Министерство образования и науки Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в телекоммуникационных сетях и вычислительных системах

Выпускная квалификационная работа бакалавра по направлению 03.03.01
«Прикладные математика и физика»

Исследование и разработка системы анализа кода для повышения производительности программ на языке программирования высокого уровня

Студентка Б01-008 группы
Лирисман К. С.

Научный руководитель
Гаврин Е. А.

Долгопрудный
2024

Содержание

Аннотация

В настоящий момент активно развивается статически типизированный управляемый язык программирования, являющийся расширенной и более быстрой версией языка *TypeScript* (далее TS). Основная идея разработки спецификации и компилятора этого языка программирования — сделать его максимально похожим на TS для упрощения перехода будущих разработчиков между TS и выбранным для исследования языком, а также для ускорения переписывания существующих на TS приложений. Таким образом, между целевым языком и TS формируется общая часть, - корректная с точки зрения TS. Оставшуюся часть называют не поддерживаемой TS.

В данной работе предлагается выделить из целевого ЯП подмножество, наиболее выгодное с точки зрения производительности, в том числе за счёт отсечения удобного для разработчиков, но медленного функционала. Таким образом, пока сам язык неминуемо развивается в сторону общности с TS‘ом ради легкости перехода, предложенная система помогает держать фокус на производительности.

Анализ происходит в момент компиляции исходного кода и предлагает включение желаемых проверок группами или по отдельности путем добавления флагов компиляции. Реализовано 8 типов проверки, опирающихся на спецификацию выбранного языка программирования высокого уровня и реализацию его компилятора. К общим с TS проверкам относятся: неявная упаковка и распаковка, ускорение проверок равенства, запрет инструкций верхнего уровня, исключая классы и функции, предложение установки модификатора для класса или метода как финального и другие. К несовместимым с TS относятся: использование корутин вместо асинхронных функций, предложение использовать модификатор, ограничивающий наследование классов и методов в случае, если на момент проверок у них нет потомков. Предлагается использование двух режимов работы проверяющей системы — в состоянии предупреждений, а именно, предложений, не обязывающих разработчика к исправлением замечаний и не влияющих на результат работы программы, и в состоянии, приводящему к ошибке при ненулевом количестве предложений, ожидающих от пользователя последующих исправлений, и считающегося за ошибку компиляции. Предложенная система формулирует рекомендации для повышения производительности. Однако существуют сценарии использования, когда разработчикам необходимо отключение этих проверок. С помощью произведенной системы регулирования режимов проверок пользователи целевого языка могут активировать или деактивировать любую опцию по одной или включать группами. Также можно

снять с проверки определенные строки или целые частей кода - всё это реализуемо с помощью предложенного аналога системы точечного отключения проверок Clang Tidy непосредственно в исходном коде программы в виде многострочных или однострочных комментариев. Таким образом, данная система значительно повышает скорость работы и время запуска приложения на выбранном статически типизированном языке высокого уровня, позволяет разработчикам простыми методами улучшить их код и потенциально избежать некоторых ошибок.

Глава 1

Введение

В современном мире существует огромное множество языков программирования. Часть из этих языков программирования одной из своей целей при разработке доступность использования для широкого круга людей, которые не глубоко знакомы с устройством компьютеров, операционных систем.

Дополнительно, каждый язык программирования оптимизирован для своего типа задач, которые предположительно будут создаваться с его использованием.

Постановка задачи создания идеального языка программирования включает в себя точное и однозначное выражение спецификации задачи и способность превратить написанное в машинные инструкции. Очевидно, что такая задача едва ли достижима, так как тяжело довести до совершенства формулировки большого количества задач, и еще сложнее реализовать нужный алгоритм.

На данный момент в мире насчитывается больше сотни языков программирования, хотя большинство людей знакомы только с несколькими десятками, и это при условии глубоких знаний и опыта в сфере.

Такое большое количество языков программирования необходимо для качественной реализации определенного класса приложений. При этом большинство современных языков программирования создаются машинно-независимыми, то есть их структура не привязана к архитектуре. Отсюда их название, - машинно-независимые языки программирования высокого уровня.

Сфера компьютерных наук развивается довольно быстро, динамично и хаотично по причине своей популярности и обширности. Это значит, что постоянно на рынке появляются новые подходы, старые становятся либо неактуальными, либо дорабатываются, чтобы соответствовать современным стандартам.

По мере появления более сложных систем и программного обеспечения возникают все новые сценарии использования, которые требуют все новых языков. Сложность приложений и систем растет, что требует новых методов решения и подходов. При этом не менее важным параметром является поддержание доступной стоимости разработки.

Как было сказано ранее, на сегодняшний день на первый план вышли языки программирования высокого уровня, которые позволяют разработчику сфокусироваться

на логике работы приложения и не переживать об управлении памятью и иными ресурсами, относящимися к низкоуровневым задачам. Все это позволяет сформулировать несколько причин для создания новых языков программирования высокого уровня:

1. Проектирование новых языков с целью оптимизации производительности может включать в качестве результата более эффективное использование ресурсов компьютера, упрощение синтаксиса в контексте целевой задачи или разработка новых синтаксических конструкций или функционала, обелегчающих работу разработчиков конкретной сферы.

2. Проектирование новых языков под новые задачи, для которых на данный момент нет удобного или оптимального решения. К таким задачам можно отнести, например, задачи, связанные с узко направленной научной деятельностью.

3. Разработка нового языка может являться чисто стратегическим действием для продвижения коммерческих решений в открытый доступ или ради развития новых идей, видений и подходов в области компьютерных наук

В настоящее время активно развивается язык программирования, представляющий собой более быструю альтернативу существующему *TypeScript* (далее TS). Основной целью этого языка является создание расширенное и более быстрой статической версией TS при условии сохранения совместимости с самим TS ради упрощения перехода будущих разработчиков с одного языка на другой.

Современные среды программирования требуют высокую производительность приложений. В таком контексте принципиально важны дополнительные инструменты, позволяющие автоматически или с определенной персонализацией под конкретную задачу проанализировать исходный код и выдвинуть предложения по его оптимизации.

Под повышением производительности в целом могут стоять разные параметры, например, энергопотребление или размер полученного бинарного файла. Эти и другие параметры обычно тесно взаимосвязаны, и потому в попытках улучшить показатели одного параметра можно получить отрицательный результат для другого.

Выделяют динамические параметры программы, к которым традиционно относят производительность и энергопотребление. Смысл динамических параметров заключается в зависимости от конкретного устройства, на котором исполняется программа, то есть при проведении замеров результаты не будут повторяться. Однако такие параметры можно измерять группами: для одной группы устройств и для другой. Замеры формата 'до' и 'после' в рамках выбранной группы должны отвечать свойству повторяемости. Дополнительно можно сказать, что тенденция должна сохраняться при переходе из одной группы в другую. Например, если использование некого алгоритма А в группе А приводит к ускорению приложения на 20% по сравнению с методом Б в рамках группы А, то при в группе Б не должно наблюдаться противоположного результата, то есть, например, замедления на 15% при таком же сравнении метода А и метода Б.

