Анализ и оптимизация запаковок переменных примитивного типа в Kotlin/Native

Александр Кузнецов
Разработка программного обеспечения / Software Engineering
Факультет информационных технологий и программирования
Университет ИТМО
Санкт-Петербург, Россия
Email: alexvangogen@gmail.com

***Аннотация*—Прямая трансляция примитивных типов языка Kotlin в примитивные типы LLVM IR — целевого промежуточного представления бэкенда Kotlin/Native — не всегда возможна. В таких случаях компилятору требуется сгенерировать операции запаковки для получения переменных типов-оберток из переменных примитивных типов или операции распаковки для обратного преобразования, что сказывается на производительности кода и потреблении памяти. В данной статье рассматриваются источники запаковок в Kotlin/Native и предлагаются способы их возможного устранения. Полученные результаты свидетельствуют об эффективности реализованных оптимизаций.**

***Index Terms*—Примитивные типы, запаковка, специализация, статический анализ.**

I. Введение

Kotlin — современный язык программирования, одной из ключевых особенностей которого является мультиплатформенность — возможность компилировать исходный код в разные представления, специфичные для разных платформ. Первая версия языка, Kotlin 1.0, была представлена в феврале 2016 года и поддерживала только компиляцию в байткод виртуальной машины Java. В настоящее время компилятор Kotlin имеет три бэкенда и, помимо генерации JVM-байткода, поддерживает транспиляцию в JavaScript, а также генерацию машинного кода.

За последнее отвечает бэкенд, именуемый Kotlin/Native[1]. Он компилирует Kotlin IR — промежуточное представление, генерируемое фронтендом компилятора Kotlin, в LLVM IR — платформонезависимое промежуточное представление, которое, в свою очередь, уже с помощью инфраструктуры LLVM можно скомпилировать в машинный код под любую архитектуру.

В LLVM IR, как и в JVM-байткоде, существует поддержка примитивных типов, например, тип *i32* представляет 32-битное целое число, а тип *double* — 64-битное число с плавающей точкой. В самом Kotlin нет разделения на примитивные типы и типы-обертки: каждый тип представляется в виде класса, и поэтому, например, возможно вызывать методы из целых чисел:

```
1 val x = 18.plus(24)    // x == 42
```

Однако компилятор не всегда может представить значение примитивного типа в Kotlin в виде значения примитивного типа в целевом представлении. В таких случаях он генерирует операцию запаковки — операцию исполнения некоторой функции, которая принимает значение примитивного типа и возвращает экземпляр класса-обертки. Например, для запаковки переменной типа *Int* Kotlin/JVM генерирует инструкцию *invokestatic*, которая вызывает статическую функцию *Integer.valueOf* языка Java. Kotlin/Native же сгенерирует вызов специальной функции *<Int-box>*, которая вернет специальную структуру *ObjHeader*, являющуюся универсальным представлением классов из Kotlin. В данном случае структура содержит само число, а также run-time информацию о классе *Int*, такую как информация о суперклассе и полное имя пакета, в котором класс объявлен.

Зачастую требуется выполнить обратную операцию и из экземпляра класса-обертки получить значение примитивного типа. В этой ситуации компилятору требуется сгенерировать функцию распаковки данного экземпляра. В Kotlin/Native это обеспечивается за счет вызова функции *<T-unbox>*, где *T* — название примитивного типа.

Таким образом, получение обертки над значением примитивного типа требует одного или двух вызовов дополнительных функций, а также выделения памяти в куче для хранения экземпляра класса-обертки, что, в свою очередь, увеличивает нагрузку на сборщик мусора. Еще одним неприятным фактом является то, что в целевом представлении операции над примитивными типами будут представлены как вызовы методов классов-оберток, а это дополнительный уровень косвенности и дополнительные затраты на поиск в таблице методов.

