [\![cont\ M\ (expr)]\!]$$

$$[\![(f\ e)]\!] = \left[\!\!\left[\begin{pmatrix} gensym() \leftarrow \$f \\ gensym() \leftarrow \$e \end{pmatrix} \right.\right.$$
$$\left.\left. [\![T\ f\ (\lambda(\$f))(T\ e\ (\lambda(\$e)(\$f\ \$e\ cont)))]\!] \right]\!\!\right]$$

Варто зазначити, що такий спосіб комбінації функцій *M* та *T* не готовий до реалізації на практиці, оскільки операція перетворення застосування функції в ньому передбачає, що функція та її аргументи завжди є комплексними виразами, що призводить до генерації надлишкових продовжень, яких можна уникнути у випадках, коли функція та аргумент будуть саме атомарними виразами. Для вирішення цієї проблеми застосовується перетворення вищого порядку, яке враховує можливість роботи з атомарними виразами. Головною проблемою попереднього методу перетворення було те, що він завжди вимагав прив'язування виклику функції і його аргументів до нових змінних, навіть якщо функція і аргументи вже були атомарними виразами. Якщо трансформація буде приймати реальну функцію, що очікує атомарну версію наданого продовження, то трансформаціє буде здатна перевіряти необхідність створення нової змінної.

В трансформації вищого порядку функція *T* набуває вигляду $expr \times (aexp \to cexp) \to cexp$ і приймає в якості аргументу функцію замість синтаксичного продовження, якій буде надано атомізовану версію виразу, що в результаті призведе до створення комплексної CPS-форми.

$$. \quad : T \to expr \to k \to cexp$$

$$[\![\lambda(\_)]\!] = [\![k\ M\ (expr)]\!]$$

$$[\![(?\ symbol\ ?)]\!] = [\![k\ M\ (expr)]\!]$$

$$[\![(f\ e)]\!] = \left[\!\!\left[\begin{pmatrix} gensym() \leftarrow \$rv \\ \lambda(\$rv)(k\ \$rv) \leftarrow cont \end{pmatrix} \right]\!\!\right]$$
$$\left[\!\!\left[T\ f\left(\lambda(\$f)\right)\left(T\ e\left(\lambda(\$e)(\$f\ \$e\ cont)\right)\right)\right]\!\!\right]$$

Така модифікація трансформації вирішує проблему продукування надлишкових змінних, проте призводить до появи η-розширення навколо продовження. Найкращих результатів можна досягти шляхом поєднання першого та другого способів трансформації у третій, гібридний спосіб трансформації. Для цього вводяться наступні функції перетворення:

- функція $TC: expr \times aexp \to cexp$, яка буде застосовуватись для трансформації комплексних виразів;
- функція $TK: expr \times (aexp \to cexp) \to cexp$, яка буде викликатись функцією M і буде приймати рішення про необхідність передачі контролю над трансформацією функції *TC*.

$$. \quad : TC \to expr \to k \to cexp$$

$$[\![\lambda(\_)]\!] = [\![k\ M\ (expr)]\!]$$

$$[\![(?\ symbol\ ?)]\!] = [\![k\ M\ (expr)]\!]$$

$$[\![(f\ e)]\!] = \left[\!\!\left[\begin{pmatrix} gensym() \leftarrow \$rv \\ \lambda(\$rv)(k\ \$rv) \leftarrow cont \end{pmatrix} \right]\!\!\right]$$
$$\left[\!\!\left[T\ f\left(\lambda(\$f)\right)\left(T\ e\left(\lambda(\$e)(\$f\ \$e\ cont)\right)\right)\right]\!\!\right]$$

$$. \quad : TK \rightarrow expr \rightarrow k \rightarrow cexp$$

$$\left[\!\left[\lambda(\_) \right]\!\right] = \left[\!\left[k \; M(expr) \right]\!\right]$$

$$\left[\!\left[(?\, symbol\, ?) \right]\!\right] = \left[\!\left[k \; M(expr) \right]\!\right]$$

$$\left[\!\left[(f \;\; e) \right]\!\right] = \left[\!\left[\begin{array}{l} \left(\begin{array}{l} gensym() \leftarrow \$rv \\ \lambda(\$rv)(k \; \$rv) \leftarrow cont \end{array} \right) \\ \left[\!\left[T \;\; f\left(\lambda(\$f)\right)\left(T \; e\left(\lambda(\$e)(\$f \; \$e \; cont)\right)\right) \right]\!\right] \end{array} \right]\!\right]$$

2.1.2. ***Задача збереження контролю над стеком викликів***

Описані вище способи трансформацій вирішують проблему отримання нормалізованої CPS-форми функціональних виразів мови JavaScript, але проблема збереження можливості відновлення стеку викликів для трансформованої програми потребує додаткової модифікації алгоритму перетворення. Для можливості подальшого відновлення стеку викликів трансформація повинна розрізняти і відмічати вирази виклику та лямбда-функції на функції-користувачі (users) та функції-продовження (continuations). Для цього до абстрактної граматики функціональних викликів вводяться наступні продукції:

$$\langle uexp \rangle ::= \left(\lambda \;\; \left(\langle uvar \rangle \langle kvar \rangle \right) \;\; \langle call \rangle \right)$$

$$\quad\quad\quad\quad | \; \langle uvar \rangle$$

$$\langle kexp \rangle ::= \left(\kappa \;\; \left(\langle uvar \rangle \right) \langle call \rangle \right)$$

$$\quad\quad\quad\quad | \; \langle kvar \rangle$$

$$\langle call \rangle ::= \langle ucall \rangle \,|\, \langle kcall \rangle$$

$$\langle ucall \rangle ::= \left(\langle uexp \rangle \; \langle uexp \rangle \; \langle kexp \rangle \right)$$

$$\langle kcall \rangle ::= \left(\langle kexp \rangle \; \langle uexp \rangle \right)$$

Всі функціональні виклики тепер можуть набувати вигляд *u*-форми та *k*-форми, які є семантично еквівалентними, проте *k*-форма свідчить про те, що процедура цієї продукції є продовженням, генерованим трансформацією. Відповідно до цього, механізм генерації нових змінних тепер має

використовувати різні генератори для *u*-форми та *k*-форми: для генерації нових змінних, прив'язаних до значень функцій-користувачів використовується функція *genusym,* для генерації нових змінних, прив'язаних до значень продовження, використовується функція *genksym*.

Анотування функціональних об'єктів за допомогою таких функцій робить можливим подальшу роботу з продовженнями, організованими у вигляді LIFO-стеку та дозволяє використання регістру покажчику стеку для генерованого коду. Відповідно, під час кожного створення нового продовження, необхідно виконувати операцію інкременту на цей покажчик, а після обробки виклику цього продовження виконувати операцію скидування значення покажчика до попереднього.

2.1.3. *Застосування трансформацій до розширеної граматики мови*

Зазначені вище способи наочно демонструють архітектуру абстрактного алгоритму трансформації функціональних об'єктів мови у CPS-форму, проте вони розглядають застосування перетворень до атомарних і комплексних функціональних виразів без урахування особливостей їх внутрішньої структури, які впливають на процес трансформації в разі її застосування до реального програмного коду. Для цього визначений формальний синтаксис функціональних виразів має бути розширений і містити визначення конструкцій, що містяться в реальному коді функцій, тобто конструкції галуження, математичні вирази, оголошення локальних змінних, наявність типів даних, обробку виключних ситуацій, змінні для позначення відсутності значення тощо.

Розширена граматика функціональних виразів виглядає наступним чином:

$$\langle aexp \rangle ::= \left(\lambda \left(\langle var \rangle^*\right)\langle cexp \rangle\right)$$
$$| \langle var \rangle$$
$$| \#t \ | \# f$$
$$| \langle number \rangle$$
$$| \langle string \rangle$$
$$| (void)$$
$$| call/ec \,|\, call/cc$$
$$\langle cexp \rangle ::= \langle aexp \rangle$$
$$| \left(begin \langle cexp \rangle^*\right)$$
$$| \left(if \ \langle cexp \rangle \langle cexp \rangle \langle cexp \rangle\right)$$
$$| \left(set! \langle var \rangle \langle cexp \rangle\right)$$
$$| \left(letrec\left(\left[\langle var \rangle \ \langle aexp \rangle\right]^*\right)\langle cexp \rangle\right)$$
$$| \left(\langle prim \rangle \langle aexp \rangle^*\right)$$
$$| \left(\langle aexp \rangle \langle aexp \rangle^*\right)$$
$$\langle prim \rangle = \{ \ + \ , \ - \ , \ / \ , \ * \ , \ = \ \}$$

Розширена граматика цільової CPS-форми:

$$\langle aexp \rangle ::= \left(\lambda \left(\langle var \rangle^*\right)\langle cexp \rangle\right)$$
$$| \langle var \rangle$$
$$| \#t \ | \# f$$
$$| \langle number \rangle$$
$$| \langle string \rangle$$
$$| (void)$$

$$\langle cexp \rangle ::= (if\ \langle aexp \rangle\ \langle cexp \rangle \langle cexp \rangle)$$
$$| (setthen!\ \langle var \rangle\ \langle aexp \rangle\ \langle cexp \rangle)$$
$$| \left(letrec\ \left(\left[\langle var \rangle\ \langle aexp \rangle \right]^* \right) \langle cexp \rangle \right)$$
$$| \left(\left(cps\ \langle prim \rangle \right) \langle aexp \rangle^* \right)$$
$$| \left(\langle aexp \rangle \langle aexp \rangle^* \right)$$

$$\langle prim \rangle = \{\ +\ ,\ -\ ,\ /\ ,\ *\ ,\ =\ \}$$

Для реалізації трансформації над розширеною граматикою до вже наявних додається нова функція перетворення $TK^* : expr^* \times \left(aexp^* \to cexp \right) \to cexp$, яка виконує трансформацію перетворення *TK* для послідовності вхідних виразів та функцій вхідної мови. Також до синтаксису атомарних виразів додаються типи, примітивні значення та обробники виключних ситуацій. Поява останніх призводить до того, що фактично тепер функціональні об'єкти розширюваної мови мають приймати два продовження – одне для обробки значення і друге для обробки виключної ситуації, що призводить до появи додаткового галуження шляху передачі контролю між функціями та продовженнями.

Розширені функція перетворення *TC* та *TK* мають наступний вигляд:

$$. \quad : TK \to expr \to k \to cexp$$
$$[\![\lambda(\_)]\!] = [\![k\ M\ (expr)]\!]$$
$$[\![(?\ symbol\ ?)]\!] = [\![k\ M\ (expr)]\!]$$
$$[\![(f\ e)]\!] = \left[\!\!\left[\begin{pmatrix} genusym() \leftarrow \$rv \\ \left(k\ (\$rv)\ (k\ \$rv) \right) \leftarrow cont \end{pmatrix} \right]\!\!\right]$$
$$\left[\!\!\left[T\ f\ \left(\lambda(\$f) \right) \left(T\ e\ \left(\lambda(\$e) (\$f\ \$e\ cont) \right) \right) \right]\!\!\right]$$

$$. : TC \to expr \to c \to cexp$$

$$[\![(?\ aexp\ ?)]\!] = [\![(cont\ M\ \ (expr))]\!]$$

$$[\![\{\ expr\ \}]\!] = TC\ \ (expr\ cont)$$

$$[\![\{\ expr\ expr*\}]\!] = TK(expr\ \lambda(\_)TC([\![\{expr*\}\ cont]\!])$$

$$`if\ cexp\ texp\ fexp\ = \left[\!\!\left[\begin{array}{l}\lambda(genksym() \leftarrow \$k) \\ [\![TK\ cexp\lambda(aexp) \\ [\![if\ \ aexp\ TC(texp\ \$k)TC(\ fexp\ \$k)]\!]\end{array}\right]\!\!\right]$$

$$[\![`(set!\ var\ expr)]\!] = TK(expr\ \lambda(aexp)\left[\!\!\left[\begin{array}{l}`set-then! \\ [\![var\ aexp\ cont(void)]\!]\end{array}\right]\!\!\right]$$

$$[\![`(and\ p\ \ (?\ prim\ ?))\ \ `es\ldots]\!] = TK^*(es\ \lambda(\$es)[\![(cps\ `p)\ \ `\$es\ `cont]\!]$$

2.2. Контифікація функціональних об'єктів

Розглянуті CPS-перетворення створюють синтаксичне розмежування між функціями та локальними продовженнями. Перші зазвичай компілюються у вигляді виділених на купі анонімних замикань або іменованих функцій, в той час як останні завжди можуть бути представлені у вигляді вбудованого коду з застосуваннями продовжень, виконаних за допомогою операторів безумовного переходу. Виходячи з цього для досягнення максимальної ефективності використання купи та мінімізації кількості викликів функцій бажано перетворити функції в продовження. Цей процес отримав назву контифікація.

Функції можуть бути контифіковані, коли вони завжди повертають контроль виконання в однакову початкову точку. Розглянемо наступний код, записаний у формальному синтаксисі, означеному в попередньому розділі:

$$let\ fun\ f(x) = \ldots$$

$$in\ g\ \ (case\ d\ of\ \ in_1\ d_1\ \ => \ f\ \ y\ \mid\ in_2\ d_2\ \ => \ f\ \ d_2)$$

Якщо функція *f* взагалі повертає значення, вона повинна передати контроль функції *g*. В даному випадку виконання цієї умови є очевидним, але для більш складних прикладів перевірка цієї умови стає окремою задачею. Тепер розглянемо його CPS-перетворення:

$letval\ f = (\lambda\ k\ x. \cdots k \cdots)\ in$

$letcont\ k_0\ \omega = g\ r\ \omega\ in$

$letcont\ j_1\ d_1 = f\ k_0\ y\ in$

$letcont\ j_2\ d_2 = f\ k_0\ d_2\ in$

$case\ d\ of\ j_1\ [\ ]\ j2$

Зрозуміло, що функції *f* завжди передається однакове продовження k_0 – і тому, якщо вона не містить розгалуджень, вона повинна повернутися через k_0 і таким чином передати потік керування функції *g*. Ми можемо перетворити *f* в локальне продовження наступним чином:

$letcont\ k_0\ \omega = g\ r\ \omega\ in$
$letcont\ j\ x = \cdots k_0 \cdots\ in$
$letcont\ j_1\ d_1 = j\ y\ in$
$letcont\ j_2\ d_2 = j\ d_2\ in$
$case\ d\ of\ j_1\ [\ ]\ j2$

За допомогою таких перетворень було досягнене наступне:

1. Функцію *f* замінено на продовження *j* з подальшим видаленням зворотного продовження в декларації і викликах функції.
2. В тілі функції *f* замінено аргумент *k* на функціональну структуру k_0.
3. Продовження *j* переміщено в область видимості k_0.