Если такие результаты и удастся получить, это может означать только некорректность постановки задачи или использованных метрик.

Очевидно, что скорость выполнения программы и энергопотребление являются двумя очень важными аспектами, что при разработке всегда хотелось бы выиграть в обоих. Тем не менее они могут быть очень по-разному оптимизированы. Тогда некоторые изменения в языке программирования или в реализации его компилятора могут привести одновременно к увеличению потребления энергии из-за увеличения числа операций или увеличения нагрузки на процессор, чем мы не можем пренебрегать в случае устройств с ограниченными ресурсами, и к увеличению скорости выполнения программы за счет более эффективного распределения ресурсов машины. Некоторые из таких изменений могут быть следующими:

1. При использовании параллелизма алгоритмы увеличивают свою скорость выполнения, но при этом могут потреблять большее количество энергии, так как нагрузка на процессор стала значительно больше.

2. При изменении алгоритма или структуры данных некоторые операции могут оказаться более эффективными по времени, но менее эффективными по памяти, и тогда расход энергии увеличится.

3. При оптимизациях компилятора запускаются сложные оптимизационные процессы, которые могут больше расходовать ресурсы. Может быть частой проблемой при нерациональном или неграмотном подходе к оптимизациям.

В данной работе фокус идет на повышение производительности в контексте времени выполнения и запуска программы за счет гибкой работы со спецификацией целевого языка программирования и реализацией его компилятора. По данной причине вышеупомянутые оптимизации для работы не являются релевантными, а использование других, более высокоуровневых абстракций предполагается заранее считать вредными для быстродействия. Таким образом, будет предлагаться поиск недорогостоящих аналогов с точки зрения скорости выполнения и спецификации выбранного языка программирования.

Итак, в данном случае уместно считать, что меньшее время исполнения программы влечет к меньшему энергопотреблению. Далее под повышением производительности будет пониматься скорость выполнения программы и время ее запуска в рамках выбранных устройств и опций компилятора.

Глава 2

Постановка проблемы

2.1 Цели работы:

1. Разработка системы анализа и предупреждений исходного кода выбранного языка программирования высокого уровня для повышения его производительности и ускорения запуска написанных на нем приложений
2. Реализация системы-аналога Clang Tidy для выборочного отключения выбранных проверок точечно, непосредственно в исходном коде.

2.2 Задачи работы:

1. Изучение существующих решений
2. Анализ текущей спецификации целевого языка программирования на предмет потенциально медленных языковых конструкций и функционала, сбор данных для дальнейшего тестирования.
3. Предоставить вариант исправления, ускоряющий работу приложения, корректный с точки зрения выбранного языка программирования, для каждой языковой единицы среди предложенных
4. Протестировать каждое предложение: замерить скорость работы приложения до и после предложенных исправлений
5. Разработка системы анализа кода на этапе компиляции
6. Реализовать соответствующую поверку в системе анализа после подтверждения положительных результатов тестирования
7. Разработать систему точечного и группового отключения выбранных разработчиком проверок, поддержать наиболее популярные сценарии использования, опираясь на данные, полученные при изучении существующих решений

8. Поддержать возможность анализа в системе многофайловой сборки

Цели работы разумно считать достигнутыми при выявлении и реализации не менее пяти предложений, ускоряющих работу приложения в среднем не менее, чем на 3 %, а также разработке системы отключения проверок, успешной поддержке проектной сборки и прохождении тестирования, составленного из некоторых потенциальных сценариев применения предложенных решений.

Глава 3

Обзор существующих решений

Сегодня представлено огромное число средств, позволяющих на разных этапах проанализировать код программы на качество, причем понимание о качестве могут подразумевать быстродействие, а могут и нет. По данной причине есть смысл рассмотреть основные классы данных инструментов. Это позволит правильно классифицировать и применить результат данной работы на практике другим пользователям и читателям.

Итак, разные системы могут включать как статический, так и динамический анализ, и они используются для различных языков программирования, таких как C, C + +, *Java*, Python и другие. Для анализа и оптимизации кода существует несколько методов:

1. **Статический анализ**

   При статическом анализе код подвергается проверке без стадии выполнения.

   Примерами инструментов для данного метода могут служить Lint для C или SonarQube.

   При статическом анализе идет проверка стиля написания кода, поиск потенциальных ошибок.

2. **Динамический анализ**

   При статическом анализе код подвергается проверке во время его выполнения.

   Примерами инструментов являются Valgrind для C/C + +, JProfiler для *Java*.

   Здесь выполняется профилирование времени выполнения, анализ используемых ресурсов, в том числе поиск потенциальных утечек в памяти.

3. **Оптимизации компиляции**

   (a) ***JIT*-компиляция (Just-In-Time)**

   Целью такого рода оптимизаций стоит компиляция кода во время его выполнения, происходит оптимизация под конкретное окружение.

   Типичным примером оптимизаций компиляции является *Java* Virtual Machine (JVM).

Преимущество данного метода состоит в адаптации кода к конкретным условиям выполнения.

(b) **AOT-компиляция (Ahead-Of-Time)**

Еще один вид оптимизаций компилятора, но здесь компиляция происходит еще раньше.

Примером в C/C++ является Low Level Virtual Machine ($LLVM$).

Метод отличается быстрой загрузкой и запуском программ, что помогает уменьшить накладные расходы на этапе JIT-компиляции.

4. **Оптимизация кода**

(a) **Инлайн-функции**

Цель: Уменьшение накладных расходов на вызов функций. При инлайне предполагается, что компилятор заменит од в определении функции вместо каждого вызова этой функции.

Вызов функции требует поэтапного обращения к стеку, что устраняется путем встраивания функции.

Стоит помнить, что общий размер программы может увеличиться при большом количестве инлайн-функций.

(b) **Удаление мертвого кода**

Мертвым кодом являются неиспользуемые и недостижимые части программы.

Удаление мертвого кода способствует уменьшению размера программы, более удобному чтению.

Мертвый код можно обнаружить методами статического анализа.

Основное, что объединяет вышеперечисленные методы в рамках данной работы, это ограниченность их анализа спецификацией языка. То есть анализ идет на основе того, как конкретный пользователь языка использует его конструкции в своей программе, не затрагивая идеи, на которых строился и писался сам язык программирования.

Теперь рассмотрим инструменты, по сути своей выделяющие из конкретных языков программирования некоторые подмножества с фокусом, например, на производительность, безопасность или надежность кода.

1. **Ada**

Язык Ada имеет концепцию ограничительных прагм, которые позволяют разработчикам ограничивать использование определенного функционала языка ради повышения производительности и надежности кода. Такие ограничения выдвигают запрет на использование динамической памяти или, например, запрет на использование исключений.

2. **SPARK (подмножество Ada)**

   У языка программирования Ada есть одно подмножество, - SPARK, которое предназначено для использования в системах с высокими требованиями к надежности и безопасности. Данное подмножество аналогично прошлому исключает некоторые синтаксические наборы, чтобы обеспечить возможность формальной верификации кода, тем самым значительно повышая безопасность и надежность написанного.

