Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Степанов Алексей Макарович

Поддержка постепенной типизации в языке программирования Kotlin

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Бреслав А. А.

Рецензент:
Подхалюзин А. В.

Санкт-Петербург
2017

Оглавление

Введение 4

1 Обзор литературы 5

1.1 Виды типизации в языках программирования 5

1.2 Постепенная типизация в языках программирования 7

1.2.1 PHP 7 и Python 3.5 7

1.2.2 Groovy 7

1.2.3 C# 8

1.2.4 Scala 9

1.2.5 Kotlin (JavaScript) 10

1.3 Постановка задачи 11

2 Подход к решению задачи 12

2.1 Динамическое поведение Groovy 13

2.2 Динамическое поведение C# 14

2.3 Предложенное взаимодействие с динамическим кодом в Kotlin 15

3 Реализация решения 19

3.1 Технические особенности 19

3.1.1 Связь языков Java и Kotlin 19

3.1.2 Осуществление вызовов на Java платформе 21

3.2 Реализация поддержки динамических вызовов на стороне компилятора 22

3.3 Реализация библиотеки для поддержки динамического поведения 24

3.3.1 Загрузочный метод 24

3.3.2 Механизм динамического разрешения перегрузок 25

3.3.3 Обзор ситуаций требующих изменения вызываемого метода . . . 29

3.3.4 Методы выполняющие настройку точки вызова 30

3.4 Запоминание результатов вычислений 33

4 Измерения производительности 34

4.1 Особенности измерения производительности на Java платформе 34

4.2 Окружение при проведении измерений производительности 35

4.3 Рассмотренные тесты производительности 35

4.3.1 Вычисление чисел Фибоначчи 36

4.3.2 Возведение матрицы в квадрат 38

4.3.3 Z-функция 40

4.3.4 Быстрое Преобразование Фурье 40

4.3.5 Разрешение перегрузок методов 40

4.3.6 Разрешение перегрузок на списке динамических объектов 43
4.3.7 Параллельное разрешение перегрузок на списке динамических объектов 46
4.4 Анализ результатов 46

Заключение **49**

Список литературы **50**

Приложение А. Таблицы результатов вычислительных тестов **53**

Приложение Б. Исходный код разработанного решения **62**

Введение

В языках программирования широко распространены приёмы статической и динамической типизации. Статическая типизация отличается ранним обнаружением ошибок в программах, широкой поддержкой программными инструментами и возможностью порождать высокопроизводительный код. Динамическая типизация обеспечивает гибкость кода и упрощает написание несложных программ. В зависимости от задачи, программисту может быть удобно использовать или один или другой вид типизации. Для объединения преимуществ обоих приёмов, используется постепенная типизация (gradual typing) [25]. Есть два традиционных способа введения постепенной типизации в язык программирования. Можно обеспечить гибкость в статически типизированном языке добавлением динамического типа, или можно добавить ограничения и проверки в рамках динамически типизированного языка посредством добавления типовой информации.

Язык программирования Kotlin был разработан как статически типизированный. Kotlin, в качестве целевой платформы, поддерживает JavaScript и байт-код Java. При компиляции в JavaScript возможно использование динамически типизированного кода, а при компиляции в байт-код Java — нет.

В рамках данной работы, будет разработана поддержка постепенной типизации в языке Kotlin при компиляции в байт-код Java. В работе будет предложено поведение языка Kotlin при взаимодействии с динамически типизированным кодом. В частности, будет рассмотрен механизм поиска перегрузок методов во время выполнения программы — процесс выбора из всех методов с требуемым именем, того, который лучше всего подходит для вызова с текущими аргументами. В работе будет описана реализация постепенной типизации в коде компилятора языка Kotlin. Будет проведено исследование, сравнивающее производительность динамического кода, сгенерированного представленным решением, со статически типизированным, генерируемым компилятором Kotlin. Также будет произведено сравнение производительности генерируемого кода, с динамически и статически типизированным кодом, генерируемым компилятором языка Groovy.

1. Обзор литературы

При разработке программного обеспечения часто используются различные методы, которые помогают убедиться, что программная система ведёт себя в соответствии с некоторой спецификацией. Одним из таких методов является *система типов*. Согласно [19], для неё можно дать следующее определение:

Определение 1.1. *Система типов — это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений.*

1.1. Виды типизации в языках программирования

В зависимости от системы типов, языки программирования можно рассматривать как представителей одной из следующих групп.

- Языки со статической типизацией (static typing languages).
- Языки с динамической типизацией (dynamic typing languages).
- Языки с постепенной типизацией (gradual typing languages).

Языки со статической типизацией отличает связывание переменных, параметров подпрограмм и возвращаемых значений функций с типом в момент объявления, причём этот тип нельзя будем изменить позднее. Таким образом, в момент компиляции мы знаем типы всех выражений. Такое знание позволяет нам проводить некоторые оптимизации и некоторый анализ, на основе которого можно предоставлять пользователю подсказки в рамках Интегрированной Среды Разработки (ИСР).

В языках с динамической типизацией, связывание происходит в момент присваивания значения, которое выполняется во время выполнения программы. Одна и та же переменная может быть связана с разными типами, в зависимости от логики программы. Рассмотрим следующий пример.

```
1 class A {
2     method1()
3 }
4 class B {
5     method1()
6 }
7
8 x.method1()
```

В зависимости от типа переменной x, за строкой *x.method1()* может стоять вызов различных методов — либо из класса A, либо из класса B. В случае, если x окажется представителем типа, для которого не определён метод *method1()*, вызов должен завершиться с ошибкой. При динамической типизации мы ограничены как в проведении оптимизаций, так и в предоставлении подсказок пользователю в рамках ИСР. Это происходит потому что в большинстве случаев, до процесса выполнения, мы ничего не знаем о переменной x.

Постепенная типизация представляет собой систему типов, в которой часть переменных и выражений может быть типизированна, и их корректность проверяется в момент компиляции, а часть может быть не типизированна, и об ошибках типизации в них мы узнаем в момент исполнения [25] [24]. При такой типизации мы можем использовать преимущества как статической, так и динамической типизации. Часть кода мы можем типизировать и выполнять над ним некоторые проверки ещё в момент компиляции. Также компилятор может обеспечивать некоторый набор оптимизаций, положительно влияющих на ускорение кода. В то же время, мы можем получить возможность выразить в рамках нашего языка некоторые удобные конструкции или упростить их написание.

- Можно выразить **eval** — функцию, которая отображает код программы записанный в строковом виде, в результат выполнения этого кода, в соответствии с текущими значениями окружающих объектов.
- Упрощается реализация предметно-ориентированных языков — языков упрощающих решение прикладных задач в некоторой специализированной области.
- Удобная работа с объектной моделью документа (ОМД) — программным интерфейсом, позволяющим унифицировано работать с HTML-, XHTML-, и XML-документами.

Возможны два подхода к постепенной типизации. У нас изначально может быть статически типизированный язык, в котором, для некоторых выражений мы будем помечать, что их тип будет известен только в момент выполнения. Таким путём пошли языки C#, Dart. Противоположным подходом является ситуация, когда у нас есть динамически типизированный язык, в котором мы добавляем возможность вводить некоторые ограничения на типы. Эти ограничения могут проверяться во время выполнения, или в момент компиляции, в зависимости от реализации языка программирования. Таким путём пошли языки PHP 7, TypeScript, Python.

При реализации постепенной типизации в статически типизированных языках, обычно вводят специальный тип, называемый *dynamic*, используемый для представления статически неизвестного типа. У нас изменяется понятие эквивалентности типов — в него добавляется отношение *согласованности*, которое говорит что динамический

тип согласован с любым другим. Заметим, что это отношение не транзитивно, иначе из него можно было бы вывести что любой тип согласован с любым другим.

1.2. Постепенная типизация в языках программирования

В данной главе мы рассмотрим примеры введения постепенной типизации как в статически, так и в динамически типизированные языки.

1.2.1. PHP 7 и Python 3.5

PHP является динамическим языком. В его седьмой версии добавили возможность аннотировать возвращаемое значение и аргументы функций скалярными типами int, float, string и bool [5]. В предыдущих версиях также поддерживалась проверка аргументов на то, что они являются массивом, корректным вызываемым типом, экземпляром какого-то класса или реализуют требуемый интерфейс [1].

```
1  <?php
2    function add(int $a, int $b): int {
3      return $a + $b;
4    }
5    var_dump(add(1, 2)); // int(3)
6  ?>
```

Все проверки типов в PHP происходят во время исполнения. При несоответствии типов в PHP 5 будет обрабатываться фатальная ошибка, а в PHP 7 будет выбрасываться исключение.

Python является динамическим языком, и начиная с версии 3.5, в нём можно аннотировать аргументы и возвращаемое значение функций [23]. В типах можно выражать и какие-то более сложные составные конструкции, например, можно составить тип суммы типов. Python не производит никаких проверок типов, однако, их можно осуществить при помощи сторонних средств. Например, с помощью инструмента статической проверки типов mypy [31] или инструментов входящих в ИСР JetBrains PyCharm [20].

1.2.2. Groovy

Groovy является динамическим языком. Для обеспечения частичной типизации, в нём есть два механизма. Рассмотрим пример динамического кода на этом языке:

```
class A {
    def foo(x) {
        return x + 1
    }
}
```

Мы можем, аналогично предыдущим языкам, проаннотировать все его компоненты типами.

```
class A {
    int foo(int x) {
        return x + 1
    }
}
```

Другим способом является добавление к классу аннотации *@CompileStatic* [14]. В проаннотированных таким образом классах, будут осуществлены проверки типов в момент компиляции.

1.2.3. C#

C# является первым представителем языков, которые изначально были статически типизированы. В версии 4.0 в него добавили возможность помечать некоторые выражения типом *dynamic*. Поведение этого типа, во многом, схоже с поведением типа *object*, который является самым общим типом в иерархии наследования. В частности, переменные помеченные типом *dynamic*, компилируются в тип *object*. Таким образом, тип *dynamic* существует только в процессе компиляции, и отсутствует во время выполнения [21]. В операциях, которые содержат в своей части выражения помеченные типом *dynamic*, не выполняется проверка типов компилятором [28]. Чтобы лучше понять различия *object* и *dynamic*, рассмотрим следующий пример:

```
class Program {
    static void Main(string[] args) {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
```

```
8           System.Console.WriteLine(dyn.GetType());
9           System.Console.WriteLine(obj.GetType());
10      }
11  }
```

В результате, для обеих переменных мы получим один и тот же тип:

System.Int32
System.Int32

Однако разница между *dynamic* и *object*, проявляется в операциях, которые можно над ними совершить:

```
1 n = dyn + 3;
2 j = obj + 3;
```

Так как объект *obj* имеет тип *object*, который не поддерживает операцию прибавления, мы получим ошибку компиляции. С другой стороны, переменная *dyn* имеет при компиляции тип *dynamic*, который тоже не поддерживает операцию прибавления, но откладывает проверку этого до момента выполнения программы. При выполнении, мы обнаруживаем что в динамической переменной лежит объект типа *System.Int32*, на котором мы успешно реализуем требуемую операцию.

1.2.4. Scala

Ещё одним представителем статически типизированных языков является Scala. В качестве экспериментальной возможности, начиная со Scala 2.10, был введён trait Dynamic. С его помощью, появляется возможность отключать статическую проверку типов, при выполнении ряда условий. Если у нас есть выражение qual.sel, на котором статическая проверка типов завершилась неудачно, у нас есть возможность успешно завершить типизацию. При условии что qual удовлетворяет trait Dynamic, а имя sel не совпадает ни с одним из следующих — applyDynamic, applyDynamicNamed, selectDynamic, или updateDynamic, то текущий вызов переписывается согласно правилам, подробно описанным в [17].

У trait Dynamic есть реализация *js.Dynamic*, которая используется в Scala.JS. С её помощью можно упростить написание кода, взаимодействующего с объектной моделью документа. В качестве примера, приведём описанный в [3] способ типизированный работы с объектной моделью.

```scala
object Window extends js.GlobalScope {
    val document: DOMDocument
}
trait DOMDocument extends js.Object {
    def getElementById(id: js.String): HTMLElement
}
trait HTMLElement extends js.Object {
    def appendChild(child: HTMLElement): Unit
}
class Image extends HTMLElement {
    var src: js.String
}
object Main {
    def main(): Unit = {
        val playground = Window.document.getElementById("playground")
        val img = new Image
        img.src = "./path/to/img.png"
        playground.appendChild(img)
    }
}
```

Такой код, может быть упрощён с помощью использования динамической типизации.

```scala
object Main {
    def main(): Unit = {
        val g = js.Dynamic.global
        val playground = g.document.getElementById("playground")
        val img = js.Dynamic.newInstance(g.Image)()
        img.src = "./path/to/img.png"
        playground.appendChild(img)
    }
}
```

1.2.5. Kotlin (JavaScript)

Kotlin является статически типизированным языком программирования. Начиная с версии 1.1, в нём появилась поддержка компиляции в JavaScript. При компиляции

в JavaScript, появляется возможность использования типа *dynamic* [4]. У этого типа есть следующие особенности:

- значение данного типа может быть присвоено любой переменной, и передано всюду, в качестве любого параметра;
- любое значение может быть присвоено переменной имеющей тип *dynamic*, или передано в качестве аргумента, который ожидал тип *dynamic* на входе;
- никаких проверок на нулевой указатель не производится.

Однако, в настоящее время, тип *dynamic* не поддерживается при компиляции в байт-код виртуальной машины Java.

1.3. Постановка задачи

Целью работы является обеспечение поддержки постепенной типизации в языке Kotlin при компиляции на Java платформе.

Для достижения поставленной цели, можно выделить следующие задачи:

- определить семантику динамических операций;
- выработать правила разрешения перегрузок;
- реализовать поддержку динамических вызовов в компиляторе языка Kotlin под JVM;
- оценить производительность представленного решения.

2. Подход к решению задачи

В этом разделе, мы определим спецификацию, согласно которой будет реализована работа с динамическими переменными. Одним из важных моментов этой спецификации является разрешение перегрузок методов. Перегрузка методов — возможность определить несколько методов с одинаковым именем, но с разным количеством или разными типами аргументов. Заметим, что используя только знания о типах, которые мы получаем из самих объектов во время исполнения программы, мы не сможем добиться такого же разрешения перегрузок, как во время компиляции. Это можно проиллюстрировать следующим примером на языке C#:

```
public class Base {};
public class Derived : Base {};

public class A {
    public String method1(Base b) {
        return "Base";
    }
    public String method1(Derived b) {
        return "Derived";
    }
}
```

Если мы попробуем вызвать метод *method1* класса *A*, передав ему переменную Base b = **new** Derived(), то мы получим в результате строку *Base*. Потому что выбор метода будет происходить согласно статическим типам. Рассмотрим динамический случай **dynamic** b = (Base) **new** Derived(). Без дополнительного хранения памяти о статических типах, мы не сможем узнать во время исполнения статический тип b. Поэтому выбор метода будет происходить согласно типам времени исполнения, и при передаче b в метод *method1* мы получим строку *Derived*.

Также заметим, что коль скоро, выбор подходящей перегрузки должен происходить во время исполнения, становится критичен вопрос производительности нашего решения.

Перед определением спецификации, мы посмотрим семантику динамических операций в языках Groovy и C#. На основание её анализа, мы попробуем сформировать правила динамического поведения в языке Kotlin.