Для контифікації програм, що складаються з послідовностей рекурсивних функцій вищого порядку використовується дерево домінаторів графу викликів програми. Флют та Вікз в своєму дослідженні дерев домінаторів [7] показують, що такий алгоритм є оптимальним: жодні функції,

які можуть бути замінені на продовження, не залишаються в коді програми після застосування перетворення. Запропонований аналіз на основі дерев домінаторів може бути адаптований до отриманої CPS-форми, що значно спрощує процес перетворення, оскільки всі визначення та використання функцій мають іменовані продовження (Флют та Вікз використовують іменовані продовження лише для викликів, які не є хвостовими). Застосування трансформації до функцій вищого порядку також спрощується, але за наявності функцій в якості об'єктів першого класу та загальної блокової структури вхідного коду трансформація стає значно складнішою для формального опису.

В цьому випадку більш ефективним є підхід, коли замість комплексного аналізу і однопрохідного перетворення описані вище атомарні трансформації застосовуються поступово до тих пір, поки їх застосування не призводить до відсутності змін в оброблюваній програмі. Далі розглядаються особливості застосування трансформації спочатку для нерекурсивних функцій вхідної мови, потім проводиться генералізація способу для рекурсивних функцій і функцій вищого порядку, а також виконується порівняння означеного способу перетворення з перетворенням на основі дерев домінаторів.

В абстрактній нетипізованій мові λ_{CPS}^{U} без рекурсії визначення функцій, придатних до контифікації, є тривіальним: такими є всі функції, які викликаються з однаковим продовженням. Визначається наступне перетворення:

$$
\begin{aligned}
&CONT\ (f\ \ not\ free\ in\ C,\ D\ and\ D\ minimal):\\
&letval\ f = \lambda kx.K\ in\ C\left[D\left[f\ k_0 x_1, \ldots, f\ k_0 x_n\right]\right],\\
&\rightarrow\\
&C\left[letcont\ j\ x = K\left[k_0 / k\right] in\ D\left[j\ x_1, \ldots, j\ x_n\right]\right]
\end{aligned}
$$

де C – контекст з однією точкою входу, а D – мінімізований контекст з багатьма точками входу, формалізацію яких не наводиться. Перетворення *CONT* виконує три операції:

1. Функцію *f* замінюється продовженням *j*, кожний виклик початкової функції замінюється викликом продовження.
2. В тілі *K* функції *f* замінено формальний параметр *k* на загальне продовження k_0.
3. Створене продовження *j* вноситься в область видимості загального продовження k_0.

Контекст *D* з декількома точками входу є мінімальним контекстом, що охоплює всі виклики *f*. Це гарантує виконання умови про знаходження продовження *j* в області видимості k_0 після завершення роботи трансформації. Аналіз коректності перетворення тривіальний: необхідно перевірити видимість аргументів всіх викликів загального продовження. Повторне застосування цього перетворення в кінцевому випадку призводить до отримання оптимального результату. Приклад:

$$letval\ h = \lambda\ k_h x_h \cdots in$$
$$letval\ g_1 = \lambda\ k_1 x_1 \cdots h\ k_1 z_1 \cdots k_1 z_8 \cdots in$$
$$letval\ g_2 = \lambda\ k_2 x_2 \cdots h\ k_2 z_2 \cdots in$$
$$letval\ f = \lambda\ k_f x_f \cdots g_1\ k_f z_3 \cdots g_2\ k_f z_4 \cdots g_2\ k_f z_5 \cdots in$$
$$letval\ m = \lambda\ k_m x_m \cdots f\ j_1 z_6 \cdots f\ j_2 z_7\ in \cdots$$

Можемо бачити, що функції g_1 та g_2, але не функція *h* завжди передають контроль одному продовженню k_f. Застосуємо перетворення *CONT* до цих функцій:

$$
\begin{array}{l}
letval\ h = \lambda\ k_h x_h \cdots in \\
letval\ \ f = \lambda\ k_f x_f \\
\left(\begin{array}{l} letcont\ kg_1 x_1 = \cdots h\ k_f z_1 \cdots k_f z_8 \cdots in \\ letcont\ kg_2 x_2 = \cdots h\ k_f z_2 \cdots kg_1 z_3 \cdots kg_2 z_4 \cdots kg_2 z_5 \cdots \end{array} \right) in \\
letval\ \lambda m\ k_m x_m \cdots\ f\ \ j_1 z_6 \cdots f\ \ j_2 z_7 = in \cdots
\end{array}
$$

В результаті цього функція h також стає придатною до контифікації, оскільки тепер вона завжди передає контроль продовженню k_f:

$$
\begin{array}{l}
letval\ \ f = \lambda\ k_f x_f \\
\left(\begin{array}{l} letcont\ kh\ x_h = \cdots in \\ letcont\ kg_1 x_1 = \cdots kh\ z_1 \cdots k_f z_8 \cdots in \\ letcont\ kg_2 x_2 = \cdots khz_2 \cdots kg_1 z_3 \cdots kg_2 z_4 \cdots kg_2 z_5 \cdots \end{array} \right) in \\
letval\ \lambda m\ k_m x_m \cdots\ f\ \ j_1 z_6 \cdots f\ \ j_2 z_7 = in \cdots
\end{array}
$$

Узагальнення цього перетворення для рекурсивних функцій та продовжень потребує введення додаткових абстрактних операцій. Припустимо, у нас є λ_{CPS}^{T} терм форми

$$
\begin{array}{ll}
letfun & f_1 k_1 h_1 x_1 = K_1 \\
\cdots & f_n k_n h_n x_n = K_n \\
in\ K. &
\end{array}
$$

Множина функцій $F \subseteq \{ f_1, \ldots, f_n \}$ може бути контифікована одночасно та позначається $Contifiable\ (F)$, якщо є певна пара продовжень k_0 і h_0 така, що кожна поява $f \in F$ у вхідній програмі є або хвостовим викликом в межах *F*, або викликом з аргументами k_0 та h_0. Кожна функція множини *F* в кінцевому випадку передає виконання продовженню k_0, або в разі виникнення виключної ситуації викликає обробник h_0. До моменту виклику продовжень k_0 і h_0 керування може передаватись між функціями множини *F* рекурсивно. Таких підмножин *F* послідовностей пов’язаних функцій може бути багато;

робиться припущення, що множина F є сильно-зв'язаною з хвостовими викликами в якості шляхів з’єднання, або однокомпонентною множиною без хвостових викликів. Тоді для заданого терму існує унікальне часткове розбиття множини , що задовольняє $Contifiable\ (-)$.

Нехай $F = \{f1, \ldots, fm\}$. Введемо наступне перетворення для функціональних виразів:

$$(f\ k\ h\ x)^* = \begin{cases} j_i\ x & якщо\ f = f_i \in F \\ f\ k\ h\ x & в\ інших\ випадках \end{cases}$$

і застосуємо його до всіх вхідних термів.

За умови існування $Contifiable\ (F)$ розглядаються два можливих випадки:

1. Всі застосування форми $f\ k_0\ h_0\ x$ для $f \in F$ належать терму *K*. Тоді є можливим застосування наступного перетворення, аналогічного *CONT*

$RECCONT_1\ (f_1, \ldots, f_m\ not\ free\ in\ C,\ and\ K\ minimal)$:
$letfun\ f_1 k_1 h_1 x_1 = K_1$
$\ldots \quad f_n k_n h_n x_n = K_n$
$in \quad C[K]$
$\rightarrow$
$letfun\ f_{m+1} k_{m+1} h_{m+1} x_{m+1} = K_{m+1}$
$\ldots \quad f_n k_n h_n x_n = K_n$
$$in\ C\begin{bmatrix} letcont\ j_1 x_1 = K_1^*[k_0 / k_1, h_0 / h_1] \\ \cdots \quad j_m x_m = K_m^*[k_0 / k_m, h_0 / h_m] in\ K^* \end{bmatrix}$$

2. В іншому випадку певні застосування форми $f\ k_0\ h_0\ x$ для $f \in F$ знаходяться в тілі однієї з функцій $f_n \notin F$.

$RECCONT_2\ (f_1,\ldots,f_m\ not\ free\ in\ C,\ and\ K\ minimal)$:

$letfun\ f_1k_1h_1x_1 = K_1$

$\quad f_{n-1}k_{n-1}h_{n-1}x_{n-1} = K_{n-1}$

$\ldots \quad f_nk_nh_nx_n = C[K_n]$

$in\ K$

$\rightarrow$

$letfun\ f_{m+1}k_{m+1}h_{m+1}x_{m+1} = K_{m+1}$

$\ldots \quad f_nk_nh_nx_n = K_n$

$$in\ C\left[\begin{array}{l} letcont\ j_1x_1 = K_1^*[k_0 / k_1, h_0 / h_1] \\ \cdots \quad j_mx_m = K_m^*[k_0 / k_m, h_0 / h_m]in\ K^* \end{array}\right]$$

Для порівняння запропонованого Флютом та Вікзом алгоритму контифікації на основі дерев домінаторів з наведеним необхідно побудувати континуаційний граф потоку керування для вхідної програми, вузлами якого є змінні продовжень з виділеним кореневим елементом. Для кожної функції *f*, що повертає продовження *k*, якщо f передається в якості значення першого порядку, створюється ребро від кореневого елементу до *k*; інакше для кожного застосування вигляду $f\ j\ x$ створюється ребро від *j* до *k*. Для кожного локального продовження *k* створюється ребро від кореневого елементу до *k*.

Застосування нерекурсивного перетворення *CONT* створює ефект злиття елементів графу (рис 2.1):

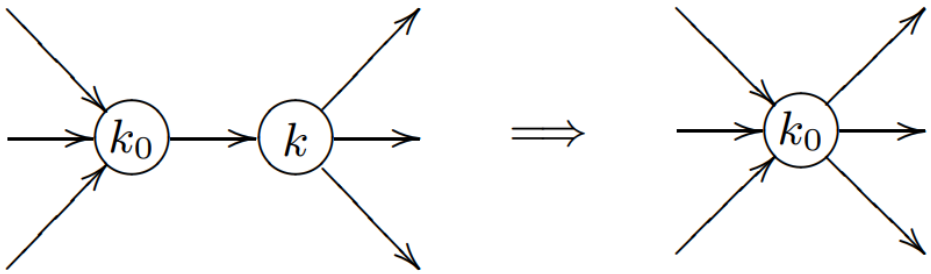


Рис. 2.1. Застосування трансформації *CONT*

Рекурсивні перетворення $RECCONT_1$ та $RECCONT_2$ мають схожий ефект за винятком того, що замість k маємо сильно зв'язаний компонент $\{k_1,\ldots,k_m\}$.

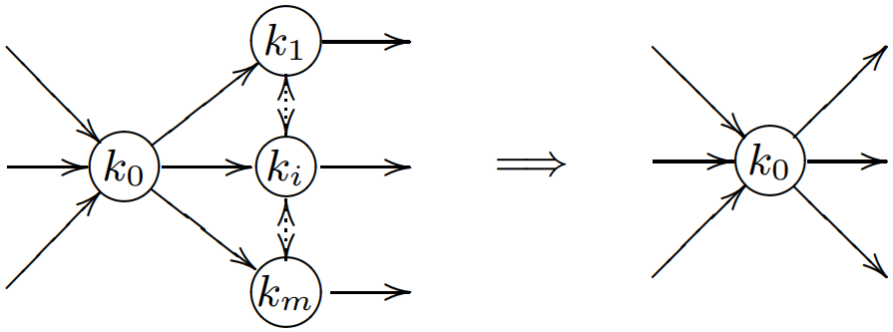


Рис. 2.2. Застосування трансформації $RECCONT_n$

І навпаки, будь-яка частина графа потоку керування, що відповідає лівій частині цієї діаграми відповідає підмножинам функцій, до яких можуть застосовуватися перетворення $RECCONT_1$ та $RECCONT_2$. Очевидно, що зазначений алгоритм вичерпного перетворення завершує роботу, коли граф, що зменшується із застосуванням кожного перетворення, перестає містити конструкції, наведені на діаграмах. Алгоритм, описаний Флютом та Вікзом, контифікує *k,* якщо над ним строго домінує певне продовження *j,* чиїм безпосереднім домінатором є кореневий елемент. Можна показати, що якщо граф містить таку пара вузлів *j* і *k*, то певний його підграф відповідає наведеній на діаграмі конструкції. Отже, зазначене вичерпне перетворення дає результат, ідентичний отриманому за допомогою оптимального алгоритму, що базується на деревах домінаторів.

2.3. Застосування оптимізації хвостових викликів функцій до отриманої CPS форми

Після зведення вхідної програми до CPS-форми стає можливим застосування до отриманого коду алгоритму оптимізації хвостових викликів. Ключовою операцією перетворення є обгорнення викликів функціональних

продовжень у спеціальні структури, що називаються thunk. Thunk є функціональним об’єктом, що являє собою відкладений виклик функції, який може зберігатись для подальшої передачі контролю виконання між відкладеними обчисленнями таким чином, що їх виконання не буде призводити до виснаження стеку викликів. Загальний алгоритм трансформації виглядає наступним чином:

1. Замінити всі анотовані виклики продовжень на thunk-структури.
2. Створити об’єкт *TC* – продовження верхнього рівня, якому будуть передавати керування інтерфейсні функції.
3. Обгорнути трансформовану програму в ітеративний опитувач.
4. Оголосити структуру даних для збереження необхідного для подальшого налагодження проміжного стану.
5. Додати відкладені виклики функціональних об’єктів в тіло ітеративного опитувача.

В результаті застосування цих трансформацій до CPS-форми внутрішнього представлення отримуємо результуючу програму, яка зберігає всю необхідну для налагодження інформацію про виклики функцій та вирішує проблему виснаження стеку викликів, оскільки глибина рекурсії тепер обмежується лише загальною кількістю наявної пам’яті. Крім того, отримана програма буде демонструвати значно ефективнішу роботу з кеш-пам’яттю, оскільки під час її виконання екстерналізований стек викликів стає придатним до кешування.

3. СТРУКТУРА СИСТЕМИ ПОБУДОВИ CPS-ФОРМИ ТА ОПТИМІЗУЮЧОГО ТРАНСЛЯТОРА

Система побудови CPS-форми внутрішнього представлення та TCO-оптимізації реалізована у формі клієнт-серверного додатку, в якій клієнт являє собою веб-застосунок, а сервер виконує всі перетворення вхідного коду. Систему організовано за допомогою архітектурного підходу REST, відповідно, сервер не зберігає жодного стану, за керування станом повністю відповідає клієнтський застосунок. Сервер містить три вхідні точки, які відповідають його модулям:

Перший модуль призначений для початкової побудови дерева розбору вхідного коду. На вхід цього модуля подається програмний код у вигляді лінійної текстової послідовності, з якого конструюється дерево розбору, представлене у форматі даних JSON. Цей модуль також отримує інформацію про наявність синтаксичних помилок у вхідному коді.