3. **Ada Ravenscar**

   Рассматривая другое подмножество Ada, названное Ravenscar, стоит упомянуть о принципиальной разнице со SPARK подмножеством: в данном уже не говорится про обеспечение формальной верификации кода. Тем не менее концепция ограничения некоторых возможностей языка сохраняется, чтобы обеспечить однозначность выполнения программы.

4. $MisraC/\mathrm{C}++$

   $MISRA$ (Motor Industry Software Reliability Association) C и $\mathrm{C}++$ — это набор рекомендаций и правил для написания безопасного и надежного кода на языках C и $\mathrm{C}++$. Эти правила часто используются в автомобильной и других критически важных отраслях, где важна высокая производительность и безопасность.

5. $CERTC$

   $CERTC$ — это набор руководящих принципов для написания безопасного и надежного кода на C, разработанный Software Engineering Institute (SEI) при Carnegie Mellon University. Стандарт фокусируется на безопасности: включает рекомендации по предотвращению распространенных уязвимостей, таких как переполнения буфера, ошибки с указателями.

6. **Standard ECMA-327**

   Данная версия сосредотачивается на минимизации использования памяти и процессорного времени для устройств с ограниченными ресурсами.

7. *Java* **ME (Micro Edition)**

   Для *Java* существует подмножество под названием $JavaME$ (Micro Edition), специально разработанное для устройств с ограниченными ресурсами, таких как встроенные системы и мобильные устройства. $JavaME$ предоставляет сокращенный набор библиотек и API, подходящих для встраиваемых сред, и исключает более ресурсоёмкие функции, типичные для $JavaSE$ (Standard Edition).

Данные варианты максимально близки идейно с тем, что будет далее представлено в работе. Теперь, когда проведен обзор разного рода решений, направленных на анализ с целью ускорения работы программы, ясны методы и инструменты, необходимые для реализации поставленных целей.

Глава 4

Теоретическая часть

Рассмотрим некоторые теоретические аспекты, которые требуются для выполнения задач данной работы.

4.1 Целевой язык программирования

Для успешной реализации системы анализа кода для повышения производительности программ на языке программирования высокого уровня требуются некоторые знания о целевом языке программирования.

Выбранный язык сочетает и поддерживает функции, которые используются во многих известных языках программирования, где эти инструменты уже доказали свою полезность и мощность. Он поддерживает императивные, объектно-ориентированные, функциональные и обобщенные парадигмы программирования, объединяя их безопасно и последовательно. В то же время целевой ЯП не поддерживает функции, позволяющие разработчикам программного обеспечения писать опасный, небезопасный или неэффективный код. В частности, язык использует принцип строгой статической типизации. Он не допускает динамических изменений типов, так как типы объектов определяются их объявлениями. Их семантическая корректность проверяется на этапе компиляции.

Основные аспекты, характеризующие данный язык в целом:

1. Объектная ориентированность

   Поддержка традиционного подхода к программированию на основе классов и объектно-ориентированного программирования (ООП). Основные понятия этого подхода следующие:

   (a) Классы с единичным наследованием

   (b) Интерфейсы как абстракции, которые реализуются классами

   (c) Виртуальные функции (члены класса) с механизмом динамического переопределения

Объектная ориентированность, общая для многих (если не всех) современных языков программирования, обеспечивает мощный, гибкий, безопасный, понятный и адекватный дизайн программного обеспечения.

2. Модульность

   Язык поддерживает компонентный подход к программированию. Предполагается, что программное обеспечение разрабатывается и реализуется как композиция единиц компиляции. Единица компиляции обычно представлена в виде модуля или пакета.

3. Статическая типизация Связь с типом, например, возвращаемого значения функции или переменной, происходит сразу, в момент объявления, и не может быть изменен позже.

4. Управляемость

   Язык работает в управляемой среде выполнения, и управление ресурсами осуществляется не напрямую операционной системой, а через промежуточное программное обеспечение, которое обеспечивает ряд услуг, таких как управление памятью и сборка мусора (garbage collection).

4.2 Бенчмаркинг

После исследования спецификации целевого языка программирования требуется для каждого потенциально медленного метода А, имеющего альтернативу в виде метода Б, провести замеры и сравнить скорость выполнения кода или время старта выполнения. Для выполнения данной задачи в работе используется подход, называемый бенчмаркингом.

В общем случае бенчмарком называется программа, которая измеряет некоторые характеристики производительности приложения или фрагментов кода. В данной работе предлагается считать бенчмарк экспериментом, так как с его помощью должны получаться результаты, позволяющие узнать более подробно поведение приложения.

Получив значения метрик, нужно объяснить их и быть уверенным в том, что предложенное объяснение является верным. В ходе работы часто будет использоваться бенчмарк для сравнения некого метода А и метода Б в рамках конкретного функционала языка.

Тривиального сравнения скорости методов с последующим предпочтением более быстрого не является корректным решением ни одной из задач данной работы, так как является в корне не верным подходом. Подобного рода вывод носит строго локальный характер, исключает анализ данных, на которых проводились замеры, состояние системы, причины увиденной разницы в скорости.

Зачастую аномальные значения в результатах замеров производительности связаны с ошибками в методологии измерений. Таким образом, фиксация прироста скорости метода А по сравнению с методом Б не является решенной задачей. Кроме того, бенчмаркинг в целом не является универсальным подходом, полезным при любом исследовании производительности.

Одним из наиважнейших требований к качественному бенчмаркингу является повторяемость полученных результатов. Предполагается, что между измерениями допустима только незначительная разница, не влияющая на выводы из эксперимента и не дающая качественных различий в полученных результатах, относящихся к разным запускам.

4.3 Абстрактное синтаксическое дерево (AST)

При реализации система анализа, возник вопрос о том, как именно его производить, чтобы это было максимально быстро и корректно. Выбор был принят в пользу изучения абстрактного синтаксического дерева (Abstract Syntax Tree или AST) после прохода основного этапа проверки, заключающей, сожержит ли программа ошибки времени компиляции. Соответственно, необходимо уметь понимать это дерево на выходе.

Итак, у каждого ЯП есть набор ключевых слов, по которым мы определяем некоторые выражения в программе, задаем инструкции и получаем из наших входных данных определенный результат. AST отличается понятной структурой для человека.

Можно определить язык программирования высокого уровня как язык с упором на читаемость человеком. Но программа на языке высокого уровня не запустится, если не перевести ее в код на языке низкого уровня. За данный этап ответственны два процесса: компиляция и интерпретация.

Разработать код на языке высокого уровня и перевести его в список команд низкого уровня позволяют AST-деревья. Абстрактное синтаксическое дерево представляет собой один из промежуточных слоев при преобразовании языков высокого уровня.

Устройство AST следующее:

1. Промежуточными узлами являются функции, методы, операторы
2. В листах дерева содержатся константы, переменные, методы классов, аргументы

AST-дерево не является деревом разбора, так как не учитывает синтаксис языка.

