САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ
Центр высшего образования

Кафедра математических и информационных технологий

Сташевский Леонид Евгеньевич

Протоколы в языке программирования Kotlin

Магистерская диссертация

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

Научный руководитель:
м.п.м.и., аспирант Жарков Д. С.

Рецензент:
к.т.н., доцент Глухих М. И.

Санкт-Петербург
2017

Оглавление

Аннотация

В работе предлагается реализация протоколов для компиляторе языка программирования *Kotlin* для платформы *JVM*. Описанный подход не использует тип-обёртку и не выполняет генерацию типов во время выполнения. По сравнению с имеющимися аналогами, выбор метода для вызова не происходит в процессе компиляции. Это позволило расширить поддержку параметрических типов, реализовать перегрузку методов, улучшить работу с примитивными типами.

При реализации используются библиотеки вызовов и рефлексии языка программирования *Java*. Были рассмотрены механизмы разрешения перегрузок, кеширование, многопоточность.

В работе приведено качественное сравнение полученной реализации с подходами в других языках. Помимо этого, сделан анализ времени вызова метода в различных ситуациях, описаны случаи, в которых полученная реализация работает быстрее, чем в языке программирования *Scala*.

Введение

В большинстве современных языков программирования существует механизм наследования [1]. Он вводит над типами отношение подтипов явно, с помощью средств языка. Однако, ручное указание подтипизации вводит ограничение на переиспользование типов из сторонних библиотек.

В качестве альтернативного подхода существует структурная типизация[15]. В языках с такой типизацией отношение подтипов вычисляет компилятор. Такой подход менее распространён из-за сложности его использования. Однако существует промежуточное решение - интеграция структурных типов в языки с номинальной типизацией. Такие типы называются протоколами.

В настоящее время язык программирования $Java$ является одним из распространённых [17]. Он исполняется на платформе $Java\ Runtime\ Environment(JRE)$. Благодаря этому, программы на нём можно запускать на множестве различных устройств. Для платформы написано большое количество библиотек и инструментов разработки. За время существования языка выпускалось несколько его версий.

Одним из ключевых достоинств $Java$ является обратная совместимость языка с предыдущими версиями[2]. С одной стороны, это позволяет использовать части приложений, написанных для разных версий, с другой, тормозит развитие языка и платформы. Вследствие чего развиваются языки, которые совместимы с платформой JVM и ранее написанным кодом на $Java$, пытающиеся упростить написание программ и решить проблемы языка $Java$. Примерами таких языков являются: $Kotlin$, $Scala$, $Ceylon$, $Clojure$. Каждый из них имеет синтаксические конструкции, упрощающие разработку.

$Kotlin$ - это язык программирования совместимый с кодом, написанным на $Java$. Его основной платформой является JVM. Как и в $Java$, в языке $Kotlin$ поддерживается отношение наследования. В данной работе рассматриваются особенности реализации протоколов для JVM в контексте языка программирования $Kotlin$.

1. Обзор предметной области

1.1. Виртуальная машина Java

В данном разделе рассматриваются особенности разработки компилятора для виртуальной машины $Java$(далее JVM).

Виртуальная машина JVM - это часть JRE, ответственная за исполнение программ. JVM представляет из себя абстрактную виртуальную машину, описанную спецификацией[4]. В спецификации описаны набор инструкций, модель памяти, формат исполняемых файлов, ожидаемое поведение. Существует много различных реализаций JVM для разных платформ: $HotSpot$, $IcedTea$, $Viva$, $Azul$ и т.д.

На уровне JVM реализуется большинство языковых инструментов: классы и наследование, интерфейсы, объявление статических полей и методов, набор примитивных типов, механизм исключений. Объявление функций вне класса невозможно.

1.1.1. Типизация

В JVM используется строгая статическая номинальная типизация. Это значит, что все инструкции строго типизированы и наследование поддерживается на уровне виртуальной машины.

Все типы делятся на 2 класса: примитивные и ссылочные. JVM определяет некоторое количество встроенных типов, так же пользователи могут определять собственные типы.

Среди встроенных типов существует набор примитивных типов. Примитивные типы всегда передаются по значению. Это сделано из соображений производительности. Для встроенных примитивных типов существуют зеркальные ссылочные типы, для передачи по ссылке. Для таких типов существует встроенная конвертация из одного типа в другой. Существуют следующие примитивные типы: int, $boolean$, $short$; парные ссылочные типы: $Integer$, $Boolean$, $Short$ соответственно. Объявить пользовательские примитивные типы невозможно.

1.1.2. Исполнение кода

Виртуальная машина $Java$ придерживается объектно-ориентированной парадигме. Единицей исполнения является файл типа $class$. Все классы находятся в общей иерархии наследования: существует встроенный класс $Object$ от которого по умолчанию наследуется каждый класс.

Скомпилированные классы собираются в архивы формата jar. Загрузчик классов виртуальной машины может загружать новые классы по ходу выполнения.

1.1.3. Видимость

Модификаторы видимости поддерживаются на уровне виртуальной машины. Существует 4 модификатора видимости: *private*, *public*, *protected*, *package*. *Private* доступен только в том же окружении, что и объявление, *public* доступен везде, *protected* доступен наследникам, *package* доступен в пределах пакета. Модификаторы доступа можно указать для объявлений классов и их полей.

1.1.4. Работа с памятью

Виртуальная машина скрывает работу с памятью внутри себя. Пользователю предоставляется возможность создать объект и работать с объектом по ссылке. Гарантий на то, что объект имеет постоянный адрес в памяти нет. Как следствие, невозможно прочитать или записать указанный адрес в памяти. Так же невозможно осуществить безусловный переход на какую либо область памяти, в том числе функцию. Механизм освобождения объектов не определён в спецификации, его реализация зависит от виртуальной машины. Ссылки, не указывающие на какой-либо объект, имеют значение *null*.

1.1.5. Байткод виртуальной машины

Рассмотрим подробно структуру классфайла. Каждый классфайл содержит в себе описание одного класса. В каждом классе хранится набор его полей, методов и информация о базовом классе. С каждым полем и методом хранится их тип. Исходный код метода состоит из последовательности однобайтовых(реже двухбайтовых) инструкций и секции локальных переменных.

Рассмотрим имеющийся набор инструкций. В байткоде присутствуют инструкции для работы с локальными переменными, для создания объектов и примитивных типов, арифметические операции, условные переходы.

В виртуальной машине поддерживается вызов методов. Существует несколько различных поведений вызова, для каждого из них существует своя инструкция:

- **invokespecial** используется для вызова конструкторов и приватных методов;
- **invokevirtual** используется для вызова методов у класса;
- **invokestatic** используется для вызова статических методов;
- **invokedynamic** инструкция добавленная для вызова функций в динамических языках, будет рассмотрена далее.

Вызов метода возможен только в том случае, когда его тип и имя возможны во время компиляции. Перед загрузкой класса виртуальная машина запустит верификатор, который проверит наличие методов и соответствие типов аргументов.

1.1.6. Библиотеки для интроспекции

Для получения информации о методах класса во время выполнения существуют две стандартные библиотеки: библиотека $reflection$ и библиотека $invoke$. Обе библиотеки имеют интерфейс на языке $Java$.

Библиотека рефлексии С помощью рефлексии[7] существует способ узнавать содержимое классов и вызывать произвольные методы во время выполнения. Во время выполнения из JVM можно получить информацию о том, какого типа объект, какие методы и поля присутствуют в конкретном объекте и их сигнатуру. У каждого объекта в JVM есть поле с именем $class$, откуда можно получить объект типа $Class$. Он хранит множество дополнительной информации о типе. В этом объекте есть функции для получения списка методов или метода по имени и типу. С каждым методом хранится информация об имени и типе. Существует возможность вызвать полученный метод. Для этого на объекте типа $Method$ вызывают процедуру $invoke$, куда передают объект, на котором происходит вызов, и массив объектов типа $Object$ с аргументами. Все примитивные типы должны быть обёрнуты в соответствующие ссылочные. При вызове происходит проверка типов объекта и аргументов, и проверка их числа. Поиск и вызов метода могут происходить длительное время: необходимо выполнить поиск по имени, выполнить проверку типов для всех аргументов. Если при вызове происходит исключительная ситуация, то порождается исключение типа $InvocationTargetException$, причиной которого является исходное исключение.

При вызове происходит проверка модификаторов доступа. Если попытаться вызвать недоступный метод, то произойдёт исключение $SecurityException$. С помощью рефлексии можно получить доступ к методам, которые не доступны из-за модификатора доступа. Для этого существует флаг доступности и методы, позволяющие его прочитать или изменить. Изменив флаг доступа, можно обойти проверки и вызвать метод. Данный механизм несколько раз предлагался к удалению из JDK и его использование не рекомендуется.

Библиотека invoke Платформа JVM используется в качестве среды выполнения для некоторых динамических языков. Изначально, их поддержка осуществлялась с помощью механизма рефлексии. Данный подход не удобен с точки зрения реализации компилятора и накладывает значительные ограничения на производительность. Разработчики платформы внесли предложение $JavaSpecificationRequest(JSR)$ 292, которое расширяет спецификацию JVM, добавляя туда возможности для работы динамических языков[3]. Предложение было принято и добавлено в спецификацию, начиная с версии 7.

JSR292 содержит в себе новую инструкцию для вызовов $invokedynamic$ и библиотеку для работы с ссылками на функции. Рассмотрим библиотеку подробнее. Два основных класса, которые она содержит: *Lookup* и *MethodHandle*. Первый необходим для получения ссылок на методы. Его главное отличие от поиска с помощью рефлексии в том, что он учитывает области видимости методов и полей. Результатом поиска является *MethodHandle*. *MethodHandle* - это класс, который является прямой ссылкой на метод. При вызове этого метода не происходит проверок типов аргументов, проверки возможности вызова. Данная процедура переносится на стадию поиска. Благодаря этому, ожидается, что поиск будет дольше, а вызов быстрей. Помимо этого существует возможность манипуляции над аргументами: можно вставить аргумент на какую-то позицию. Существует возможность вызова с автоматическим преобразованием типов аргументов.