2.1. Динамическое поведение Groovy

Язык Groovy изначально динамический. Рассмотрим основные моменты, возникающие при взаимодействии статического кода с динамическим. При присваивании динамической переменной в статически типизированную, во время выполнения происходит попытка приведения типа. В случае её неуспешности, мы получим исключение *GroovyCastException*. В языке Groovy существуют неявные приведения следующих типов:

- при присваивании различных типов в переменную типа *String*, у этого типа неявно вызывается метод *toString*;
- при присваивании в *Boolean* или *boolean* значения другого типа, происходят преобразования согласно конвенции *Groovy Truth* [7];
- при присваивании значения типа *String* в переменную типа *Class*, происходит попытка поиска класса с соответствующим именем;
- происходит автоматическое преобразование списков в массивы;
- происходит авто упаковка и распаковка[1];
- происходит автоматическое преобразование замыканий в классы с единственным абстрактным методом;
- классы могут неявно преобразовываться в их родителей или в реализуемые ими интерфейсы;
- выполняются некоторые преобразования между численными типами [29].

При присваивании статически типизированной переменной в динамическую, никаких дополнительных проверок и преобразований не происходит. Поиск нужной перегрузки метода осуществляется в момент выполнения, как в случае когда метод вызывается на динамической переменной, так и в случае когда динамическая переменная участвует среди аргументов.

Алгоритм используемый для разрешения перегрузок в Groovy, согласно [22], можно описать следующими словами:

1. Создать список всех методов с подходящих именем.
2. Удалить методы, параметры которых не подходят к текущему вызову.
3. Если текущий список содержит ровно один элемент, то выбираем его и завершаем процесс поиска перегрузки.

[1] Про упаковку и распаковку будет рассказано в главе 3.1.1.

4. Для всех методов считаем метрику *расстояние между методами*, вычисленную между типами аргументов каждого метода и реальными аргументами переданными в него.

5. Если есть метод, на котором достигается строгий минимум метрики, то выбираем этот метод и завершаем процесс поиска перегрузки.

6. Завершаем процесс выбора перегрузки с неудачей.

Для вычисления метрики *расстояние между методами*, можно воспользоваться следующим алгоритмом:

1. Для произвольного аргумента метода, который является кандидатом на разрешение перегрузки, считаем расстояние в иерархии наследования между этим типом, и типом времени исполнения параметра, переданного в этот аргумент.

2. Метрикой для конкретного метода-кандидата является сумма метрик посчитанных для всех его аргументов, согласно правилу описанному на первом шаге.

2.2. Динамическое поведение C#

Язык C# изначально статически типизированный. При присваивании статически типизированной переменной в динамическую переменную не происходит никаких проверок. При обратном присваивании, возможны два случая:

- При присваивании используется явное преобразование динамической переменной. Тогда во время компиляции происходит статическая проверка возможности присваивания двух статических типов. При её неуспешности, мы получаем ошибку времени компиляции. Далее, во время выполнения программы происходит попытка выполнить неявное преобразование. Если оно было не успешно, происходит ошибка во время выполнения.

- Если явное преобразование отсутствует, то во время выполнения программы происходит поиск неявного преобразования из типа который лежит в динамической переменной в статический тип [2]. При отсутствии неявного преобразования, мы получаем ошибку во время выполнения.

При перегрузке методов мы обычно не знаем множество кандидатов до момента выполнения. Однако, есть набор случаев, при котором он известен:

- статический вызов с динамическими аргументами;

- вызов методов на статически типизированной переменной;

- вызов по индексу, осуществлённый на статически типизированной переменной;

- вызов конструктора с динамическими аргументами.

Во всех перечисленных случаях, во время компиляции происходит частичная проверка, игнорирующая динамически типизированный код.

В C#, при разрешении перегрузок, методы-кандидаты выбираются из следующих наборов [2]:

- вызов именованного метода, посредством использования вызываемого выражения;
- вызов именованного конструктора, посредством использования вызываемого выражения создающего новый объект;
- вызов метода доступа по индексу;
- вызов предопределённого или пользовательского оператора, участвовавшего в выражении.

Каждый из вышеперечисленных наборов задаёт множество кандидатов. Рассмотрев объединение этих множеств, очищенное от методов неподходящих по имени или по набору аргументов, можно определить лучший метод, согласно следующим правилам:

- Если целевое множество содержит ровно один метод, то он признаётся лучшим.
- В противном случае, лучшим методом считается тот, который лучше всех остальных методов, с учётом их списков аргументов. Целевой метод определяется путём его сравнивания со всеми другими кандидатами *специальным предикатом*, который должен показать истину при всех сравнениях.
- Если мы не смогли успешно определить лучший метод, то мы считаем что текущий выбор является неоднозначным, и мы продуцируем ошибку времени выполнения.

Определение лучшего метода довольно объёмно, желающие могут ознакомиться с ним в главе 7.5.3.2 Better function member [2].

2.3. Предложенное взаимодействие с динамическим кодом в Kotlin

Принимая во внимание обзор особенностей разрешения перегрузок в других языках, сформируем правила, согласно которым, должны происходить операции с участием динамических переменных в языке Kotlin. При разработке правил операций мы будем стараться учитывать следующие критерии:

- Предсказуемость динамического поведения.
- Схожесть динамического поведения со статическим.

Мы хотим чтобы в динамическую переменную можно было положить любой объект. Поэтому нам не приемлем подход Scala, при котором мы можем использовать только объекты удовлетворяющие определённому интерфейсу.

Составим список операций с участием динамического кода. Для каждой операции выработаем правила, работу согласно которым мы будем ожидать от языка Kotlin.

- Присваивание в динамическую переменную.
- Присваивание динамической переменной в типизированную.
- Вызов метода на динамической переменной.
- Вызов метода на типизированной переменной.
- Запрос поля у динамической переменной.
- Запрос динамического поля у не динамической переменной.

При присваивании в динамическую переменную, мы будем ожидать точно такое же поведение, как при присваивании в тип *Any?*.

При присваивании динамической переменной в типизированную, мы хотим ожидать такое же поведение как при привидении *Any?* к этому типу — во время выполнения должна произойти проверка, что согласно иерархии наследования, данное присваивание действительно может произойти.

При вызове метода на динамической переменной, мы хотим поведение, которое похоже на статическое поведение языка Kotlin. Поэтому, мы будем хотеть, чтобы среди всех методов, которые можно вызвать на лежащем в динамической переменной типе, выбрался такой, что:

1. Его имя совпадает с динамически вызванным методом.
2. К его аргументам подходят аргументы времени выполнения у динамического метода.
3. Он является *более специфичный*, чем все другие методы, которые удовлетворяют пунктам 1-2.

Если у нас не получилось выбрать метод согласно вышеуказанным правилам, мы хотим получить исключение времени исполнения.

Про два типа φ и ψ, будем говорить:

- Что φ и ψ *эквивалентные типы*, если они в точности совпадают.
- Что φ и ψ *похожие типы*, если один из них, является упакованной[2]версией другого.
- Что φ *лучший тип* чем ψ, если φ реализует интерфейс ψ, или является его потомком.
- Во всех других случаях, будем говорить, что φ *худший тип*, чем ψ.

Для сравнения специфичности методов $f(\{a_i\})$ и $g(\{b_i\})$, можно использовать следующий алгоритм:

1. Если f совпадает с g, то он *более специфичный* чем g.
2. Если f является методом-помощником, то он *более специфичный* чем g.
3. Если g является методом-помощником, то он *менее специфичный* чем f.
4. Если возвращаемый тип f *лучше*, чем тип g, то f *более специфичный.*
5. Если возвращаемый тип f *хуже*, чем тип g, то f *менее специфичный.*
6. Если у f существует такой индекс i, что i-ый параметр f *хуже* чем i-ый параметр g, то f *менее специфичный* чем g.
7. Если у f существует такой индекс i, что i-ый параметр f *лучше* чем i-ый параметр g, то f *более специфичный* чем g.
8. Если g является методом с переменным числом аргументов, а f — нет, то f *более специфичный.*
9. Во всех остальных случаях, f *менее специфичный.*

При вызове метода на типизированной переменной, мы хотим запускать разрешение перегрузки во время компиляции, считая что *dynamic* это наименее специфичный тип из всех. Стоит отметить, что это может вызывать некоторые не интуитивные ситуации, например если у нас есть две перегрузки:

```
1 fun foo(s: String)
2 fun foo(d: dynamic)
```

То следующие вызовы будут разрешены в пользу перегрузки foo(String):

[2]Про упаковку будет рассказано в главе 3.1.1.

```
foo("")
foo(dyn) // dyn: dynamic
```

С другой стороны, вызов foo(1), будет разрешён в пользу foo(**dynamic**), потому что он не подходит к никаким другим перегрузкам.

При запросе поля у динамической переменной, мы хотим следующее поведение:

- Если у класса объекта, который в данный момент находится в динамической переменной, существует поле с запрошенным именем, мы хотим получить значение этого поля на текущем объекте.
- В противном случае, если у класса объекта, который в данный момент находится в динамической переменной, существует метод чтения с запрошенным именем, мы хотим получить результат его работы на текущем объекте.
- В противном случае, мы хотим получить исключение времени выполнения.

При запросе динамического поля у типизированной переменной, мы хотим такое же поведение, как при запросе обычного поля класса.

3. Реализация решения

Для осуществления поддержки динамической типизации для компилятора Kotlin в байт-код Java, необходима реализация двух пунктов:

1. Требуется разработать изменения в компиляторе, которые в тех местах, где проверка типов отложена до момента выполнения, будут реализовывать вызов динамических инструкций.

2. Необходимо предоставить библиотеку, которая содержит набор загрузочных методов, обеспечивающих необходимое динамическое поведение в требуемых случаях — при вызове методов и при чтении или записи в поля.

3.1. Технические особенности

Рассмотрим детали реализации языков Kotlin и Java, знания о которых нам понадобятся в нашей работе.

3.1.1. Связь языков Java и Kotlin

В настоящий момент Java является одним из самых популярных языков программирования [27]. Программы на Java транслируются в промежуточное представление — байт-код Java [12]. Байт-код состоит из набора инструкций, которые интерпретируются с помощью виртуальной машины Java (JVM).

Несмотря на свою популярность, язык Java имеет ряд недостатков, например, требование к обратной совместимости с предыдущими версиями, громоздкость синтаксиса и медленное развитие. Последнее время, стали появляться новые языки, такие как Scala, Kotlin, Groovy. Они тоже компилируются в Java байт-код и используют для выполнения JVM. Некоторые из них поддерживают выполнение не только на JVM, но и на других платформах, например — JavaScript. Новые языки решают часть проблем Java, но иногда создают свои собственные.

В языке Java типы бывают примитивными и ссылочными. В настоящий момент существует 8 примитивных типов. Для каждого из них существует его ссылочная «обёртка». Java поддерживает автоматические преобразования между ними, такой процесс называется *автоупаковка и распаковка*. Каждому примитивному типу Java, соответствуют свои типы в Kotlin, согласно таблице 1. Однако, аналоги примитивных типов, в языке Kotlin существуют только до момента компиляции. Для улучшения производительности, в процессе компиляции, они заменяются своими аналогами из Java, и отсутствуют во время выполнения как таковые. Выбор, заменить аналог в Kotlin на примитивный тип Java или на его ссылочную обёртку, компилятор языка

Kotlin выполняет самостоятельно. В Java также существует вспомогательный примитивный тип *void*. Он отличается тем, что мы не можем создавать его экземпляр. В Kotlin для него существует аналог — тип *Unit*, у которого есть только один экземпляр. Для оптимизации производительности, в Kotlin, если функция возвращает значение типа *Unit*, то на уровне Java байт-кода, она преобразуется в функцию, которая ничего не возвращает, то есть имеет тип *void*. Однако, с точки зрения языка Kotlin, мы возвращаем из функции объект типа *Unit*, и после этого, мы можем производить на полученном объекте какие-то действия. Поэтому, для обеспечения корректности, в месте использования возвращённого *Unit*-значения, компилятор генерирует байт-код, который загружает тот самый, единственный экземпляр типа *Unit* из стандартной библиотеки. Так как возвращённое *Unit*-значение используется не всегда, такой байт-код иногда будет опущен, что позволит улучшить производительность.

Таблица 1: Соответствие примитивных типов

Примитивный тип Java	Ссылочная обёртка Java	Аналог Kotlin
boolean	java.lang.Boolean	kotlin.Bool
byte	java.lang.Byte	kotlin.Byte
char	java.lang.Character	kotlin.Char
short	java.lang.Short	kotlin.Short
int	java.lang.Integer	kotlin.Int
long	java.lang.Long	kotlin.Long
float	java.lang.Float	kotlin.Float
double	java.lang.Double	kotlin.Double

Объекты класса и всевозможные переменные массива являются ссылочными типами данных. Все классы объединены в общую иерархию наследования. На вершине иерархии находится тип *Object*, все остальные наследуются от него, или, говоря иначе, *Object* является их *родителем*. Любой класс может быть передан в любое место программы, в котором ожидается его родительский класс. В языке Kotlin аналогом типа *Object* является тип *Any?*. Каждый класс языка Kotlin, является наследником класса *Any*. При компиляции, этот класс превращается в *Object*. Особым значением, которое может храниться в любой ссылочной переменной в Java, является *null*. Это значение показывает что в данной переменной не лежит никакого значения. В языке Kotlin, чтобы разрешить хранение в переменной *null* значения, необходимо приписать к её типу «?». Например, любое значение можно положить в переменную типа «*Any?*».

Помимо примитивных типов, в языке Kotlin ещё есть набор типов, которые перестают существовать после процесса компиляции. Некоторые из них перечислены в таблице 2.

Таблица 2: Преобразование типов языка Kotlin

Преобразованный тип в Java	Тип в языке Kotlin
(java.lang.) String	(kotlin.) String
boolean[]	BooleanArray
byte[]	ByteArray
char[]	CharArray
short[]	ShortArray
int[]	IntArray
long[]	LongArray
float[]	FloatArray
double[]	DoubleArray

3.1.2. Осуществление вызовов на Java платформе

В языке Java поля и методы можно разделить на две группы [9]. Обычные поля ассоциированы с конкретным экземпляром класса — объектом. С другой стороны, статические поля ассоциированы с классом, и они общие для всех его экземпляров. Внутри статических методов мы имеем право взаимодействовать только со статическими полями. Обычные методы имеют право взаимодействовать как с полями текущего экземпляра класса, так и со статическими полями данного класса. Важно заметить, что статичность поля или метода никак не связано со статическим разрешением вызова. В языке Java, на этапе компиляции, компилятор может определить, осуществляем мы вызов статического или обычного метода класса (поля).

Для осуществления статически разрешаемых вызовов методов и взаимодействия с полями, в байт-коде Java предназначены следующие инструкции [12]:

- **invokespecial** — используется для создания новых экземпляров классов, вызовов приватных методов, и методов родительского класса;
- **invokestatic** — используется для вызова статических методов;
- **invokeinterface** — используется для вызова методов интерфейса;
- **invokevirtual** — используется для вызова всех остальных методов;
- **getfield**, **getstatic** — используется для получения поля объекта или поля класса соответственно;
- **putfield**, **putstatic** — используется для установки значения поля объекта или поля класса соответственно.

Для поддержки динамических языков на платформе JVM, в JSR 292, была предложена новая байт-код инструкция *invokedynamic*, которая поддерживает эффективное

и гибкое решение, помогающее с выполнением вызовов, в условиях отсутствия статической информации о типах [11].