4.4 Спецификация языка программирования

Спецификация языка программирования — это документ, который определяет язык программирования, чтобы пользователи и разработчики могли однозначно понимать, что означают программы на этом языке.

Спецификация языка программирования может принимать несколько форм, включая следующие:

1. Явное определение синтаксиса и семантики языка. В целом, синтаксис обычно задается с использованием формальной грамматики, семантические определения могут быть написаны на естественном языке (например, подход, использованный для языка C) или формальной семантики (например, спецификации Standard ML и Scheme).

2. Описание поведения транслятора языка (например, C++ и *Fortran*). Синтаксис и семантика языка должны быть выведены из этого описания так, чтобы его можно было написать на естественном или формальном языке.

3. Модельная реализация, иногда написанная на целевом языке. Модельная реализация является по определению стандартом, на основе которой измеряются все другие реализации с соответствующими начальными условиями и к которой добавляются все улучшения.Синтаксис и семантика языка явны в поведении модельной реализации.

Синтаксис языка программирования обычно описывается, используя комбинацию следующих двух компонент:

1. Регулярные выражения для описания лексем.

2. Грамматики, которые описывают, какие именно комбинации лексем корректны синтаксически.

Формулирование строгой семантики большого, сложного и применимого на практике языка программирования является непростой задачей даже для опытных специалистов, а полученная спецификация может быть трудной для понимания пользователем. Рассмотрим некоторые способы описания семантики языка программирования:

1. Естественный язык: описание на естественном языке

2. Формальная семантика: математическое описание.

3. Эталонные реализации: описание с помощью кода программы.

4. Тестовые наборы: описание с использованием примеров программ и их ожидаемого поведения.

Стоит отметить, что все языки используют хотя бы один из этих методов описания, а некоторые языки сочетают в себе сразу несколько.

Наиболее широко используемые языки определяются с использованием описаний их семантики на естественном языке. Это описание обычно принимает форму справочного руководства по языку. Причем такие документы могут содержать сотни страниц,

например, печатная версия «Спецификации языка Java», 3-е издание имеет объем 596 страниц.

Неточность естественного языка как средства описания семантики языка программирования может привести к проблемам с интерпретацией спецификации. Например, семантика потоков в *Java* была определена на английском языке, и позже выяснилось, что спецификация не дает качественных указаний для разработчиков.

4.5 Clang-Tidy

Clang-Tidy — это инструмент, представляющий собой «линтер» C++ на основе clang. Его цель — предоставить расширяемую среду для диагностики и исправления типичных ошибок программирования, таких как нарушения стиля, неправильное использование интерфейса или поиск ошибок с применением статического анализа. $Clang-Tidy$ является модульным и предоставляет удобный интерфейс для написания новых проверок.

$Clang-Tidy$ имеет свои собственные проверки, а также может запускать проверки Clang Static Analyser. Каждая проверка имеет имя, а сам список проверок можно выбрать с помощью специального параметра $-checks=$.

Однако опция $-checks=$ не влияет на аргументы компиляции, поэтому она не может включать предупреждения Clang, которые еще не включены в конфигурации сборки. Параметр $-warnings-as-errors=$ преобразует любые предупреждения, выдаваемые под флагом $-checks=$, в ошибки (но сам по себе никаких проверок не включает).

Диагностика clang-tidy предназначена для вызова кода, который не соответствует стандарту кодирования или каким-либо образом проблематичен. Однако, если известно, что код правильный, может быть полезно отключить предупреждение.

Если конкретный механизм подавления недоступен для определенного предупреждения или его использование по какой-либо причине нежелательно, $Clang-Tidy$ имеет общий механизм подавления диагностики через комментарии формата $NOLINT$, $NOLINTNEXTLINE$ и $NOLINTBEGIN...NOLINTEND$.

Комментарий $NOLINT$ предписывает Clang-tidy игнорировать предупреждения в той же строке, то есть он не применяется к функции, блоку кода или любой другой языковой конструкции. Комментарии $NOLINTBEGIN$ и $NOLINTEND$ позволяют подавлять предупреждения Clang-tidy в нескольких строках, расположенных между двумя комментариями, включающих, соответственно, $NOLINTBEGIN$ и $NOLINTEND$.

За всеми комментариями может следовать необязательный список имен проверок в круглых скобках. Список имен проверок поддерживает подстановку с тем же форматом и семантикой, что и при включении проверок.

Формальный синтаксис имеет следующее описание:

```
lint-comment:
  lint-command
  lint-command lint-args
```

```
lint-args:
  ( check-name-list )

check-name-list:
  check-name
  check-name-list , check-name

lint-command:
  NOLINT
  NOLINTNEXTLINE
  NOLINTBEGIN
  NOLINTEND
```

Конкретный пример:

```
class Foo {
  // Отключение диагностики для данной линии
  Foo(int param); // NOLINT

  // Отключение конкретной проверки со следующей линии
  // NOLINTNEXTLINE(google-explicit-constructor, google-runtime-int)
  Foo(bool param);

  // Отключение всех проверок из google модуля
  // для всех линий между BEGIN и END
  // NOLINTBEGIN(google*)
  Foo(bool param);
  // NOLINTEND(google*)
};
```

Все рассмотренные выше разделы в комбинации будут применяться для выполнения задач данной работы.

Глава 5

Практическая часть

Изучив существующие решения, перейдем непосредственно к анализу текущей спецификации целевого языка программирования на предмет потенциально медленных языковых конструкций и функционала и сбору данных для дальнейшего тестирования.

Недостаток данной работы заключается в невозможности полного анализа спецификации, так как выбранный ЯП находится в стадии активной разработки. Несколько раз в месяц документ обновляется, и одни положения из него уходят, одновременно с тем как приходит много новых глав и подразделов, исправляются ошибки.

По данной причине на данном этапе была выбрана стратегия исследования двух категорий подмножеств языка:

1. Подмножество, общее с *TypeScript*. То есть та часть языка, которая может быть аналогично воспроизведена с помощью *TypeScript*, без внесения каких-либо изменений в текущий код программы.

2. 'Фундаментальное' подмножество. Будем условно считать в рамках данной работы фундаментальными те конструкции языка, при изменении которых потребуются значительные изменения в комплементарных главах спецификации, и значительно меняющие поведение программы на целевом языке программирования.

Конечно, спецификация ЯП не делится на описание подмножеств подобного характера. Поэтому их поиск заключает в себе прежде всего последовательное чтение документа с претензией на его неоптимальность. В текущих условиях то, какие именно гипотезы пошли на дальнейшее тестирование, зависит исключительно от опыта и знаний автора работы.

5.1 Работа со спецификацией

5.1.1 Имплицитный боксинг и анбоксинг

Одна из первых глав спецификации, логично, содержит описание типов в данном языке программирования. Упоминается о том, что выбрана статическая типизация,

что, как было упомянуто в соответствующем теоретическом разделе, означает, что тип каждой объявленной сущности и каждого выражения известен в время компиляции. Тип объекта либо явно задается разработчиком, либо неявно выводится компилятором.

Существует две категории типов:

1. Типы значений (Value Types).
2. Ссылочные типы (Reference Types).