Если рассматривать оптимизирующую компиляцию, выполняемую со стороны LLVM, то даже в этом случае в скомпилированном коде остаются операции выделе-

[1]https://kotlinlang.org/docs/reference/native-overview.html

ния памяти под объект-обертку и получения значения из обертки, что мешает проводить ряд некоторых других оптимизаций, например, склеивание избыточных инструкций[2].

Данная работа посвящена тому, как компилятор Kotlin/Native может уменьшить количество генерируемых запаковок. В разделе II рассказывается о том, каким образом осуществляется подсчет количества запаковок, и какой код привносит излишние запаковки. В разделах III-V описываются причины излишних запаковок, формулируются подходы к решению. В разделе VI обозреваются существующие решения. Раздел VII описывает реализацию предложенных оптимизаций, в разделе VIII приводится оценка их эффективности.

II. Анализ количества запаковок

Для измерения точного количества запаковок, совершенных функцией во время ее выполнения, в компилятор был добавлен дополнительный опциональный этап понижений (lowerings). Суть его заключается в добавлении глобального счетчика типа *Int* с общим связыванием[3]. В начало каждой функции, выполняющей запаковку, добавляется вызов функции *Int.inc*, увеличивающей значение счетчика.

Подсчет запаковок осуществляется только для функций, имеющих аннотацию *@CountBoxings*. Тело аннотированной функции оборачивается в блок try-finally, перед началом блока добавляется инструкция обнуления счетика, а в тело finally записывается вызов функции *println*, печатающей значение счетчика в стандартный поток вывода.

Анализ осуществляется на существующих в компиляторе Kotlin/Native наборах тестов производительности. Были найдены следующие источники запаковок:

1) Передача значений примитивных типов в обобщенные функции и методы обобщенных классов, в том числе и передача в лямбда-функции;
2) Проверка на *null* значений примитивных типов, не являющихся nullable-типами.

При этом в Kotlin/Native уже существует ряд оптимизаций, направленных на снижение количества запаковок, например, наличие кеша (cache) для оберток над небольшими числами и преобразование методов *equals*, *hashCode*, *toString* классов-оберток в функции, принимающие вместо оберток значения примитивных типов.

III. Передача значений примитивных типов в обобщенные функции и классы

Язык Kotlin поддерживает концепцию обобщенных (generic) классов и функций. Например, следующая функция может принимать значения любого типа, реализующего интерфейс *Comparable*:

```
fun <T : Comparable<T>> min(a: T, b: T): T {
    return if (a < b) a else b
}
```

Однако обобщения не поддерживаются в LLVM IR, поэтому компилятор Kotlin/Native производит стирание типа. В результате параметры функции, как и возвращаемое значение, становятся указателями на *ObjHeader*. Ввиду того, что тело функции в LLVM IR получается слишком громоздким, ниже приведено его словесное описание:

1) Вычисляется адрес реализации метода *Comparable.compareTo* для типа, указанного в структуре *ObjHeader*, соответствующей первому параметру функции;
2) Найденный метод вызывается с аргументами, соответствующими параметрам функции;
3) Как и в Java, результат сравнения *compareTo* выражается в знаке возвращаемого целочисленного значения. В зависимости от этого знака, функция возвращает либо первый, либо второй параметр.

При передаче в функцию *min* двух целочисленных значений они будут запакованы в тип-обертку, так как функция не может работать с примитивными типами.

Теперь рассмотрим следующую функцию:

```
fun minInt(a: Int, b: Int): Int {
    return if (a < b) a else b
}
```

Она отличается от предыдущей только тем, что может принимать только аргументы типа *Int*. Так как данной функции уже не требуется стирание типов, Kotlin/Native генерирует довольно прямолинейное представление:

```
1  entry:
2    %6 = load i32, i32* %p-a
3    %7 = load i32, i32* %p-b
4    %8 = icmp slt i32 %6, %7
5    br i1 %8, label %when_case,
6              label %when_next
7
8  when_case:
9    %9 = load i32, i32* %p-a
10   br label %when_exit
11
12 when_exit:
13   %10 = phi i32 [ %9, %when_case ],
14                 [ %11, %when_next ]
15   br label %epilogue
16
17 when_next:
18   %11 = load i32, i32* %p-b
19   br label %when_exit
20
21 epilogue:
22   %12 = phi i32 [ %10, %when_exit ]
23   ret i32 %12
```

Буквально, два числа сравниваются при помощи встроенной инструкции *icmp* (строка 4), и, в зависимости от результата сравнения (phi-инструкция, строка 13), возвращается одно из этих чисел (строка 23).