Другий модуль відповідає за зведення синтаксичного дерева до CPS-форми, збереження інформації про стек викликів та мінімізацію отриманої форми внутрішнього представлення. Приймає на вхід дерево розбору, отримане в результаті роботи першого модуля, на основі якого будує мінімізовану CPS-форму вхідної програми.

Третій модуль застосовує алгоритм оптимізації хвостових викликів та генерує вихідну програму. На вхід приймає мінімізовану CPS-форму програми та структури даних, що репрезентують стек виконання початкової програми. Результатом роботи є синтаксично еквівалента програма мовою JavaScript з оптимізованими хвостовими викликами.

Перевагою такої організації системних модулів є незалежність від форми представлення вхідної програми (одиничні функції або повний програмний проект) ізольованість кожного перетворення від інших, що захищає чутливі

перетворення від помилок у вхідному коді та дозволяє виконання генерації коду для будь-якого проміжного представлення програми.

3.1. Модуль побудови дерева розбору вхідного коду

Даний модуль викликається з контролеру завантаження файлів проекту на сервер після отримання відповідного запиту від користувача за допомогою веб-інтерфейсу. Після отримання контролером повного проекту і перевірки файлів на відповідність розширень для кожного програмного JavaScript файлу викликається функція первинної побудови синтаксичного дерева. На цьому етапі збирається також інформація про помилки у вхідному коді, яка разом з отриманим деревом розбору подається на вхід наступним трансформаціям.

Головною функцією модуля є асинхронна функція parse, яка викликається з інтерфейсного методу контролера uploadFiles для програмного коду, що міститься у завантажених файлах. Ця функція будує дерево розбору, а також оголошує функцію-обробник синтаксичних помилок вхідного коду, яка буде додавати інформацію про помилки до відповіді. Застосування такого підходу до обробки введення некоректного коду дозволяє реалізувати часткову побудову дерев розбору для вхідної проекту, що в подальшому зробить можливим часткове застосування перетворень з метою отримання максимально оптимізованого коду, зважаючи на некоректність певних вхідних елементів.

Після завершення роботи функція повертає контролеру завантаження дерево розбору і список всіх помилок у вхідному коді. Контролер в свою чергу викликає проміжний генератор коду для файлів, що не містять помилок, генеровані проміжні файли без помилок об’єднуються в список з вхідними файлами, що містять помилки.

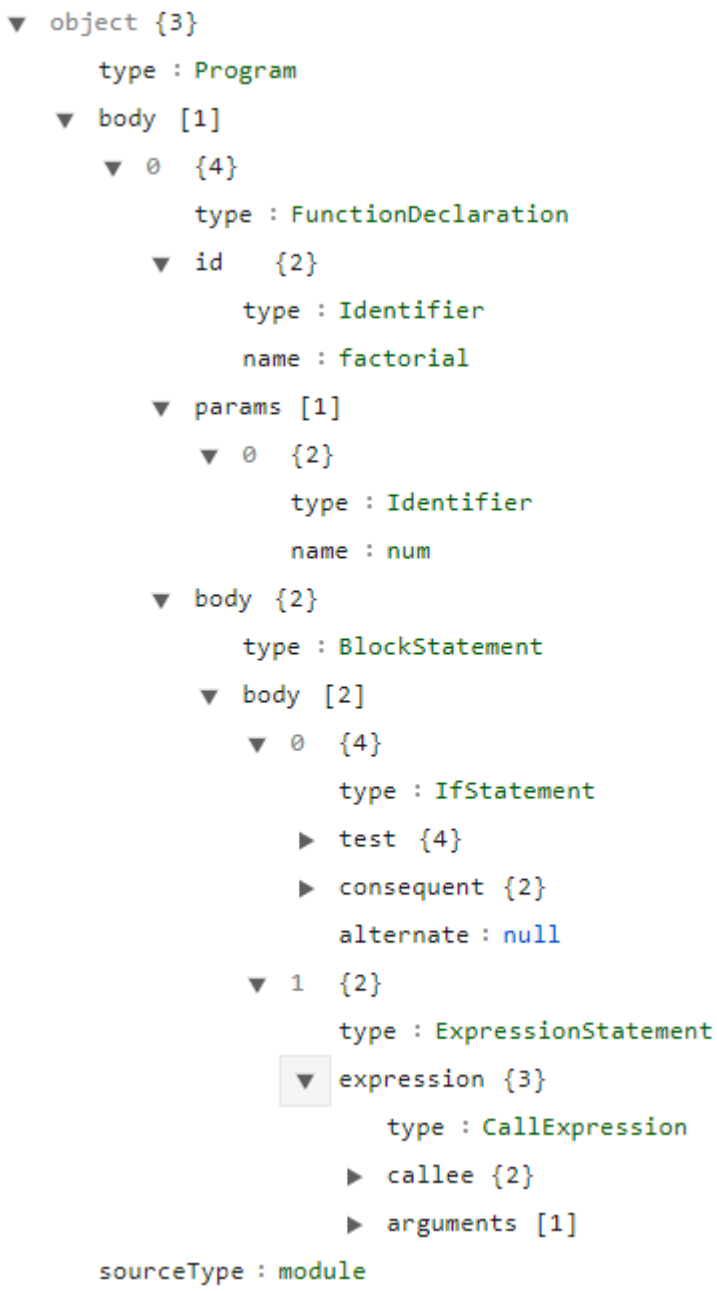


Рис. 3.1. Дерево розбору, що відповідає програмі обчислення факторіалу

З отриманого синтаксичного дерева генерується проміжний код, що віддається клієнту для початкового відображення разом з анотованим списком помилок. Формат дерева розбору, наведений на рис. 3.1 відповідає формату вхідного параметра інтерфейсної функції наступного модуля.

3.2. Система побудови CPS-форми внутрішнього представлення

Модуль побудови CPS-форми внутрішнього представлення приймає на вхід дерево розбору початкової програми та інформацію про помилки у файлах. Саме цей модуль є центральним в роботі системи, адже саме в ньому визначені функції трансформацій та порядок їх застосування до елементів вхідного коду. Основною синтаксичною одиницею для побудови CPS-форми є стрілочна функція, до якої зводяться всі функції вхідної програми. В граматиці мови JavaScript такі функції відповідають нетерміналу «anonymousFunction» (рис. 3.1).

```
anoymousFunction
    : functionDeclaration
    | Async? Function '*'? '(' formalParameterList? ')' '{' functionBody '}'
    | Async? arrowFunctionParameters '=>' arrowFunctionBody
    ;

arrowFunctionParameters
    : Identifier
    | '(' formalParameterList? ')'
    ;

arrowFunctionBody
    : singleExpression
    | '{' functionBody '}'
    ;
```

Рис. 3.2. Правило граматики, що відповідає стрілочним функціям

Для реалізації абстрактного алгоритму перетворення реалізовано наступні функції, що працюють з деревом розбору вхідної програми.

1. Transform. Є інтерфейсною функцією центрального модуля, що викликається контролером. Приймає на вхід дерево розбору програми, оголошує множину необхідних функцій-відвідувачів та викликає функцію traverse для синтаксичного дерева. Після завершення функції traverse, яка повертає трансформоване дерево розбору, передає керування контролеру.

2. Traverse. Є головною функцією обходу дерева розбору, що викликає наявні функції-відвідувачі для відповідних елементів дерева та піддерев. В разі наявності виклику функції replaceWith над результатом роботи функції-відвідувача, повертає змінений елемент дерева, за рахунок чого виконуються недеструктивні перетворення дерева розбору.
3. FuncVisitor, TopLevelVisitor, BodyVisitor, CallExpVisitor. Функції-відвідувачі конкретних елементів оброблюваного дерева, що реалізують безпосередньо правила перетворення продукцій граматики мови. Можуть містити як безпосередньо інструкції заміщення елементів для кінцевих правил, так і виконувати додатковий аналіз елементів дерева.
4. GenerateIdentifier, generateLambda. Функції, що генерують нові унікальні ідентифікатори та лямбда-вирази у вигляді елементів дерева розбору, що використовуються для створення нових продовжень функцій під час перетворення в CPS-форму.
5. Допоміжні функції, тобто функції-предикати вигляду isSomeCondition, які використовуються для перевірки певних властивостей елементів дерева, функції пошуку заданих елементів в піддеревах, функції генерації унікальних імен ідентифікаторів тощо.

Загальний механізм роботи цього модуля працює наступним чином: виконується обробка кожного елемента дерева типу FunctionDeclaration, яка генерує унікальне продовження для цієї функції, виконує пошук та заміну інструкцій повернення значення цієї функції на виклики генерованого продовження з результатом функції в якості вхідного аргументу, після чого послідовно викликає функції обробки внутрішнього та зовнішнього контексту цієї функції, які рекурсивно викликають обробники всіх інших функцій.

3.3. Система оптимізації хвостових викликів та збереження стеку викликів

Даний модуль містить завершальне перетворення, що виконує підняття викликів продовжень, додає виклики до загального продовження *TC* для точок входу програми, оголошує ітеративний опитувач, в який додає реструктуризовані виклики продовжень, а також створює об'єкт, що зберігає інформацію, необхідну для коректної роботи налагоджувача.

3.4. Інтерфейс користувача

Інтерфейс користувача являє собою одностраничний веб-додаток, що окрім функцій відображення та керування відповідає також за збереження проміжного стану перетворень вхідного коду.

На початку роботи користувач бачить вікно завантаження файлу, в яке необхідно завантажити або файл з розширенням .js, що містить вхідну програму, або zip-архів, що містить проект мови JavaScript, який може складатись з декількох файлів, організованих у директорії.

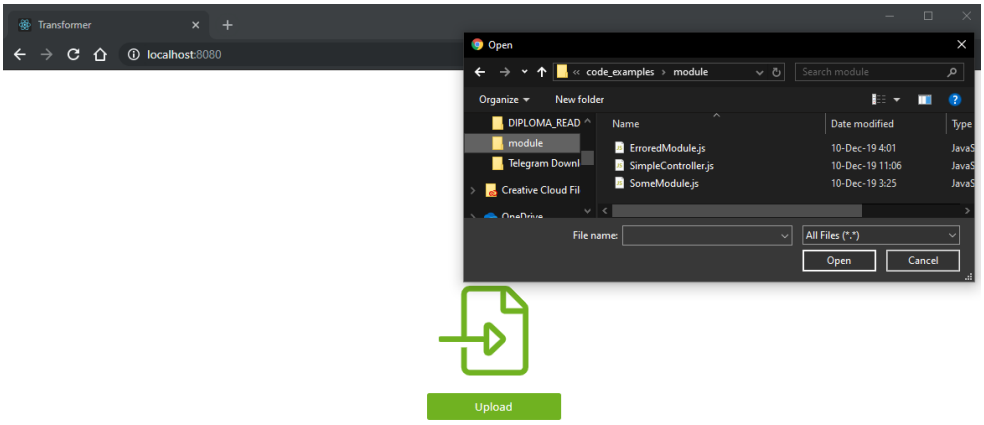


Рис. 3.3. Початкове вікно завантаження файлу

Після завершення завантаження файлів та виконання первинної побудови дерева розбору відображається основне вікно додатку, що містить

відображення вхідного коду у вигляді вкладок з текстом вхідних файлів, кнопки запуску перетворення для поточного файлу та всіх файлів проекту, область відображення отриманого коду. Повідомлення про помилки у вхідних файлах виводяться у вигляді спливаючих повідомлень, що містять назву файлу, тип помилки та детальне повідомлення про помилку. Вкладки з файлами, що містять помилки, виділяються окремим кольором.

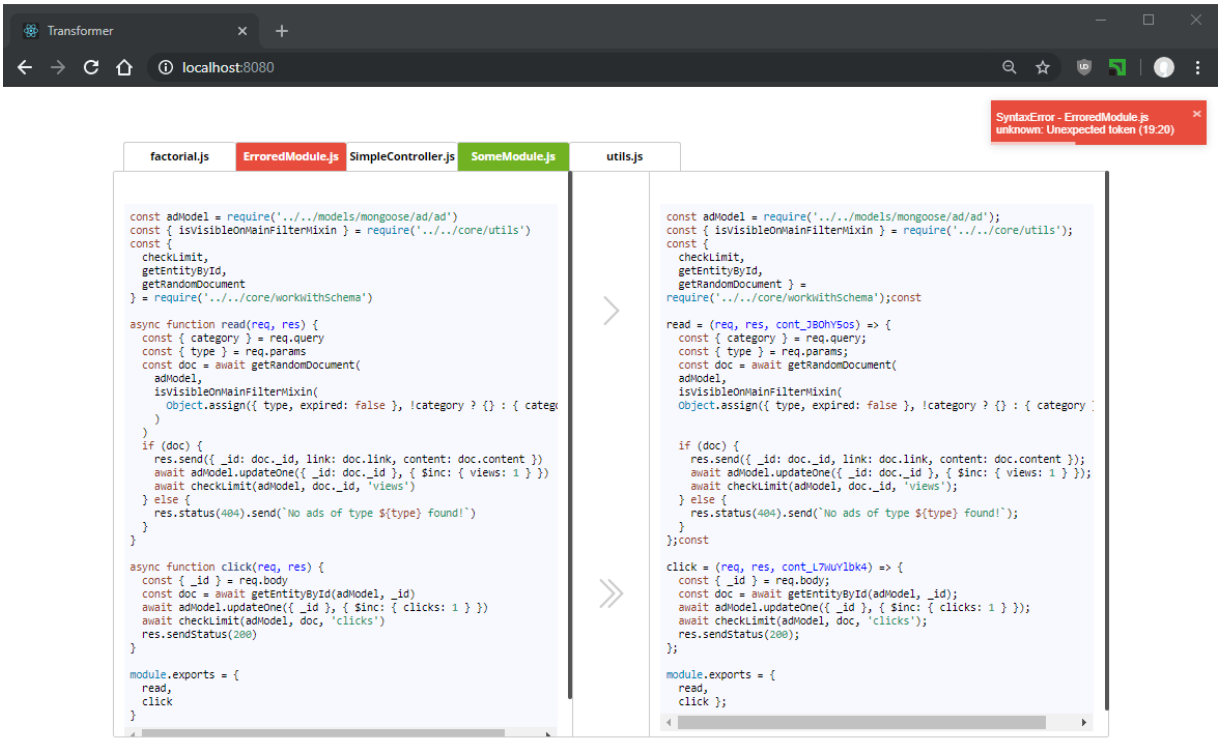


Рис. 3.4. Головне вікно додатку

В лівій області екрану доступний для перегляду вхідний код компоненту SomeModule.js, у верхній області виведені вкладки інших файлів проекту, причому в модулі ErroredModule.js міститься синтаксична помилка, а саме неочікуваний токен на позиції 19 рядка і 20 стовпчика, про що свідчить виділення вкладки цього модуля червоним кольором і наявність спливаючого повідомлення в правому верхньому куту екрану. Користувач може обрати режим перетворення, що буде застосований до файлів проекту:

1. Перетворення одиничного файлу. Для виконання такого перетворення необхідно натиснути одиничну стрілку в центрі екрану. Після цього в правій частині екрану з’явиться трансформована версія вхідного коду програми поточного файлу. Інші файли залишаться без змін.