Типы, встроенные в целевой ЯП, называются предопределенными.

К предопределенным типам относятся примитивные типы данных, в том числе *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, *char*.

Каждому предопределенному типу значения соответствует предопределенный тип класса, называемый упаковочным (или боксингом), который обертывает значение предопределенного Value Type: *Number*, *Byte*, *Short*, *Int*, *Long*, *Float*, *Double*, *Boolean*, *Char*.

Перевод из Value Type в соответствующий Reference Type называется упаковкой (боксингом). Обратный перевод называется распаковкой (анбоксингом).

Спецификация языка программирования позволяет для обеих категорий типов выполнять ряд базовых операций, например, арифметических. То есть корректно с точки зрения языка объявление двух переменных, допустим, Value Type *int* и их последующее сложение с записью результата в переменную такого же типа наравне с аналогичной операцией для Reference Type *Int*.

Справедливо предположение о том, что для выполнения примитивных операций и ряда других сценариев необязательно использование более 'сложных' типов данных, которые несут в себе дополнительный функционал. Для подобных сценариев отлично подходят более 'легкие' Value Types, так как для работы с ссылочным типом надо аллоцировать место в куче. Значимый тип может работать на стеке, не уходя в кучу, а может стать частью ссылочного типа. Это свойство может значительно повысить производительность для некоторых программ. Дополнительно, при многих, казалось бы, обычных операциях, происходит постоянная неявная конвертация из Value Type в Reference Type, что просто по определению несет в себе дополнительные затраты ресурсов, увеличивая время исполнения программы.

Итак, предложена проверка на выявление неявной упаковки и распаковки с предложением переписывания на соответствующие упрощающие типа данных в каждом контексте.

5.1.2 Модификатор *final* для методов и классов

Некоторая вводная информация про классы в выбранном ЯП: объявления классов вводят новые ссылочные типы и описывают способ их реализации; классы могут быть верхнего уровня (Top-level) или локальными; тело класса содержит объявления

и инициализаторы классов; объявления могут представлять члены или конструкторы классов; тело объявления члена включает в себя область действия объявления; члены класса включают в себя поля, методы и аксессоры; члены класса могут быть объявлены или унаследованы.

Что принципиально важно, по умолчанию все классы в целевом ЯП являются открытыми для наследования. То есть при отсутствии модификатора *final* у метода или у самого класса, можно производить наследование.

Именно с этого момента возникает проблема. Язык мог бы иметь противоположный модификатор по умолчанию, как это сделано в *Kotlin*, однако это подразумевает большой пласт несовместимости с *TypeScript*. Рассмотрим следующий частый пример: разработчик пишет свою программу, определяет классы, интерфейсы и не продумывает детально механизмы взаимодействия между ними. Соответственно, в итоге программа содержит много классов, которые не имеют никаких наследников, ровно как и их методы, и данная программа даже отлично работает и не выдает никак проблем.

Однако очевидна неоптимальность: классы и методы, не имеющие наследников, не помечены модификатором *final*. Например, при вызове метода *TypeScript* проверяет, есть ли этот метод в самом классе. Если метод не найден, он ищет его в родительском классе, продолжая вверх по цепочке наследования до тех пор, пока не найдет метод или не достигнет верхнего уровня (базового) класса. Вставка *final* позволяет не делать лишние проверки и поиски в случае отсутствия наследников.

5.1.3 Инструкции верхнего уровня

Инструкции верхнего уровня (Top-level statements) - это языковые конструкции, допустимые к использованию без необходимости оборачивать их в функции, классы или иные структуры данных. Отдельный модуль может содержать последовательности операторов, которые логически составляют одну единую последовательность операторов.

Если один модуль импортирует другой, то семантика операторов состоит в определении импортируемого модуля. Таким образом, все операторы верхнего уровня выполняются один раз до вызова какой-либо другой функции или до доступа к какой-либо переменной верхнего уровня.

Если отдельный модуль используется в качестве программы, то инструкции верхнего уровня используются как точка входа программы. Если отдельный модуль содержит *main* функцию, то она выполняется сразу после исполнения инструкций верхнего уровня.

Ошибка времени компиляции (Compile Time Error) в целевом ЯП возникает, если инструкции верхнего уровня содержат *return*.

Инструкции верхнего уровня можно использовать непосредственно в теле файла без необходимости оборачивать их в функции, классы или другие структуры. Эта возможность существует во многих ЯП, например, в C# и Kotlin. Это позволяет исполнять код сразу при запуске программы, что делает её удобной для написания простых скриптов

и небольших программ.

Инструкции верхнего уровня просты в использовании и потому очень удобны для небольших приложений и скриптов, где не требуется большой работы с ресурсами. Кроме того, линейность этих и инструкций делает код более простым для понимания.

Выскажем предположение, что это же порождает проблемы в более высоко нагруженных системах. Например, если выполняются какие-либо тяжелые вычисления или операции ввода-вывода, это уже значительно увеличивает время запуска приложения, что может уже быть критично при разработке в мобильной сфере, так как пользователи не любят долго ждать. Кроме того, второй популярный сценарий разработки - использование библиотек и фреймворков, и при их подгрузке время запуска растет значительно.

Перемещение инструкций верхнего уровня в ту же функцию *main* избавляет от потенциально лишних операций и увеличивает вероятность быть оптимизированным за счет $JustInTime$ (JIT) компиляции. Например, если модуль импортируется, но не вся импортированная функциональность используется, то инструкции верхнего уровня все равно выполнятся полностью и засорят код. В случае перемещения в функцию *main* гарантируется, что импорты будут только при необходимости.

5.1.4 Операторы равенства

Согласно спецификации выбранного ЯП, выражения равенства используют операторы равенства ==, ===, ! = и ! ==. Во всех случаях выражение $a! = b$ дает тот же результат, что и $!(a == b)$, а выражение $a! == b$ дает тот же результат, что и $!(a === b)$.

Рассмотрим случай конкретно ссылочного типа данных. Сравнение ссылок применяется, если оба операнда являются совместимыми ссылочными типами, за исключением типов *string*, *bigint*, *Object*, объединенных типов ($Union$) и типовых параметров ($TypeParameters$).

Целевой статически типизированный ЯП предоставляет расширенную семантику для сравнения с *null* и $undefined$ ради более полного соответствия $TypeScript$. Любая сущность может быть сравнена с *null* с использованием операторов == и ! =. Это сравнение может вернуть значение *true* только для *nullable* типов, если они действительно *null* во время выполнения программы ($runtime$). В остальных случаях сравнение с *null* возвращает $false$, и это известно во время компиляции.

Аналогично, сравнение с $undefined$ есть $false$. Но, если переменная, с которой производится сравнение, является типа $undefined$ или объединенного типа, который включает $undefined$ как один из своих типов - получаем *true*.

Изучив реализацию данного языка программирования на основе спецификации, было обнаружено, что одна и та же проверка может быть выполнена с разной скоростью в зависимости от порядка расположения операндов в операторе равенства, а именно $ref == null$ или $null == ref$.