Инструкция *invokedynamic* осуществляет вызов метода, на который ссылается специальный объект, называемый точкой вызова (call site). Этот объект получается путём вызова специального загрузочного метода (bootstrap method), ассоциированного с каждой *invokedynamic* инструкцией [12]. Путём создания специального алгоритма поведения загрузочного метода, у нас появляется возможность, во время выполнения программы, подстраиваясь под актуальную типовую информацию, производить вызов нужных методов.

В языке Java поддерживается *рефлексия* — механизм исследования данных о программе, во время её выполнения. Она осуществляется при помощи Java Reflection API. Основной список информации, которую можно получить с помощью интерфейса API [6]:

- определить класс объекта во время исполнения;
- получить список полей и методов класса;
- получить информацию о типах аргументов метода;
- вызвать требуемый метод с конкретными аргументами.

Однако, стоит отменить, что некоторые операции Java Reflection API выполняются довольно продолжительное время.

В языке Kotlin, для любого объекта можно обеспечить возможность быть вызванным. Для этого, в его классе достаточно определить оператор *invoke* — специальный метод, который помечен ключевым словом *operator*. При наличии такого оператора, вызов a(i_1, ..., i_N) будет преобразован в вызов a.invoke(i_1, ..., i_N).

В языках Kotlin и Java поддерживаются перегрузки методов — возможность определить несколько методов с одним и тем же именем, но с разным числом или разными типами аргументов. Также поддерживаются методы с переменным числом аргументов. Со стороны Java платформы, в месте вызова аргументы упаковываются в массив (набор элементов расположенных непрерывно в памяти) и передаются в вызываемую функцию. На стороне вызываемой функции, работа с переданными аргументами никак не отличается от ситуации, если бы аргументы были бы переданы в массиве.

3.2. Реализация поддержки динамических вызовов на стороне компилятора

Благодаря тому, что при компиляции Kotlin в JavaScript уже поддерживаются динамические вызовы, первая задача решается определением того кода, который будет генерироваться при компиляции в байт-код Java. В качестве основного

средства осуществления динамических вызовов, мы будем использовать инструкцию *invokedynamic*. Рассмотрим подробнее как мы будем с ней работать.

Первым аргументом этой инструкции мы будем передавать идентификатор действия, согласно таблице 3.

Таблица 3: Идентификаторы динамического действия

Идентификатор	Требуемое действие
getField	Получение значение поля или свойства класса
setField	Установка нового значения поля или свойства класса
invoke	Вызов метода или объекта умеющего обработать вызов

Вторым аргументом нам необходимо передать ожидаемый тип метода, который мы хотим получить в результате вызова. Этот тип мы можем определить во время компиляции, путём составления *дескриптора метода*, согласно правилам описанным в главе 4.3 Descriptors [12]. Несмотря на то что во время компиляции мы не знаем будущие типы, которые будут лежать в *dynamic*-переменных во время исполнения программы, известно что они точно удовлетворяют типу *Object*. Поэтому мы можем использовать его сигнатуру в дескрипторе.

Третьим аргументом является *загрузочный метод*, реализация которого будет рассмотрена в главе 3.3.

Четвёртым аргументом будет идти имя метода или поля, оно понадобится для осуществления поиска требуемого метода во время выполнения программы.

В языке Kotlin поддерживается использование аргументов по умолчанию. Это означает, что при определении метода, некоторым из его последних аргументов мы можем указать значение по умолчанию. Тогда мы получаем возможность опускать часть последних аргументов при осуществлении вызова [13]. При вызове метода с меньшим числом аргументов, компилятор осуществит вызов специально сгенерированного для этого метода помощника, который подготовит все недостающие аргументы метода, вычислив их согласно значениям по умолчанию. После этого, помощник осуществит вызов требуемого метода с вычисленными аргументами.

Аргументы по умолчанию являются мощным инструментом в сочетании с другой возможностью языка — именованными аргументами. В аргументах по умолчанию, мы не имели возможность опускать произвольные аргументы, имеющие значение по умолчанию. Если мы опустили какой-то аргумент метода, то следующие мы тоже должны были опустить, потому что компилятор не мог понять какой именно аргумент мы хотим передать. Одним из способов решения этой проблемы являются именованные аргументы. Мы можем аннотировать часть параметров именем соответствующих им аргументов в вызываемом методе [13]. Для того чтобы избежать неопределённостей, все именованные аргументы должны тоже идти последними.

Заметим, что если мы хотим поддержать именованные аргументы в динамическом вызове, нам необходимо сохранить все аннотации имён аргументов до момента разрешения динамического вызова. Для этого мы их передадим в качестве последних аргументов в *invokedynamic* инструкцию. Заметим, что поскольку именованные аргументы всегда находятся последними, нам не нужно сохранять соответствие между именами аргументов и переданными значениями, т.к. мы их сможем однозначно восстановить во время выполнения.

3.3. Реализация библиотеки для поддержки динамического поведения

Рассмотрим основные компоненты библиотеки времени выполнения, которую нам потребуется реализовать:

- Загрузочный метод.
- Механизм динамического разрешения перегрузок.
- Методы выполняющие настройку точки вызова для различных типов динамических операций.
- Механизм кеширующего взаимодействия с лежащими в полях или свойствах класса вызываемыми объектами.
- Маркер выполнения операции составного присваивания.

Детальней рассмотрим каждую из этих компонент.

3.3.1. Загрузочный метод

Основная задача загрузочного метода состоит в том, чтобы в зависимости от требуемого динамического действия, обеспечить возвращение объекта, в котором лежит точка вызова, с соответствующей действию ссылкой на метод. Для этого, в качестве параметров мы получаем специальный объект типа *MethodHandles.Lookup*, который будет осуществлять проверку разрешений доступа — возможен ли из текущего места вызова доступ к выбранному методу или полю и свойству класса. Также мы получаем все параметры, которые мы передали в *invokedynamic* инструкцию при компиляции, согласно главе 3.2.

В самом загрузочном методе, мы перебираем все типы динамических действий согласно таблице 3, и выбираем нужный метод-настройщик. После его установки, он будет вызван с актуальными аргументами пользовательского вызова. Заметим, что после вызова метода-настройщика, мы можем потерять данные переданные в загрузочный метод. Но в загрузочном методе мы ещё не можем осуществить поиск

нужного метода или поля, потому что мы не знаем актуальных параметров времени выполнения. Поэтому определение целевого метода или поля и свойства класса, необходимо осуществлять в методе-настройщике. Но там нам например понадобится объект типа *MethodHandles.Lookup*, для осуществления проверки корректности прав доступа. Поэтому мы воспользуемся механизмом *MethodHandles.insertArguments* [15]. С его помощью мы получим ссылку на метод, которому можно будет передать аргументы в соответствии с их ожидаемым типом. Ожидаемый тип мы указали при генерации *invokedynamic* инструкции, которая в себе вызывает наш метод-настройщик, и передаёт в него необходимые в дальнейшем аргументы из загрузочного метода. Мы передадим в метод-настройщик следующие объекты, которые понадобятся нам в дальнейшем:

- точка вызова;
- объект типа *MethodHandles.Lookup*;
- ожидаемый тип ссылки на метод;
- имя вызываемого метода или поля и свойства класса;
- именованные аргументы (только при осуществлении вызова метода).

3.3.2. Механизм динамического разрешения перегрузок

В настоящий момент, всё разрешение методов в компиляторе языка Kotlin, при компиляции в байт-код Java, происходит во время компиляции. Оно основывается на знании о типах, которое мы имеем во время компиляции. Для осуществления динамического поведения необходимо разработать инструмент, который на основании имени метода, всех его параметров и типовой информации времени выполнения, правильно выберет вызываемый метод. Данный механизм должен поддерживать выбор метода с учётом его перегрузок. Ему необходимо принимать во внимание методы с переменным числом аргументов, методы с аргументами по умолчанию, а также вызовы методов с использованием именованных аргументов.

Предположим, у нас осуществляется следующий динамический вызов:

```
receiver.method(arg1, ..., argN)
```

Алгоритм разрешения перегрузок можно описать следующим образом:

1. Получить список всех методов, которые есть у типа, находящегося в момент выполнения в переменной *receiever*.

2. Удалить все методы с неподходящим именем.

3. Удалить те методы, тип которых не подходит к актуальным аргументам динамического вызова.

4. Если полученный набор методов пуст, то попробовать осуществить шаги 2-3 на наборе встроенных методов.

5. Среди оставшихся методов выбрать наиболее специфичный, по сравнению с другими.

6. Если наиболее специфичный метод не найден, то завершить поиск с ошибкой.

7. Проверить согласованность с правами доступа.

8. При необходимости учесть аргументы по умолчанию и именованные аргументы.

9. Завершить алгоритм, вернув результирующую ссылку на метод.

Остановимся подробнее на реализации этих пунктов. Для поиска всех методов класса можно использовать механизм Java Reflection API. Однако, не все методы мы сможем найти. Как мы обсуждали в главе 3.1.1, некоторых классов языка Kotlin, не существует во время выполнения, так как они превращаются в соответствующие им классы Java. Однако, у соответствующих им классов Java уже не будет всех методов, которые мы могли вызвать у их аналога из Kotlin. Поэтому нам необходимо реализовать *библиотеку встроенных методов*, в которой присутствуют все недостающие методы, и при необходимости, выполнить разрешение в пользу методов из данной библиотеки. Результатом поиска с использованием Java Reflection API будет массив объектов-кандидатов типа *Method* или *Field* для методов и полей соответственно.

Полученный список методов или полей нам необходимо отфильтровать, удалив из него все те поля, которые имеют неподходящее имя. Стоит учесть, что для поддержки разрешения методов, у которых есть значения по умолчанию, согласно главе 3.2 необходимо учесть наличие метода-помощника. От оригинального метода его отличает наличие суффикса *$default* и большее количество аргументов. В метод-помощник дополнительно передаются специальные битовые маски, при анализе установленных битов которых, можно понять, какие аргументы необходимо вычислить согласно их значениям по умолчанию.

При отборе тех методов, чьи параметры удовлетворяют актуальным аргументам динамического вызова, стоит ориентироваться на несколько моментов. Во первых, быстрым критерием отбора является количество аргументов метода. Оно должно совпадать с количеством переданных динамических аргументов, однако, отдельно стоит обработать случаи, когда у метода переменное число аргументов, и когда есть аргументы по умолчанию. Для проверки возможности передать объект в качестве

нужного аргумента, необходимо посмотреть, отличаются ли их типы с точностью до упаковки и распаковки, либо они удовлетворяют ограничениям иерархии наследования. Для проверки ограничений иерархии наследования, в Java Reflection API присутствует метод *isAssignableFrom()*.

Для проверки согласованности прав доступа, нам понадобится заранее сохранённый объект типа *MethodHandles.Lookup*. Благодаря ему, по имеющемуся у нас объекту типа *Method*, с помощью метода *unreflect*, мы можем получить ссылку на метод, которую в дальнейшем установим нашей точке вызова. В случае поля, с помощью методов *unreflectGetter* и *unreflectSetter*, мы можем объект типа *Field* преобразовать в ссылку на нужный метод чтения или метод записи. В ситуации, когда согласно правилам языка, мы не имеем прав доступа из места вызова к данному методу или полю, мы получим исключение *IllegalAccessException*.

Особое внимание стоит уделить отдельному случаю:

```
1  public class MySuperClass {
2      List toList() {
3          return new MyList();
4      }
5
6      private static class MyList extends AbstractList {
7          @Override
8          public Object get(int index) {
9              return 1;
10         }
11
12         @Override
13         public int size() {
14             return 0;
15         }
16     }
17 }
```

Мы отдаём приватный класс по публичному интерфейсу. В статическом варианте мы бы знали что возвращённый объект удовлетворяет интерфейсу *List*, и мы бы могли вызывать его методы используя байт-код инструкцию *invokeinterface*. Однако, в динамическом варианте, у нас отсутствует знание о статически возвращённом типе. Мы знаем что возвращённый класс имеет тип *MySuperClass.MyList*. При попытке у него вызвать метод *get*, мы получим *IllegalAccessException*, потому что мы не имеем прав

доступа к классу *MySuperClass.MyList*. Одним из решений этой проблемы, является перебор родителя (с учётом его родителей) и всех интерфейсов (с учётом их родителей), которые реализует класс динамического объекта. При переборе, нам необходимо составить множество методов, которые удовлетворяют данному имени и аргументам, а также доступные к вызову из текущего места вызова. Из данного множества необходимо выбрать наиболее специфичный метод. Заметим, что если какой-то метод у класса сейчас недоступен, то у одного из его родителей или интерфейсов он может быть доступен. Если у класса нет подходящего метода, то его и не будет ни у кого выше по иерархии наследования. Если мы нашли доступный метод у класса, то нам нет необходимости рассматривать его родителей и интерфейсы.

Если у нас есть аргументы по умолчанию, то вызов будет происходить при помощи метода-помощника. Нам необходимо будет создать ссылку на метод, который на позиции параметров по умолчанию, передавал бы какое-то значение подходящего типа. После всех параметров, в этом методе должны передаваться битовые маски. Установленные биты этих масок будут показывать позиции аргументов, которые мы попросим вычислить метод-помощник. Битовые маски будут передаваться в виде 32-х битных целых чисел. Таким образом, одно такое число может характеризовать 32 аргумента. Обратим внимание, что при большем количестве аргументов, нам потребуется передавать большее количество чисел. Для передачи битовых масок и значений нужного типа, мы будем использовать *MethodHandles.insertArguments*. Для получения какого-то значения требуемого типа, мы можем перебрать тип аргумента, и для типа boolean вернуть false, для числовых типов вернуть представление нуля в этих типах, а для ссылочных типов вернуть null. При передаче именованных аргументов нам необходимо создать ссылку на такой метод, который будет расставлять аргументы на нужные места, и после вызывать целевой метод. Для этого нам понадобится метод *MethodHandles.permuteArguments*.

При запросе разрешения имени поля и свойства, важным моментом является выработка осознания того, что перед нами — поле или свойство класса. Предположим динамический вызов выглядит так:

```
receiver.field
```

Однако, мы точно не знаем, это поле *field* класса или свойство *field* класса. Если это свойство класса, то необходимо осуществлять доступ посредством специального *метода чтения*, который в байт-коде Java будет выглядеть как метод с именем *getField()*. Таким образом, у нас возможна одна из трёх ситуаций, каждую из которых мы должны обработать:

- У типа, принимаемого переменной *receiver* во время выполнения, есть поле *field*,

но нет метода *getField()*. Тогда мы должны связать текущий вызов с получением этого поля.

- У типа, принимаемого переменной *receiver* во время выполнения, нет поля *field*, но есть метод *getField()*. Тогда мы должны связать текущий вызов с вызовом этого метода.

- У типа, который принимает переменная *receiver* во время выполнения, есть поле *field* и метод *getField()*. Тогда мы должны связать текущий вызов с получением этого поля *field*, потому что так происходит в языке Kotlin при статической типизации.

3.3.3. Обзор ситуаций требующих изменения вызываемого метода

Рассмотрим подробно ситуации в языке Kotlin, в которых у нас может возникнуть желание поменять решение о выборе вызываемого метода.