Функцию *minInt* можно получить из функции *min* автоматически во время компиляции: если компилятор

[2]https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions

[3]https://llvm.org/docs/LangRef.html

встречает вызов вида *min(x, y)*, где x и y имеют тип *Int*, то он может создать копию функции *min*, изменить имя копии на *minInt*, удалить в ней параметр T и заменить все его вхождения на конкретный тип *Int*, после чего изменить сам вызов функции *min* на вызов функции *minInt*; иными словами, специализировать функцию *min* для типа *Int*.

Подобные трансформации можно осуществлять и для классов. В следующем примере класс *MyPairIntDouble* является специализацией класса *MyPair* для типов *Int* и *Double*, и создание его экземпляра не требует никаких запаковок:

```
class MyPair<S, T>(
    val first: S,
    val second: T
)

class MyPairIntDouble(
    val first: Int,
    val second: Double
)
```

Для некоторых классов в Kotlin уже существуют специализированные версии. Например, вместе с обобщенным классом *Array<T>* реализованы дополнительные версии для примитивных типов: *IntArray*, *LongArray* и так далее.

IV. Повторные запаковки и их удаление

Одна и та же переменная может запаковываться несколько раз. Подобная ситуация может произойти не только по вине программиста, но также после других оптимизаций, таких как специализация, описанная выше, или встраивание (inlining) функций.

Для примера рассмотрим следующие две функции:

```
fun <T> putIfAbsent(list: MutableList<T>, x: T) {
    if (x in list) return
    list += x
}

fun putIfAbsentInt(list: MutableList<Int>, x: Int) {
    if (x in list) return
    list += x
}
```

Заметим, что функция *putifAbsentInt* является специализацией функции *put*, использующей вместо типового параметра T конкретный тип *Int*. Однако в функцию *putIfAbsent* переменная сразу передается в запакованном состоянии, в то время как в теле *putIfAbsentInt* параметр x будет запакован дважды, в строках 7 и 8.

Уменьшить количество запаковок можно, если создать временную переменную и записать в нее запакованное значение переменной x, после чего использовать эту временную переменную вместо x, где это необходимо. При этом функция принимает следующий вид:

```
fun putIfAbsentInt(list: MutableList<Int>, x: Int) {
    val xBoxed = <Int-box>(x)
    if (xBoxed in list) return
    list += xBoxed
}
```

Данная задача является частным случаем forward-анализа доступных выражений (Available Expressions) [1][2]. Для каждой вершины графа потока управления этот анализ определяет, какие подвыражения в данной вершине гарантированно уже были вычислены ранее в коде. Эта информация позволяет не пересчитывать несколько раз одно и то же выражение, а записать значение выражения в отдельную переменную и использовать ее вместо повторного вычисления.

Результат анализа для вершины v — набор доступных выражений — рассчитывается по формуле

$$[\![v]\!] = \bigcap_{u \in pred(v)} \Big(\big([\![u]\!] \cup gen(u) \big) \setminus kill(u) \Big),$$

где $gen(u)$ — выражения, созданные в вершине u, $kill(u)$ — выражения, изменившиеся в u и переставшие быть доступными.

То есть, чтобы получить доступные выражения для вершины v, нужно для каждой вершины-предшественника u:

1) рекурсивно вычислить доступные выражения в u;
2) добавить к доступным выражениям те, что создаются в u;
3) убрать из доступных выражений те, что содержат переменные, изменяющиеся в u,

после чего оставить только те выражения, которые доступны для каждой вершины u.