Данный эффект ожидаем в связи с тем, что первым шагом при вычислении значении такого равенства идет проверка левой части на *null*, а только потом уже сравнение операндов. Во втором же случае можно предположить, что проверка на *null* по сути опускается, что сильно должно уменьшать время исполнения, ведь подобного рода проверки делаются сильно чаще и не являются примитивными, так как речь идет о ссылочном сравнении.

5.1.5 Использование корутин вместо асинхронных функций

Асинхронные функции в *TypeScript* представляют собой функции, которые используют ключевое слово *async*. Они позволяют выполнять операции асинхронно и возвращать *Promise*. Внутри такой функции могут использоваться операторы *await*, которые приостанавливают выполнение функции до тех пор, пока не завершится операция, указанная после *await*.

Корутины же позволяют явно управлять состоянием выполнения, что может быть полезно для сложных потоков работы или взаимодействия с внешними ресурсами. Корутины могут приостанавливать своё выполнение (*yield*) и возобновлять его позже, сохраняя состояние между вызовами.

Асинхронная функция является неявной корутиной, которую нельзя вызывать как обычную функцию. Использование такой аннотации не рекомендуется данным ЯП. Этот тип функций поддерживается только для обратной совместимости с *TypeScript*. Так как данный язык в целом ориентирован на ускорение TS, это добавляет аргумент в пользу необходимости сравнения асинхронных функций и корутин.

5.1.6 Использование вызова функций вместо лямбд

Конечно, вот перевод на русский:

Лямбда — это короткий блок кода, который принимает параметры и может возвращать значение. Лямбда похожи на функции, но не требуют имен в описаниях. Лямбды могут быть реализованы непосредственно в выражениях.

Под обычными функциями в данном контексте подразумеваются функции, объявленные с использованием ключевых слов $function$ и других аналогичных.

При вызове лямбда-функции создаётся объект и определяется виртуальная функция. Затем, при использовании, будет выполнен виртуальный вызов, который требует больше времени, чем вызов обычной функции. В среднем, при повторном вызове в лямбда-функции, эти различия должны занимать больше времени. Поэтому данная гипотеза пригодна для проверки бенчмаркингом на иследующем этапе.

5.1.7 Остаточные параметры (rest parameters)

Остаточные параметры в *TypeScript* позволяют функции принимать переменное количество аргументов в виде массива. Обозначение таких параметров реализовано с помощью троеточия (...) перед именем параметра.

Таким образом, когда мы используеем (...) перед именем параметра, *TypeScript* собирает все переданные аргументы, начиная с этого места, в массив.

Возникает соответствующая мысль о том, что можно сразу положить все переданные аргументы в массив и не создавать его каждый раз с нуля при остаточных параметрах в сценарии, например, большого цикла. Тогда массив создастся ровно один раз с проверкой наличия необязательных параметров. Относительно неплохой альтернативой между вариантами массива и остаточных параметров могут быть опциональные параметры.

В спецификации выбранного языка программирования говорится о двух формах опциональных параметров. Первая форма содержит выражение, которое указывает значение по умолчанию. Значение параметра устанавливается в значение по умолчанию, если аргумент, соответствующий этому параметру, не передается при вызове функции.

Вторая форма является краткой записью для параметра объединенного типа $T|undefined$ с значением по умолчанию $undefined$.

Есть необходимость сравнить поведение массивов, опциональных и остаточных параметров

5.2 Бенчмаркинг выдвинутых гипотез

Итак, когда в рамках просмотра было выдвинуто по меньшей мере 7 идей для ускорения, необходимо их протестировать. Тестирование производится путем бенчмаркинга в стиле сравнения варианта А с вариантом Б.

Пройдемся по каждой из выдвинутых гипотез.

5.2.1 Имплицитный боксинг и анбоксинг

Имплицитный боксинг и анбоксинг типов случается огромное множество раз в рядовой программе на том же *TypeScript*. Постоянное преобразование типов в *switch* и просто при выполнении примитивных операций, сравнений или передаче аргументов в функцию довольно сильно замедляет программу, согласно замерам на устройстве.

При бенчмаркинге стандартной настройкой являются максимальные оптимизации компилятора и JIT-компиляция. Однако приводятся замеры и без JIT, чтобы оценить вклад оптимизаций в быстродействие варианта А или варианта Б.

Приведу результаты бенчмарков нескольких форматов.

1. Тип аргументов *double*

Тип запуска	Боксинг аргумента	Аргумент без боксинга	Ускорение
Время в сек, *JIT*	3,75	1,75	53,3%
Время в сек	3,768	1,75	53,5%

Таблица 1. Тип запуска *double*

2. Тип аргументов *int*

Тип запуска	Боксинг аргумента	Аргумент без боксинга	Ускорение
Время в сек, *JIT*	19,776	1,148	94,2%
Время в сек	20,084	1,115	94,4%

Таблица 2. Тип запуска *int*

3. Выполнение в *switch*

Тип запуска	Боксинг	Без боксинга	Ускорение
Время в сек, *JIT*	2,149	1,061	50,6%
Время в сек	2,129	1,039	51,1%

Таблица 3. Выполнение в *switch*

По данным таблицы очевиден большой прирост в скорости в отсутствии лишних перегонок из упакованного типа в неупакованный и наоборот. Таким образом, проверка на неявную упаковку и распаковку должна присутствовать в системе анализа и предупреждений.

5.2.2 Модификатор *final* для методов и классов

Согласно выдвинутой гипотезе, предложение использовать модификатор *final* для классов и методов, не имеющих наследников, должно существенно ускорить выполнение программы, так как больше нет необходимости в поисках по таблице методов при вызове.

Наследники ожидаются в том числе в качестве локальных классов. Одновременно с этим, нельзя предлагать использование данного модификатора в местах, где это в соответствии со спецификацией выбранного языка программирования повлечет ошибку времени компиляции. То есть во всех случаях вариант А и вариант Б приводят к одинаковому результату выполнения программы.

Запрещено использование ключевого слова *final* для абстрактных классов, классов или методов с наследниками, а также статических.

Так как целевая метрика предполагает включение *JIT*-компиляции, итоговый прирост составляет порядка 13%, что является отличным результатом для легковесного бенчмарка.

Очевидно, что на более крупных системах итоговый прирост будет в процентном соотношении не таким заметным. Однако много маленьких ускорений влекут хороший суммарный результат.

Тип запуска	$final$	Без $final$	Ускорение
Время в сек, JIT	21,800	25,107	13,1%
Время в сек	21,88	25,106	12,8%

Таблица 4. Модификатор в $final$

Таким образом, данный тип проверки тоже рекомендован к включению в анализирующую систему.

5.2.3 Инструкции верхнего уровня

Запрет на использование инструкций верхнего уровня предполагает ускорение времени запуска приложения. То есть время, которое требуется затратить, чтобы приступить к исполнению $main$ функции.

В распоряжении автора работы для оценки такого эффекта не имеется полноценных программных инструментов, чтобы произвести корректную и точную оценки, поэтому замеры для данной гипотезы были проведены с использованием подручных средств и имеют большую погрешность в результатах.