2. Перетворення всіх файлів проекту. Для застосування перетворення для всіх файлів проекту необхідно натиснути здвоєну стрілку в центрі екрану, після чого трансформацію буде застосовано для всіх коректних файлів проекту послідовно. Код файлів, що містять помилки, змінено не буде.

4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ОТРИМАНОГО КОДУ ТА ТЕСТУВАННЯ СИСТЕМИ

4.1. Тестування системи

Метою проведення тестування розробленої системи є перевірка коректності роботи основних її функцій, до яких належать:

- отримання вхідних файлів, придатних до перетворення;
- обхід дерева розбору;
- перетворення елементів дерева розбору;
- визначення властивостей елементів дерева розбору;
- пошук елементів дерева розбору.

Вхідними даними для тестування першого етапу є проект цільової платформи Node.js, що містить файли з використанням різноманітного функціоналу мови JavaScript, в тому числі з елементами, непридатними до перетворення, а також елементами, що містять синтаксичні помилки. Подальші елементи проміжного представлення, необхідні для тестування інших функцій, отримуються з програмних елементів вхідного проекту, придатних до аналізу та перетворення. Модульне тестування системи виконується за допомогою фреймворку Jest. Модульними тестами покритий код основних модулів серверу, а саме модулю побудови CPS-форми та виконання оптимізації хвостових викликів. Також написано модульні тести для перевірки коректності роботи допоміжних функцій генерації нових елементів синтаксичного дерева та функцій-предикатів. Результати модульного тестування системи наведено в табл. 4.1.

Таблиця 4.1

Результати тестування функцій системи

№	Назва функції	Вхідні дані	Очікуваний результат	Отриманий результат	Чи пройдено тест
Т1.	controller.upload	Файл з кодом .js, не містить помилок у вхідному коді	files: [{ name: String, text: String }]	files: [{ name: String, text: String }]	+
Т2.	controller.upload	Zip архів з проектом, не містить помилок у вхідному коді	files: [{ name: String, text: String }, …]	files: [{ name: String, text: String }, …]	+
Т3.	controller.upload	Файл з кодом .js, містить помилки у вхідному коді	files: [{…}], errors: [{…}, …]	files: [{…}], errors: [{…}, …]	+
Т4.	controller.upload	Zip архів з проектом, містить помилки у вхідному коді	files: [{…}, …], errors: [{…}, …]	files: [{…}, …], errors: [{…}, …]	+
Т5.	controller.upload	Файл з довільним вмістом без розширення .js	Error: please provide .js files only	Error: please provide .js files only	+
Т6.	controller.upload	Zip архів з проектом, містить не-js файли	files: [{…}, …], errors: [{…}, …]	files: [{…}, …], errors: [{…}, …]	+

Продовження табл. 4.1

№	Назва функції	Вхідні дані	Очікуваний результат	Отриманий результат	Чи пройдено тест
T1.	controller.transform	files: [{ name: String, text: String }]	Object<AST>	Object<AST>	+
T2.	controller.transform	files: [{ name: String, text: String }, …]	[Object<AST>, …]	[Object<AST>, …]	+

T3.	singleFileComposer	{
		name: String,
		text: String
		}
			Object<AST>	Object<AST>	+

T4.	core.transform	Object<AST>	Object<CPS>	Object<CPS>	+
T5.	core.traverse	Object<CPS>	TCO-optimized Object<CPS>	TCO-optimized Object<CPS>	+
T6.	core.codegen	TCO-optimized Object<CPS>	{ text: String }	{ text: String }	+

4.2. Підходи до порівняльного аналізу отриманого коду

Головними вимогами до коду, що отримується в результаті перетворення, окрім його коректності, є його стійкість до виключної ситуації переповнення стеку викликів, а також можливість покрокового виконання з доступом до стеку викликів. Виконати це порівняння можна за допомогою запуску вхідного і вихідного коду у вбудованому засобі налагодження інтегрованого середовища розробки Visual Studio Code. Тестування необхідно виконувати перш за все для функції, що має велику глибину рекурсії, яка призводить до переповнення стеку викликів.

Тестування швидкодії отриманого коду має демонструвати незначне погіршення швидкодії, пов'язане з додатковим перемиканням контексту введеними продовженнями та додаткові операції звернення до загального продовження. Тестування необхідно проводити для різних наборів вхідних даних: програм зі звичайними рекурсивними функціями, програм з великою вкладеністю викликів функцій без використання рекурсії, програм з перехресною рекурсією. Крім цього слід провести тестування системи на модулях, що не містять придатного для перетворень коду, для яких система не повинна вносити жодних змін. Отриманий код з метою перевірки працездатності слід подати на виконання рушію V8 версії не нижче 10.0.0. Для порівняння ефективності роботи початкового і отриманого коду слід використовувати npm-модуль perf.

Перевіряти можливості роботи зі стеком викликів та покрокового виконання результуючої програми необхідно за допомогою інструменту налагодження node inspector, інтегрованого у будь-яке середовище розробки. Вихідна програма має дозволяти отримання проміжних значень для будь-якого елементу стеку викликів та вільний перехід між розташованими в довільних місцях вхідного коду точками зупинки.

4.3. Порівняння можливостей роботи з отриманим кодом

Для наочності в якості вхідних даних для цього порівняння використано синтетичну функцію f1, яка має наступний вигляд:

```
'use strict'

function f1(a) {
  // setTimeout(() => {
  if (a === 0) {
    debugger // a must be 0
    console.log(a)
    return 0
  } else {
    if (a === 49500) debugger
    return f1(a - 1)
  }
  // }, 0)
}

console.log(f1(50000))
```

Глибина рекурсії під час роботи даної функції становить 50000, що значно перевищує розмір стеку виконання. Результат її запуску без застосування оптимізації хвостовий викликів наведено на рис. 4.1.

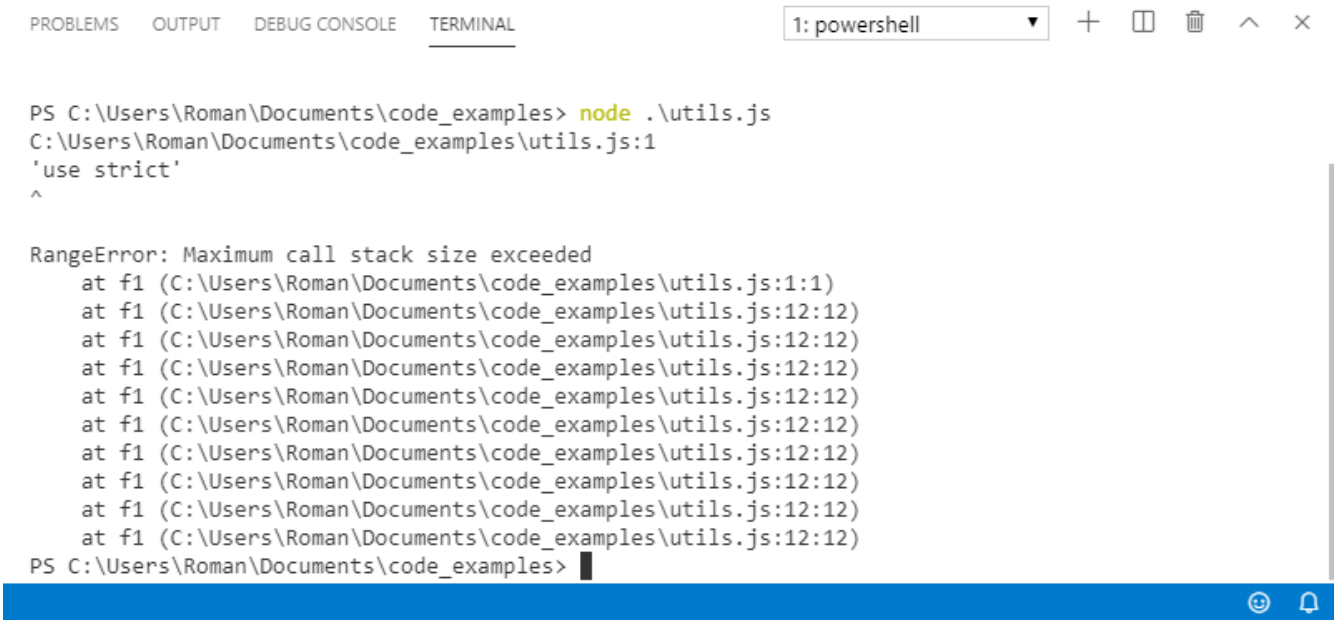


Рис. 4.1. Результат запуску початкової функції

Функція, яку отримано в результаті застосування перетворень, має наступний вигляд:

```
const f1 = (a, cont_VDiCCR1Hv) => {
  if (a === 0) {
    debugger
    console.log(a)
    cont_VDiCCR1Hv(0)
  } else { if (a === 49500) debugger
    return f1(a - 1, x_OgUbggk1x => cont_VDiCCR1Hv(x_OgUbggk1x))
  }
}
f1(50000, x_ZTKmHNmve => console.log(x_ZTKmHNmve))
```

Покрокове виконання отриманої функції та наявні елементи стеку її викликів зображено на рис. 4.2.

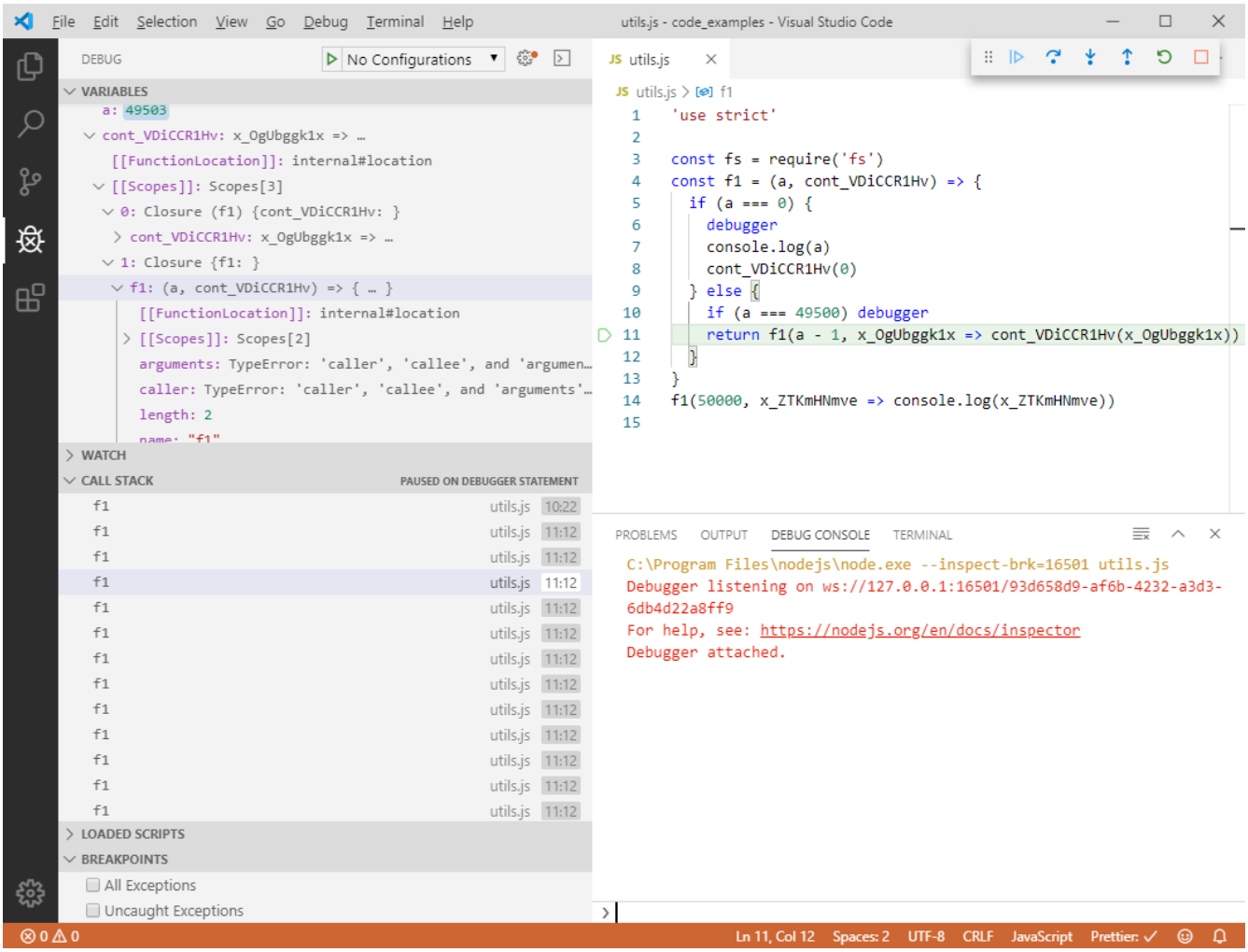


Рис. 4.2. Налагодження оптимізованої функції

Тестування роботи розробленої системи на реальних даних проводилось на вхідному коді системи керування ресурсами Cody, що є у вільному доступі, засобами Chrome Profiler. Результат роботи системи для великих файлів та проектів можна бачити на рис. 4.3.