Рассмотрим подробнее инструкцию *invokedynamic*. Эта инструкция принимает на вход ссылку на специальный статический метод, который называется *bootstrap* методом. Этот метод определяется разработчиком и вызывается только при первой интерпретации инструкции. Результат вызова этого метода сохраняется внутри виртуальной машины и не вычисляется при следующем обращении. Метод имеет специальную сигнатуру: ему передаётся экземпляр класса *Lookup* и произвольный набор пользовательских параметров. Типы всех параметров должны быть из следующего набора: *Type*, *MethodHandle*, *String*, примитивные типы, массивы примитивных типов. Данный метод обязан возвращать объект типа *CallSite*. Он может быть любым *CallSite* из библиотеки примитивов: *ConstantCallSite*, *MutableCallSite* и т.д.

В отличии от рефлексии, управление модификаторами доступа не предусмотрено. Все вызовы функций через класс *MethodHandle* привязаны к месту вызова, в котором был создан *Lookup* объект, более того, невозможно получить метод, который недоступен в месте вызова из-за модификаторов доступа.

Если при вызове функции внутри происходит исключение, то его передача осуществляется как есть.

1.2. Язык Kotlin

Целевой платформой языка *Kotlin* является *JVM*. *Kotlin* поддерживает совместимость с языком *Java* [11], поддерживается раздельная компиляция. Несмотря на это, существуют большие различия.

Для абстрактных методов в интерфейсах и классах в *Kotlin* можно задавать реализацию по умолчанию. Для методов присутствует перегрузка по параметрам. Аргументы могут иметь значения по умолчанию.

1.2.1. Типизация

В *Kotlin* присутствует проверка пустоты ссылок на уровне системы типов. В обычные ссылки невозможно положить пустую. Для работы с пустыми ссылками существует специальный синтаксис.

В $Kotlin$ присутствует собственный набор встроенных типов: Int, $Short$, $Boolean$, и т.д. Для объявления типов, в которых можно хранить пустую ссылку, в конце добавляется вопросительный знак: $Int?$, $Short?$, $Boolean?$. Благодаря тому, что типы проаннотированны, в процессе компиляции $Kotlin$ проверяет корректность присваивания на уровне типов. Во время трансляции встроенный тип может стать как примитивным, так и ссылочным, в зависимости от контекста использования и возможности хранения пустой ссылки. Например, в большинстве случаев $Int?$ будет оттранслирован в $Integer$, а Int в int.

$Kotlin$ использует номинальную систему типов с наследованием. Поддерживается механизм интерфейсов. В языке присутствуют операторы is и as. С помощью первого, во время выполнения можно проверить является ли объект экземпляром указанного типа. Второй оператор пытается выполнить приведение типов.

В $Kotlin$ существует возможность создавать интерфейсы. В интерфейсах можно задать реализацию методов по умолчанию.

Параметрические типы В $Kotlin$ существует возможность создавать параметрические типы. Тип параметра указывается в треугольных скобках. На параметрических типах можно задавать отношение порядка, в зависимости от типа параметра. Это можно сделать, как и в объявлении типа, так и при определении шаблонного метода.

1.3. Структурная и номинальная типизации

В данном разделе рассмотрены основы структурной типизации и номинальной типизации.

Номинальная типизация Номинальная типизация во многих языках реализована с помощью механизма наследования[15]. Пользователь явно указывает наследников каждого класса. Исходя из этого, компилятор разрешает вызовы методов. В некоторых языках происходят опциональные проверки перегрузок.

У номинальной типизации существует множество плюсов. В первую очередь, она является интуитивной для программиста: программист знает какие классы могут быть переданы и явно поддерживает иерархию наследования. В процессе компиляции можно использовать информацию о наследовании и расположить поля объектов так, чтоб к ним удобно было обращаться по смещению, тем самым существенно уменьшается время доступа к полям объектов.

Проверка типов при номинальной типизации сводится к обходу направленного ациклического графа иерархии типов. Связывание двух объектов разных типов является корректным, если существует путь из типа присваиваемого объекта к типу целевого объекта. Граф иерархии строится из отношений наследования: типы являются вершинами, отношение "наследник-родитель" задаёт рёбра.

Подведя итог, можно привести следующие плюсы номинальной типизации:

- быстрее во время выполнения;
- помогает избежать случайных отношений;
- проще объявлять рекурсивные типы;
- быстрее проверять.

Структурная типизация Структурная типизация основывается на содержимом типа[15]. Имя типа имеет меньшее значение.

Тип A является подтипом типа B, если выполнены следующие условия:

- A содержит все именованные методы B;
- Тип каждого метода A является подтипом соответствующего метода в B.

Как следствие, любой тип является подтипом пустого типа. Типы функций являются подтипами только при полном совпадении типов аргументов и возвращаемого значения.

Одним из основных минусов структурной типизации является семантическая неочевидность: два не связанных по смыслу типа могут иметь схожую структуру. Благодаря этому, структурная типизация менее распространена. В основном она используется в функциональных языках программирования, а также в некоторых скриптовых языках как альтернатива динамической. Структурная типизация схожа с динамической, но является более безопасной: гарантия наличия полей и методов обеспечивается на этапе компиляции.

У структурной типизации есть свои плюсы. Она является более гибкой: имена типов не имеют значения, имеет значение только содержимое. Два типа с одинаковым содержимым считаются эквивалентными. Проверка типов при структурной типизации: проверка вложенности двух типов.

Плюсы структурной типизации:

- замкнуты: тип содержит в себе всё описание структуры;
- имя типа имеет символический характер и не влияет на отношение подтипизации.

Несмотря на то, что повсеместное использование структурной типизации часто не оправдано, существуют ситуации, в которых необходимо её использование. К примеру с помощью структурной типизации можно писать обобщённые функции, которые работают с любыми объектами, обладающими заданным набором методов [14].

1.4. Существующие реализации протоколов

В этом разделе проведён анализ существующих реализация протоколов, приведены плюсы и минусы различных подходов. В качестве реализаций, рассмотрены языки: *Scala*, *Go* и прототип *Whiteoak*.

1.4.1. Scala

Рассмотрим реализацию протоколов в языке программирования *Scala*[13]. *Scala* компилируется в *JVM* байткод. Реализацию можно разделить на 2 части: проверку типов при компиляции и генерация байткода для вызова метода.

Типы протоколов являются обобщением типов *Scala*. Проверка типов протоколов изначально поддерживается компилятором, с одним исключением: накладывается ограничение на явное наследование. Поэтому, для проверки типов достаточно отключить только проверку указания отношения наследования.

Рассмотрим генерацию байткода. В *Scala* типы протоколов существуют только во время компиляции, в байткоде не генерируется новый тип. В процессе компиляции, тип протокола стираются до типа *Object*. Вызов метода у объекта типа протокола происходит с помощью специального механизма *applyDynamic*: для каждого места обращения к полю объекта генерируется статический метод. В статическом методе происходит поиск нужного метода. Сперва метод ищется в кеше[10], затем с помощью рефлексии. Так как внутри вызова может произойти исключительная ситуация, в этом методе происходит обработка исключения и разворачивание исходного исключения из *InvokationTargetException*. Таким образом, вызов дополнительного метода выглядит прозрачно для пользователя. В реализации используют кеширование для всех найденных методов. В качестве кеша используют ссылочный список, где сохраняют каждый увиденный тип. В связи с тем, что во время выполнения отсутствуют типы - запрещено создание протоколов с параметрическим типом или использование в протоколах внешних параметрических типов.

Внимательное тестирование подхода, используемого в *Scala* показало, что есть возможность написать код, порождающий некорректное поведение и ошибку времени выполнения, несмотря на корректность синтаксиса. Такое поведение связано с использованием примитивных типов в параметрических классах:

```
class Impl[T] {
  def x(i: T): T = i
}

class Main {
  type Proto = { def x(i: Int): Int }
```

```

  def foo(arg: Proto) {
    print(arg.x(42))
  }

  def main(args: Array[String]) {
    foo(Impl[Int]())
  }
}
```

С точки зрения языка *Scala*, такой код является корректным. Он проходит проверку типов и компиляцию, но во время выполнения происходит исключительная ситуация. Причиной этому является несовпадение типов методов. Во время генерации класса *Impl* тип T становится наиболее общим типом *Object*. В то же время, для вызова в методе *foo* сгенерировался метод, ищущий функции с типом *Int*. Поэтому, во время выполнения нужная функция в объекте *Impl* отсутствует и происходит исключение.

Для того чтобы обойти проверки модификаторов доступа, каждый раз выставляется флаг модификации доступа. Исключения, связанные с модификаторами доступа, игнорируются.

1.4.2. Whiteoak

Whiteoak - это расширение языка *Java*, которое добавляет поддержку структурных типов[9]. *Whiteoak* состоит из двух частей: модификации компилятора языка программирования *Java*, с целью добавить новую синтаксическую конструкцию с проверкой типов и библиотеки времени выполнения. Для каждого протокола, во время компиляции, генерируется абстрактный класс. Во время выполнения, для каждого типа, который присваивается к типу протокола, генерируется специальный класс-наследник абстрактного класса протокола. Этот класс делегирует вызов методов протокола реальным вызовам на объекте.

Из такой реализации следует, что невозможно создание самостоятельных структурных типов и определение конструкторов, определение статических методов.

```
struct Source {
  int read();
}

class Impl {
  int read() {
```

```
    return 0;
  }
}

void exhaust(Source s) {
  while (s.read() >= 0) {
  }
}

void run() {
  Impl impl = new Impl();
  exhaust(impl);
}
```

Преимущество данного подхода - высокая скорость работы. После генерации класса обёртки, скорость вызова метода протокола сводится к двум выполнениям инструкции *invokevirtual*, что практически не отличается от обычного вызова метода.