Тем не менее прирост очевиден и велик, если в качестве сценария взять, например вычисление суммы двух чисел в цикле, и вариант А заключить в виде инструкций верхнего уровня, а в случае варианта Б в виде тела функции $main$. Этот пример легко масштабируется до уровня библиотек и фреймворков.

Потому, данную категорию запретов стоит внести в анализирующую систему аналогично паре предыдущих.

5.2.4 Операторы равенства

Обоснование гипотезы в разделе работы со спецификацией дало аналогичные результаты в виде целевого байт-кода, и гипотеза подтвердилась. Данный результат абсолютно закономерен, ведь уменьшение количества проверок за счет без уменьшения безопасности используемого кода является очевидной оптимизацией.

Тип запуска	$ref == null$	$null == ref$	Ускорение
Время в сек, JIT	5,261	4,0728	22,5%
Время в сек	5,113	4,203	17,8%

Таблица 5. Операторы равенства

К тому же, хороший прирост дала JIT-компиляция. Отдельно отметим, что переписывание кода с варианта А на вариант Б абсолютно тривиально и не требует никаких дополнительных исследований на корректность и валидность с точки зрения языка.

Таким образом, проверка на операторы равенства тоже будет добавлена в анализирующую систему.

5.2.5 Использование корутин вместо асинхронных функций

Использование корутин носит несовместимый с *TypeScript* характер. Кроме того, предложение по переписыванию асинхронных функций на корутины имеет больше стилевой и инженерный смысл к текущему моменту развития спецификации. Это означает, что большого прироста производительности не ожидается. Однако замеры имеют следующий характер:

1. Использование со *string*

Тип запуска	Асинхронный	Корутины	Ускорение
Время в сек, *JIT*	1,318	1,426	-7,6%
Время в сек	1,326	1,432	-7,4%

Таблица 6. Использование со *string*

2. Использование с *int*

Тип запуска	Асинхронный	Корутины	Ускорение
Время в сек, *JIT*	1,915	1,886	1,5%
Время в сек	1,838	1,819	1,03%

Таблица 7. Использование с *int*

Видно, что результаты носят не общий характер, и все зависит от использования конкретных типов. Тем не менее есть случаи, удачные для оптимизации. С учетом позиции, указанной в спецификации языка как приоритетной с точки зрения стиля и, возможно, будущего видения развития проекта, данная проверка будет внесена в анализирующую систему с ограничениями.

5.2.6 Использование вызова функции вместо лямбд

В общем случае, вызов обычных функций ожидается немного более эффективным, чем вызов лямбд. Это ожидание базируется на дополнительной работе, которую компилятор должен выполнить для обработки лямбд.

Конечно, конкретные примеры стоит замерять инструментами другого типа, нежели системой, подобной разрабатываемой в данной работе.

Выдвинутая гипотеза подтверждается в замерах бенчмаркингом:

Тип запуска	Функции	Лямбды	Ускорение
Время в сек	9,718	11,195	13,2%
Время в сек, *JIT*	9,479	11,643	18,5%

Таблица 8. Функции и лямбды

И снова целевая метрика с *JIT* дает существенный прирост в и без того немалом различии между вариантом А и вариантом Б. Данный тип проверок имеет смысл включить в предложенную в теме работы систему.

5.2.7 Остаточные параметры (rest parameters)

Итак, существует 3 способа передать неопределенное количество параметров функции: остаточные параметры, опциональные параметры и массивы.

В данном бенчмарке используем максимальные оптимизации компилятора и всегда с *JIT*-компиляцией.

Результаты имеют следующий характер:

№	Массив, с	Опциональные, с	*Rest*, с	Комментарии
1	556	6768	10331	Массив строится 1 раз до цикла с проверкой неиспользуемых параметров
2	10325	6715	10319	Массив строится на каждой итерации с аналогичной проверкой
3	10410	6682	10286	Массив строится на каждой итерации без проверки неиспользуемых параметров

Таблица 9. Функции и лямбды

Из данной таблицы совершенно отчетливо прослеживается тенденция: остаточные параметры медленнее в разы, чем массивы или опциональные параметры. Выбор использования массива или опциональных параметров для переписывания больше зависит от конкретной задачи.

Остаточные параметры по определению являются синтаксическим сахаром, то есть является более кратким, интуитивно понятным способом записи с точки зрения человека, не добавляя новых возможностей или изменений в язык. В данной ситуации это отчетливо прослеживается наличием двух конструкций, позволяющих получить тот же результат работы программы.

Все эти наблюдения позволяют внести остаточные параметры в список пунктов проверки будущей системы анализа и предупреждений.

5.3 Разработка анализирующей системы

Переходим к разработке самой системы, выдающей предупреждения по неэффективности кода.

Совершенно бессмысленно выполнять анализ на эффективность кода до потверждения его корректности на этапе компиляции. Поэтому данная система будет идти следующим этапом и работать по тому же принципу, что проверка на мертвый код. Мертвым кодом называются участки программы, которые либо не могут быть исполнены, либо их выполнение никак не влияет на результат работы программы.

Так как система анализа и предупреждений не является обязательной для достижения корректности приложений, для реализации предложена следующая структура анализатора:

1. Все проверки активируются или деактивируются через механизм флагов компиляции.

2. Флаги разделены на группы по принципу общности с *TypeScript* и, наоборот, отсутствию совместимости с ним.

3. Существует флаг, запускающий абсолютно все проверки системы.

4. По аналогии с предупреждениями компиляторов *gcc* и *clang*, есть возможность воспринимать все выдвинутые системой предупреждения как ошибки времени компиляции.

Данная структура является понятной и довольно простой в реализации при условии корректной работы с *AST*.

Процесс работы анализатора на основе предложенной схемы в итоге выглядит следующим образом:

1. Происходит считывание флагов компиляции. Если среди них есть флаги, относящиеся к системе анализа и предупреждений, она запустится после проверки программы на корректность.

2. Система анализа при вызове получает программу как набор узлов, имеющих свой тип после прохождения этапа компиляции. Например, существуют узлы присваивающего выражения, узлы декларации классов и другие.

3. В таком виде программа проходит каждую проверку из списка активированных. Обход дерева происходит от самого его корня, постепенно спускаясь от родительских узлов к дочерним.

4. Сами проверки завязаны на имеющийся тип текущего узла и на контекст остального поддерева, необходимого для однозначного решения о вынесении предупреждения о неэффективности. Например, в случае проверки на модификатор *final* для классов и методов, происходит поиск наследников по всей программе, в том числе в других модулях и среди локальных классов.

5. При выполнении условий, соответствующих возможности оптимизировать код, выносится соответствующее сообщение, которое содержит имя проверенного файла, строку и индекс проверенного узла и текст, описывающий неэффективность с предложением исправления.

6. В случае, если активна опция за восприятие предупреждений как ошибок, работа анализирующей системы останавливается сразу после первого выданного сообщения, происходит ошибка времени компиляции.

Этот процесс работы получится несколько видоизмененным после внедрения запрещающей системы.