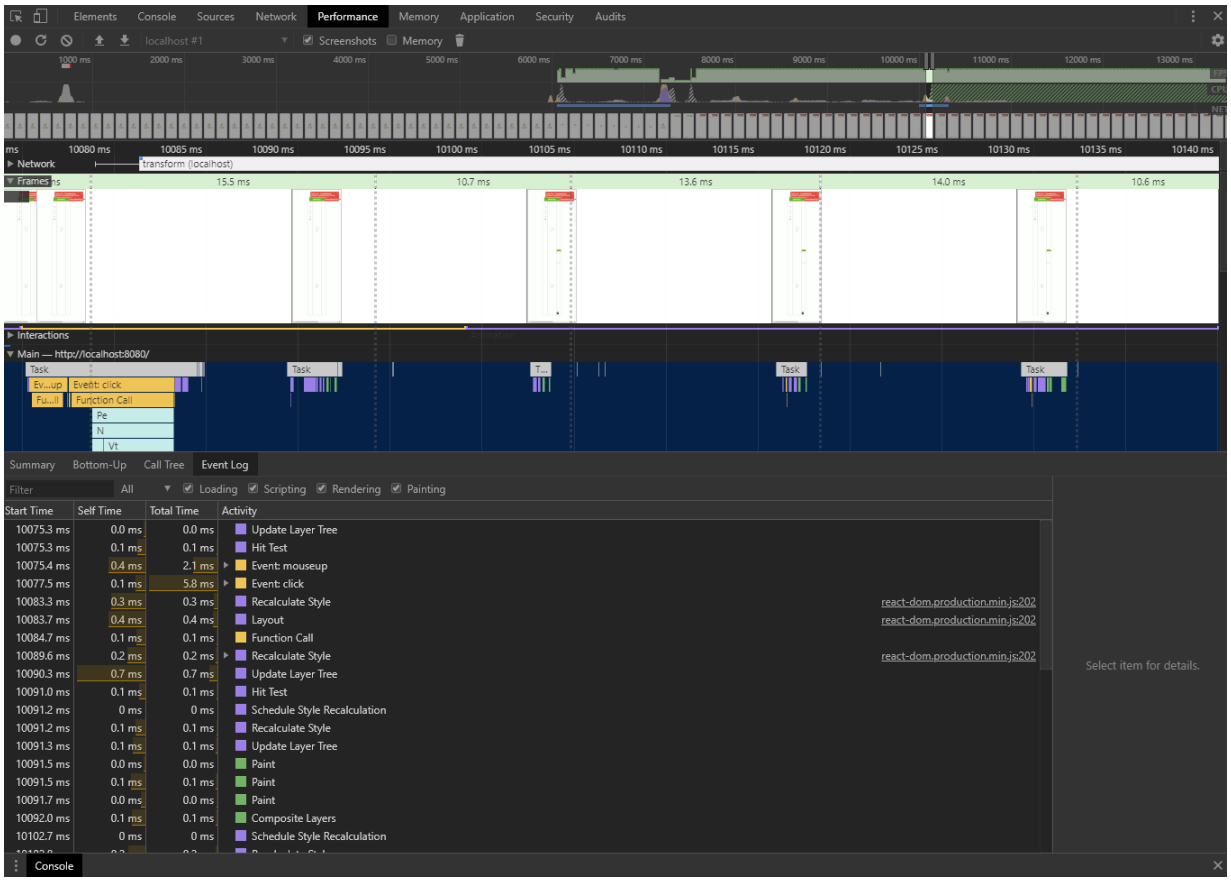


Рис. 4.3. Аналіз швидкодії модулів системи

Перевірка швидкодії отриманого коду використано npm-модуль perf. Результати виконання команди npm run perf для початкового і перетвореного коду наступні:

```
Function                    | Time (ms) | Relative
:---------------------------|----------:|--------:
f1                          |    56.231 |    40.4%
cont_VDiCCR1Hv              |    43.003 |    30.9%
f2                          |    39.874 |    28.6%
```

Даний час роботи отриманого коду повністю відповідає прогнозованій швидкодії.

5. ПОБУДОВА БІЗНЕС МОДЕЛІ

5.1. Опис проблеми

Під час розвитку інформаційних технологій і програмування як такого обсяги інформації, що оброблюється, рівень абстракцій та, відповідно, складність програмних систем, що розроблюються, невпинно зростають. Разом з цим стрімко зростають витрати людських та фінансових ресурсів, необхідних як на розробку нового, так і на підтримку і модернізацію існуючого програмного забезпечення. Цікаво, що саме останній вид робіт найчастіше виходить за рамки планованого бюджету і призводить до зриву термінів завершення проектів.

Яскравим прикладом може слугувати сфера банківського програмного забезпечення, в якій навіть сьогодні домінуючою технологією, на основі якої розроблено щонайменше 50% (80% за даними 1997 року, 60% за даними 2012) існуючого програмного забезпечення в галузі, є COBOL, мова 1959 року. За даними опитування Computerworld 2014 року 36% опитаних управлінців заявили, що вони планували міграцію своїх програмних систем на інші мови, а 25% сказали, що могли б планувати міграцію за умови її нижчої вартості.

Іншим прикладом може слугувати сфера web-розробки, яка, хоч і набагато молодша за попередньо розглянуту, все одно вже встигла набрати критичну масу так званого legacy-коду, який є застарілим і малопридатним, а отже, дорогим для підтримки і оновлення. Так, в найпоширенішій мові для розробки web-застосунків JavaScript існують фундаментальні помилки дизайну, які ніколи не будуть виправлені саме через те, що вони вже стали частиною настільки великої кількості непідтримуваних проектів, що їх виправлення призведе фактично до краху величезної частини WWW.

Розглянуті приклади об’єднують наступні проблеми: наявність великої кількості застарілого коду, який неможливо переписати з нуля і майже

неможливо (невиправдано дорого) модернізувати. Можливим вирішенням цих проблем, яке буде дешевшим, ніж наймання програмістів для ручного аналізу і переписування, є використання інструментів для автоматичного перетворення коду, тобто трансляторів.

Останнім часом зростає популярність функціонального підходу до розробки ПЗ, в тому числі в сфері web-технологій. Однією з головних проблем, пов’язаних з підтримкою функціональної парадигми у головній web-мові – JavaScript – є відсутність оптимізації рекурсивних викликів функцій. Розробники вимушені кожного разу вручну створювати відповідні оптимізації, або взагалі відмовлятися від рекурсії на користь ітеративного підходу.

Визначені проблеми узагальнено на схемі «Дерева проблем», рис. 5.1.

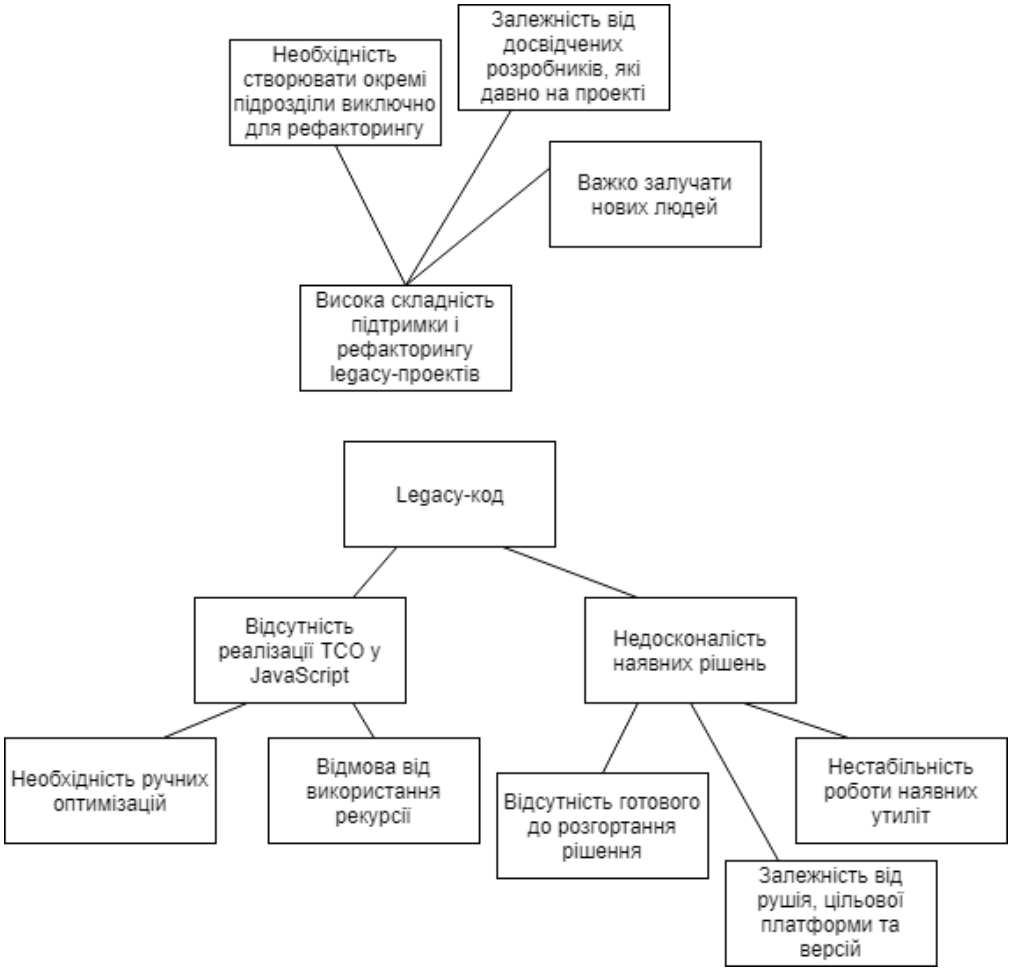


Рис. 5.1. Дерево проблем

5.2. Зацікавлені сторони

Існує декілька зацікавлених сторін у вирішенні визначених проблем.

Станом на сьогодні в індустрії web-розробки провідними компаніями є т. зв. «FANG» - Facebook, Amazon, Netflix, Google. Саме ці компанії, окрім створення і розвитку власних основних продуктів, інвестують величезні ресурси у створення сприятливої інфраструктури для web-розробників, в тому числі у розробку засобів автоматичної трансляції та оптимізації коду. Для цих компаній однією з найбільших статей витрат є саме витрати на підтримку і рефакторинг legacy-продуктів. Так, наприклад, Facebook розпочав свою роботу близько 15-ти років тому і був спочатку написаний на PHP, тому можна тільки уявити, наскільки великою є його кодова база зараз, коли він являє собою нагромадження з величезної кількості компонентів, написаних на різних мовах з використанням кардинально різних технологій, які інтегруються з центральним ядром. При цьому, якщо провести аналіз зручності використання як самого Facebook, так і його компонентів, як-то Messenger, Watch або Instagram, за критеріями часу відповіді, відсутності помилок, кількості «кліків» для виконання типової дії та іншими, то можна зрозуміти, чому Facebook настільки зацікавлені у модернізації наявної кодової бази. Приблизно така ж ситуація спостерігається і з іншими «титанами» галузі, які запустили свої продукти на початку-середині 2000-х років, коли web-технології тільки починали розвиватись, і сьогодні змушені вкладати величезні кошти в масштабування, підтримку і оновлення застарілих програмних продуктів.

Існує також величезна кількість середніх та малих компаній, які не можуть собі дозволити витрачати таку ж кількість ресурсів на масштабне оновлення своїх legacy-проектів. Для них фактично єдиним засобом для вирішення цієї проблеми є використання автоматичних засобів трансляції та оптимізації коду для поступового оновлення кодової бази зі збереженням

наявної бізнес-логіки. Прикладами можуть слугувати деякі українські продуктові IT компанії, такі як EVO, LUN, Terrasoft, а також багато інших, менш відомих. Аутсорс-компанії також можуть бути зацікавлені у використанні подібних інструментів, якщо в них є контракти на тривалу підтримку розроблюваних продуктів.

Останньою зацікавленою стороною є малий IT-бізнес, тобто молоді стартапи, маленькі компанії, орієнтовані на внутрішній ринок web-сторінок (розробники типових інтернет-магазинів, лендінгів тощо), та поодинокі фрілансери. Представники цієї сторони під час розробки також постійно зустрічаються з legacy-кодом, хоч і не в таких масштабах, як більш великі компанії. У випадку коротких проектів з розробки основною проблемою, пов'язаною з legacy, є брак часу, у зв'язку з чим вони також зацікавлені у використанні інструментів автоматичного перетворення коду.

У табл. 5.1 зведено визначені групи зацікавлених сторін, їх інтереси та вплив.

Таблиця 5.1

Зацікавлені сторони

Зацікавлена сторона	**Інтерес**	**Вплив**	**Стратегії**
Великі продуктові IT компанії	Ефективний та порівняно недорогий спосіб модернізації наявної кодової бази	Високий	Демонстрація переваг використання засобів автоматизованого перетворення та оптимізації в порівнянні з ручним переписуванням
Середні продуктові IT компанії		Середній	
Аутсорс компанії	Зниження витрат на підтримку продуктів	Середній	
Малі IT компанії та програмісти-фрілансери	Економія часу на рефакторинг коду	Низький	

5.3. Комерційне рішення. Основні характеристики

На основі аналізу визначених проблем, можна описати кінцевий продукт для їх вирішення. Даний програмний продукт буде реалізовувати алгоритм оптимізації вхідного коду TCO для клієнтських та серверних застосунків мовою JavaScript. Результатом роботи даного алгоритму буде код з оптимізованими рекурсивними викликами, що дозволить використовувати код, написаний у функціональному стилі, з гарантованою відсутністю помилок виснаження стеку викликів. Дане рішення може бути корисним як для великих проектів, оскільки в них такі помилки можуть виникати внаслідок великих обсягів оброблюваних даних, так і для малих або нещодавно розпочатих, оскільки одразу дозволять користуватись перевагами функціонального підходу до розробки без необхідності додатково оброблювати ці помилки в майбутньому.

Для керівників великих та середніх проектів наслідки запровадження даного рішення будуть більш відчутними, особливо в тривалій перспективі, оскільки окрім оптимізації вже наявної кодової бази це рішення ліквідує подальше накопичення неоптимізованого коду, а отже, дозволить зосередити зусилля розробників на інші, більш важливі і корисні задачі, ніж відлагодження і оптимізація legacy. Крім того, проект позбавиться певної долі залежності від досвідчених розробників, які знаються на недоліках наявної кодової бази і звільнення яких до цього призвело б до катастрофічних для проекту наслідків.

Виходячи з цього можна зробити висновок, що основним клієнтом даного продукту є великі та середні продуктові IT компанії, а подальша співпраця буде ґрунтуватись на моделі B2B. Від моделі B2C повністю відмовлятись не планується, хоча її підтримка буде обмеженою.

Враховуючи дане орієнтування, розроблюваний програмний продукт має підтримувати обробку коду для широкого кола розповсюджених

фреймворків та середовищ виконання JavaScript, як клієнтських – jQuery, Angular, React, Vue – так і серверних, тобто Node.js. Крім цього, оскільки в основному оброблятися буде застарілий код, необхідно забезпечити підтримку низки застарілих стандартів ECMAScript та неактуальних версій перелічених вище технологій. Це не має викликати суттєвих проблем під час розробки, оскільки нові стандарти ECMAScript гарантують зворотну сумісність з попередніми, а документація до неактуальних версій фреймворків та платформ майже завжди зберігається в повному обсязі.