Существенным недостатком данного подхода является потеря идентичности объекта при оборачивании. Дело в том, что один и тот же объект, дважды приведённый к типу протокола, имеет разные обёртки, а, следовательно, и разные ссылки. В *Whiteoak* проблема решена следующим образом: везде, где пользователь использует объект как сущность, используется ссылка на реальный объект. Это вносит определённые ограничения на использование протоколов в типовых параметрах. Появляется неоднозначность, какой объект ожидает метод: обёртку или ссылку. Эта проблема не решается в момент выполнения, поэтому *Whiteoak* запрещает использование параметрических типов для протоколов.

1.4.3. Язык Go

Язык *Go* компилируется в бинарный код. Интерфейсы в *Go* являются структурными типами[18].

```
type geometry interface {
  area() float64
  perim() float64
}

type rect struct {
  width, height float64
}
```

```

func (r rect) area() float64 {
  return r.width * r.height
}
func (r rect) perim() float64 {
  return 2*r.width + 2*r.height
}

func measure(g geometry) {
  fmt.Println(g)
  fmt.Println(g.area())
  fmt.Println(g.perim())
}

func main() {
  r := rect{width: 3, height: 4}
  measure(r)
}
```

Рассмотрим реализацию структурных типов в компиляторе *gccgo*. Так как *Go* компилируется в бинарный код, разрешён вызов метода по указателю. Для каждого интерфейса компилируется специальный дескриптор типа. Каждый дескриптор содержит в себе список методов. Для каждого метода сохраняются: имя, сигнатура, указатель на реализацию. Объект содержит в себе ссылку на набор дескрипторов. По возможности, разрешение метода происходит во время компиляции. В ситуациях, когда ссылки на необходимые дескрипторы не получается разрешить во время компиляции, это происходит во время выполнения, при создании объекта.

Плюс подхода заключается в высокой скорости вызова и гибкости реализации для различных случаев использования. Такой подход не реализуем на *JVM* из-за отсутствия прямого доступа к памяти.

1.5. Постановка задачи

Главная цель данной работы - реализация поддержки протоколов в компиляторе языка *Kotlin*. Для её достижения необходимо выполнить следующие задачи:

- Исследовать способы реализации структурных типов в *JVM*;
- Реализовать и сравнить различные подходы;
- Придумать механизм работы протоколов и проверку типов для них в компиляторе языка *Kotlin*;

- Реализовать генерацию кода протоколов в компиляторе;
- Измерить производительность и сравнить полученное решение с другими языками.

Важным ограничением для реализации является прозрачность протоколов для пользователей: должна сохраниться поддержка раздельной компиляции, генерируемый код должен быть потокобезопасным, использование должно быть схоже с обычными типами.

Требования к реализации Перед тем как приступить к реализации, необходимо определить, какие ограничения необходимо соблюдать для получения результатов, которые могут быть применимы на практике:

- Язык $Kotlin$ совместим с $Java$. Это значит, что объекты из языка $Kotlin$ должны быть корректными объектами для языка $Java$ и в обратную сторону. Одной из важных особенностей, которую необходимо сохранить, - это идентичность объекта, то есть приведение к его к типу протокола не должно модифицировать ссылку на объект.
- Другим важным требованием является возможность использования протоколов как в качестве типов коллекций, так и в качестве элементов существующих коллекций. Таким образом, необходимо чтобы протоколы поддерживали типовые параметры.
- Протоколы не должны вносить ощутимых расходов на потребление памяти и время работы при приведении типов и вызове метода.
- Возможности использование протоколы должны быть схожи с обычными типами.
- Необходимо избегать генерации кода в процессе выполнения

2. Описание реализации

В данном разделе описано поведение полученной реализации протоколов в языке *Kotlin*.

2.1. Постановка проблемы

Рассмотрим обычную ситуацию использования протокола с точки зрения пользователя. Существует протокол, являющийся структурным типом. Существует уже заранее скомпилированный тип, который содержит методы протокола(а значит и является его подтипом), но явное наследование не указывается. С точки зрения виртуальной машины, привести существующий тип можно лишь к тому типу, который содержится в его иерархии наследования. Протокол был создан независимо от подходящего типа, поэтому в иерархии наследования он отсутствует. Более того, в месте вызова гипотетически может быть любой тип, удовлетворяющий требованиям протокола. Это значит, что тип во время компиляции не известен и приведение типа к типу протокола невозможно. Следовательно, невозможно сгенерировать вызов с помощью любой инструкции в момент компиляции.

Как было рассмотрено в разделе 1.4, для решения данной проблемы существует два подхода. Первый подход заключается в генерации класса обёртки для каждого типа, который приводится к структурной и делегирует вызов объекту конкретного типа. В момент генерации класса обёртки должен быть известен тип объекта, который приводится к протоколу, а значит, либо необходима информация о всех типах в момент компиляции, либо обёртки для примитивных типов придётся генерировать в момент первого приведения каждого типа. Таким образом, необходимо, либо генерировать новые классы во время выполнения, либо отказаться от раздельной компиляции. Более того, возникает проблема определения идентичности объекта: дважды приведённый объект будет иметь разные экземпляры классов обёрток и проверка на идентичность не пройдёт.

Второй подход заключается в использовании библиотек, которые работают с информацией об объекте во время выполнения: библиотеки вызовов или библиотеки invoke. При таком подходе, информацию о типах знать не нужно, её можно получить во время компиляции. Данный подход может работать медленнее, но не накладывает ограничений на разработку компилятора, поэтому он был выбран в качестве базового подхода.

2.2. Прототип

Для реализации структурных типов необходимо уметь производить поиск и вызов любой функции на произвольном объекте. Такой функциональности можно добиться, используя одну из двух стандартных библиотек: библиотеку рефлексии или библиотеку вызовов. Поскольку в данном контексте обе библиотеки предоставляют схожую функциональность, выбор между ними сводится к поиску более производительного решения. Для того, чтобы выбрать между ними, реализован прототип, задачей которого является измерение производительности двух подходов. В качестве модели была выбрана ситуация вызова одного метода протокола с одним ссылочным аргументом, возвращающая ссылочный аргумент. В прототипе осуществляется ручная генерация байткода для двух подходов.

2.2.1. Реализация с помощью библиотеки рефлексии

Рассмотрим реализацию с помощью библиотеки рефлексии. Для места вызова метода протокола генерируется статический метод. Этот метод принимает на вход объект, на котором происходит вызов, и возвращает экземпляр класса $Method$. После этого, все аргументы упаковываются в массив и вызывается метод *invoke*, который делает вызов нужного метода и сохраняет на стек результат. Поскольку сигнатура метода *invoke* не типизирована, то необходимо выполнить приведение типов. Внутри статического метода делается обращение к классу на поиск метода с нужным именем и типом. При использовании протоколов имеет место следующее предположение: типов, которые будут приведены к протоколу в конкретном месте вызова, будет значительно меньше, чем самих вызовов метода. Тип и имя метода известны в момент компиляции и не меняются в процессе выполнения. Процедура поиска метода происходит значительно дольше чем вызов, поэтому для поиска метода выгодно использовать кеширование результата. Так как тип объекта, на котором осуществляется вызов один, результат поиска сохраняется в статическое поле класса при первом вызове и используется как результат в дальнейшем.

2.2.2. Реализация с помощью библиотеки вызовов

Рассмотрим реализацию с помощью библиотеки вызовов. Аналогично предыдущему варианту, для каждого места вызова генерируется статический метод. В этом методе происходит вызов инструкции *invokedynamic*, которой передаётся ссылка на вспомогательный статический метод и тип вызываемого метода. Помимо этого, JVM передаёт экземпляр класса *Lookup* для данного места вызова. Внутри вспомогательного метода создаётся объект типа $ConstantCallSite$, который содержит в себе вспомогательный класс с функциями, отвечающими за поиск нужного метода. Для поиска метода происходит вызов метода $find$ для объекта, который возвращает экземпляр класса $MethodHandle$, ссылающийся на необходимый метод. Из вызова статического метода возвращается $MethodHandle$, вызов у него *invokeExact*, вызывает необходимый метод. В отличии от механизма рефлексии, оборачивание аргументов в массив

Тип метода	Рефлексия(нс)	Вызовы(нс)	Фактор
Класс	4.815 ± 0.072	5.002 ± 0.031	0.963
Наследник	4.797 ± 0.023	5.002 ± 0.037	0.959
Интерфейс	4.818 ± 0.055	5.016 ± 0.104	0.961
Анонимная функция	153.892 ± 0.909	5.069 ± 0.111	30.359

Таблица 1: Результаты бенчмарка для реализации с помощью рефлексии

не требуется. Помимо *invokeExact* существует так же метод *invoke*, который в случае необходимости производит приведение типов.

2.2.3. Сравнение производительности

Цель данного измерения - понять скорость вызова метода с помощью двух решений в различных случаях. В ходе различных тестирований была выявлена зависимость между типом реализации вызываемого объекта и временем работы, данный критерий был взят в качестве параметра тестирования. Для тестирования были выделены следующие типы целевых объектов

- финальный метод класса;
- переопределённый метод наследника, вызванный через ссылку на базовый класс;
- метод интерфейса, реализованный классом;
- метод интерфейса, реализованный анонимной функцией.

Тестирование производилось для функции с одним аргументом, внутри которой вызывается специальный метод JMH[8] предотвращающий удаление непродуктивного кода. Вызовы производились в одном потоке. В процессе тестирования было обнаружено, что существует влияние типа реализации протокола на скорость вызова. Время выполнения измерялось в наносекундах на вызов, каждый бенчмарк запускался в течении секунды, результаты были усреднены по 20 запускам. Результаты измерений для рефлексии приведены в таблице 1.

Из результатов измерения видно, что в среднем библиотека вызовов немного медленнее чем библиотека рефлексии. Однако, существуют ситуации, в которых библиотека рефлексии в 30 раз медленнее. Для поиска причин замедления было принято решение изучить машинные инструкции, с целью найти причину замедления и, по возможности, устранить её. Для этого были сделаны снимки с помощью утилиты $perfasm$, которая позволяет вывести горячие точки - байткод и машинные инструкции, выполнявшиеся дольше всего. Листинг горячих точек можно найти в приложении 1. Изучение горячих точек показало, что основное время выполнения происходит внутри исходного кода виртуальной машины, в пределах одной инструкции вызова.