---

receiver.field

Предположим, во время компиляции, переменная *receiver*, помечена как *dynamic*. При осуществлении первого вызова мы должны запустить *процесс разрешения свойства класса или поля.*

При повторном вызове, в зависимости от изменения типа, который принимает переменная *receiver* во время выполнения, у нас может быть две ситуации:

- Тип не изменился. Тогда мы имеем право вызвать то же поле или метод, которые мы разрешили при первом вызове.

- Тип изменился. Тогда мы не можем точно сказать что мы должны сейчас вызвать, поэтому мы должны снова запустить *процесс разрешения поля или свойства класса.*

---

receiver.field = object

Ситуация присваивания значения полю или вычислимому свойству разрешается аналогично предыдущему пункту, проверяя изменение типа объекта лежащего в переменной *receiver*.

---

receiver.method(arg1, ..., argN)

Предположим, во время компиляции переменная *receiver* помечена как *dynamic*. При осуществлении первого вызова мы должны запустить *процесс разрешения целевого метода*.

При повторном вызове нам также необходимо проверить изменения типа *receiver*. Так как язык Kotlin поддерживает перегрузку методов по типам аргументов, нам также необходимо проверить изменения типов всех аргументов arg_i. Существует ещё одна ситуация. При первом вызове может обнаружиться, что у класса, представителем которого является *receiver*, нет метода *method*, но зато есть поле или свойство класса с именем *method*, по которому лежит объект, который может быть вызван. Такие объекты характеризуются тем, что у них есть как минимум один метод с именем *invoke*, помеченный что он является оператором [13]. Заметим, что при наличии нескольких таких методов, нам необходимо запускать процедуру определения нужной перегрузки. Таким образом, чтобы осуществить требуемый вызов, нам необходимо вызвать получение значения поля *method*, и на этом значении вызывать нужный метод с учётом перегрузок. Поэтому знания того, что класс объекта лежащего в *receiver* нам недостаточно. Нам необходимо ещё знать не изменился тот объект, который лежит в поле *method* или возвращается соответственным свойством класса.

3.3.4. Методы выполняющие настройку точки вызова

Выполнение настройки точки вызова в общем случае можно описать так:

1. Разрешение требуемой ссылки на метод.

2. Установка условий, при которых необходимо будет признавать точку вызова недействительной и запускать процесс настройки заново.

3. Установка разрешённой ссылки на метод как целевой, для текущей точки вызова.

4. Осуществление динамического вызова в первый раз.

Пройдёмся подробнее по каждому из пунктов. Для разрешения ссылки, при динамических операциях с полями или свойствами класса, достаточно запустить динамическое разрешение с запрошенным именем и текущими аргументами, используя описанные в главе 3.3.2 механизмы.

При разрешении ссылки при динамических вызовах методов, ситуацию, при которых метод с данным именем при текущих аргументах не найден, можно исправить в некоторых случаях:

1. Имя вызываемого метода входит в список операторов составных арифметических присваиваний.

2. У класса существует поле или свойство, совпадающее по имени с текущим методом, в котором лежит объект который можно вызвать.

Таблица 4: Соответствие операторов составного присваивания

Ключевой символ	Составной оператор	Обычный аналог
+=	plusAssign	plus
-=	minusAssign	minus
=	timesAssign	times
/=	divAssign	div
%=	modAssign	mod
%=	remAssign[3]	rem[3]

В языке Kotlin, если в коде программы встречается ключевой символ из таблицы 4, то сначала осуществляется попытка найти у класса соответствующий этому ключевому символу составной оператор. Если у класса отсутствует требуемый составной оператор, то осуществляется попытка выполнить текущую операцию путём использования аналога этого составного оператора. Например, выражение a += b может быть преобразовано либо в a.plusAssign(b), либо в a = a.plus(b).

Однако, до момента осуществления динамического вызова, мы не можем точно знать, есть ли у класса, который в данный момент находится в динамической переменной, составной оператор присваивания, либо у него есть только аналог. В первом случае, точка вызова должна быть связана с самим составным оператором, во втором — с его аналогом, и затем должна быть произведена операция записи результата в переменную. Чтобы отличать эти случаи, если во время выполнения был найден настоящий составной оператор, мы будем в качестве результата возвращать объект специального типа. Такой объект мы будем называть *маркер выполнения операции составного присваивания.* В самом компиляторе, будет генерироваться код использующий специальную инструкцию *instanceof.* Эта инструкция выполняет проверку того, что объект является представителем какого-то класса. С её помощью мы сможем понять, необходимо ли производить присваивание.

Заметим, что целевой метод может возвращать значение типа *Unit.* В главе 3.1.1 мы обсуждали тонкости реализации методов с таким возвращаемым значением. Однако, при динамическом вызове, мы не можем знать что метод, который будет выбран во время выполнения, возвращает *Unit.* Поэтому необходимо осуществить явный возврат значения этого типа. Мы должны выбрать ссылку на такой метод, который вызывает разрешённый ранее метод, а после возвращает экземпляр класса *Unit.*

Мы уже обсуждали в главе 3.3.3, что у нас возможна ситуация, при которой отсутствует метод с запрошенным именем, но присутствует поле или свойство класса,

[3] Начиная с Kotlin 1.1

возвращающее объект поддерживающий операцию вызова. Для осуществления этого вызова, нам надо разрешить у полученного объекта метод *invoke*. Заметим, что ввиду поддержки перегрузок методов, разрешать необходимо с учётом актуальных аргументов. Для обеспечения поддержки данной функциональности, был создан специальный класс *ObjectInvoker*. Мы будем связывать точку вызова, со специальным методом этого объекта — *performInvoke*. В этом объекте мы можем запомнить ссылку на метод, выполняющий роль метода чтения поля или свойства. Во время выполнения вызова, мы будем вызывать метод чтения, а после этого, на полученном объекте разрешать метод *invoke*.

Ситуации, при которых необходимо признавать точку вызова недействительной, детально описаны в главе 3.3.3. Для обеспечения поддержки этого механизма, в Java можно воспользоваться *MethodHandle.guardWithTest()*. Этот метод позволяет обернуть текущую ссылку на метод таким образом, что перед её выполнением будет проверено какое-то условие. При не выполнении предиката, мы будем осуществлять весь процесс разрешения заново. Нам понадобятся следующие условия:

```
1  boolean isInstance(Class c, Object o) {
2      return o != null && o.getClass() == c;
3  }
4
5  boolean isNull(Object o) {
6      return o == null;
7  }
```

В случае *ObjectInvoker*, условные методы нам должны помогать только проверить изменение класса объекта, на котором производится динамический вызов. Если этот класс не изменился, мы имеем право использовать метод доступа к полю или свойству, сохранённый при разрешении прошлого вызова. Заметим, что разрешение этого метода не зависит от типов аргументов переданных в вызов. Типы аргументов будут влиять только на разрешение метода *invoke* у полученного из метода доступа объекта. Сохранив последний разрешённый метод и список типов аргументов, при котором это разрешение произошло, мы имеем право повторно вызвать метод, при условии отсутствия изменения типов аргументов.

Таким образом, получив результирующую ссылку на метод, мы можем её установить для текущей точки вызова, при помощи вызова метода *setTarget()* [16]. В качестве завершения настройки точки вызова, мы возвращаем значение, полученное путём вызова установленной ссылки на метод.

3.4. Запоминание результатов вычислений

В нашем решении два места можно ускорить за счёт запоминания результатов вычислений. Во первых, при разрешении перегрузки методов, мы можем запоминать соответствия между типами аргументов и разрешёнными методами. Можно сделать предположение, что через данную точку вызова чаще всего проходит ограниченное небольшой константой количество наборов типов. Тогда мы имеем право перебрать набор сохранённых типов, и при нахождении точного соответствия аргументов, мы можем вызвать сохранённый разрешённый метод. В противном случае, нам необходимо провести полноценный поиск нужного перегруженного метода. В случае наличия свободного места в наборе сохранённых методов, мы можем сохранить вычисленную перегрузку для последующего использования. В случае отсутствия свободного места, имеет смысл использовать один из алгоритмов вытеснения устаревших результатов. Мы будем использовать алгоритм, вытесняющий самую старую добавленную запись. В нашей реализации мы будем сохранять 15 последних использованных наборов. При реализации взаимодействия с наборами, стоит не забывать о поддержке многопоточности в динамическом коде. Чтение и запись сохранённых результатов необходимо реализовывать с использованием различных примитивов многопоточной синхронизации. В работе, для запоминания результатов используется связанный список *LinkedList*, чтение и запись в который происходит из методов, помеченных ключевым словом *synchronize*.

Ещё одним случаем, при котором может пригодиться сохранение результатов, является нахождение перегрузок у вызываемых объектов, лежащих в полях класса. В нём можно использовать те же самые стратегии, которые были востребованы в первом случае. Однако, этот случай требует отдельной реализации в коде. В реализации решения, описанного в настоящей работе, сохраняется только последний результат разрешения перегрузки.

4. Измерения производительности

Одним из важных вопросов при использовании динамического кода является его производительность. При статической типизации множество проверок и вычислений происходит во время компиляции, тогда как при динамической типизации их приходится производить во время выполнения. Такие проверки могут сильно сказываться на времени работы программы.

4.1. Особенности измерения производительности на Java платформе

Важным фактором в любом измерении является степень доверия результату. Код, предназначенный для Java платформы, запускается при помощи виртуальной машины, поэтому нам необходимо учитывать её особенности. Во время выполнения, Java машина часть байт-кода может интерпретировать, выполняя инструкцию за инструкцией, а часть — компилировать в машинный код. Машинный код выполняется быстрее чем интерпретируемый, однако требует времени на свою генерацию. Во время работы, Java машина накапливает статистику по выполняемому коду, анализируя которую, она может производить различные оптимизации и решать каким способом выполнять каждый участок кода. Часто, чтобы позволить JVM накопить статистическую информацию, перед замером, выполняют несколько, так называемых «разогревочных», запусков.

С накоплением достаточного количества статистики, возникает ещё одна проблема. Часть кода может быть удалена, потому что Java машина посчитает что этот код никак не используется и не влияет на результат работы программы. Также, если во время выполнения станет ясно что результат работы некоторых методов не зависит от внешних факторов, их вызов быть заменён на результат вычисления. Выполнение замеров производительности на оптимизированном коде может давать ложную картину о скорости работы программы.

Во время выполнения «разогревочных» запусков, виртуальная машина Java накапливает информацию об используемой памяти. На основании этой информации, JVM может выделить для себя большее или меньшее количество памяти. Если JVM не хватает свободной памяти, она может запустить процедуру *сборки мусора* — процесс поиска и удаления объектов, которые ещё лежат в памяти, но уже не нужны для выполнения программы. Сама процедура сборки мусора может занимать достаточно внушительное время, серьёзно влияя на время работы замеряемого участка кода.

Для повышения точности показателей, при выполнении замеров времени работы программы, рекомендуется отключать сторонние приложения. Ввиду особенности обеспечения планирования процессов в многозадачных системах, отключая приложе-

ния мы можем уменьшить влияние фактора того, что планировщик операционной системы имеет возможность снимать процесс JVM с выполнения.

Одним из популярных способов уменьшить количество проблем при измерениях написанного для JVM платформы кода, является использованием специального программного инструмента — JMH [18]. Данный программный инструмент, анализируя переданные ему параметры, самостоятельно обеспечивает требуемое количество «разогревочных» и «основных» запусков. В рамках каждого из основных запусков, каждый тест производительности запускается несколько раз, после чего вычисляется среднее время работы в рамках этого запуска. Результаты всех запусков представляются как выборка из нормального распределения с неизвестной дисперсией. По полученной выборке можно рассчитать различные характеристики случайной величины, например математическое ожидание и доверительный интервал.

Для того чтобы избежать удаления неиспользуемого кода, JMH предоставляет специальный объект типа *Blackhole*. Передача результата в этот объект позволяет избежать случая, при котором Java машина считает что результат операции ни на что не влияет, и не выполняет часть кода.

4.2. Окружение при проведении измерений производительности

- **Операционная система:** Linux version 4.4.0-71-generic (buildd@lcy01-05) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4)) #92-Ubuntu SMP Fri Mar 24 12:59:01 UTC 2017.
- **Центральный процессор:** Intel(R) Core(TM) i7-6700 CPU @3.40GHz.
- **Версия JVM:** Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode).
- **Версия JRE:** Java(TM) SE Runtime Environment (build 1.8.0_121-b13).

4.3. Рассмотренные тесты производительности

Для оценки практичности решения, был разработан набор тестов производительности. В нём сравниваются решения написанные на языках Kotlin и Groovy. В языке Groovy, начиная с версии 2.0, появилась поддержка инструкции *invokedynamic* JVM платформы [10]. Есть измерения, что в некоторых случаях, Groovy код, скомпилированный с поддержкой данной инструкции, работает быстрее [30]. В каждом тесте производится сравнение одинакового кода, переписанного с учётом особенностей каждого языка или способа компиляции. В качестве целевых платформ, для тестирования производительности, использовались следующие:

- Kotlin (статическая типизация, версия 1.1 rc);
- Kotlin (максимально возможное число объектов помечено типом *dynamic*, на основе версии 1.1 rc);
- Groovy (версия 2.4.5);
- Groovy Invoke Dynamic (версия 2.4.5);
- Groovy (с аннотацией CompileStatic) (версия 2.4.5).

Для поддержания объективности, каждый тест производился на всех целевых платформах, при различных размерах входных данных. Исходный код тестов для всех платформ, можно найти в репозитории [26]. При анализе результатов, основное внимание мы будем обращать на следующее:

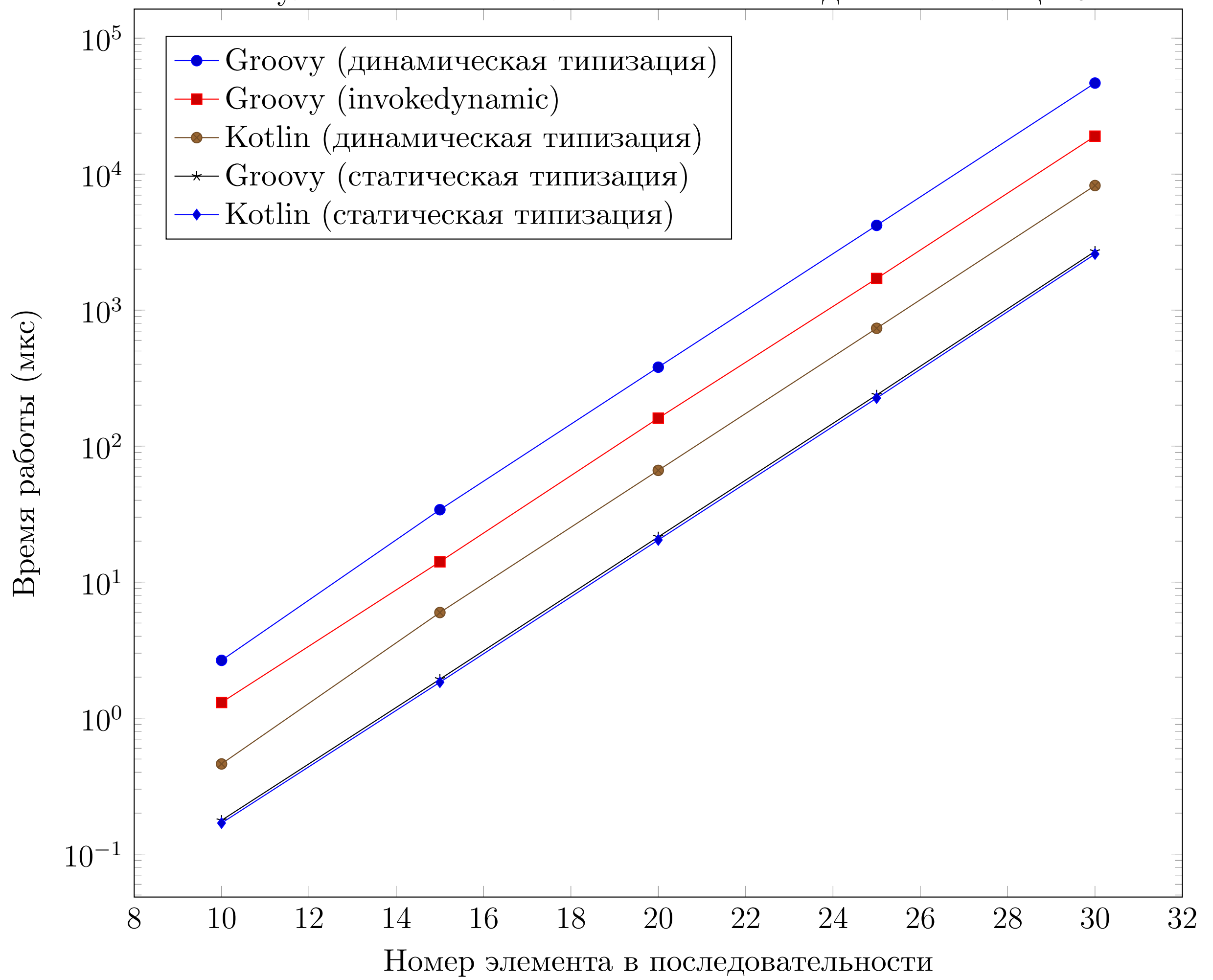
- Относительное различие статически и динамически типизированного кода.
- Сравнение скорости работы динамической типизированного кода на языке Kotlin, с динамически типизируемым кодом Groovy (при компиляции «обычным способом» и с помощью *invokedynamic* инструкции).