5.4. Конкурентні переваги рішення

Команди розробників рушіїв Chakra від Microsoft та V8 від Google застосовували алгоритм оптимізації TCO до внутрішніх структур даних у своїх рушіях. Для користувачів дані реалізації пропонувались в якості експериментальних режимів роботи рушія, які не рекомендувалося застосовувати в production-коді. Через деякий час інтерфейс для застосування цих функцій взагалі було вилучено з рушіїв, через що користувачі повністю втратили можливість застосовувати дану оптимізацію до свого коду. Розробники рушіїв повідомляли про дві основні проблеми, пов’язані з використанням цього алгоритму: погіршення швидкодії оптимізованого коду та значне ускладнення аналізу стеку викликів після закінчення роботи алгоритму.

Легко бачити головний недолік цих рішень: вони працювали з закритими структурами даних внутрішнього представлення рушіїв, не були виокремлені в самостійний програмний компонент, тобто повністю залежали від рушія.

Запропоноване рішення пропонує принципово інший підхід до реалізації цієї оптимізації, а саме використання native-js структур даних для внутрішнього представлення та комбінування попередніх і оптимізуючих трансляторів для отримання оптимізованого коду мовою JavaScript.

Відповідно, конкурентними перевагами запропонованого програмного продукту є:

- JavaScript в якості цільової мови трансляції, що надає користувачу свободу використання отриманого коду;
- Використання native структур даних, незалежних від рушія/фреймворку/середовища виконання;
- Відсутність впливу на складність відлагодження отриманого коду;
- Підтримка широкого кола різновидів web-застосувань для оптимізації.

5.5. Клієнти. Сегменти ринку споживання

Виходячи з аналізу зацікавлених сторін, клієнтами даного програмного продукту є IT компанії та підприємці, що спеціалізуються на розробці web-застосунків. Сегментацію ринку можна проводити за наступними ознаками: розмір компанії, орієнтація на внутрішній або зовнішній ринок, наявність власного продукту або контрактів на тривалу підтримку аутсорс-продуктів, а також середня тривалість проекту компанії.

Сегментація ринку за розміром спричинена тим, що у великих компаніях для модернізації кодової бази часто створюють окремі підрозділи, які займаються виключно роботою з legacy, що тягне за собою значні витрати на менеджмент, заробітні плати, оренду робочих місць та інші, пов’язані з найманими співробітниками. Для середніх компаній часто саме ці витрати взагалі не дозволяють провести модернізацію, хоча, з іншого боку, наслідки про відмову від оптимізації кодової бази для них з меншою ймовірністю призведуть до критичних наслідків. Зазвичай існує пряма кореляція між розміром компанії і тривалістю проектів, якими ця компанія займається, тому можна зробити висновок, що чим більшою є компанія, тим критичнішою для неї є потреба у модернізації кодової бази саме за допомогою автоматизованих засобів роботи з кодом.

З точки зору орієнтації на локальний чи зовнішній ринки практика показує, що в разі розроблення програмного забезпечення для найбільш розвинених країн (США, Канада, Західна Європа) дуже багато уваги звертається саме на якість програмного коду продукту: замовники часто цікавляться різноманітними метриками аналізу коду та загалом зацікавлені у стабільній роботі та масштабованості отримуваного програмного продукту. На відміну від цього в країнах, де ІТ галузь за темпами розвитку та іншими економічними показниками випереджає інші (Східна та Центральна Європа, Індія), замовники часто не звертають уваги на власне код і зацікавлені тільки у термінах завершення проектів та їх функціональних властивостях.

Очевидно, що компанії, які розробляють власний продукт, набагато зацікавленіші в його оптимізації, аніж представники аутсорс спільноти, проте в разі наявності тривалих контрактів зацікавленість аутсорс-розробників у запровадженні засобів автоматичної оптимізації різко зростає.

5.6. Унікальна ціннісна пропозиція

На основі дерева проблем та аналізу зацікавлених сторін було визначено очікування відповідних сторін від пропонованого продукту. Великі компанії бажають замість створення окремих підрозділів з рефакторингу отримати «коробочне» рішення для автоматизованого перетворення вхідного коду, яке в той же час буде сумісним з якомога ширшим технологічним стеком, середні продуктові компанії бажають отримати максимально доступне рішення для оптимізації власного продукту, аутсорс-компанії сподіваються спростити підтримку розроблюваних продуктів, а малі компанії і поодинокі розробники бажають зекономити час, який можна витратити на розробку інших нетривалих проектів. Запропоноване рішення в певній мірі задовольняє всі перелічені вимоги зацікавлених сторін та вирішує наведені проблеми.

Таким чином, унікальною ціннісною пропозицією є розроблений метод проведення TCO-оптимізації програмного коду, а основною перевагою є його незалежність від рушія мови, об’єкта оптимізації та цільової платформи отриманого коду, яка до цього ніким не пропонувалась. Інша назва – «Continuation Passing Style based implementation of Tail Call Optimization algorithm».

5.7. Доходи та витрати

Основний дохід планується забезпечувати шляхом продажу комерційних ліцензій на використання утиліти для автоматизованого перетворення коду. Клієнтам буде надаватись право використовувати утиліту для власних проектів у вигляді cloud-сервісу, тобто за допомогою віддаленого доступу без надання програмного коду та можливості внесення змін. Опціонально буде надаватись шифрований контейнер для розгортання на власних серверах і подальшого використання для оптимізації обмежених за розміром проектів. За допомогою контейнеризації вирішується проблема неконтрольованого використання. Буде пропонуватись кілька типів ліцензій з певними обмеженнями.

- Single use license. Демо-версія продукту, яка дозволяє один раз завантажити і оптимізувати невелику програму.
- Self-hosted license (for projects up to 25K LOC). Орієнтована на одного користувача. Постачається тільки контейнер з утилітою з метою ліквідації невиправданих для даної ліцензії витрат на оренду серверів.
- Small company license (for projects up to 100K LOC). На вибір надається або контейнер з утилітою для розміщення на власних серверах, або доступ до віддаленого серверу з розгорнутою утилітою і API.
- Enterprise license (for projects more than 100K LOC, cost scales with project size). Те саме, що і Small Company License, але на потужніших серверах і без обмежень на максимальний розмір проекту.

До витрат належать:

- утримання персоналу для розгортання і підтримки cloud-рішень;
- утримання іншого персоналу, або витрати на разові послуги (бухгалтерія, прибирання, юридичні консультації);
- утримання робочих місць для постійного персоналу;
- оренда серверів;
- податки.

Детальніше ознайомитися з прогнозованими витратами і прибутками можна з табл. 5.2.

Таблиця 5.2

Витрати на реалізацію проекту

Найменування витрат	**1-й місяць, т. $**	**2-й місяць, т. $**	**3-й місяць, т. $**	**4-й місяць, т. $**	**5-й місяць, т. $**	**6-й місяць, т. $**
Загальні витрати	1.5	1.5	1.5	1.5	1.5	1.5
ЗП	0	27	27	27	27	27
Оренда серверів	0	0	0	0	0	0
Витрати	1.5	28.5	28.5	28.5	28.5	28.5
Заплановані прибутки	0	0	0	0	0	0
Результат	–1.5	–28.5	–28.5	–28.5	–28.5	–28.5

Продовження табл. 5.2

Найменування витрат	7-й місяць, т. $	8-й місяць, т. $	9-й місяць, т. $	10-й місяць, т. $	11-й місяць, т. $	12-й місяць, т. $	Загальні результати
Загальні витрати	1.5	1.5	1.5	1.5	1.5	1.5	18
ЗП	32	32	32	32	32	32	327
Оренда серверів	4	4	4	4	4	4	24
Витрати	37.5	37.5	37.5	37.5	37.5	37.5	369
Заплановані прибутки	72.5	72.5	72.5	72.5	72.5	72.5	435
Результат	35	35	35	35	35	35	66

5.8. Бізнес-модель

Все написане вище узагальнимо в бізнес-модель у вигляді lean canvas.

- Споживачі: великі та середні IT компанії з власним продуктом або тривалими контрактами на розробку/підтримку, одиничні користувачі.
- Проблема: надмірна складність роботи з великими legacy-проектами, висока вартість ручної підтримки і модернізації кодової бази; відсутність рішень для автоматизованої оптимізації у вигляді окремих сервісів; відмова від функціонального підходу під час розробки web-застосувань через відсутність TCO.
- Рішення: програмне забезпечення, що пропонує автоматичну оптимізацію TCO web-застосунків, написаних мовою JavaScript.

- Унікальна ціннісна пропозиція: програмне забезпечення, незалежне від рушія мови, об’єкта оптимізації та цільової платформи отриманого коду; зменшення витрат ресурсів на оптимізацію програмного коду.
- Потоки доходів: доходи від продажу ліцензій.
- Структура витрат: утримання персоналу для розгортання і підтримки cloud-сервісів; оренда серверів для розгортання cloud-сервісів, оплата послуг бухгалтера, юриста та клінінґ-менеджера, податки.
- Канали: відділи з питань співробітництва IT-компаній, SPA з рекламними матеріалами.
- Ключові метрики: кількість проданих ліцензій, кількість завантажених контейнерів.
- Прихована перевага: значна універсальність алгоритму з точки зору підтримуваних технологій.

Бізнес-модель у зведеному вигляді наведена в табл. 5.3.

Таким чином, можна зробити висновок, що запропонований проект, який використовує описаний у дисертації метод оптимізації TCO має перспективи у подальшій реалізації. Проведений аналіз не враховує всіх ризиків і факторів, пов’язаних з веденням IT-бізнесу в Україні, проте припускається, що наявних досліджень достатньо для впевненого прогнозування комерційної успішності продукту.

Таблиця 5.3

Канва бізнес-моделі

<table>
<tr>
<td rowspan="2">Проблема
надмірна складність роботи з великими legacy-проектами, висока вартість ручної підтримки і модернізації кодової бази; відсутність рішень для автоматизованої оптимізації у вигляді окремих сервісів; відмова від функціонального підходу під час розробки web-застосувань через відсутність TCO.</td>
<td>Рішення
програмне забезпечення, що пропонує автоматичну оптимізацію TCO web-застосунків, написаних мовою JavaScript.</td>
<td rowspan="2">Унікальна ціннісна пропозиція
програмне забезпечення, незалежне від рушія мови, об’єкта оптимізації та цільової платформи отриманого коду; зменшення витрат ресурсів на оптимізацію програмного коду.</td>
<td>Прихована перевага
значна універсальність алгоритму з точки зору підтримуваних технологій.</td>
<td rowspan="2">Споживачі
великі та середні IT компанії з власним продуктом або тривалими контрактами на розробку/підтримку, одиничні користувачі.</td>
</tr>
<tr>
<td>Ключові метрики
кількість проданих ліцензій, кількість завантажених контейнерів.</td>
<td>Канали
відділи з питань співробітництва IT-компаній, SPA з рекламними матеріалами.</td>
</tr>
<tr>
<td colspan="3">Структура витрат
утримання персоналу для розгортання і підтримки cloud-сервісів; оренда серверів для розгортання cloud-сервісів, оплата послуг бухгалтера, юриста та клінінг-менеджера, податки.</td>
<td colspan="2">Потоки доходів
доходи від продажу ліцензій.</td>
</tr>
</table>

5.9. Висновки до розділу 5

В даному розділі було проведено аналіз поточної ситуації у сфері автоматизованої оптимізації програмного коду web-застосунків, виявлено ключові проблеми та підсумовано їх у відповідному дереві проблем. Також було проведено аналіз основних зацікавлених сторін у вирішенні існуючих недоліків та ступінь їх впливу на вирішення проблем. На основі цього було запропоновано комерційне рішення з конкурентними перевагами, що задовольняє вимоги зацікавлених сторін, та синтезовано унікальну ціннісну пропозицію розроблюваного продукту. Було проведено аналіз потенційних клієнтів, досліджено сегменти ринку споживачів, що дозволило спрогнозувати потенційні доходи та витрати на реалізацію продукту. В результаті було сформовано бізнес-модель, що обґрунтовує доцільність реалізації даного продукту та прогнозує його потенційну прибутковість в майбутньому.

ВИСНОВКИ

Метою даної роботи було створення способу побудови CPS-форми внутрішнього представлення для вхідного коду мовою JavaScript та застосування даної форми для розробки способу оптимізації хвостових викликів програм для платформи Node.js.

Аналіз схожих систем, виконаний у дипломному проекті, показав наявність реалізованих систем для інших мов програмування. Проте системи оптимізації хвостових викликів для мови JavaScript на разі не існує, хоча аналіз певної підмножини хвостових викликів був доступний в якості експериментального режиму роботи рушія V8 старих версій.

Розроблена система являє собою клієнт-серверний застосунок з користувацьким інтерфейсом у вигляді односторінкового веб-застосунку.

Практична цінність розробленої системи полягає в наступному:

1. Розроблений спосіб оптимізації хвостових викликів, окрім безпосереднього вирішення проблеми виснаження стеку викликів, дозволяє подальше покрокове налагодження отриманої програми стандартними засобами платформи Node.js.
2. Існує можливість застосування перетворення як для окремих функцій мови, так і для модулів, що спрощує подальше використання і інтеграцію отриманого коду у вже існуючі програмні продукти.
3. Існує можливість окремого використання побудованої CPS-форми без застосування оптимізації хвостових викликів у вигляді програмного коду для певних цілей розробника.

Розробка виконана у повному обсязі, тестування системи виконано у відповідності до затвердженої програми та методики тестування.

Використання розробленої системи спростить використання функціонального стилю під час написання і налагодження програм мовою JavaScript платформи Node.js.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Bright, W. Inheriting Purity [Electronic Resource] / W. Bright, A. Alexandrescu // CPP Articles / Dr. Dobb's Bloggers — Access mode : http://www.drdobbs.com/cpp/inheriting-purity/232601305. — Access date : Sept. 2019.
2. SonarTS README.md [Electronic Resource]. — Access mode : https://github.com/SonarSource/SonarTS. — Access date: Sept. 2019.
3. Crockford, D. JavaScript: The Good Parts [Text] / Douglas Crockford. — 1st edition. — O'Reilly, 2008. — 172 p.
4. JShint Application Programming Interface [Electronic Resource]. — Access mode : http://jshint.com/docs/api/. — Access date : April 2018.
5. Pokorny, R. Do pure functions exist in JavaScript? [Electronic Resource] / R. Pokorny. — Access mode : https://hackernoon.com/do-pure-functions-exist-in-javascript-b128ed5f0ed2. — Access date: Oct. 2019
6. Okasaki, C. Purely Functional Data Structures [Text] / Chris Okasaki. — 2nd edition. — Cambridge University Press, 1999. — 232 p.
7. Hindley, J. R. Lambda-Calculus and Combinators: An Introduction [Text] / Hindley J. R., Seldin J.P. — 1st edition. — Cambridge University Press, 2008. — 358 p.
8. Simpson, K. You Don't Know JS: Scope & Closures [Text] / Kyle Simpson. — 1st edition. — O'Reilly, 2014. — 98 p.
9. Simpson, K. You Don't Know JS: Types & Grammar [Text] / Kyle Simpson. — 1st edition. — O'Reilly, 2015. — 200 p.
10. Boulanger, J. Static Analysis of Software: The Abstract Interpretation [Text] / Jean-Louis Boulanger. — Wiley-ISTE, 2011. — 331 p.