Данный анализ не считает выражения, вычисляемые внутри цикла, доступными для вершин вне этого цикла, так как он не может гарантировать, что количество итераций цикла всегда строго положительное и что при каждом исполнении программы его тело будет исполнено хотя бы один раз. Как следствие, запаковку одной и той же переменной в теле цикла не получится вынести за его пределы. Для этого нужно проводить дополнительные оптимизации, такие как вынос инварианта за пределы цикла (Loop Invariant Code Motion) [1].

V. Избыточные проверки на null

В языке Kotlin существует так называемый elvis-оператор. Он бинарный и возвращает левый операнд, если тот не равен *null*, и правый операнд в противном случае. Так как значения примитивных типов не могут равняться *null*, компилятор предварительно их запаковывает. Однако в ряде случаев эта процедура избыточна. Так, в следующем примере функции *f1*, *f2* и *f3* порождают лишнюю запаковку и распаковку:

```
fun f1(x: Int) = x ?: 0

inline fun Int?.orZero() = this ?: 0
fun f2(x: Int) = x.orZero()

class Value(val x: Int)
fun f3(value: Value?) = value?.x ?: 0
```

Во всех трех случаях компилятор запаковывает выражение в левой части операнда, так как работает в

предположении, что левая часть должна иметь тип *Int?*. Сам же оператор возвращает значение типа *Int*, что порождает распаковку.

В функции *f1* избыточна сама проверка, так как левый операнд по типу никогда не может быть равен *null*. Аналогичная ситуация для *f2*: при встраивании тела функции *orZero* тип приемника (receiver) уточняется с *Int?* до *Int*. В случае с *f3* проверка необходима, но, тем не менее, можно обойтись без запаковки. Например, следующая функция имеет такую же семантику, что и *f3*:

```
fun better_f3(value: Value?) = if (value != null) value.x else 0
```

Функция *better_f3* не требует никаких запаковок, так как в ней нигде не встречается тип *Int?*. Однако компилятор представит функцию *f3* в следующем виде:

```
fun lowered_f3(value: Value?): Int {
    val tmp: Int? = if (value == null) null else value.x
    return if (tmp == null) 0 else tmp
}
```

Стоит отметить, что оптимизации LLVM удаляют проверки на *null* в данных примерах, однако операции запаковок и распаковок остаются.

VI. Обзор существующих решений

Существует несколько способов обеспечения автоматической специализации функций и классов. Один из них — создание специализированной версии «по необходимости», то есть, только в том случае, если компилятор обнаружил вызов обобщенной функции или создание экземпляра обобщенного класса с примитивными типовыми аргументами. Так работает, например, инстанцирование шаблонов в C++ и мономорфизация обобщений в Rust. Подобная стратегия предполагается и при специализации классов и интерфейсов в языке Java, реализация которой находится на стадии разработки как одна из частей Project Valhalla[4]. В .NET CLR загрузка специализированных объявлений осуществляется на стадии связывания (linking) [3].

Другой способ — дать возможность явно указать, для каких классов и функций, каких типовых параметров и для каких конкретных примитивных типов компилятор должен сгенерировать специализацию. Подобная функциональность была реализована в языке Scala в версии 2.8 [4]. Типовой параметр может быть проаннотирован специальной аннотацией *@specialized*, которая указывает, что компилятор должен сгенерировать по копии класса на каждый примитивный тип, существующий в языке. Аннотация может принимать параметры, являющиеся классами примитивных типов; в этом случае компилятор сгенерирует специализации только для тех типов, что присутствуют среди параметров. При этом компилятор создает специализации, основываясь на аннотации, а не на реальной необходимости. Явные указания компилятору сгенерировать специализации функций также поддерживаются в языке Swift[5].

Заметим, что оба способа могут потенциально повлиять на создание неиспользуемых копий. В случае с созданием специализации «по необходимости» специализация может потребоваться внутри неиспользуемой функции. В случае явной специализации можно сгенерировать неиспользуемый класс или функцию.