Для удобства анализа результатов, мы будем сортировать данные по увеличению времени работы. В качестве времени работы, мы будем замерять время выполнения одного запуска теста в секундах. Если в описании теста не указано иного, перед запуском тестов мы не будем проводить «разогревочных» итераций. Это сделано для того, чтобы исключить запоминание результатов вычисления перегрузок. После этого мы будем запускать код на выполнение один раз в рамках «основных» итераций. Такой процесс мы повторим 20 раз, используя каждый раз новый java процесс. На основании всех результатов «основных» итераций, мы будем вычислять ошибку — максимальное замеченное отклонение, относительно посчитанного времени работы. Нами также были проведены замеры времени работы «разогретого» кода, в котором происходил запуск 20 «разогревочных» итераций перед выполнением 20 «основных». Результаты таких замеров отличался мультипликативно от результатов описанных в работе, однако относительный порядок скорости работы был аналогичный.

4.3.1. Вычисление чисел Фибоначчи

Числа Фибоначчи представляют собой возрастающую последовательность, в которой первые два элемента равны 1, а следующие равны сумме двух предыдущих [32]. Данный тест реализован в рекурсивном варианте решения, в котором расчёт результирующего значения для i-го элемента, выполняется путём запуска вычисления

Рис. 1: Результаты теста: Числа Фибоначчи. По данным таблицы 5.



алгоритма от *i-1*-го, и *i-2*-го элемента, без выполнения каких-то техник запоминания промежуточных результатов.

Результаты измерений представлены на рисунке 1. Статические решения языков Groovy и Kotlin показывают примерно одинаковую производительность. Однако, статическое решение Groovy чуть-чуть медленнее, потому что порождаемый им класс, в конструкторе производит набор операций, необходимых для реализации интерфейса *groovy.lang.GroovyObject*.

Традиционное динамическое решение Groovy медленнее, чем решение с использованием инструкции *invokedynamic*. Это можно объяснить тем, что во втором решении, в начале работы, происходит связывание каждой точки вызова с целевой ссылкой на метод. Так как во время работы программы не меняются типы приходящие на каждую точку вызова, каждый вызов будет происходить с использованием сохранённых в JVM значений. Такие вызовы будут оптимизироваться, и происходить быстрее чем в традиционном решении.

Можно заметить, что разработанное динамическое решение Kotlin оказалось быстрее, чем решение Groovy (invokedynamic), несмотря на то, что они оба используют инструкцию *invokedynamic*. Это можно объяснить тем, что описанное в настоящей работе решение эффективней реализует операции над стандартными типами. Решение Groovy, для обеспечения корректности операции сложения двух чисел, использует методы класса *org.codehaus.groovy.runtime.dgmimpl.NumberNumberPlus*, который делает множество тяжеловесных операций, таких как например *instanceof*.

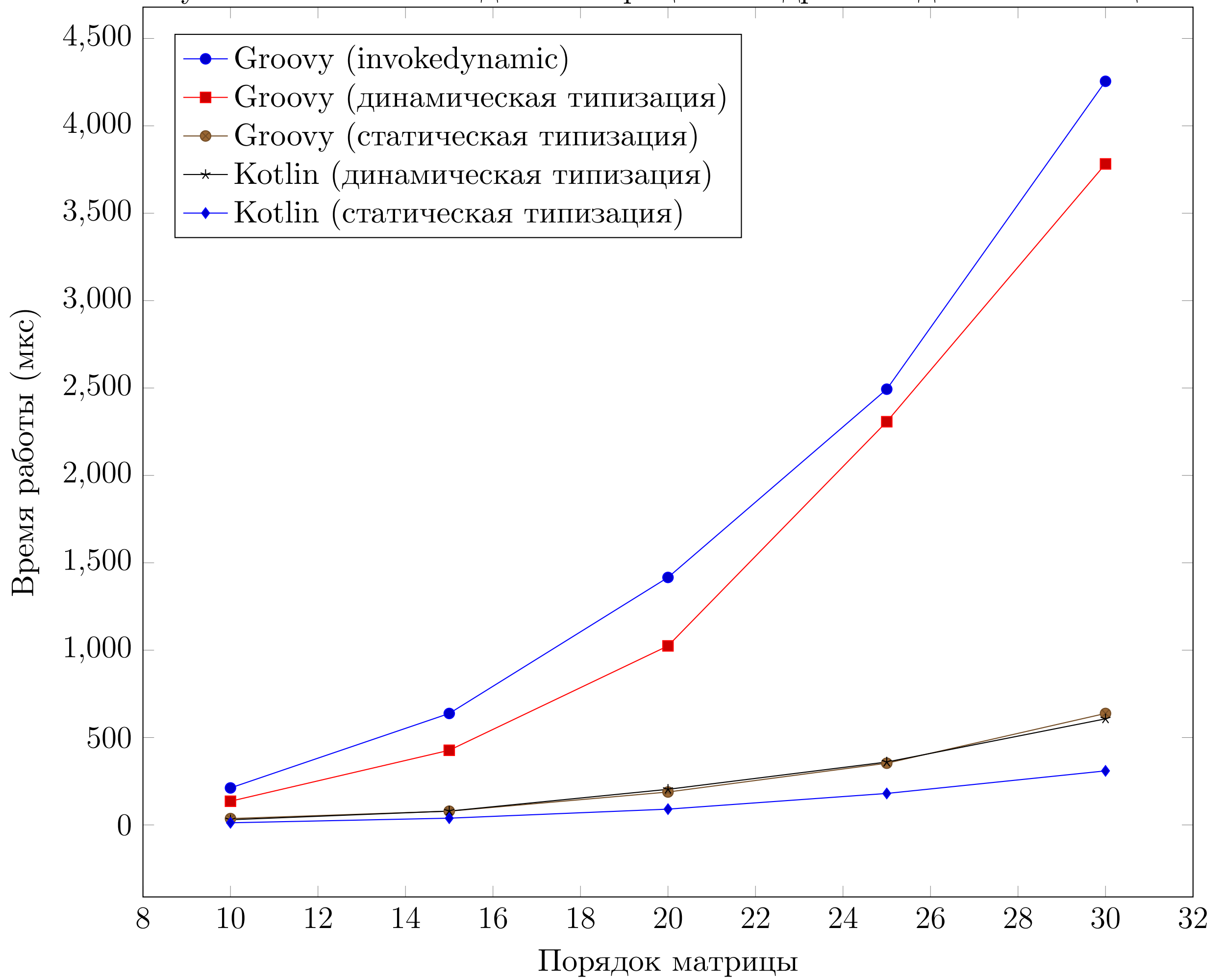
Важным является факт того, что сложность динамического решения не ухудшается с ростом параметра. Решение Groovy (invokedynamic) на протяжении всех измерений, хуже своего статического аналога примерно в 7.26 раз, а решение Kotlin — в 3.13 раз.

4.3.2. Возведение матрицы в квадрат

Перемножение матриц является классической задачей для тестирования производительности. В рамках нашей задачи нет необходимости вдаваться в тонкости оптимизаций размещения матрицы в памяти. Мы будем возводить в матрицу, хранящуюся как массив массивов чисел, в квадрат, по стандартной формуле произведения матриц, описанной например в [34].

Согласно рисунку 2, мы видим, что решение рассмотренное в настоящей работе обыгрывает динамические решения языка Groovy. Интересным является факт того, что динамическое решение Kotlin примерно сравнимо по производительности со статическим решением Groovy. Это можно объяснить тем, что в этом случае Groovy генерирует достаточно сложный байт-код. По сравнению со статическим решением Kotlin, он использует достаточно много таких операций как:

Рис. 2: Результаты теста: Возведение матрицы в квадрат. По данным таблицы 6.



- *ScriptBytecodeAdapter.castToType* — используется для приведения типов, с учётом их особенностей в языке Groovy.
- *NumberNumberMultiply.multiply* и *NumberNumberPlus.plus* — используются для осуществления операций сложения и умножения. В статически типизированном коде Kotlin, вместо них выполняются быстрые байт-код инструкции *imul* и *iadd* соответственно.
- *IntRange* — используется для итерирования по элементам матрицы. В то время как в языке Kotlin, используется увеличение индексной переменной и её сравнение порядком матрицы.

Важным является факт того, что сложность динамического решения языка Kotlin не ухудшается с ростом порядка матрицы. Разработанное решение хуже своего статического аналога примерно в 2.13 раз. Решение Groovy (invokedynamic) хуже своего статического аналога примерно в 7.03 раз.

4.3.3. Z-функция

Z-функция от строки это массив совпадающий с ней по длине, в котором i-ый элемент, совпадает с числом символов, которые начиная с i-ой позиции, совпадают с первыми символами данной строки. Z-функция помогает решать множество задач связанных с поиском подстрок в строке. Детали её реализации подробно описаны например в [8].

По рисунку 3, мы видим, что все решения языка Kotlin обыгрывают все решения языка Groovy. Стоит заметить, что динамическое решение Kotlin обыгрывает в среднем в 1.92 раз статическое решение Groovy. Это во многом происходит благодаря тому, что Groovy довольно большой объём кода выполняет для простых операций.

4.3.4. Быстрое Преобразование Фурье

Быстрое преобразование Фурье — алгоритм быстрого вычисления дискретного преобразования Фурье, которое широко используется в области цифровой обработки сигналов. Подробно о реализации алгоритма описано в [33].

Рассмотрев рисунок 4, мы можем сделать вывод о том, что динамическое решение на языке Kotlin, обыгрывает все решения на языке Groovy. Стоит заметить, что динамическое решение Kotlin в среднем в 7.44 раз быстрее статического решения Groovy.

4.3.5. Разрешение перегрузок методов

Одним из важных процессов при выполнении динамического кода является выбор нужной перегрузки метода. Для определения лучшей перегрузки, обычно необходимо произвести много действий во время выполнения кода — перебрать все методы и

Рис. 3: Результаты теста: Z-функция. По данным таблицы 7.