11. Chess, J. Secure Programming with Static Analysis [Text] / Brian Chess, Jacob West. — Addison-Wesley Professional, 2007. — 624 p.
12. Cantelon, M. Node.js in Action [Text] / Mike Cantelon, Marc Harter, TJ Holowaychuk, Nathan Rajlich. — 2nd edition. — Manning Publications, 2013. — 416 p.
13. Cousineau, G. The Functional Approach to Programming [Text] / Guy Cousineau, Michel Mauny. — 2nd edition. — Cambridge University Press, 1998. — 460 p.
14. SpiderMonkey Parser API [Electronic Resource]. — Access mode : https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API. — Access date: Nov. 2019.
15. Flanagan, D. JavaScript: The Definitive Guide [Text] / David Flanagan. — O'Reilly, 2011. — 1096 p.
16. Stefanov, S. JavaScript Patterns: Build Better Applications with Coding and Design Patterns [Text] / Stoyan Stefanov. — 3rd edition. — O'Reilly, 2010. — 236 p.
17. Сулема, Є.С. Дипломне проектування за напрямами підготовки «Прикладна математика», «Комп'ютерна інженерія», «Програмна інженерія» [Текст] : навч.-метод. посіб. / Є.С. Сулема ; за заг. ред. І.А. Дички. — К. : НТУУ «КПІ», 2011. — 224 с.

Спосіб оптимізації хвостових викликів функцій для Node.js на основі CPS-форми внутрішнього представлення

Доповідач: студент групи КП-81мп

Стилик Роман Григорович

Керівник: доцент, к.т.н.

Марченко Олександр Іванович

Актуальність дослідження

TCO (Tail Call Optimization) є частиною стандарту ES6, та, відповідно має підтримуватись рушіями, які відповідають цьому стандарту.

TCO було реалізовано в якості експериментальної функції рушія V8, але **наразі підтримку цієї функції було скасовано**.

Таким чином, **платформа Node.js наразі не підтримує TCO**, що певним чином ускладнює використання функціонального стилю під час розробки програм для цієї платформи.

Постановка наукової задачі та мета досліджень

Науковою задачею є створення способу оптимізації хвостових викликів функцій на основі CPS-форми.

Метою досліджень є створення алгоритму побудови CPS-форми внутрішнього представлення для вхідного коду мовою JavaScript та застосування даної форми для розробки способу оптимізації хвостових викликів програм для платформи Node.js.

Об’єкт та предмет досліджень

Об’єктом досліджень є процес оптимізації хвостових викликів програм для платформи Node.js.

Предметом досліджень є способи оптимізації хвостових викликів програм для платформи Node.js.

Технічні задачі

Задачею є розробка програмного продукту,
який виконує наступне:

1) Побудова CPS-форми внутрішнього представлення для вхідної програми мови JavaScript;
2) Виконання оптимізації хвостових викликів побудованої CPS-форми і генерація оптимізованої вихідної програми.

Термінологія

CPS (Continuation Passing Style) – спосіб побудови програм, в якому контроль потоку виконання (Control Flow) передається в явному вигляді за допомогою продовжень (continuation).

Continuation – представлення стану програми в певний момент часу, який є доступним для переходу в нього. В контексті теми доповіді продовження реалізуються за допомогою функцій вищого порядку.

Tail Call – виклик процедури/функції в якості останньої дії поточної процедури. Виклик тієї самої процедури називається хвостовою рекурсією.

TCO/TCE (Tail Call Optimization/Elimination) – реалізація Tail Call, за якої вони оброблюються без використання стеку викликів (Call Stack) і таким чином не призводять до його виснаження.

Дослідники

Ґеральд Джей Сасмен, Гай Льюіс Стіл молодший – AI Memo 349 (1975) – дали визначення і заклали концепцію CPS, запропонували першу реалізацію CPS-компілятора для мови Scheme;

Джон Рейнольдс – «The Discoveries of Continuations» (1993) – детальний аналіз продовжень (continuations) як форми внутрішнього представлення.

Спенсер Тіппінг – «JS in ten minutes» (2013) – запропонував спосіб ручного комбінування CPS та TCO у JS-програмах. Пізніше було доведено, що цей спосіб був частинним випадком загальної техніки Trampolining.

Чому хвостові виклики є проблемою?

Для зберігання поточної позиції інтерпретатора використовується call stack, який містить дані про викликані функції.

Будь який виклик функції додає викликану функцію в стек, після завершення виконання функція прибирається зі стеку.

У випадку хвостових викликів головна функція буде видалена зі стеку викликів тільки тоді, коли завершить виконання вкладена функція.

У випадку, коли має місце глибока вкладеність хвостових викликів, стек може переповнитись.

Приклад

```
function greeting() {
    // [1] Some code here
    sayHi();
}
function sayHi() {
    return "Hi!";
}
// Invoke the `greeting` function
greeting();

// [3] Some code here
```

Під час виконання даної програми стек викликів виглядатиме наступним чином:

1) [greeting]
2) [greeting, sayHi] ← **проблема**
3) [greeting]
4) [ ]

В даному випадку стек використовується неоптимально.

За допомогою TCO можна досягти наступного:

1) [greeting]
2) [sayHi]
3) [ ]

Навіщо потрібна CPS-форма?

Не всі функції, що потребують оптимізації, одразу містять хвостові виклики.

Наприклад:

```
function factorial (n) {
  if (n < 2) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