Таким образом, оптимизации, с точки зрения байткода, не представляются возможными.

После оценки результатов измерений было принято решение реализовать оба способа в компиляторе и предоставить пользователю возможность выбрать одну из реализаций с помощью опций компиляции. Обе библиотеки достаточно динамично развиваются, поэтому в будущем возможна оптимизация отдельных случаев использования в виртуальной машине.

2.3. Реализация протоколов в компиляторе Kotlin

Поддержка протоколов в компиляторе заключается в введении способа объявить тип протокола и определении набора правил для приведения и вызове метода. Для того, чтобы объявить тип протокола в компиляторе был введён новый тип интерфейса. Если перед именем интерфейса указать ключевое слово *protocol*, интерфейс будет автоматически считаться объявлением протокола типа:

```
protocol interface Proto {
  fun id(i: Int): Int
}

class Impl {
  fun id(i: Int): Int {
    return i
  }
}

fun foo(arg: Proto) {
  println(arg.id(42))
}

foo(Impl())
```

Объявленный протокол может использоваться так же, как и обычный интерфейс: являться типовым параметром, быть родительским классом и т.д. Однако информация о нём отсутствует во время выполнения: у классов, явно реализующих тип протокола, он отсутствует в списке интерфейсов.

Для того чтобы избежать неоднозначного поведения, в протоколах отключены методы и аргументы по умолчанию.

2.3.1. Выбор метода

В зависимости от способов объявления и реализации протокола, у разных типов, приведённых к протоколу, могут быть разные методы в одном месте вызова. Помимо этого, в языке $Kotlin$ поддерживаются перегрузки методов с разным числом и типом параметров. Подобные ситуации могут приводить к неоднозначности в выборе метода.

Примером такой ситуации может послужить использование параметрических типов в протоколе:

```kotlin
protocol interface Sample<T> {
  fun foo(arg: T)
}

class X {
  fun foo(i: Int)
  fun foo(i: Int?)
}

...
val x = X()
val first: Sample<Int> = x
val second: Sample<Int?> = x
```

На уровне виртуальной машины отсутствует информация о типовых параметрах. В случае метода $foo(Int)$ транслированный тип будет примитивным, а в $foo(Int?)$ ссылочным. Если бы тип $Sample$ не был протоколом, то из двух ссылок $first$ и $second$ был бы доступен метод $foo(Int?)$. В случае протокола, будет запущен алгоритм выбора метода, который выберет наиболее близкий по типам метод в каждом случае. В данном случае, из ссылки $first$ доступен для вызова метод $foo(Int)$, а из ссылки $second$ доступен $foo(Int?)$.

Алгоритм выбора метода независимо запускается во время компиляции и во время выполнения. Выбор метода во время компиляции необходим при приведении типа объекта к типу протокола. В данном случае, происходит проверка наличия всех методов протокола в объекте во время компиляции. В случае, если не существует метода с необходимым возвращаемым значением, но существует метод с более конкретным типом возвращаемого значения, то он будет считаться подходящим.

Во время выполнения, выбор метода происходит непосредственно во время вызова. Так как представление компилятора и виртуальной машины о методах различны, сохранить результат выбора во время компиляции как есть не представляется возможным. Данный метод гарантировано существует, так как это было проверено на этапе компиляции.

2.3.2. Исключения

Вызов метода протокола отличается от вызова обычного метода. Некоторые реализации могут предполагать оборачивание исключения в вспомогательный класс. В случае, если в методе протокола происходит исключение, оно будет передано без оборачивания, независимо от типа реализации протокола.

2.3.3. Типовые параметры

В отличии от реализаций протоколов для других JVM языков, протоколы в $Kotlin$ полностью поддерживают работу с типовыми параметрами, так же, как и обычные типы. Протокол может являться, как типовым аргументом, так и самостоятельно содержать типовые параметры. Более того, реализация параметризованного класса может являться подтипом не параметризованного протокола, и наоборот: реализация не параметризованного класса может быть подтипом параметризованного протокола с подставленным типом.

Операторы проверки и приведения типа Для обычных типов в $Kotlin$ разрешено приведение или проверка принадлежности любого объекта к любому типу. В большинстве случаев, объекты, реализующие протоколы, не являются типами протоколов, с точки зрения JVM. В таком случае возможно два поведения: реальная проверка типов инструкцией JVM, либо проверка соответствия протоколу во время выполнения. С точки зрения пользователя, каждое из поведений может являться не ожидаемым, поэтому было принято решение запретить использование протоколов в правой части операторов as и is. В тоже время, разрешено присваивание объекта обычного типа к типу протокола, если проверка ограничений на тип проходит во время компиляции. Если известен исходный тип объекта типа протокола, то можно выполнить приведение типов с помощью встроенных операторов. Так же можно выполнить проверку типов на объекте типа протокола. Таким образом существует возможность перехода от обычных типов к протоколам и от протоколов к обычным типам.

3. Особенности реализации

Рассмотрим детали реализации протоколов в *Kotlin*. Основная часть реализации заключается в механизме вызова метода. Проблема заключается в том, что типов виртуальной машины(реальных типов методов) может быть не достаточно для выполнения корректного вызова. Идея реализации состоит в том, что во время компиляции вся информация о типах, необходимая в процессе выполнения для вызова метода, отдельно сохраняется в сгенерированном байткоде для использования во время вызова. Информация о том, какой метод необходимо вызывать выясняется в момент выполнения.

3.1. Компиляция

Для корректной работы протоколов в языке необходима генерация байткода в процессе компиляции. Реализация протоколов добавляет необходимость в введении новой разновидности типов, поэтому необходима модификация существующего алгоритма проверки типов для корректного приведения традиционных типов к типам протокола. Помимо проведения проверок, так же необходима генерация кода для вызова методов. Как было замечено ранее, было реализовано два независимых генератора кода в компиляторе для различных подходов: библиотеки рефлексии и библиотеки вызовов.

3.1.1. Проверка типов

При написании программы, тип протокола можно получить только с помощью приведения типа: ручного или автоматического. При приведении типов компилятор обязан проверять условие подтипизации: является ли тип присваиваемого объекта подтипом целевого. В случае если целевой тип является протоколом, обычная проверка заменяется проверкой вложенности типов.

Для того чтоб проверить вложенность присваиваемого типа в целевой, нужно убедиться, что для каждого метода целевого типа существует аналогичный метод с таким же именем, тип которого является подтипом исходного метода. Тип возвращаемого значения может быть более конкретным.

При проверке типов могут встречаться типовые параметры. В данном контексте возможны две ситуации: типовой параметр имеет подстановку, типовой параметр не имеет подстановки. В первом случае, проверка типов происходит так, будто бы вместо типового параметра стояло подставленное значение. Данная подстановка является ожидаемой, с точки зрения использования, но, может быть, некорректной для сгенерированного кода во время вызова. Эта ситуация будет разобрана подробнее в разделе 3.2. Если подстановка для типового параметра отсутствует, он рассматривается как тип *Object*.

Отдельно стоит упомянуть про отсутствие подтипизации на методах, в случае, если аргументы не совпадают, но являются надтипами аргументов целевого метода. Данное поведение разрешено во многих функциональных языках, однако, с точки зрения разработчика, такое поведение может быть неоднозначным и запрещено на уровне проверки типов.

В компилятор были добавлены соответствующие сообщения об ошибке компиляции, в случае, если проверка типов завершается ошибкой при проверке протоколов.

3.1.2. Генерация байткода

Рассмотрим как происходит трансляция протоколов в байткод виртуальной машины. При генерации байткода тип протокола заменяется на тип $Object$. Заметим, что такой переход корректен с точки зрения приведения типов, так как $Object$ является базовым типом для любого типа в виртуальной машине. Таким образом информация о протоколах отсутствует во время выполнения. В результате стирания типов протоколов не требуется генерировать дополнительное приведение типов, оно происходит автоматически.

Рассмотрим генерацию байткода для вызова методов. Для каждого места вызова метода протокола внутри соответствующего класса генерируется специальный статический метод. Данный метод принимает объект, на котором происходит вызов и возвращает экземпляр класса $MethodHandle$ или $Method$, в зависимости от выбранного способа реализации.

Внутри статического метода генерируется вызов инструкции *invokedynamic* со следующими аргументами:

- объект типа $Handle$ - ссылка статический метод $getBootstrap$ класса стандартной библиотеки $ProtocolCallSite$, сконструированный во время компиляции;
- строка с именем вызываемого метода;
- тип вызываемого метода.

Данный метод будет вызван только в первый раз, затем результат вызова будет сохранён в виртуальной машине и возвращён при следующих вызовах. Результатом данного вызова является объект типа $ProtocolCallSite$. Так как данный объект привязан к месту вызова, он является удобным для хранения кеша. На полученном объекте типа $ProtocolCallSite$ генерируется вызов метода $getMethod$(или $getReflectMethod$). Механизм работы данных методов рассмотрен в разделе 3.2. Результатом вызова является ссылка на объект типа $MethodHandle$ или $Method$, которая и возвращается из статического метода.

3.2. Поддержка времени выполнения

Для поддержки времени выполнения в стандартную библиотеку был добавлен класс *ProtocolCallSite*. Основная функциональность данного класса - это поиск и кеширование метода. Как было рассмотрено ранее, имя и тип вызываемого метода были сохранены во время компиляции и переданы данному классу при создании. Помимо этого, с помощью *JVM*, в класс передаётся экземпляр класса *Lookup* из библиотеки вызовов, привязанный к месту вызова инструкции *invokedynamic*.

3.2.1. Поиск подходящего метода

По сохранённой информации, во время выполнения, можно выполнить поиск метода. В библиотеках интроспекции присутствуют методы для поиска метода по имени и типу, но использование их напрямую связано с рядом проблем.