Другие оптимизации, такие как удаление проверок на *null*, для JVM-языков описаны в [5], [6]. В [5] описан ряд оптимизаций, позволяющих убрать лишние запаковки, на стороне just-in-time компилятора JVM, что не совсем актуально для языка с другой инфраструктурой бэкенда. В [6] описаны более высокоуровневые подходы к оптимизации байт-кода, генерируемого компилятором Kotlin/JVM. В частности, предложен алгоритм удаления лишних проверок при проверке на *null* с использованием elvis-оператора.

VII. Реализация

A. Специализация классов и функций

Для реализации был выбран вариант явного аннотирования классов и функций, которые требуется специализировать. Была реализована аннотация *@Specialized* для типовых параметров, принимающая массив классов примитивных типов, для которых нужно сгенерировать специализированные версии. Однако на данный момент невозможно аннотировать типовые параметры класса, поэтому вместе со *@Specialized* добавлена аннотация для классов *@SpecializedClass*.

В компилятор были добавлены два дополнительных этапа понижений. На первом этапе происходит непосредственно генерация новых функций и классов, за что отвечают две сущности — специализатор и элиминатор типовых параметров. Специализатор обходит промежуточное представление кода и для текущей вложенности объявлений запоминает, какие типовые параметры заменены на конкретные примитивные типы и что это за примитивные типы. Таким образом, во время обхода поддерживаются две структуры: отображение $\mathbb{M}$ вида «типовой параметр — аргумент, являющийся конкретным примитивным типом» и список $\mathbb{L}$ последовательных конкретных примитивных типов. Когда специализатор обнаруживает, что данное объявление (класс или функцию) необходимо специализировать, он передает это объявление, а также отображение $\mathbb{M}$ и список $\mathbb{L}$ в качестве параметров элиминатору. Элиминатор копирует это объявление, заменяя типовые параметры на конкретные типы согласно отображению $\mathbb{M}$. Типы из $\mathbb{L}$ в элиминаторе используются для кодирования имени специализированной версии, что позволяет избежать конфликтов имен.

Полученная специализированная версия передается специализатору, который осуществляет обход этого объявления для рекурсивного поиска вложенных специализированных объявлений, после чего добавляет

[4]Brian Goetz. State of the Specialization. Url: https://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html

[5]https://github.com/apple/swift/blob/master/docs/Generics.rst

его в промежуточное представление текущего файла, а также сохраняет тройку (оригинальное объявление, список $\mathbb{L}$, специализированная версия) в структуре $\mathbb{T}$. Сама структура представляет собой отображение и отвечает на следующие вопросы:

- Какие специализации существуют для данного объявления?
- Какое объявление является специализацией данного объявления для данных конкретных типов?

Также в структуру $\mathbb{T}$ записываются уже существующие в языке специализации, например, тройка (*Array*, [*Int*], *IntArray*).

Рассмотрим работу первого этапа на следующем примере:

```
1 @SpecializedClass(forTypes = [Int::class])
2 class A<S> {
3     @SpecializedClass(forTypes = [Float::↩
          class])
4     class B<T>
5 }
```

Предположим, что это весь код, содержащийся в файле, и что структуры $\mathbb{M}$ и $\mathbb{L}$ пусты.