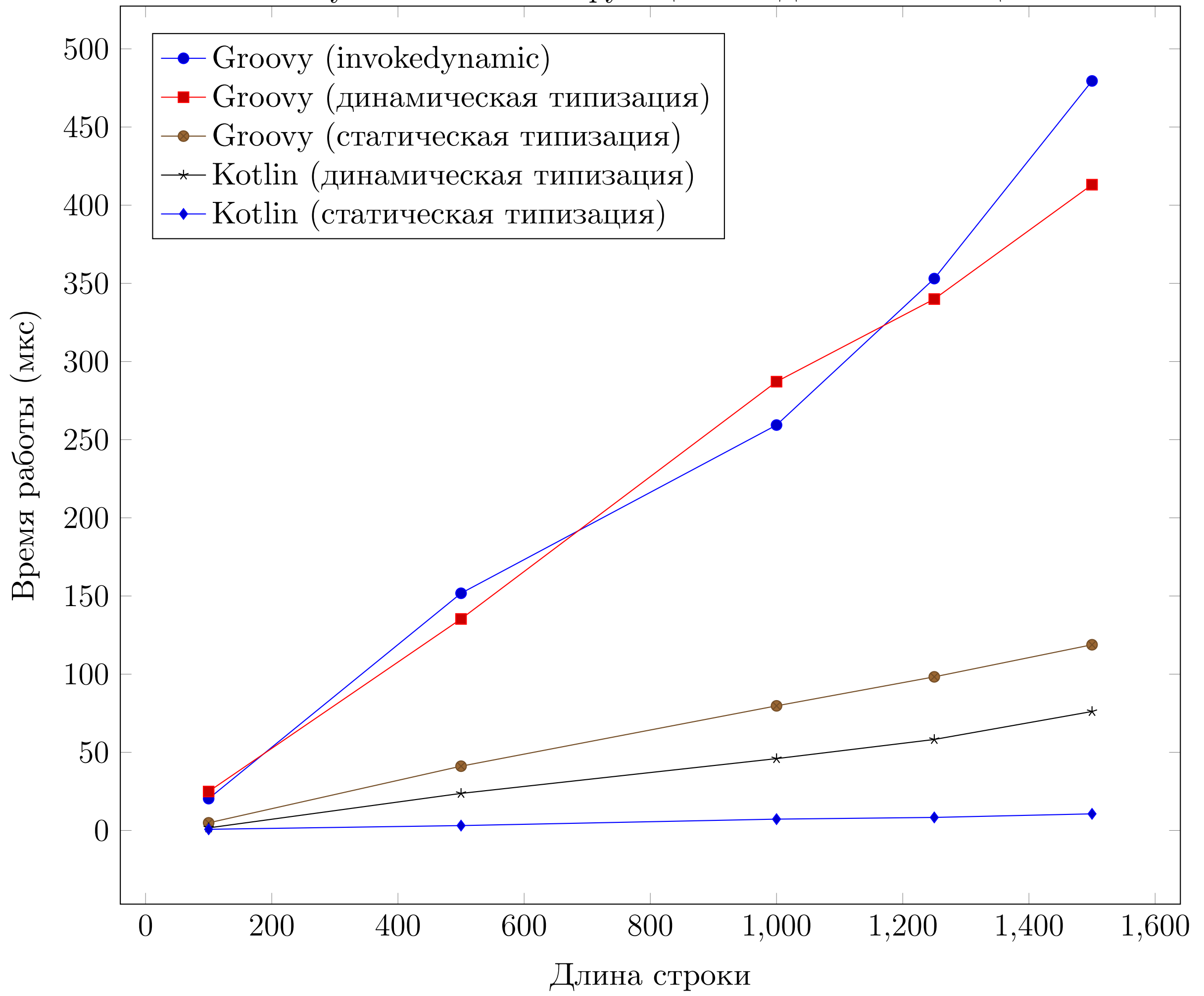
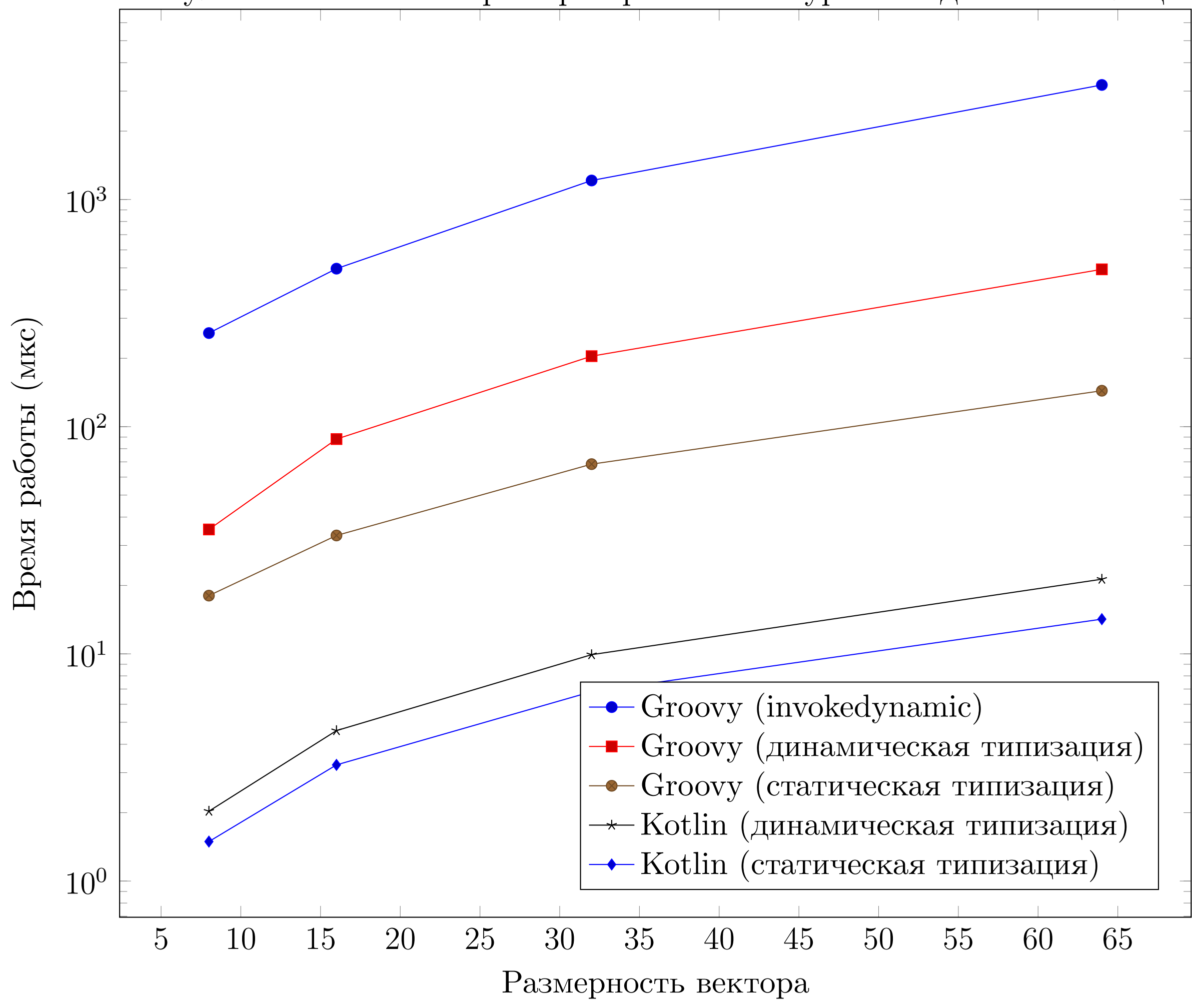


Рис. 4: Результаты теста: Быстрое преобразование Фурье. По данным таблицы 8.



определить наиболее подходящий из них. Ввиду большого количества действий, определение перегрузок становится критически влияющим на производительность фактором.

Мы будем замерять время выполнения методов, у которых различное число перегрузок с разным количеством аргументов. Результаты можно наблюдать в таблице 9. Мы можем сделать вывод о том, что динамическое решение на языке Kotlin, обыгрывает все решения на языке Groovy. Статическое и динамическое решения на языке Kotlin также отличаются не сильно.

4.3.6. Разрешение перегрузок на списке динамических объектов

Произведём тест, аналогичный рассмотренному в разделе 4.3.5, только теперь возьмём список из 1000 объектов типа *int* и 1000 объектов типа *String*. Элементы этого списка мы случайно перемешаем. Каждый объект, который мы достанем из списка, будет участвовать в качестве одного из аргументов перегруженного метода. Все остальные аргументы будут фиксированы. Причём при комбинации фиксированных аргументов с типом *int* и с типом *String* будут подходить разные перегрузки метода. Заметим, что данный тест возможен только при динамической типизации, поэтому мы его будем воспроизводить не на всех целевых платформах.

Перед запуском теста мы будем проводить 20 «разогревочных» итераций. После этого мы будем запускать код на выполнение ещё 20 раз в рамках «основных» итераций. Полученные результаты представлены в таблице 10.

Мы рассмотрим три модификации решения для языка Kotlin. Первая, после изменения типов аргументов, выполняет переназначение целевого метода в точке вызова, при помощи метода *setTarget*. Результаты замеров этой модификации представлены в таблице 11. Мы видим, что такое решение сильно проигрывает традиционному решению Groovy.

Для второй модификации воспользуемся раздельной блокировкой на чтение и запись[4]. Результаты замеров представлены в таблице 12. Мы видим ухудшение времени работы.

Откажемся от того, чтобы при изменении типов аргументов, изменять установленную ссылку на метод в точке вызова. Таким образом, при изменённых аргументах будет запускаться реализованный нами процесс поиска нужной перегрузки. Результаты этого решения мы можем наблюдать в таблице 13 и на рисунках 5 и 6. Мы получили выигрыш производительности в среднем в 3.94 раз, по сравнению с решением Groovy и в 16.95 раз, по сравнению с первой модификацией. Однако, такой подход будет хуже себя проявлять в случаях менее агрессивной смены сохранённых методов.

[4] Был использован стандартный класс *ReentrantReadWriteLock*.

Рис. 5: Результаты теста: Разрешение перегрузок на списке динамических объектов. По данным таблиц 10 и 13.

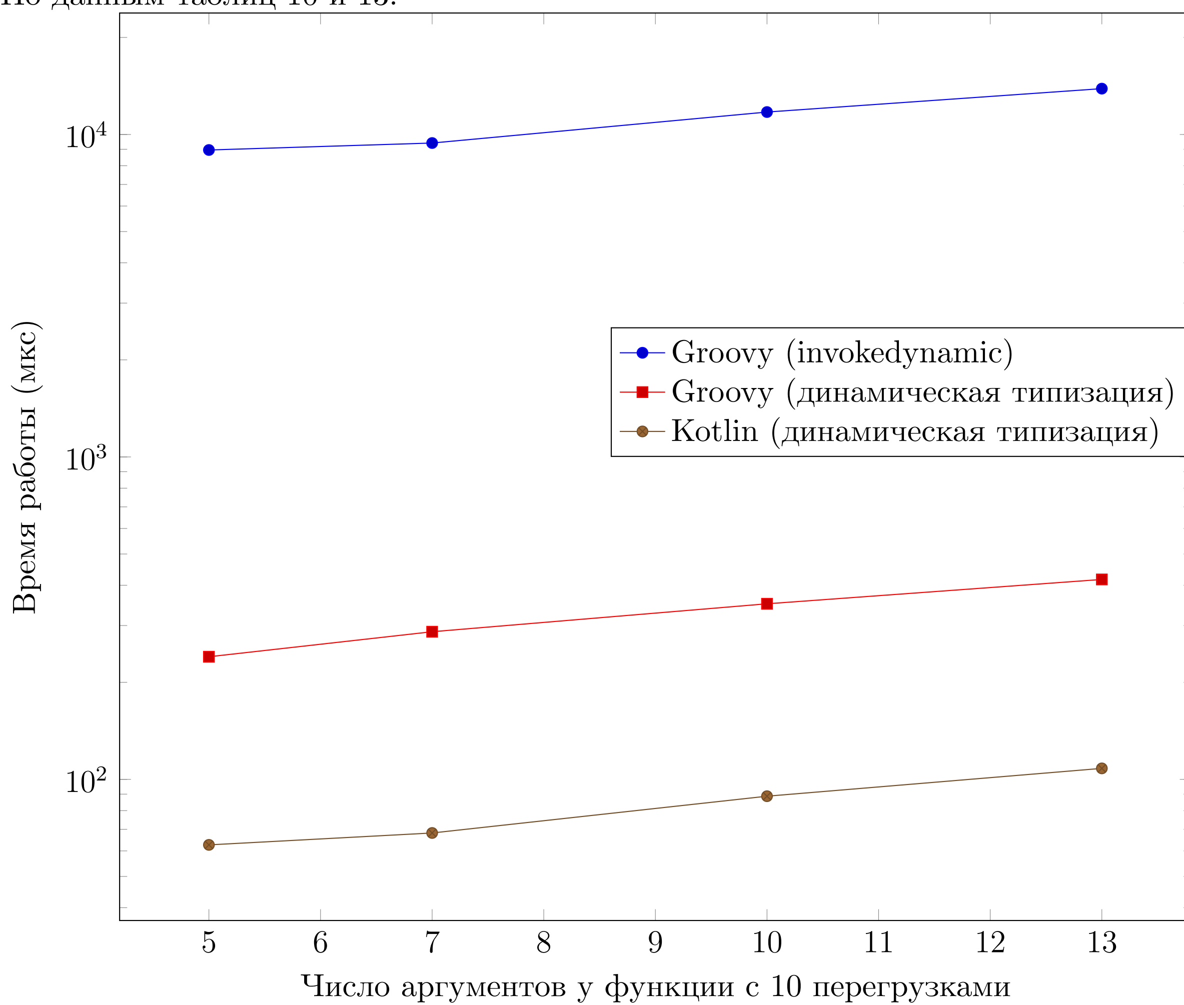
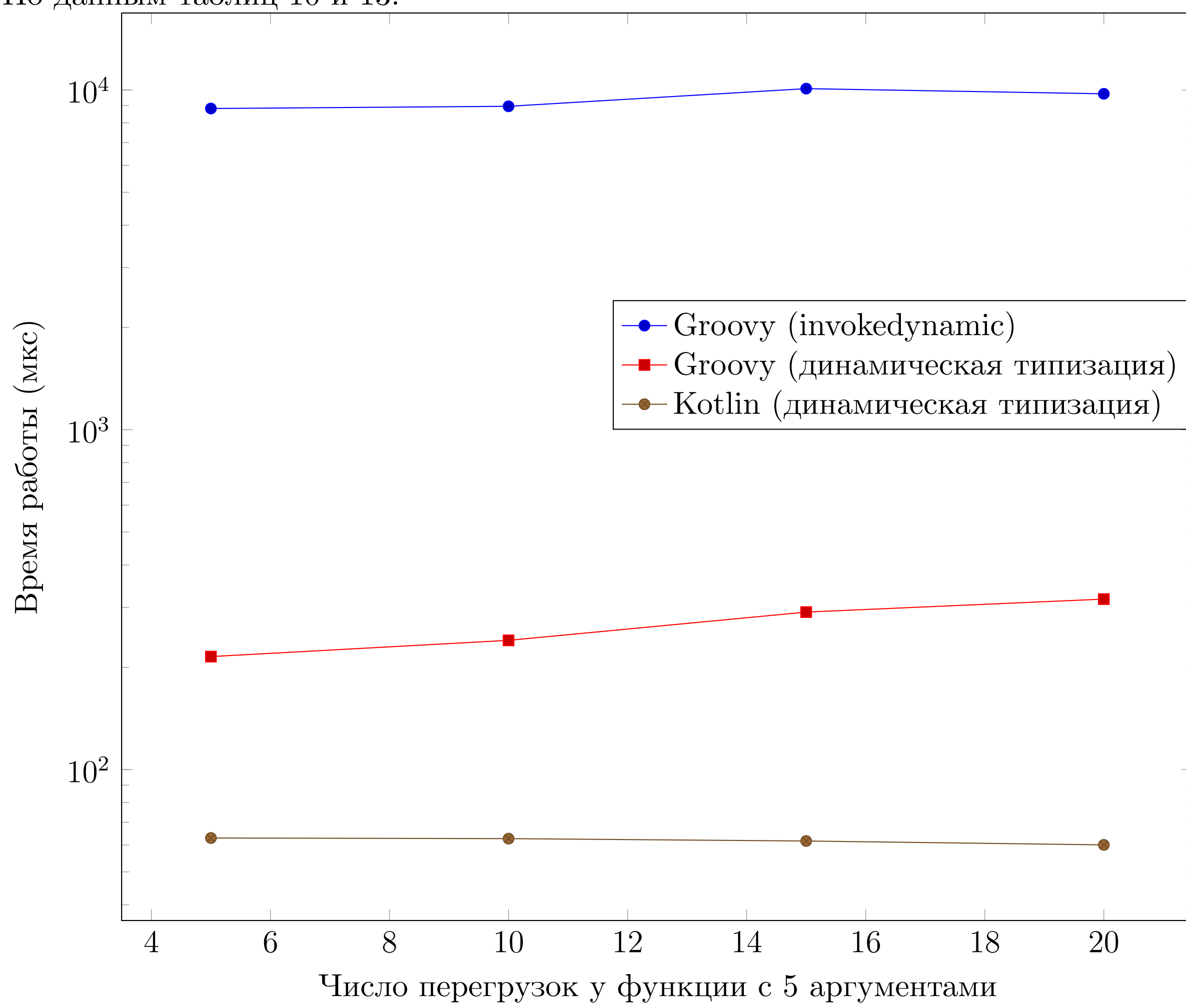


Рис. 6: Результаты теста: Разрешение перегрузок на списке динамических объектов. По данным таблиц 10 и 13.



4.3.7. Параллельное разрешение перегрузок на списке динамических объектов

Интересным будет воспроизведение ситуации, описанной в разделе 4.3.6, в многопоточной среде. Запустим тот же самый тест параллельно в 8 потоков. Перед запуском теста мы будем проводить 20 «разогревочных» итераций. После этого мы будем запускать код на выполнение ещё 20 раз в рамках «основных» итераций. Полученные результаты представлены в таблице 14.

Мы рассмотрим три модификации решения для языка Kotlin. Первая, после изменения типов аргументов, выполняет переназначение целевого метода в точке вызова, при помощи метода *setTarget*. Анализируя результаты таблицы 15, мы видим, что такое решение хуже чем традиционное решение Groovy в 10.89 раз, и лучше *invokedynamic* решения в 4.36 раз.

В однопоточной версии теста мы не получили прироста производительности при использовании раздельной блокировки на чтение и запись. Анализируя результаты таблицы 16, мы полностью отвергаем этот вариант.