Одной из проблем является невозможность вызова метода типа, скрытого модификатором доступа. Рассмотрим дизайн типичного библиотечного интерфейса *List*. Интерфейс класса объявлен публичным. Однако библиотечные методы (например *Arrays.asList*) возвращают экземпляр приватного класса, приведённый к типу интерфейса. При обращении к полю *class*, данного объекта, получаем ссылку на информацию о приватном классе. Если попытаться вызвать метод этого класса произойдёт исключение *SecurityException*. Аналогичную проблему в *Scala* решают с помощью смены модификатора доступа во время выполнения.

Другой проблемой является необходимость соответствия запрашиваемого типа с тем, что присутствует в объекте. Во время использования примитивных типов, при компиляции может быть сохранён ссылочный или примитивный тип, в зависимости от вызова. Но не известно какой метод с каким типом является ожидаемым во время выполнения. Например, при использовании класса с параметрическим типом, может присутствовать метод с параметрическим аргументом типа *Int*. При трансляции, параметрический тип станет типом *Object*, в то же время, объект мог быть приведён к типу протокола с ожидаемым типом *Int*. Из описанных выше проблем следует, что встроенный в библиотеки поиск метода не подходит для реализации, поэтому было принято решение самостоятельно реализовать поиск метода с помощью библиотеки рефлексии, который бы учитывал все особенности поиска методов для протоколов. Результатом такого поиска является экземпляр класса *Method*, который, в случае необходимости, можно привести к объекту типа *MethodHandle*, с помощью имеющегося экземпляра класса *Lookup*.

Поиск метода разбивается на две части: выбор метода в объекте для вызова, поиск доступного метода для вызова.

Поиск подходящего метода для вызова С помощью библиотеки рефлексии возможно запросить все методы объекта. Заметим, что для получения подходящего метода среди всех методов, можно выполнить разновидность алгоритма выбора перегрузки. Рассмотрим используемый алгоритм детальнее: для каждого метода с подходящим именем считаем вектор расстояния по типам аргументов. В качестве образцового вектора используем тот, что был сохранён во время компиляции. Для оценки расстояния между аргументами используем следующую градацию:

- Точное совпадение;
- Совпадение с точностью до оборачивания примитивных типов;
- Возможность присвоить объект необходимого типа в объект типа в аргументах.

В случае, если ни один из пунктов не выполняется, хотя бы для одного аргумента, метод считается не подходящим и исключается из рассмотрения. Среди всех методов выбираем тот, у которого в векторе все компоненты минимальны. Наличие и единственность такого метода проверяется в процессе компиляции при проверке типов.

Поиск доступного метода Найденный на этапе выбора метод, может быть не доступен для вызова из-за того, что его тип является скрытым модификатором доступа. Однако в момент приведения типа существовал доступный метод, не скрытый модификатором доступа. Это значит, что в иерархии наследования существует тип, не скрытый модификатором доступа, на котором можно сделать вызов. Искомый метод можно получить из обхода иерархии интерфейсов и базовых классов целевого типа, с поиском выбранного метода с помощью библиотеки рефлексии

3.2.2. Кеширование результатов поиска метода

Самый частый сценарий использования протокола - это вызов метода. Написание прототипа показало, что процедура вызова метода протокола выполняются заметно дольше, чем вызов аналогичного метода через интерфейс. Вызов метода протокола состоит из двух частей: первая часть заключается в выборе наиболее подходящего метода в объекте. Вторая в укладке аргументов и непосредственном вызове. Заметим, что результат поиска метода для одного типа метода и одного типа объекта будет всегда одинаков. Это значит, что можно запомнить результат поиска. Тогда выполнять поиск метода придётся только при первом вызове на одном типе.

Заметим, что во время компиляции одного места вызова тип вызываемого метода известен и не будет изменён во время выполнения.

Будем предполагать, что вызовов на протоколах будет происходить больше, чем типов объектов, на которых происходит вызов в одном месте вызова. Данное предположение вполне оправдано: оно начинает выполняться когда на одном типе дважды происходит вызов.

Исходя из сделанных предположений, можно сделать вывод о том, что использование кеша для хранения результата поиска может быть оправданно в случае, если обращение к кешу происходит быстрее чем поиск метода.

Для поиска оптимального кеша было реализовано 3 варианта кеширования:

- ссылочный список;
- циклический массив с вытеснением последнего добавленного элемента;
- хеш-таблица.

Ключом для поиска является объект класса. Значением - экземпляр класса $Method$ или $MethodHandle$, в зависимости от выбранной реализации. Размер кеша задаётся с помощью опций компиляции. По умолчанию размер кеша не ограничен(за исключением циклического списка, размер по умолчанию которого равен 20 элементам). Сравнение производительности кешей будет приведено в разделе 5.

3.2.3. Вызов метода из нескольких потоков

Механизм вызова метода объекта в JVM не зависит от того из скольки потоков происходят вызовы. Поэтому, одним из требований вызова является вызов метода из нескольких потоков. Отличие вызова метода протокола от вызова обычного метода заключается в наличии изменяемого состояния: кеша. Это значит, что если работа с кешом является потокобезопасной, то вызов метода протокола тоже является потокобезопасным.

В нашем решении было рассмотрено 2 подхода к обеспечению потокобезопасности кеша: $synchronized$[6] конструкция и $read - write$ блокировка. Конструкция $synchronized$ является флагом метода и обеспечивает гарантию того, что помеченный метод исполняется только в одном потоке. Блокировка $read - write$ реализована с помощью класса стандартной библиотеки $ReentrantReadWriteLock$[5]. Кеш протоколов состоит из двух методов, первый метод для поиска, второй для добавления элемента, следовательно можно производить раздельную блокировку для чтения и записи.

Каждый из подходов имеет свои преимущества и недостатки. Например, в $synchronized$ методов блокировка может не производиться если работает только один поток, помимо этого может выполняться оптимистичная блокировка. В то же время $read - write$ блокировка работает оптимально в случае, если происходят только чтения из кеша. Таким образом, если метод протокола уже был найден, блокировки не происходит.

3.2.4. Опции компиляции

Исходя из реализации, видно, что производительность протоколов сильно зависит от случаев использования. Например, использование протоколов в одном потоке избавит от необходимости использовать синхронизацию кеша, ограничение размера кеша уменьшит размер потребляемой памяти. Помимо этого, существует два независимых способа реализации: библиотека вызовов и рефлексия.

В связи с этим было принято решение предоставить пользователю контроль над параметрами компиляции протоколов с помощью опций компиляции. Так же это позволит детально измерить различные конфигурации протоколов для определения оптимальной реализации. При компиляции пользователю доступны следующие параметры:

- $protocols-backend$ - способ реализации;
- $protocols-cache$ - используемый тип кеширования;
- $protocols-cache-size$ - размер кеша.

Для выбора доступно несколько различных видов кеширования: ссылочный список, LRU кеш[12] на основе хеш-таблицы, циклический массив. Для кеша доступна многопоточная реализация. В качестве опций по умолчанию используется однопоточный ссылочный список неограниченного размера.

4. Качественное сравнение

Одной из важнейших характеристик в реализации протоколов является выразительность получившегося решениям в языке. Иными словами, интерес представляет мощность полученной конструкции. Подобное сравнение ранее приводилось для реализации протоколов в языке $Whiteoak$[9]. Функциональность приведённой реализации не зависит от опций компиляции. Подходы реализации для разных платформ существенно отличаются, поэтому в контексте рассмотренного решения интересно сравнение реализация протоколов для языков платформы JVM. Сравнение особенностей использования протоколов приведено в таблице 2.

Критерий	Scala	Whiteoak	Kotlin
Объявление метода	+	+	+
Определение метода	−	+	−
Параметрический тип	−	−	+
Функция с параметрическим аргументом	+	−	+
Ограничения на типовой параметр	−	−	+
Операторы приведения	+	+	+
Корректная работа с примитивными типами	−	+	+
Сохранение идентичности	+	−	+
Объявление конструктора	−	+	−

Таблица 2: Сравнение функциональной выразительности языков

Из сравнения видно, что выразительность решений схожа в основной функциональности. Однако с помощью решения описанного в работе удалось преодолеть недостатки в выразительности, присутствующие в других языках в достаточно строгих условиях: сохранения идентичности и раздельной компиляции. Одной из ключевых особенностей является поддержка работы с параметрическими типами. Это существенно расширяет контекст использования протоколов. Помимо этого рассмотренный подход позволяет избежать некорректного поведения при использовании функций с аргументами примитивных типов.

5. Измерение производительности

Одним из важнейших вопросов является производительность протоколов. Случаи использования протоколов разделяются на две части: приведение типов и вызов метода. Как известно данная реализация не предполагает модификации целевого объекта, поэтому приведение типов происходит прозрачно.

С другой стороны, вызов метода связан с выполнением множества трудоёмких операций. Задача данного раздела заключается в том, чтобы понять насколько дольше происходит вызов метода протокола по сравнению с обычным методом, а так же сравнить поведение различных способов реализации протоколов в $Kotlin$ в различных ситуациях. Это поможет сформировать понимание о том, какие настройки более актуальны в различных ситуациях.

Приведено сравнение производительности с реализацией в языке $Scala$.

5.1. Особенности измерений

Для корректной интерпретации результатов измерений необходимо чтобы измерения были сделаны с учётом того, как исходный код исполняется на целевой платформе. Для этого рассмотрим, как происходит выполнение исходного кода.

Целевой платформой для компиляции является виртуальная машина языка $Java$. Программы представляют из себя инструкции байткода, который, в зависимости от ситуации, может быть исполнен интерпретатором, либо он может быть скомпилирован в машинный код и выполнен на процессоре. Трансляция байткода в исходный код может занимать длительное время. Во время трансляции исходного кода так же происходит его анализ: исключаются неиспользуемые участки кода, вычисляются константы. Выполнение машинного кода может быть в десятки раз быстрее интерпретации.