5.4 Разработка запрещающей системы

Как было сказано в предыдущем пункте, опции компиляции регулируют, какие проверки будут активированными для конкретной программы, а какие будут отключенными. Это позволяет хорошо персонализировать анализатор под нужды разработчика. Тем не менее этого недостаточно.

Популярный сценарий использования аналогичных систем, как описанный ранее $Clang-Tidy$ позволяет точечно добавлять или отключать проверки в самом коде программы в виде комментариев специального формата.

Для данной системы внедрен аналогичный функционал: отключение проверок для текущей строки, для следующей строки, для блока кода, заключенного между комментариями, обозначающими начало и конец выбранных строк. Реализация выполнена в том же стиле, что и $Clang-Tidy$, то есть в виде специализированных однострочных и многострочных комментариев.

Процесс работы запрещающей системы выглядит следующим образом:

1. При триггере анализирующей системы происходит парсинг кода на предмет наличия специальных комментариев запрещающей системы.

2. Все узлы, имеющие запрет, помечаются специальным флагом типа, связанного с запретом конкретной проверки.

3. При работе анализирующей системы все узлы перед началом анализа проверяются на наличие запрещающего флага типа, относящегося к данной категории проверки. То есть, если есть узел, помеченный запретом только на проверку неявной упаковки, запрет сработает при проверке этого узла на неявную упаковку, но никакого запрета не будет при анализе данного узла на, например, остаточные параметры.

4. При отсутствии запрета узел будет проверяться, а при его присутствии будет просто пропущен анализирующей системой для анализа, но не будет удален из анализирующего поддерева. Также рекурсивный спуск от корня к дочерним узлам не останавливается при встрече запрета, что позволяет не упускать связанные блоки из анализатора.

Таким образом, разработана запрещающая система для данного анализатора и сам анализатор. Для достижения целей работы осталось выполнить одну задачу, связанную с системой проектной сборки.

5.5 Поддержка анализа через систему сборки

Как известно, язык программирования *TypeScript* в качестве системы сборки использует так называемый $tsconfig$, который представляет собой обычный $json$-файл. Этот файл выполняет следующие задачи:

1. Установка корневого каталога проекта *TypeScript*.
2. Установка файлов проекта.
3. Настройка параметров компиляции.

Для его использования нужно вручную добавить новый файл с именем $tsconfig.json$ в корень проекта.

Каждый $tsconfig.json$ представляет собой файл, который содержит ряд секций. Например, секция *compilerOptions* настраивает параметры компиляции. Здесь можно указать необходимые параметры и их значения. Параметры называются так же, как и в командной строке. Это значит, что выставить значения можно те же, что передаются в командной строке. Например:

```
{
    "compilerOptions": {
        "target": "es5",
        "removeComments": true,
        "outFile": "../../built/local/tsc.js"
    },
    "files":[
        "app.ts",
        "interfaces.ts",
        "classes.ts",
    ]
}
```

Здесь используется все те же параметры, которые применяются при компиляции в командной строке. Например, параметр "$target$" указывает, какой стандарт *JavaScript* будет применяться при компиляции, параметр "$removeComments$" удаляет комментарии, параметр "$outFile$" задает название выходного файла, а с помощью секции $files$ можно установить набор включаемых в проект файлов.

При необходимости можно включать другие опции компиляции.

Целевой язык программирования имеет систему сборки, абсолютно аналогичную $tsconfig$. Это значит, что включение опций анализатора можно повторить по аналогии, как в командной строке. Таким образом, остается только научить преобразовывать написанный в конфигурационном файле текст в активацию нужных проверок. При добавлении такого раздела вышеуказанный пример изменяется следующим образом:

```
    {
        "compilerOptions": {
            "target": "es5",
            "removeComments": true,
```

```
        "outFile": "../../built/local/tsc.js",
        "analizer": ["--enable-all-checks"]
    },
    "files":[
        "app.ts",
        "interfaces.ts",
        "classes.ts",
    ]
}
```

То есть просто добавляется строчка в раздел *compilerOptions*, что потребовало добавления одной функции в реализации компилятора целевого ЯП.

Таким образом, все поставленные в работе задачи выполнены.

Глава 6

Заключение

Обратимся к разделу постановки проблемы и пройдем по каждому выдвинутому пункту.

1. Существующие решения изучены и качественно рассмотрены. На основании нескольких из них выстроена схема работы системы анализа и предупреждений, а именно: $Clang-Tidy$ система и $StandardECMA$-327 послужили основными опорными точками при построении решения.

2. Проведен анализ текущей спецификации целевого языка программирования. Было выделено 7 гипотез для дальнейшего тестирования средствами бенчмаркинга. Среди выдвинутых предложений оказались: выявление скрытой упаковки и распаковки типов, применение модификатора $final$ для методов и классов при отсутствии наследования, запрет инструкций верхнего уровня, перестановка порядка операндов в выражениях равенства, использование корутин вместо асинхронных функций, использование вызова функций вместо лямбд, замена остаточных параметров на массивы или опциональные параметры.

3. Для каждой гипотезы представлен вариант потенциального ускорения, заключающийся в тривиальном переписывании участка кода, и протестирован с помощью бенчмаркинга.

4. При бенчмаркинге такие проверки, как имплицитный боксинг и анбоксинг типов, операторы равенства, rest параметры показали наиболее значительный прирост в скорости, а проверки на модификатор $final$ в классах и методах, использование вызова функций вместо лямбд дали меньшую, но тоже очень заметную разницу во времени исполнения. Проверка на замену асинхронных функций корутинами дала прирост порядка 1.5% для int типа и показала значительный регресс на тестировании при использовании типа $string$. Эта проверка необходима с точки зрения идеологии целевого языка согласно его спецификации, но требует осторожного применения, более конкретных замеров с использованием других оптимизирующих инструментов. Проверка на запрет использования инструкций верхнего уров-

ня дала уменьшение времени запуска, что также является целевым результатом в рамках поставленных целей и задач работы.

5. Разработана система, анализирующая исходный код согласно включенному набору проверок. Система работает с типами узлов, поддерживает режимы предупреждений и ошибки времени компиляции. Количество проверок составляет семь наименований, причем возможно групповое включение с учетом совместимости и несовместимости с $TypeScript$, а также включение всех проверок сразу одной опцией компиляции.

6. Разработана система-аналог $Clang-Tidy$, позволяющая точечно в исходном коде контролировать то, для какие участки кода следует исключить из проверяющей системы. Поддержаны те же типы специализированных комментариев, что и в $Clang-Tidy$, а именно: отключение для текущей строки, для следующей строки и отключение блока кода, заключенного между комментариями, обозначающими начало и конец запрещающей секции.

7. Поддержана возможность анализа в многофайловой сборке через специальный файл, задающий конфигурацию проекта.

В анализирующую систему включено 7 проверок, которые в среднем дают ускорение не менее, чем на 10% при замере бенчмарками.

Таким образом, поставленные задачи выполнены и цели работы были успешно достигнуты.

Литература

Будет добавлена.