```
function factorial (n) {
  function fact(n, acc) {
    if (n < 2) {
      return acc;
    } else {
      return fact(n-1, n * acc);
    }
  }
  return fact(n, 1)
}
```

Способи вирішення проблеми

CPS

1) Звести вхідну програму у CPS-форму, в якій кожен виклик гарантовано буде хвостовим;
2) Провести TCO отриманої програми будь-яким відомим способом.

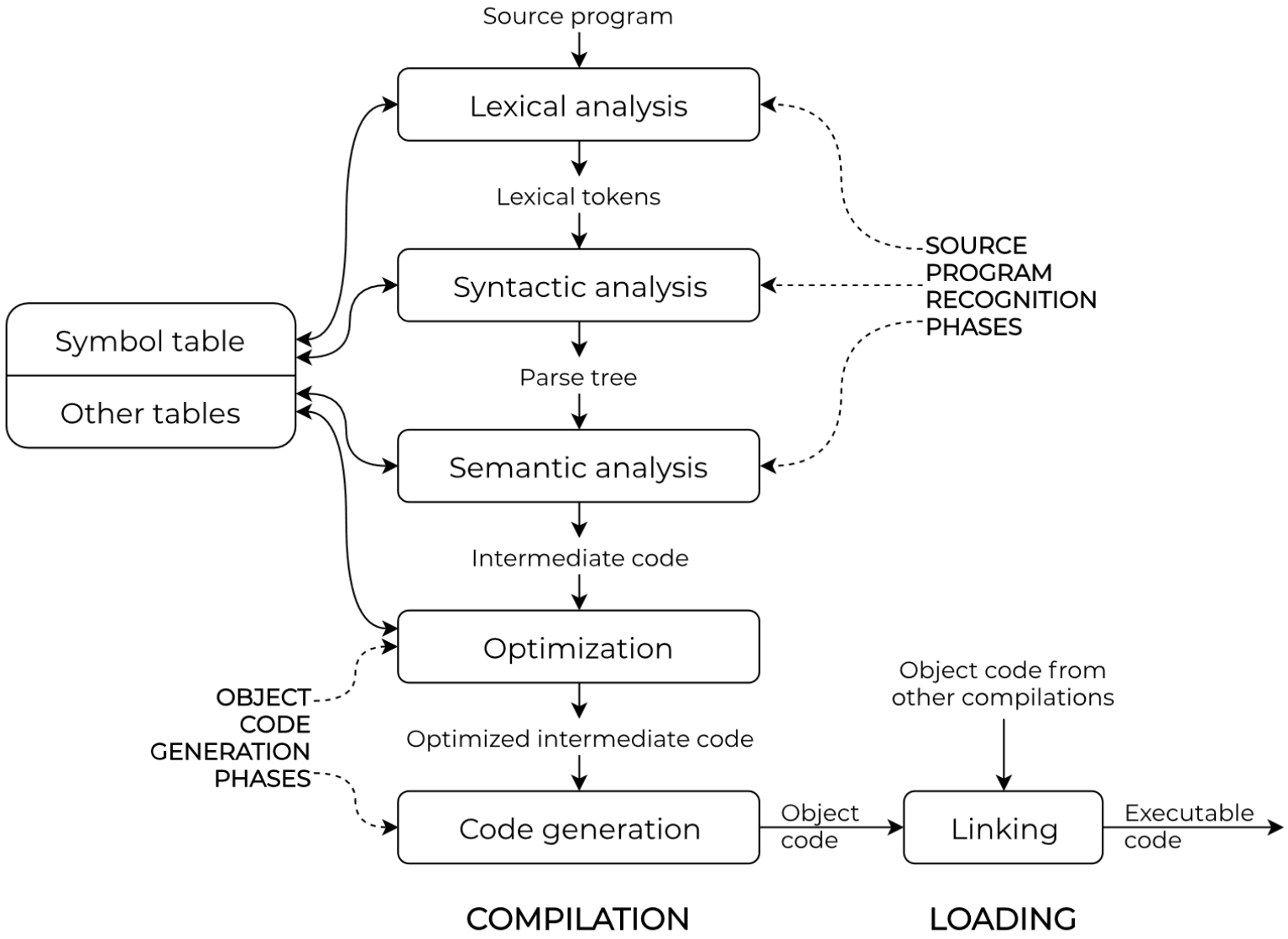
PTC (Proper Tail Calls)

1) Ввести нову синтаксичну конструкцію для явного виділення хвостових викликів;
2) Розробити механізм TCO на основі аналізу явних хвостових викликів;
3) Програміст власноруч переписує програму з використанням нової конструкції, TCO працює на рівні рушія.

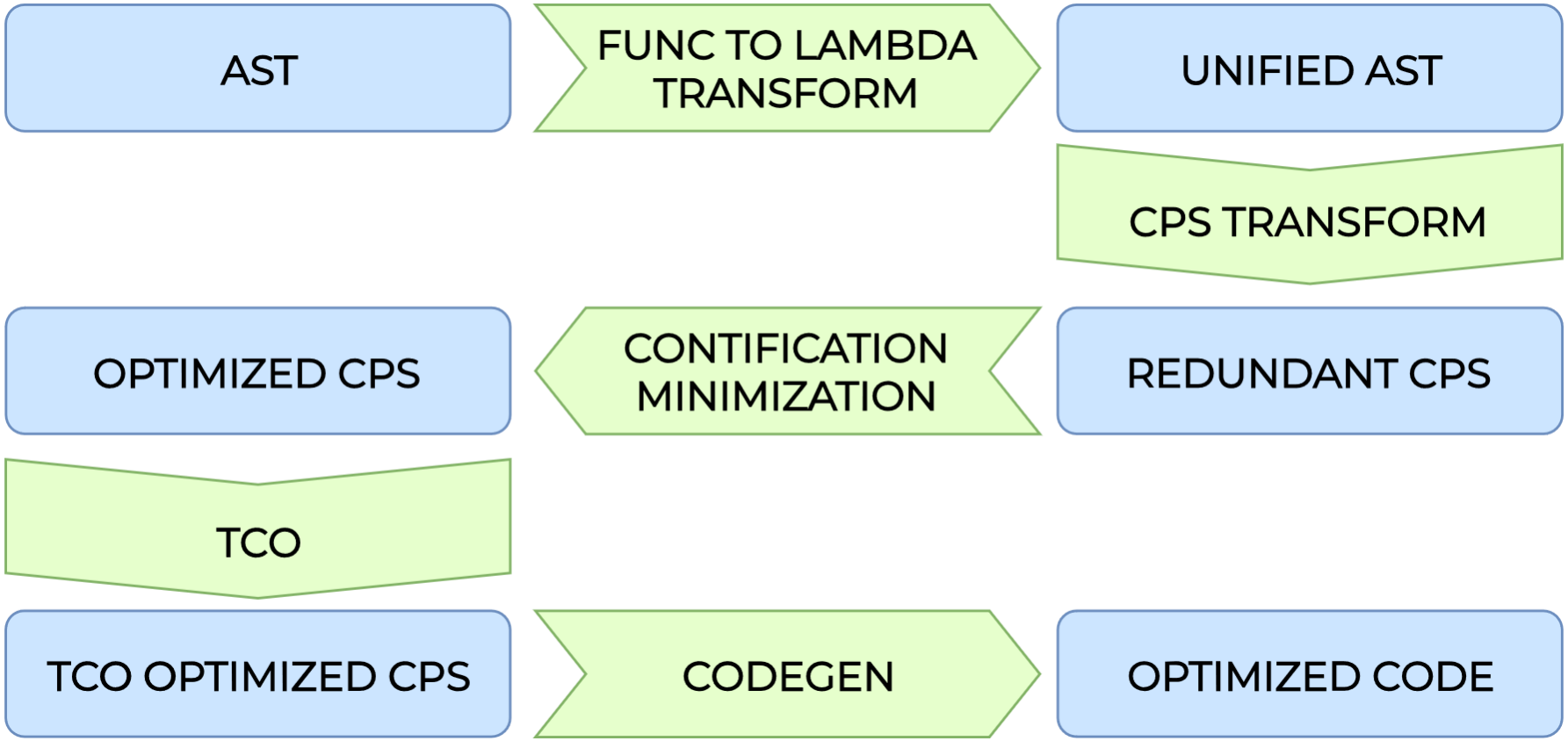
Аналіз способів TCO

	CPS-based	PTC
Реалізується на основі	Форми внутрішнього представлення, перетворення на графах	Введення явних синтаксичних конструкцій для продовжень
Можливість автоматизації	+	-
Швидкодія	Compilation time – падає Runtime – незначно падає	Важко спрогнозувати, навряд чи зміниться суттєво
Пам’ять	Програш через необхідність зберігати проміжну форму	Важко спрогнозувати, припускається виграш за рахунок зменшення стеку викликів

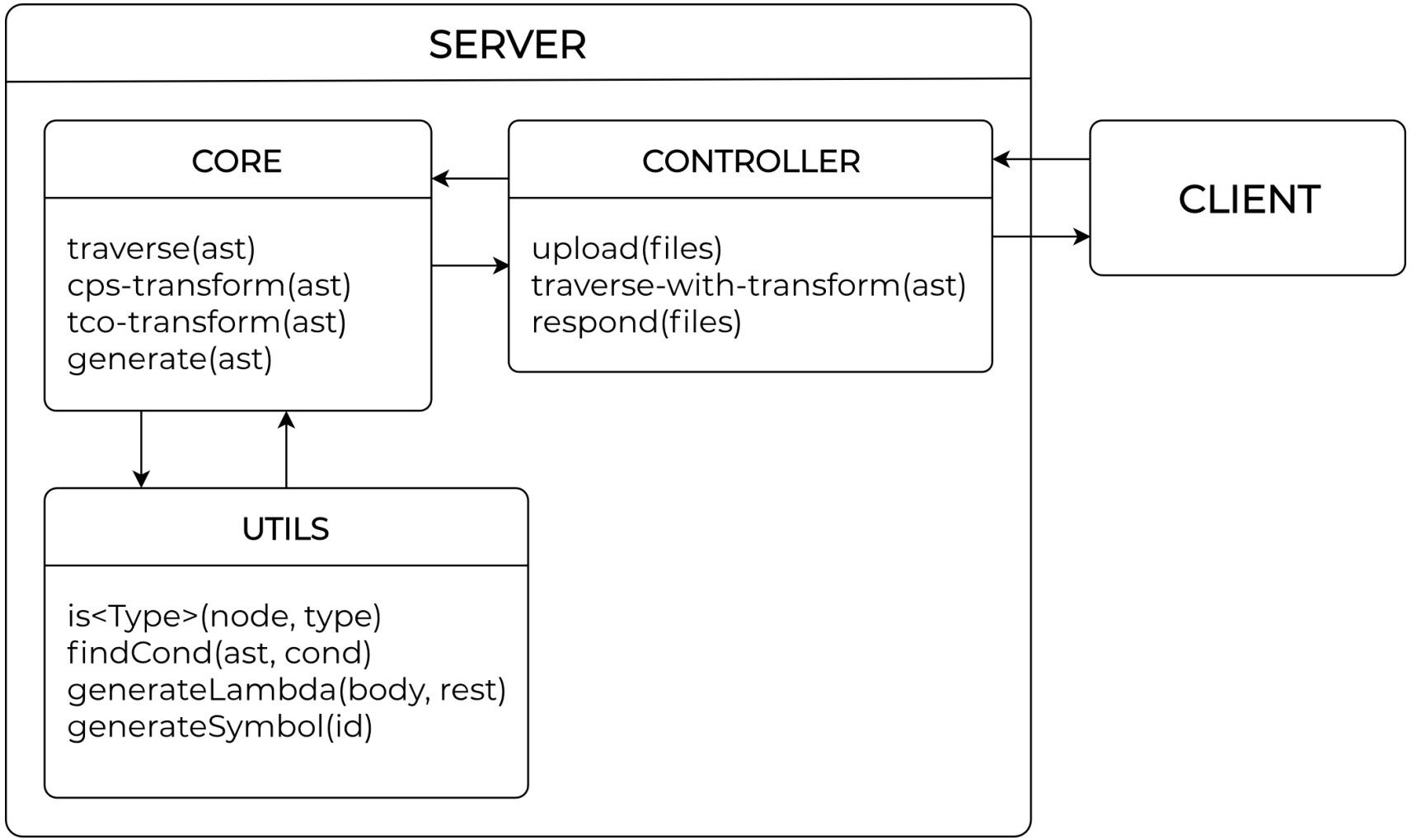
Етапи перетворень вхідного коду



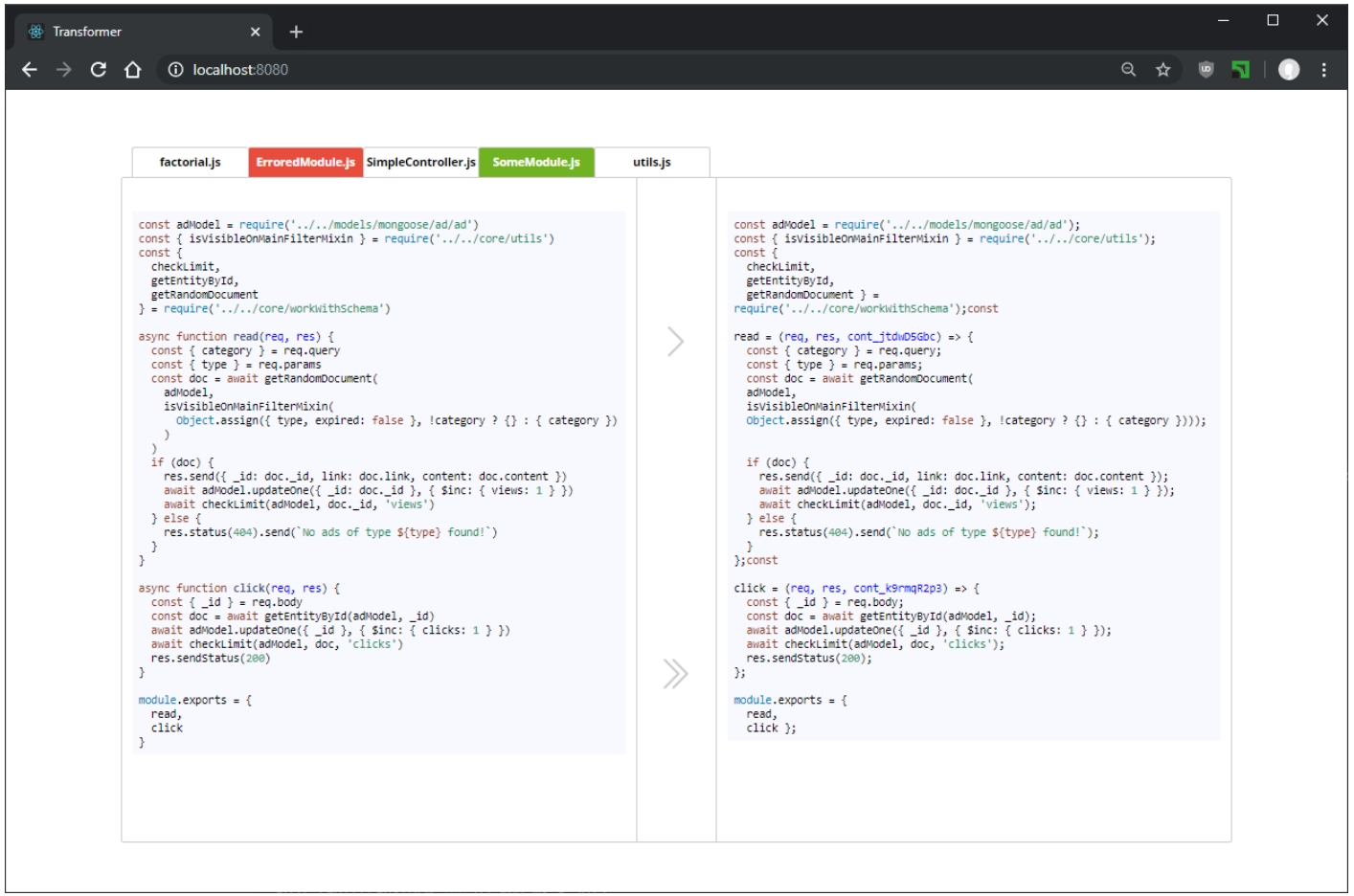
Фази роботи алгоритму



Структура розробленої системи



Користувацький інтерфейс системи



Використані технології

- Node.js
- Express.js
- React.js
- HTML5, CSS
- git

Наукова новизна

1. Запропоновано алгоритм побудови CPS-форми внутрішнього представлення для програм мови JavaScript стандарту ES6, який відрізняється від наявних реалізацій побудови CPS-форми можливістю як міжпроцедурної, так і часткової побудови, а також незалежністю від цільової платформи вхідного JavaScript коду.
2. Запропоновано спосіб оптимізації хвостових викликів функцій на основі побудованої CPS-форми, який відрізняється від існуючих збереженням контролю над стеком виконання, що в свою чергу дозволяє використання точок зупинки і покрокового виконання оптимізованої програми.

Бізнес модель

Проблема
надмірна складність роботи з великими legacy-проектами, висока вартість ручної підтримки і модернізації кодової бази; відсутність рішень для автоматизованої оптимізації у вигляді окремих сервісів; відмова від функціонального підходу під час розробки web-застосувань через відсутність TCO.

Рішення
програмне забезпечення, що пропонує автоматичну оптимізацію TCO web-застосунків, написаних мовою JavaScript.

Ключові метрики
кількість проданих ліцензій, кількість завантажених контейнерів.

Унікальна ціннісна пропозиція
програмне забезпечення, незалежне від рушія мови, об'єкта оптимізації та цільової платформи отриманого коду; зменшення витрат ресурсів на оптимізацію програмного коду.

Прихована перевага
значна універсальність алгоритму з точки зору підтримуваних технологій.

Канали
відділи з питань співробітництва IT-компаній, SPA з рекламними матеріалами.

Споживачі
великі та середні IT компанії з власним продуктом або тривалими контрактами на розробку/підтримку, одиничні користувачі.

Структура витрат
утримання персоналу для розгортання і підтримки cloud-сервісів; оренда серверів для розгортання cloud-сервісів, оплата послуг бухгалтера, юриста та клінінґ-менеджера, податки.

Потоки доходів
доходи від продажу ліцензій.

Висновки

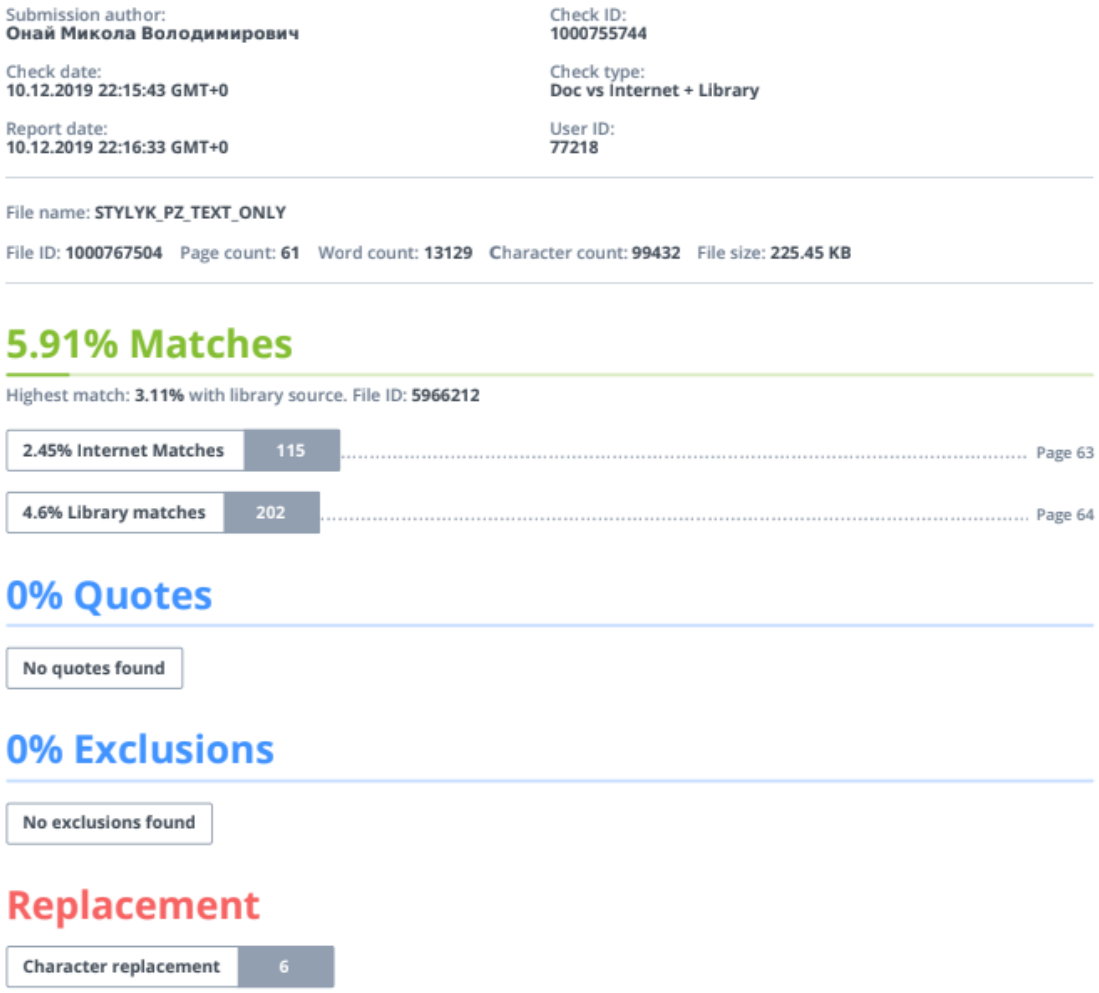
Проведено аналіз існуючих способів оптимізації хвостових викликів для платформи Node.js та інших мов програмування.

Визначено основні недоліки альтернативних способів оптимізації, показано відсутність задовільних рішень для платформи Node.js

Запропоновано спосіб оптимізації хвостових викликів для платформи Node.js на основі CPS-форми внутрішнього представлення.

Розроблено програмну систему, що виконує CPS-трансформацію та TCO для програмного коду платформи Node.js.

Результати перевірки UNICHECK



Submission author:
Онай Микола Володимирович

Check ID:
1000755744

Check date:
10.12.2019 22:15:43 GMT+0

Check type:
Doc vs Internet + Library

Report date:
10.12.2019 22:16:33 GMT+0

User ID:
77218

File name: **STYLYK_PZ_TEXT_ONLY**

File ID: **1000767504** Page count: **61** Word count: **13129** Character count: **99432** File size: **225.45 KB**

5.91% Matches

Highest match: **3.11%** with library source. File ID: **5966212**

2.45% Internet Matches	115	Page 63
4.6% Library matches	202	Page 64

0% Quotes

No quotes found

0% Exclusions

No exclusions found

Replacement

Character replacement	6

ДЯКУЮ ЗА УВАГУ!