Одной из существенных проблем является измерение времени работы метода: функция для получения текущего времени не специфицирована. Существует множество способов реализации функции времени, каждый из них зависит от конкретной реализации виртуальной машины. Например существуют виртуальные машины синхронизирующие значение времени между потоками, поэтому может наблюдаться два эффекта с вызовом метода для получения текущего времени:

- длительное время работы(значительно более длительное чем время выполнения теста);
- большое время обновления.

Решение о трансляции байткода в машинный код принимается в момент выполнения на основе собранной статистики об исходном коде. Для учёта данных особенностей используют методику прогрева: метод, который необходимо протестировать, выполняют некоторый промежуток времени. За это время виртуальная машина соберёт статистику необходимую для трансляции метода.

Ещё одной важной особенностью является автоматическая сборка мусора - процесс освобождения памяти объектов, ссылки на которых не достижимы в программе, с целью освободить занимаемую ими память. Сборка мусора длительная процедура, в процессе которой выполнение программы приостанавливается. В случае если сборка мусора запустится во время выполнения измерений, результаты измерений будут искажены.

Для того чтоб избежать подобных эффектов была использована библиотека JMH[8]($Java$ *microbenchmark harness*). В данной библиотеке содержится инструментарий написания тестов производительности. Эти инструменты автоматически позволяют измерить время работы методов, решить проблемы с удалением неиспользуемого кода, провести прогрев и провести измерения. Результаты измерений представлены как выборка из нормального распределения с заданным средним и стандартным отклонением.

5.2. Условия измерения производительности

- **ЦПУ:** Intel(R) Core(TM) i7-6700 CPU @3.40GHz.
- **Оперативная память:** 32GB DDR4.
- **Операционная система:** Linux 4.10.0-21-generic 23-Ubuntu SMP x64.
- **Версия Kotlin:** 1.1.1.
- **Версия JDK:** OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode).
- **Версия JMH:** 1.18.

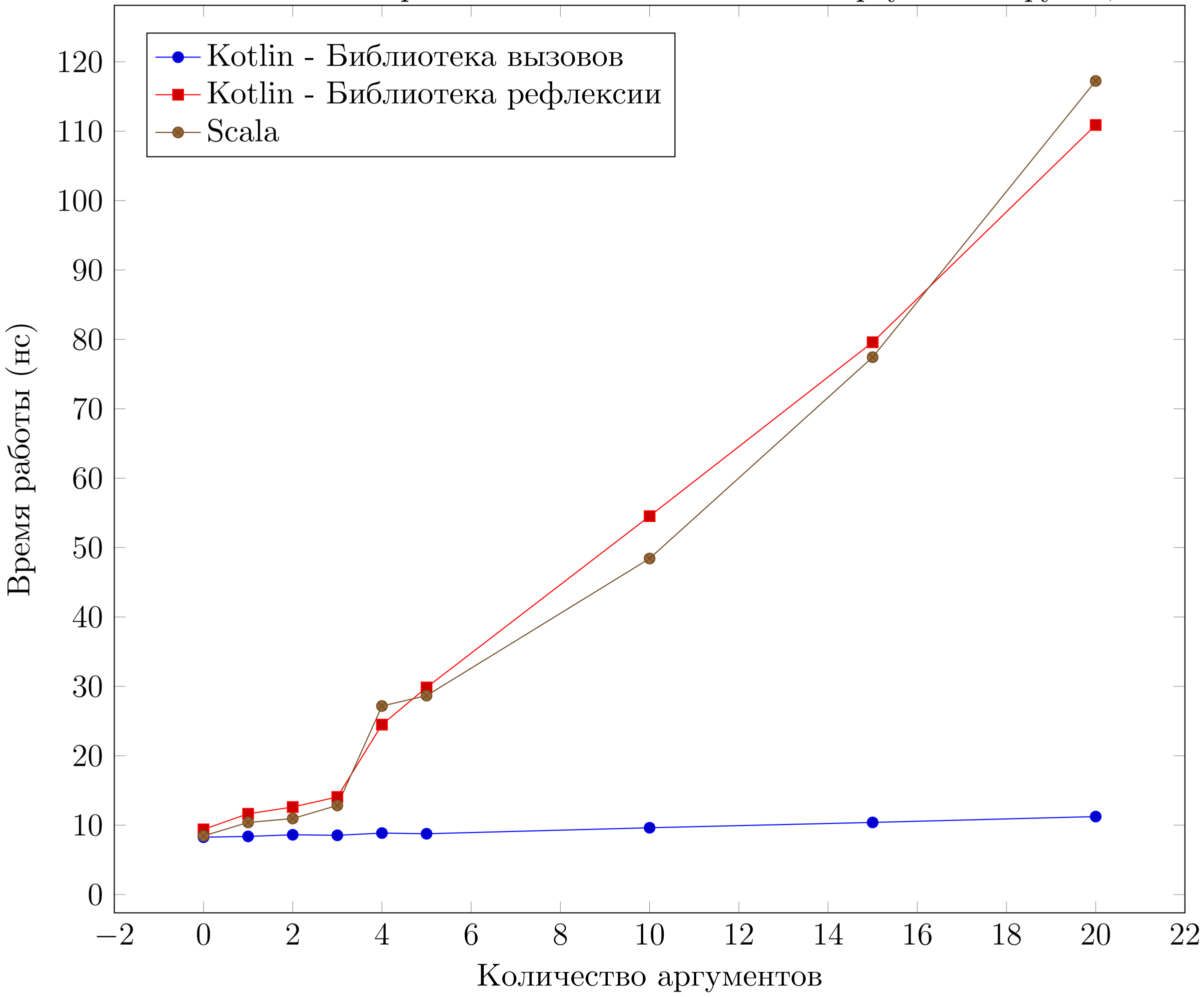
5.3. Рассмотренные тесты

В ходе тестирования работы протоколов было написано множество тестов производительности. В данном разделе приведены тесты интересные с точки зрения сравнения реализации протоколов в $Kotlin$ с другими реализациями, а также тесты, демонстрирующие типичные сценарии использования протоколов в реальном окружении.

5.3.1. Сравнение времени работы методов с различным числом аргументов

Одной из особенностей прототипа была возможность создания только одного метода с одним аргументом. После реализации полноценной поддержки протоколов, была выявлена зависимость времени вызова метода от числа аргументов. В случаях использования библиотеки вызовов, с ростом числа аргументов, время работы было значительно ниже чем с использованием библиотекой рефлексии. На рисунке 1 приведено сравнение реализаций с помощью библиотеки вызовов и рефлексии с использованием целевых объектов одного типа. В качестве кеша поиска использовалась хеш-таблица неограниченного размера. Помимо графиков для реализаций протоколов в *Kotlin*, был добавлен график для языка *Scala*, так как в нём используется реализация с помощью библиотеки рефлексии.

Рис. 1: Зависимость времени вызова от количества аргументов функции.



Данное поведение объясняется способом работы с аргументами у различных библиотек. В библиотеке рефлексии аргументы передаются в массиве, как набор ссылок типа *Object*. При каждом вызове необходимо проверить, подходит ли тип приведённого аргумента к типу вызова. Более того, необходимо время на выделение и заполнение массива аргументов на стеке.

В случае библиотеки вызовов проверка типов произошла во время поиска, поэтому дополнительные время работы слабо зависит от количества аргументов.

Время вызова в языке *Scala* сравнимо с соответствующей реализацией в языке

Kotlin, однако местами работает быстрее. Это объясняется необходимостью запуска алгоритма поиска перегрузки метода в реализации на языке *Kotlin*, в то время как в *Scala* используется встроенный метод для поиска.

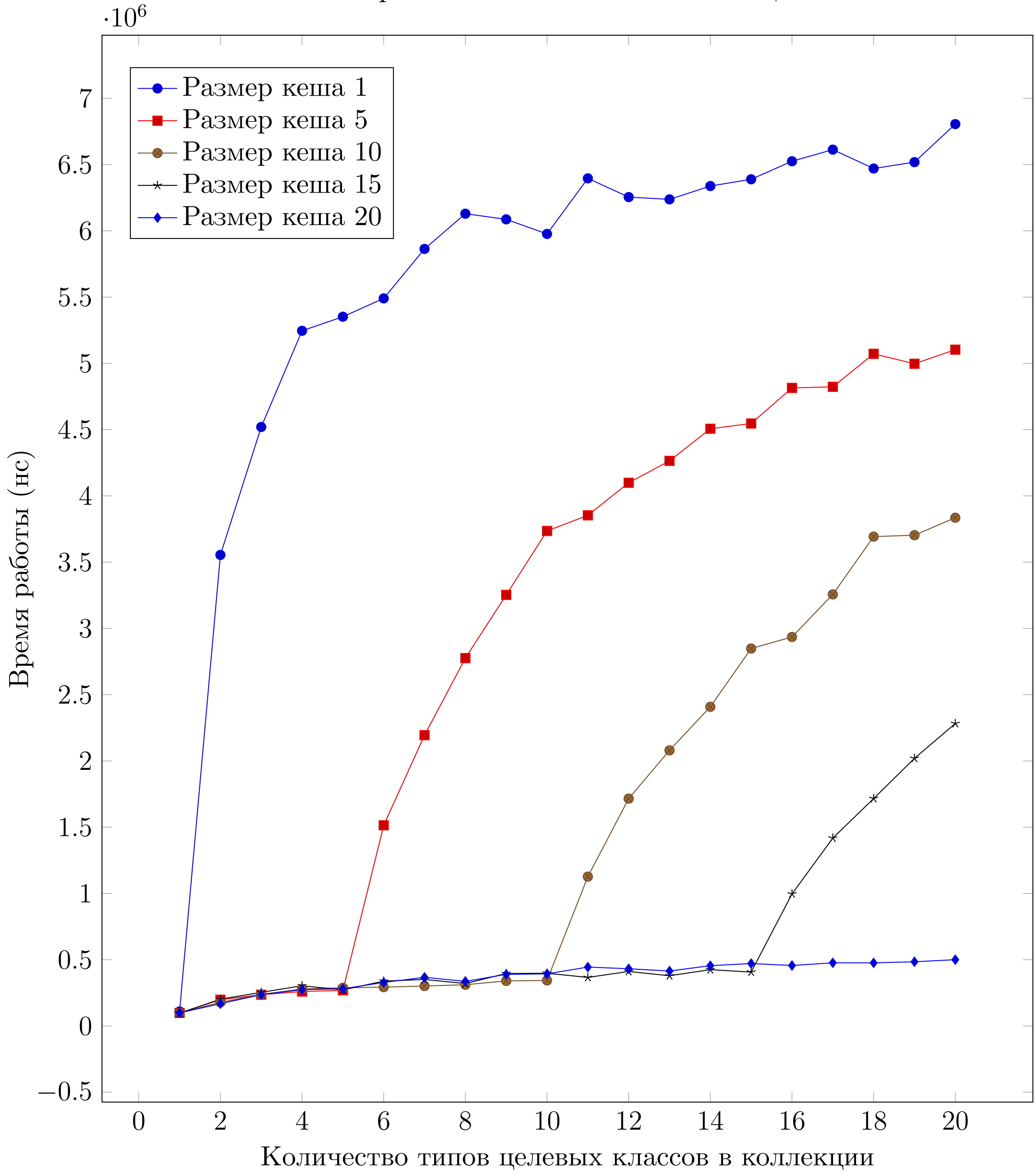
Подробное сравнение реализаций приведено в приложении 1.