На основании опыта однопоточной версии, имеет смысл рассмотреть версию решения, при которой не выполняется установка целевого метода. В таблице 17 и на рисунках 7 и 8 представлены результаты теста. Мы видим улучшение производительности по сравнению с первым решением в среднем в 5.46 раз, однако оно медленнее традиционного решения Groovy в 1.65 раз. Это решение мы и выберем в качестве основного.

4.4. Анализ результатов

Посмотрев на результаты работы всех тестов, мы можем сделать вывод, что представленное динамическое расширение языка Kotlin обыгрывает решение языка Groovy при отсутствии агрессивного изменения типов в точке вызова. При агрессивной смене вызываемых методов в однопоточном режиме, решение в среднем быстрее в 3.94 раз. Однако в многопоточной версии, при активной смене типов, решение пока проигрывает в среднем в 1.65 раз. В некоторых тестах наблюдается ухудшение производительности динамического решения Kotlin, по сравнению со статическим решением, обладающее хуже чем линейной зависимостью.

Рис. 7: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов. По данным таблиц 14 и 17.

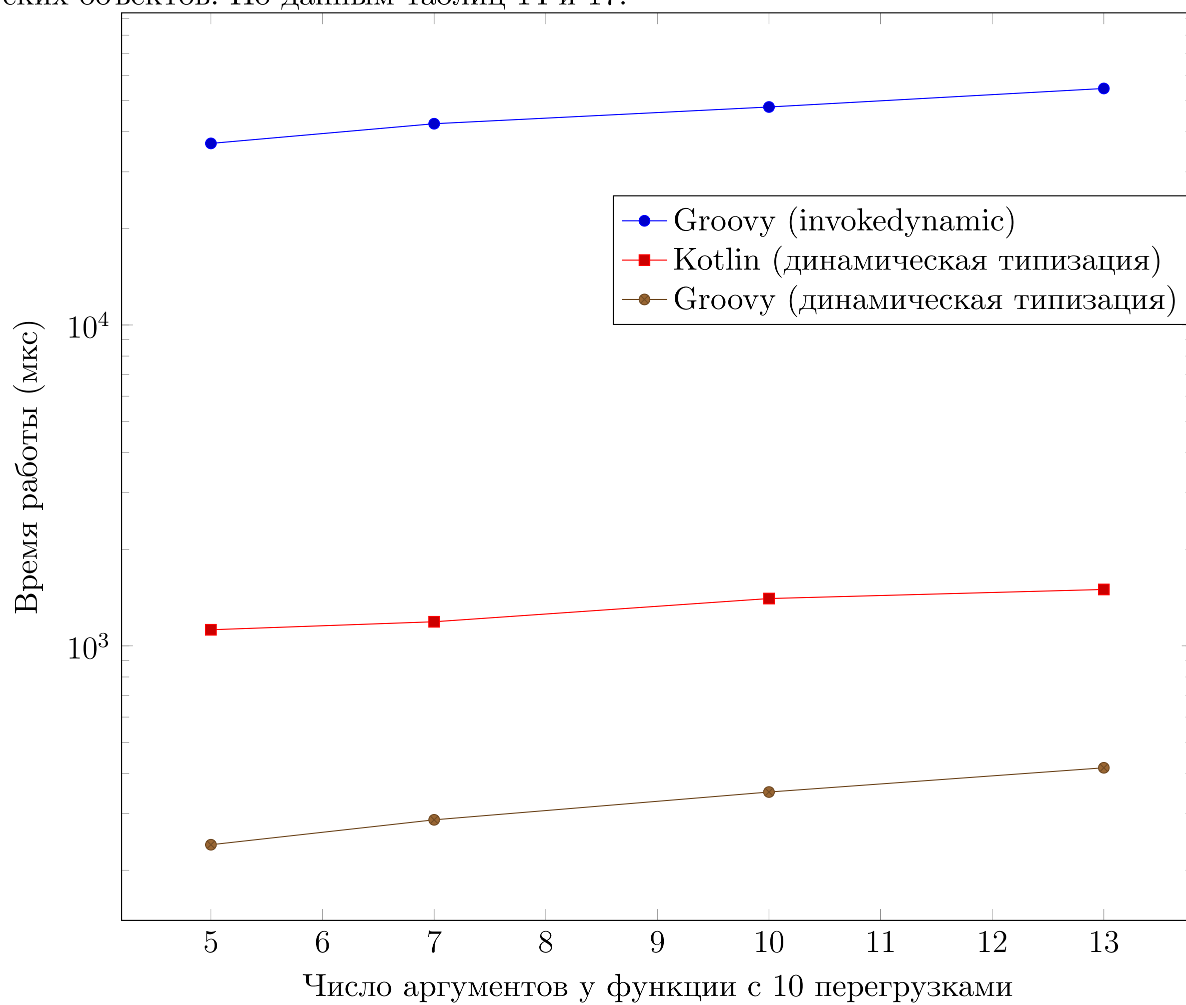
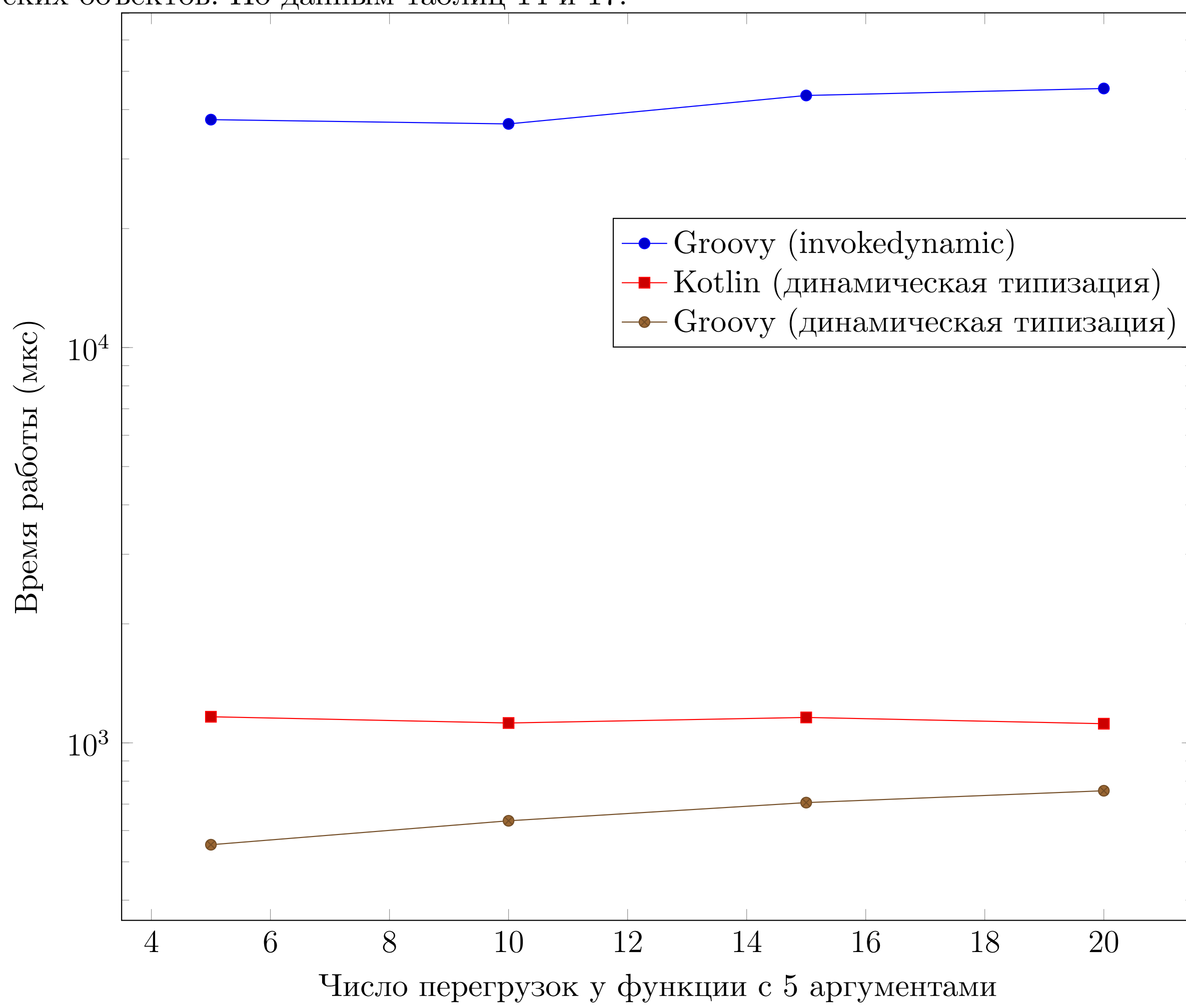


Рис. 8: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов. По данным таблиц 14 и 17.



Заключение

Настоящая работа посвящена поддержке динамической типизации в языке Kotlin при компиляции в Java байт-код. В работе была предложена семантика динамических операций в языке Kotlin. В рамках реализации этого поведения, была написана библиотека, определяющая во время выполнения программы ссылки на методы с учётом перегрузок. В работе были описаны изменения компилятора Kotlin, реализующие разработанное динамическое поведение.

Для получения представления об эффективности разработанного решения, были проведены замеры производительности, которые были сравнены с производительностью решения языка Groovy. Измерения показали, что при отсутствии активной смены типов в месте вызова, предложенное решение эффективней решения Groovy более чем в 3 раза. При однопоточной активной смене типов в точке вызова, решение в среднем быстрее в 3.94 раз.

В качестве дальнейшего направления исследований, можно рассмотреть поиск способа улучшения производительности в случае многопоточной активной смены типов на точке вызова. Для повышения удобства написания динамического кода стоит разработать проверки времени компиляции в тех случаях, в которых множество потенциальных кандидатов можно определить до момента выполнения. Особый интерес состоит в задаче предоставления возможности вызова из языка Kotlin, динамического кода написанного на языке Groovy.

Список литературы

[1] Achour Mehdi и др. Аргументы функции // Руководство по PHP. — 2015. — URL: http://php.net/manual/ru/functions.arguments.php#functions.arguments.type-declaration (дата обращения: 04.04.2017).

[2] C# Language Specification 5.0. — 2013. — URL: https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/language-specification (online; accessed: 05.04.2017).

[3] Scala. js: Type-directed interoperability with dynamically typed languages : Rep. ; Executor: Sébastien Doeraene : 2013.

[4] Dynamic Type - Kotlin Programming Language. — URL: http://kotlinlang.org/docs/reference/dynamic-type.html (online; accessed: 04.04.2017).

[5] Ferrara Anthony. PHP RFC: Scalar Type Declarations // PHP Wiki. — 2015. — URL: https://wiki.php.net/rfc/scalar_type_hints_v5 (online; accessed: 03.04.2017).

[6] Forman Ira R, Forman Nate, Ibm John Vlissides. Java reflection in action. — 2004.

[7] Groovy Truth. — 2014. — URL: http://groovy-lang.org/semantics.html#Groovy-Truth (online; accessed: 02.05.2017).

[8] Gusfield Dan. Algorithms on strings, trees and sequences: computer science and computational biology. — Cambridge university press, 1997.

[9] Horstmann Cay S, Cornell Gary. Core Java: Volume I, Fundamentals. — Pearson Education, 2012.

[10] InvokeDynamic support. — 2014. — URL: http://docs.groovy-lang.org/2.3.3/html/documentation/invokedynamic-support.html (online; accessed: 11.04.2017).

[11] JSR 292: Supporting Dynamically Typed Languages on the Java Platform, 20 11 / J Rose, D Coward, O Bini et al. // URL https://jcp. org/en/jsr/detail.

[12] The Java Virtual Machine Specification. Java SE 8 Edition / Frank Yellin, Tim Lindholm, Gilad Bracha, Alex Buckley // Addison-W esley. — 2015.

[13] Jemerov D., Isakova S. Kotlin in Action. — Manning Publications Company, 2017. — ISBN: 9781617293290.

[14] Jochen Theodorou Cedric Champeau. Annotation Type CompileStatic // GroovyDoc. — 2017. — URL: http://docs.groovy-lang.org/latest/html/gapi/groovy/transform/CompileStatic.html (online; accessed: 03.04.2017).

[15] MethodHandles.Lookup (Java Platform SE 7). — 2016. — URL: https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.Lookup.html (online; accessed: 07.04.2017).

[16] MutableCallSite (Java Platform SE 7). — 2016. — URL: https://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MutableCallSite.html (online; accessed: 10.04.2017).

[17] Odersky Martin. SIP-17 - Type Dynamic // Scala Improvement Process. — 2012. — URL: http://docs.scala-lang.org/sips/completed/type-dynamic.html (online; accessed: 04.04.2017).

[18] OpenJDK: jmh. — 2016. — URL: http://openjdk.java.net/projects/code-tools/jmh/ (online; accessed: 10.04.2017).

[19] Pierce Benjamin C. Types and programming languages. — MIT press, 2002.

[20] PyCharm - Python IDE for Professional Developers. — URL: https://www.jetbrains.com/pycharm/ (online; accessed: 04.04.2017).

[21] Reference C#. dynamic // MSDN. — 2015. — URL: https://msdn.microsoft.com/en-us/library/dd264741.aspx (online; accessed: 03.04.2017).

[22] Selecting overloaded methods at runtime. — URL: https://groups.google.com/forum/#!topic/jvm-languages/J-7GQf7sMLk (online; accessed: 05.04.2017).

[23] Shannon Mark. PEP 484 – Type Hints // PEP Index. — 2014. — URL: https://www.python.org/dev/peps/pep-0484/ (online; accessed: 03.04.2017).

[24] Siek Jeremy, Taha Walid. Gradual typing for objects // European Conference on Object-Oriented Programming / Springer. — 2007. — P. 2–27.

[25] Siek Jeremy G, Taha Walid. Gradual typing for functional languages // Scheme and Functional Programming Workshop. — Vol. 6. — 2006. — P. 81–92.

[26] Stepanov Alexey. Benchmarks for Kotlin and Groovy dynamic code. — 2017. — URL: https://github.com/alejes/kotlin-dynamic-benchmarks (online; accessed: 18.04.2017).

[27] TIOBE Index for March 2017. — URL: https://www.tiobe.com/tiobe-index/ (online; accessed: 05.04.2017).

[28] Troelsen Andrew. Pro C# 5.0 and the. NET 4.5 Framework. — Apress, 2012.

[29] Type checking assignments in Groovy. — 2014. — URL: http://groovy-lang.org/semantics.html#_type_checking_assignments (online; accessed: 02.05.2017).

[30] ”blackdrag” Theodorou Jochen. Indy and CompileStatic as tag team to defeat array access times. — 2015. — URL: http://blackdragsview.blogspot.ru/2015/01/indy-and-compilestatic-as-tag-team-to.html (online; accessed: 11.04.2017).

[31] mypy - Optional Static Typing for Python. — URL: http://www.mypy-lang.org (online; accessed: 04.04.2017).

[32] Виленкин Н.Я., Виленкин А.Н., Виленкин П.А. Комбинаторика. — ФИМА, МЦНМО, 2006.

[33] Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы. — М.: МНЦМО, 2014.

[34] Кряквин В.Д. Линейная алгебра в задачах и упражнениях, М., изд. — Vol. 2.