5.3.2. Сравнение времени работы вызова метода с различным числом целевых классов

Одной из проблем при использовании протоколов является выбор размера кеша. С фиксированным размером кеша большое количество типов в одном месте вызова метода протокола может значительно повлиять на производительность. С другой стороны, чем больше размер кеша, тем дольше осуществляется поиск в нём.

Для сравнения производительности протоколов с различным размером кеша и разным количеством целевых объектов, был написан тест, в котором на списке из фиксированного числа объектов типа протокола вызывается один метод. В списке содержатся объекты типа протокола различных типов с точки зрения виртуальной машины. В каждом тесте был использован список фиксированной длины 10^5, равномерно перемешанных элементов. На каждом объекте вызывался метод без аргументов. Результаты измерений приведены на рисунке 2.

Рис. 2: Зависимость времени вызова от количества целевых классов.



Из измерения видно, что до тех пор, пока размер кеша меньше, чем количество типов, посетивших место вызова, время вызова является небольшой константной. В то же время, если число типов превышает размер кеша, время работы существенно возрастает. Исходя из этого, можно дать рекомендацию к настройке компиляции: размер кеша должен быть не меньше количества типов, приводимых к протоколу.

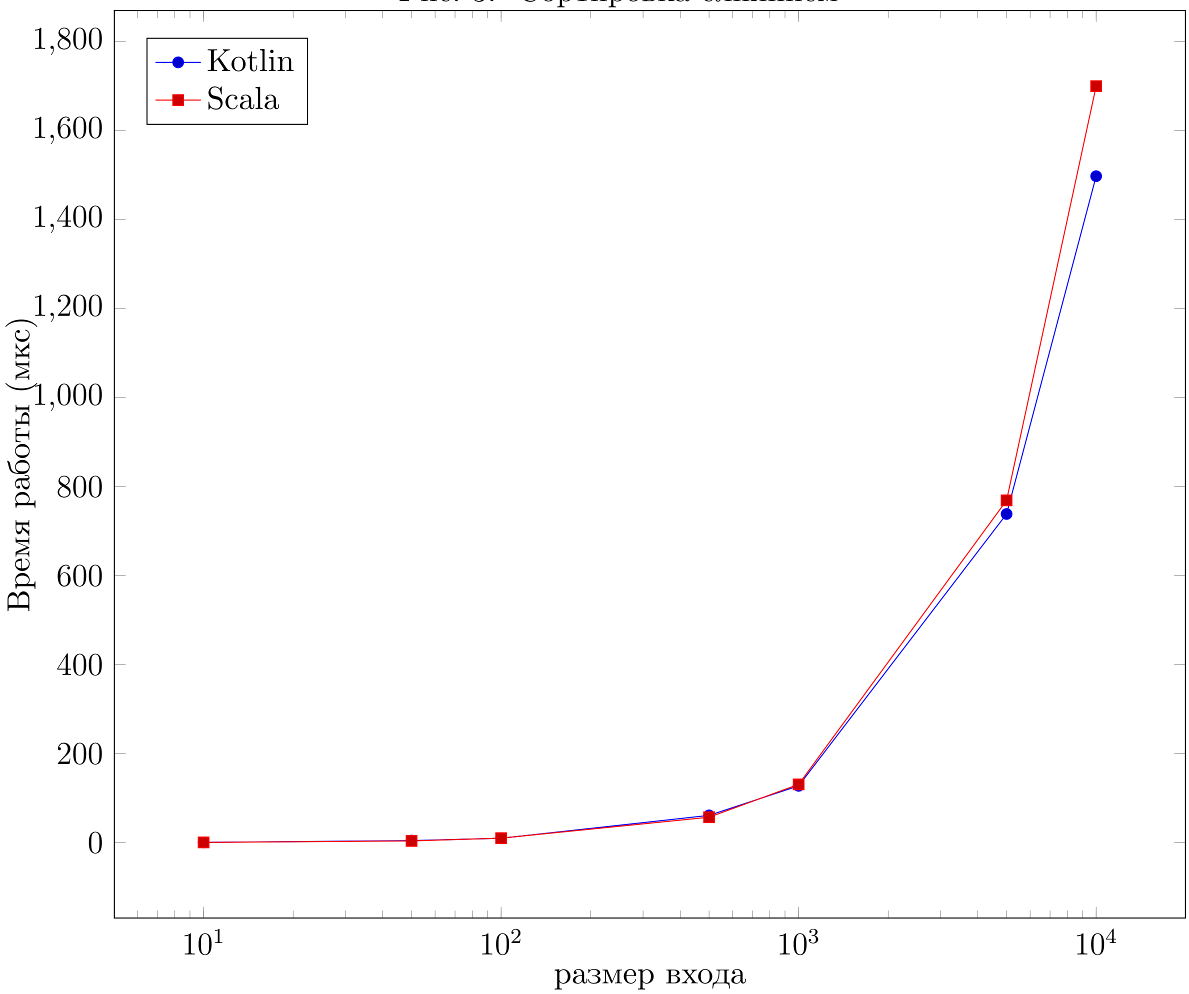
Подробные результаты измерений приведены в приложении 2.

5.3.3. Сравнение реализаций в различных языках

Для сравнения полученной реализации с другими языками необходим тест, который будет отражать случаи реального использования протоколов в программах. Одним из таких случаев использования служит объявление интерфейса *Comparable* протоколом. Введение протокола *Comparable* позволят использовать любые объекты, обладающие оператором сравнения, без явного наследования.

Был написан тест, сортирующий числа с помощью сортировки слиянием[10]. В качестве сравниваемых объектов выступают объекты типа протокола, содержащего оператор сравнения. В качестве реализации были выбраны JVM языки использующие схожий подход к реализации: $Scala$, $Kotlin$. Для компиляции $Kotlin$ используется реализация с использованием библиотеки вызовов и хеш-таблицей в качестве кеша. Результаты измерений приведены на рисунке 3.

Рис. 3: Сортировка слиянием



Исходя из графика, видно, что вызов функции с одним аргументом на протоколе в обоих языках работает примерно одно время. Разница становится заметной при росте числа вызовов. Подробное сравнение реализаций приведено в приложении 3.

Заключение

Данная работа посвящена протоколам в языке программирования $Kotlin$. В работе были рассмотрены способы реализации протоколов для различных платформ. Была предложена реализация протоколов в языке $Kotlin$ для платформы JVM. Приводится качественное сравнение полученной реализации с реализациями в других языках. В отличии от других языков сохраняется идентичность объектов, приведённых к типу протокола; реализована поддержка работы с параметрическими и примитивными типами, перегрузки методов по аргументам.

Для работы не требуется генерация новых типов во время выполнения, вместо этого, для вызова метода используются библиотека рефлексии и библиотека вызовов. В отличии от реализаций в других языках используется алгоритмы разрешения перегрузок во время выполнения. Для уменьшения расходов на поиск метода при вызове, было использовано кеширование.

По ходу реализации был написан прототип и набор бенчмарков, для сравнения полученной реализации с реализациями в других языках. В общем случае решение работает не медленнее решения в языке $Scala$. Было выявлено, что при использовании библиотеки вызовов время вызова метода протокола почти не зависит от числа аргументов. В случае вызова методов с более чем одним аргументом, полученная реализация существенно превосходит реализацию в языке $Scala$.

Исходные коды полученной реализации, прототипа и бенчмарков, а так же результаты измерений доступны в репозитории[16].

Список литературы

[1] Cook William R., Hill Walter, Canning Peter S. Inheritance is Not Subtyping // Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '90. — New York, NY, USA : ACM, 1990. — P. 125–135. — Access mode: http://doi.acm.org/10.1145/96709.96721.

[2] Corporation Oracle. OpenJDK Developers' Guide. Compatibility // OpenJDK Developers' Guide. — 2010. — Access mode: https://goo.gl/yYrhre (online; accessed: "12.04.2017").

[3] Corporation Oracle. JSR 292: Supporting Dynamically Typed Languages on the JavaTM Platform // JSRs: Java Specification Requests. — 2011. — Access mode: https://goo.gl/6mDxU1 (online; accessed: "14.04.2017").

[4] Corporation Oracle. The Java® Virtual Machine Specification // Specification: JSR-337 Java® SE 8 Release Contents. — 2015. — Access mode: https://goo.gl/WCbpCi (online; accessed: "12.04.2017").

[5] Corporation Oracle. ReentrantReadWriteLock // Java Documentation. — 2015. — Access mode: https://goo.gl/YmRKwA (online; accessed: 17.04.2017).

[6] Corporation Oracle. Synchronized Methods // Java Documentation. — 2015. — Access mode: https://goo.gl/zSs6iI (online; accessed: 17.04.2017).