ПРИЛОЖЕНИЕ А

Таблицы результатов вычислительных тестов

Таблица 5: Результаты теста: Числа Фибоначчи.

Способ компиляции	Номер элемента	Время работы	Ошибка
Kotlin (статическая типизация)	10	0.169	± 0.001
Groovy (статическая типизация)	10	0.177	± 0.018
Kotlin (динамическая типизация)	10	0.460	± 0.016
Groovy (invokedynamic)	10	1.303	± 0.066
Kotlin (статическая типизация)	15	1.834	± 0.002
Groovy (статическая типизация)	15	1.932	± 0.005
Groovy (динамическая типизация)	10	2.657	± 0.072
Kotlin (динамическая типизация)	15	5.972	± 0.040
Groovy (invokedynamic)	15	14.101	± 0.205
Kotlin (статическая типизация)	20	20.391	± 0.076
Groovy (статическая типизация)	20	21.484	± 0.056
Groovy (динамическая типизация)	15	34.054	± 0.633
Kotlin (динамическая типизация)	20	66.279	± 0.393
Groovy (invokedynamic)	20	160.403	± 9.158
Kotlin (статическая типизация)	25	225.264	± 0.540
Groovy (статическая типизация)	25	237.520	± 0.378
Groovy (динамическая типизация)	20	380.277	± 5.394
Kotlin (динамическая типизация)	25	734.782	± 6.521
Groovy (invokedynamic)	25	1706.428	± 24.858
Kotlin (статическая типизация)	30	2581.537	± 286.164
Groovy (статическая типизация)	30	2704.712	± 257.355
Groovy (динамическая типизация)	25	4196.605	± 60.303
Kotlin (динамическая типизация)	30	8235.996	± 95.351
Groovy (invokedynamic)	30	19001.554	± 238.500
Groovy (динамическая типизация)	30	46693.945	± 598.644

Таблица 6: Результаты теста: Возведение матрицы в квадрат.

Способ компиляции	Порядок матрицы	Время работы	Ошибка
Kotlin (статическая типизация)	10	12.299	± 0.151
Kotlin (динамическая типизация)	10	29.745	± 1.744
Groovy (статическая типизация)	10	36.447	± 5.781
Kotlin (статическая типизация)	15	38.676	± 0.494
Groovy (статическая типизация)	15	78.785	± 3.823
Kotlin (динамическая типизация)	15	79.319	± 2.792
Kotlin (статическая типизация)	20	90.442	± 0.665
Groovy (динамическая типизация)	10	135.590	± 9.310
Kotlin (статическая типизация)	25	180.148	± 1.966
Groovy (статическая типизация)	20	188.443	± 19.409
Kotlin (динамическая типизация)	20	203.864	± 5.650
Groovy (invokedynamic)	10	211.729	± 20.056
Kotlin (статическая типизация)	30	309.295	± 4.209
Groovy (статическая типизация)	25	353.132	± 11.320
Kotlin (динамическая типизация)	25	360.012	± 8.796
Groovy (динамическая типизация)	15	427.245	± 25.696
Kotlin (динамическая типизация)	30	607.052	± 13.380
Groovy (invokedynamic)	15	637.817	± 55.380
Groovy (статическая типизация)	30	637.911	± 40.819
Groovy (динамическая типизация)	20	1024.478	± 63.994
Groovy (invokedynamic)	20	1416.187	± 101.069
Groovy (динамическая типизация)	25	2307.156	± 153.459
Groovy (invokedynamic)	25	2493.244	± 137.665
Groovy (динамическая типизация)	30	3782.256	± 233.662
Groovy (invokedynamic)	30	4254.972	± 284.244

Таблица 7: Результаты теста: Z-функция.

Способ компиляции	Длина строки	Время работы	Ошибка
Kotlin (статическая типизация)	100	0.750	± 0.025
Kotlin (динамическая типизация)	100	1.666	± 0.043
Kotlin (статическая типизация)	500	3.122	± 0.038
Groovy (статическая типизация)	100	4.878	± 0.114
Kotlin (статическая типизация)	1000	7.253	± 0.198
Kotlin (статическая типизация)	1250	8.347	± 0.416
Kotlin (статическая типизация)	1500	10.636	± 0.447
Groovy (invokedynamic)	100	20.357	± 1.384
Kotlin (динамическая типизация)	500	23.708	± 1.497
Groovy (динамическая типизация)	100	24.929	± 1.869
Groovy (статическая типизация)	500	41.122	± 1.147
Kotlin (динамическая типизация)	1000	45.962	± 2.063
Kotlin (динамическая типизация)	1250	58.214	± 1.935
Kotlin (динамическая типизация)	1500	76.108	± 4.272
Groovy (статическая типизация)	1000	79.720	± 2.677
Groovy (статическая типизация)	1250	98.272	± 3.451
Groovy (статическая типизация)	1500	118.800	± 2.833
Groovy (динамическая типизация)	500	135.307	± 14.502
Groovy (invokedynamic)	500	151.746	± 37.113
Groovy (invokedynamic)	1000	259.352	± 24.459
Groovy (динамическая типизация)	1000	287.103	± 16.253
Groovy (динамическая типизация)	1250	339.957	± 36.523
Groovy (invokedynamic)	1250	353.048	± 32.751
Groovy (динамическая типизация)	1500	413.121	± 29.404
Groovy (invokedynamic)	1500	479.484	± 151.532

Таблица 8: Результаты теста: Быстрое преобразование Фурье.

Способ компиляции	Размерность вектора	Время работы	Ошибка
Kotlin (статическая типизация)	8	1.492	± 0.009
Kotlin (динамическая типизация)	8	2.030	± 0.046
Kotlin (статическая типизация)	16	3.246	± 0.036
Kotlin (динамическая типизация)	16	4.594	± 0.104
Kotlin (статическая типизация)	32	6.816	± 0.067
Kotlin (динамическая типизация)	32	9.918	± 0.280
Kotlin (статическая типизация)	64	14.217	± 0.126
Groovy (статическая типизация)	8	18.047	± 1.514
Kotlin (динамическая типизация)	64	21.308	± 0.713
Groovy (статическая типизация)	16	33.198	± 2.790
Groovy (динамическая типизация)	8	35.283	± 3.321
Groovy (статическая типизация)	32	68.408	± 5.189
Groovy (динамическая типизация)	16	88.188	± 8.492
Groovy (статическая типизация)	64	143.962	± 22.465
Groovy (динамическая типизация)	32	204.344	± 13.628
Groovy (invokedynamic)	8	258.238	± 151.568
Groovy (динамическая типизация)	64	492.665	± 60.565
Groovy (invokedynamic)	16	496.029	± 184.143
Groovy (invokedynamic)	32	1212.422	± 266.178
Groovy (invokedynamic)	64	3192.987	± 432.527

Таблица 9: Результаты теста: Разрешение перегрузок методов.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (статическая типизация)	0	0	0	0.002	± 0.001
Kotlin (динамическая типизация)	0	0	0	0.002	± 0.001
Groovy (invokedynamic)	0	0	0	0.003	± 0.001
Kotlin (статическая типизация)	3	3	0	0.003	± 0.001
Kotlin (статическая типизация)	1	5	3	0.003	± 0.001
Kotlin (статическая типизация)	5	5	0	0.003	± 0.001
Kotlin (статическая типизация)	10	5	0	0.003	± 0.001
Groovy (статическая типизация)	3	3	0	0.004	± 0.001
Groovy (статическая типизация)	1	5	3	0.004	± 0.001
Groovy (статическая типизация)	5	5	0	0.005	± 0.001
Groovy (статическая типизация)	10	5	0	0.005	± 0.001
Groovy (invokedynamic)	3	3	0	0.005	± 0.001
Groovy (статическая типизация)	0	0	0	0.006	± 0.001
Groovy (динамическая типизация)	0	0	0	0.006	± 0.001
Kotlin (динамическая типизация)	3	3	0	0.006	± 0.001
Kotlin (динамическая типизация)	1	5	3	0.006	± 0.001
Groovy (invokedynamic)	1	5	3	0.007	± 0.001
Kotlin (динамическая типизация)	5	5	0	0.007	± 0.001
Kotlin (динамическая типизация)	10	5	0	0.007	± 0.001
Groovy (invokedynamic)	5	5	0	0.008	± 0.001
Groovy (invokedynamic)	10	5	0	0.008	± 0.001
Groovy (динамическая типизация)	1	5	3	0.011	± 0.001
Groovy (динамическая типизация)	3	3	0	0.015	± 0.001
Groovy (динамическая типизация)	5	5	0	0.025	± 0.001
Groovy (динамическая типизация)	10	5	0	0.025	± 0.001

Таблица 10: Результаты теста: Разрешение перегрузок на списке динамических объектов.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Groovy (invokedynamic)	0	0	0	0.003	± 0.001
Groovy (динамическая типизация)	0	0	0	0.007	± 0.001
Groovy (динамическая типизация)	3	3	0	176.398	± 0.459
Groovy (динамическая типизация)	1	5	3	175.940	± 0.574
Groovy (динамическая типизация)	5	5	0	215.067	± 0.585
Groovy (динамическая типизация)	10	5	0	240.078	± 0.797
Groovy (динамическая типизация)	10	7	0	287.055	± 1.897
Groovy (динамическая типизация)	15	5	0	290.764	± 2.197
Groovy (динамическая типизация)	20	5	0	317.475	± 1.687
Groovy (динамическая типизация)	10	10	0	350.449	± 1.424
Groovy (динамическая типизация)	10	13	0	416.978	± 9.107
Groovy (invokedynamic)	3	3	0	7581.322	± 179.628
Groovy (invokedynamic)	1	5	3	7618.218	± 114.657
Groovy (invokedynamic)	5	5	0	8818.104	± 71.919
Groovy (invokedynamic)	10	5	0	8952.193	± 66.315
Groovy (invokedynamic)	10	7	0	9408.969	± 123.710
Groovy (invokedynamic)	20	5	0	9743.906	± 289.547
Groovy (invokedynamic)	15	5	0	10089.979	± 57.990
Groovy (invokedynamic)	10	10	0	11739.163	± 208.934
Groovy (invokedynamic)	10	13	0	13875.644	± 255.966

Таблица 11: Результаты теста: Разрешение перегрузок на списке динамических объектов в Kotlin.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.002	± 0.001
Kotlin (динамическая типизация)	3	3	0	1088.359	± 13.620
Kotlin (динамическая типизация)	1	5	3	1097.858	± 8.839
Kotlin (динамическая типизация)	5	5	0	1322.772	± 8.575
Kotlin (динамическая типизация)	10	5	0	1318.157	± 38.420

Таблица 12: Результаты теста: Разрешение перегрузок на списке динамических объектов с блокировкой для читателей и писателей.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.002	± 0.001
Kotlin (динамическая типизация)	1	5	3	1208.974	± 11.049
Kotlin (динамическая типизация)	3	3	0	1263.557	± 11.474
Kotlin (динамическая типизация)	5	5	0	1390.524	± 14.243
Kotlin (динамическая типизация)	10	5	0	1401.248	± 38.100

Таблица 13: Результаты теста: Разрешение перегрузок на списке динамических объектов без установки ссылки в точку вызова.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.002	± 0.001
Kotlin (динамическая типизация)	3	3	0	51.263	± 0.165
Kotlin (динамическая типизация)	1	5	3	53.558	± 0.181
Kotlin (динамическая типизация)	10	5	0	62.641	± 0.182
Kotlin (динамическая типизация)	5	5	0	62.914	± 0.149
Kotlin (динамическая типизация)	15	5	0	61.639	± 0.754
Kotlin (динамическая типизация)	20	5	0	60.047	± 0.590
Kotlin (динамическая типизация)	10	7	0	68.197	± 0.188
Kotlin (динамическая типизация)	10	10	0	88.680	± 0.934
Kotlin (динамическая типизация)	10	13	0	108.177	± 0.424

Таблица 14: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Groovy (invokedynamic)	0	0	0	0.005	± 0.001
Groovy (динамическая типизация)	0	0	0	0.013	± 0.001
Groovy (динамическая типизация)	3	3	0	485.750	± 1.285
Groovy (динамическая типизация)	1	5	3	503.761	± 1.160
Groovy (динамическая типизация)	5	5	0	552.467	± 2.044
Groovy (динамическая типизация)	10	5	0	635.103	± 1.941
Groovy (динамическая типизация)	15	5	0	706.075	± 16.708
Groovy (динамическая типизация)	20	5	0	756.651	± 14.786
Groovy (динамическая типизация)	10	7	0	757.137	± 11.724
Groovy (динамическая типизация)	10	10	0	814.036	± 14.640
Groovy (динамическая типизация)	10	13	0	961.995	± 10.399
Groovy (invokedynamic)	10	5	0	36797.990	± 770.071
Groovy (invokedynamic)	1	5	3	37134.258	± 393.078
Groovy (invokedynamic)	5	5	0	37728.824	± 699.825
Groovy (invokedynamic)	3	3	0	37964.294	± 811.578
Groovy (invokedynamic)	10	7	0	42375.937	± 1703.173
Groovy (invokedynamic)	15	5	0	43421.477	± 978.869
Groovy (invokedynamic)	20	5	0	45227.596	± 576.416
Groovy (invokedynamic)	10	10	0	47780.838	± 497.715
Groovy (invokedynamic)	10	13	0	54573.148	± 1450.541

Таблица 15: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов в Kotlin.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.004	± 0.001
Kotlin (динамическая типизация)	5	5	0	6752.358	± 141.164
Kotlin (динамическая типизация)	1	5	3	6771.015	± 169.342
Kotlin (динамическая типизация)	3	3	0	7607.360	± 138.753
Kotlin (динамическая типизация)	10	5	0	8166.267	± 151.733

Таблица 16: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов с блокировкой для читателей и писателей.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.004	± 0.001
Kotlin (динамическая типизация)	1	5	3	9136.311	± 135.171
Kotlin (динамическая типизация)	3	3	0	9788.677	± 81.199
Kotlin (динамическая типизация)	5	5	0	6688.780	± 100.597
Kotlin (динамическая типизация)	10	5	0	9970.722	± 135.417

Таблица 17: Результаты теста: Параллельное разрешение перегрузок на списке динамических объектов без установки ссылки в точку вызова.

Способ компиляции	Число перегрузок	Общее число аргументов	Число аргументов по умолчанию	Время работы	Ошибка
Kotlin (динамическая типизация)	0	0	0	0.004	± 0.001
Kotlin (динамическая типизация)	1	5	3	1065.585	± 30.004
Kotlin (динамическая типизация)	3	3	0	1105.764	± 29.187
Kotlin (динамическая типизация)	20	5	0	1117.987	± 14.512
Kotlin (динамическая типизация)	10	5	0	1122.688	± 20.080
Kotlin (динамическая типизация)	15	5	0	1159.626	± 30.247
Kotlin (динамическая типизация)	5	5	0	1164.237	± 13.570
Kotlin (динамическая типизация)	10	7	0	1189.027	± 37.498
Kotlin (динамическая типизация)	10	10	0	1404.494	± 27.432
Kotlin (динамическая типизация)	10	13	0	1499.107	± 16.909

ПРИЛОЖЕНИЕ Б

Исходный код разработанного решения

Исходный код библиотеки времени выполнения расположен по интернет адресу https://github.com/alejes/kotlin-dynamic-runtime.

Исходный код компилятора и изменения в нём, расположены по интернет адресу https://github.com/alejes/kotlin-dynamic-compiler.