[7] Corporation Oracle. Package java.lang.reflect // Package java.lang.reflect Description. — 2016. — Access mode: https://goo.gl/8ocfVZ (online; accessed: "12.04.2017").

[8] Corporation Oracle. Code Tools: jmh. — 2017. — Access mode: https://goo.gl/Wr2H4S (online; accessed: 17.04.2017).

[9] Gil Joseph, Maman Itay. Whiteoak: Introducing Structural Typing into Java // SIGPLAN Not. — 2008. — Oct. — Vol. 43, no. 10. — P. 73–90. — Access mode: http://doi.acm.org/10.1145/1449955.1449771.

[10] Introduction to Algorithms, Third Edition (International Edition) / Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. — The MIT Press, 2009. — ISBN: 0262533057. — Access mode: https://www.amazon.com/Introduction-Algorithms-International-Thomas-Cormen/dp/0262533057?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0262533057.

[11] JetBrains. Compatibility // Compatibility - Kotlin Programming language. — 2017. — Access mode: https://goo.gl/8PbMDd (online; accessed: "14.04.2017").

[12] Johnson Theodore, Shasha Dennis. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm // Proceedings of the 20th International Conference on Very Large Data Bases. — VLDB '94. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1994. — P. 439–450. — Access mode: http://dl.acm.org/citation.cfm?id=645920.672996.

[13] Martin Odersky Gilles Dubochet. Compiling Structural Types on the JVM // A Comparison of Reflective and Generative Techniques from Scala's Perspective. — 2009. — Access mode: https://goo.gl/umxr4B (online; accessed: 17.04.2017).

[14] Ostermann Klaus. Nominal and Structural Subtyping in Component-Based Programming // Journal of Object Technology. — 2008. — Jan. — Vol. 7, no. 1. — P. 121–145. — Access mode: http://www.jot.fm/contents/issue_2008_01/article4.html.

[15] Pierce Benjamin C. Types and Programming Languages. MIT Press. — MIT Press, 2002. — ISBN: 0-262-16209-1. — Access mode: https://www.cis.upenn.edu/~bcpierce/tapl/.

[16] Stashevskii Leonid. Kotlin protocols implementation. — 2017. — Access mode: https://github.com/e5l/ProtocolsGenerator (online; accessed: 17.04.2017).

[17] TIOBE. TIOBE Index for April 2017 // TIOBE software BV. — 2017. — Access mode: https://goo.gl/qtX6AP (online; accessed: 12.04.2017).

[18] Taylor Airs – Ian Lance. Go Interfaces. — 2009. — Access mode: https://goo.gl/wlUa0k (online; accessed: 17.04.2017).

Зависимость времени вызова(нс) от числа аргументов

Количество аргументов	Kotlin(рефлексия)	Kotlin(вызовы)	Scala
0	12.027 ± 0.058	11.32 ± 0.091	8.453 ± 0.048
1	14.494 ± 0.036	11.617 ± 0.548	10.392 ± 0.068
2	16.31 ± 0.048	11.582 ± 0.002	10.952 ± 0.045
3	17.948 ± 0.289	11.674 ± 0.009	12.838 ± 0.049
4	27.699 ± 0.079	12.2 ± 1.137	27.169 ± 0.152
5	31.318 ± 0.09	12.414 ± 0.001	28.665 ± 0.214
10	53.635 ± 0.148	14.805 ± 0.964	48.425 ± 0.324
15	79.463 ± 0.314	15.078 ± 0.876	77.437 ± 0.603
20	124.527 ± 0.334	15.695 ± 0.006	117.241 ± 0.619

Зависимость времени вызова от размера кеша и числа целевых объектов в месте вызова

кеш / типы	1	5	10	15	20
1	1.095	0.983	0.991	0.967	0.990
2	35.543	1.981	1.780	2.022	1.683
3	45.198	2.364	2.387	2.538	2.365
4	52.456	2.600	2.797	3.038	2.740
5	53.517	2.694	2.885	2.723	2.808
6	54.901	15.144	2.930	3.387	3.286
7	58.633	21.945	3.016	3.512	3.672
8	61.292	27.759	3.112	3.203	3.368
9	60.865	32.528	3.403	3.959	3.895
10	59.764	37.353	3.439	3.982	3.935
11	63.958	38.532	11.270	3.665	4.446
12	62.544	40.988	17.156	4.115	4.320
13	62.372	42.641	20.799	3.791	4.149
14	63.379	45.066	24.085	4.251	4.551
15	63.887	45.461	28.482	4.070	4.714
16	65.248	48.146	29.352	9.962	4.565
17	66.125	48.228	32.565	14.179	4.769
18	64.699	50.720	36.928	17.160	4.763
19	65.183	49.968	37.031	20.195	4.847
20	68.051	51.031	38.352	22.822	5.004

ПРИЛОЖЕНИЕ 3

Сравнение времени работы протоколов в JVM языках

Размер массива	Scala	Kotlin	Фактор
10	0.516 ± 0.009	0.599 ± 0.002	0.861
50	3.819 ± 0.018	4.612 ± 0.059	0.828
100	9.492 ± 0.307	9.868 ± 0.044	0.961
500	57.09 ± 0.211	61.271 ± 0.944	0.931
1000	130.902 ± 6.631	127.616 ± 1.463	1.025
5000	769.18 ± 2.255	738.57 ± 11.148	1.041
10000	1700.018 ± 66.679	1497.701 ± 19.117	1.135

Листинг *perfasm* для вызова метода в прототипе

```
Hottest code regions (>10.00% "cycles" events):
[Hottest Region 1].....................C2, level 4
ProtocolsBenchmark_reflectInterfaceLambda_jmhTest

and    $0xfff,%ebx
cmp    $0xff0,%ebx
jbe    0x00007fe97d20dfee
sub    $0x10,%rsp
mov    %edx,%eax
movzwl -0x2(%rdi,%rax,2),%ebx
mov    %bx,-0x2(%rsp,%rax,2)
dec    %rax
jne    0x00007fe97d20dfdc
mov    %rsp,%rdi
mov    $0x1,%eax
push   %rcx
mov    %rdi,%rbx
vpcmpestri $0xd,(%rbx),%xmm2
jb     0x00007fe97d20e01d
sub    $0x8,%edx
jle    0x00007fe97d20e023
cmp    %eax,%edx
js     0x00007fe97d20e023
add    $0x10,%rbx
cmp    $0x8,%edx
jge    0x00007fe97d20dff7
lea    -0x10(%rbx,%rdx,2),%rbx
mov    $0x8,%edx
jmp    0x00007fe97d20dff7
sub    %ecx,%edx
cmp    %eax,%edx
jge    0x00007fe97d20e02a
mov    $0xffffffff,%ebx
jmp    0x00007fe97d20e039
lea    (%rbx,%rcx,2),%rbx
```

```
cmp    $0x7,%ecx
jg     0x00007fe97d20e00c
sub    %rdi,%rbx
shr    $0x1,%ebx
pop    %rsp
mov    %r10,%rdi
mov    %r11d,%r10d
mov    %r9,%rcx
vmovq  %xmm0,%r11
vmovq  %xmm1,%r9
cmp    $0xffffffffffffff,%ebx
jle    0x00007fe97d20e4c9
mov    %r14,0x70(%rsp)
mov    %rdi,0x60(%rsp)
mov    %r13,(%rsp)
mov    %rcx,%rdx
mov    %r11,%rcx
mov    %rbp,%rsi
shl    $0x3,%rsi
mov    $0x718fac3b0,%rbp
callq  0x00007fe97d046160
mov    0x60(%rsp),%rdi
movzbl 0x94(%rdi),%r11d
mov    (%rsp),%r13
add    $0x1,%r13
mov    $0x718fac3b0,%r10
test   %eax,0x16f45f5b(%rip)
test   %r11d,%r11d
jne    0x00007fe97d20e360
mov    0x70(%rsp),%r14
mov    0x18(%r14),%r10d
mov    0x8(%r12,%r10,8),%r11d
mov    $0x71a3f8478,%r8
mov    0x18(%r8),%r8d
shl    $0x3,%r11
mov    0x68(%r11),%r11
test   %r8d,%r8d
je     0x00007fe97d20e451
mov    %r8,%r9
```

```
shl    $0x3,%r9
cmp    %r11,%r9
jne    0x00007fe97d20e48d
mov    0x60(%r15),%rax
mov    $0x71a3f8478,%r11
mov    0x20(%r11),%r11d
mov    %r11d,0xc(%rsp)
mov    %rax,%r11
add    $0x10,%r11
lea    (%r12,%r10,8),%rcx
cmp    0x70(%r15),%r11
jae    0x00007fe97d20e29a
mov    %r11,0x60(%r15)
prefetchw 0xc0(%r11)
movq   $0x1,(%rax)
prefetchw 0x100(%r11)
movl   $0xf8002213,0x8(%rax)
prefetchw 0x140(%r11)
mov    %r12d,0xc(%rax)
prefetchw 0x180(%r11)
mov    %rax,%r11
mov    0xc(%rsp),%ebp
test   %r8d,%r8d
je     0x00007fe97d20e2ed
mov    0xc(%rsp),%ebp
mov    0x50(%r12,%rbp,8),%r8d
cmp    $0xf80088c5,%r9d
jne    0x00007fe97d20e3e6
shl    $0x3,%r8
mov    0xc(%r8),%ebx
cmp    $0xf8008883,%r9d
jne    0x00007fe97d20e41d
mov    $0x718f869a0,%r8
mov    0x74(%r8),%r8d
lea    (%r12,%rbx,8),%r9
mov    0xc(%r9),%ebx
mov    0x10(%r9),%ebp
inc    %ebx
mov    %ebx,0xc(%r9)
```

```
cmp     %r8d,%ebx
jle     0x00007fe97d20e281
mov     0x28(%r12,%rbp,8),%r8d
mov     %r8d,0xc(%rsp)
mov     0x14(%r12,%r8,8),%r8d
test    %r8d,%r8d
jne     0x00007fe97d20df80
```