1) Класс A нужно специализировать. Сам класс, а также структуры $\mathbb{M} = \{S \to Int\}$ и $\mathbb{L} = [Int]$ передаются элиминатору.
2) Элиминатор создает класс *A-Int* без типовых параметров и возвращает его специализатору;
3) Во время рекурсивного обхода класса *A-Int* обнаруживается, что необходимо специализировать вложенный класс B. В элиминатор передается B, а также структуры $\mathbb{M} = \{S \to Int, T \to Float\}$ и $\mathbb{L} = [Int, Float]$;
4) Элиминатор создает внутри класса *A-Int* класс *B-Int-Float* без типовых параметров;
5) После посещения класса *B-Int-Float* специализатор добавляет этот класс в *A-Int* и запоминает тройку (B, [*Int*, *Float*], *B-Int-Float*);
6) После посещения класса *A-Int* специализатор добавляет этот класс в промежуточное представление файла и запоминает тройку (A, [*Int*], *A-Int*);
7) Специализатор продолжает посещение обобщенного класса A. Важно заметить, что структура $\mathbb{M}$ на данном шаге пуста, а $\mathbb{L} = [null]$;
8) Во время обхода класса A посещается вложенный класс B. В элиминатор передается B, а также структуры $\mathbb{M} = \{T \to Float\}$ и $\mathbb{L} = [null, Float]$;
9) Элиминатор создает класс *B-Float* без типовых параметров;
10) После посещения класса *B-Float* специализатор добавляет этот класс в A и запоминает тройку (B, [*null*, *Float*], *B-Float*); обход файла завершается.

На втором этапе происходит замена типов и выражений в соответствии с созданными специализациями. Для замены используются тройки из $\mathbb{T}$. Например, если где-то в коде встретилось выражение *val b = A<Int>.B<Float>()*, то компилятор определяет, что создается экземпляр класса B, и что типовые аргументы образуют последовательность [*Int*, *Float*]. Согласно структуре $\mathbb{T}$, данному классу и данной последовательности соответствует класс *B-Int-Float*. Также в $\mathbb{T}$ присутствует тройка (A, [*Int*], *A-Int*), что позволяет заменить обращение к классу *A<Int>*. В итоге выражение принимает вид *val b = A-Int.B-Int-Float()*.

B. Удаление избыточных запаковок

Данный алгоритм был реализован только для параметров функции и неизменяемых переменных. Анализ реализован в виде понижения компилятора и разбивается на два этапа. На первом этапе происходит непосредственно сбор информации о доступных переменных. После этого для каждой области видимости в программе известно множество переменных, которые являются доступными внутри них.

На втором этапе происходит создание оберток. Если посещаемое объявление переменной или параметр функции является доступным в данной области видимости, то обертка помещается на место следующей инструкции. То есть, объявления новых переменных помещаются сразу после объявлений соответствующих им переменных примитивных типов, а обертки над параметрами функции помещаются в начало функции. Если посещаемый вызов функции запаковки принимает в качестве аргумента переменную x, для которой была сгенерирована обертка *x-Boxed*, то этот вызов заменяется на получение *x-Boxed*. Если для x не существует обертки, но переменная является доступной в данной области видимости (это возможно, например, когда переменная была объявлена без инициализатора и активно используется лишь в теле одной ветки), то инструкция создания обертки помещается перед вызовом функции запаковки, после чего вызов заменяется на получение этой самой обертки.

C. Удаление проверок на null

После этапа встраивания функций каждый вызов elvis-оператора проверяется на то, имеет ли его левая часть nullable-тип, и если не имеет, то весь вызов заменяется на возвращение его левого операнда. Так, тело функции *f1* заменяется на возвращение переменной x.

Трансформация вызова elvis-оператора, у которого в левой части находится цепочка безопасных вызовов (safe call chain), как в функции *lowered_f3* из раздела V, использует идеи из [6]. Ее можно описать следующим алгоритмом.

Обозначим как $[tmp, a, b, c]$ выражение

```
1 val tmp = if (a == null) b else c
```

Тогда $[tmp2, [tmp, a, null, c], b, tmp]$ эквивалентно $[tmp2, a, b, c]$. Таким образом, собрав во время прохода по elvis-оператору соответствующие значения a, b и c, можно получить новую, сокращенную версию проверки на *null*. В частности, в функции *lowered_f3* a соответствует параметру *value*, b соответствует числу 0, а c соответствует получению свойства $value.x$. Можно заметить, что $[tmp, value, 0, value.x]$

раскрывается в выражение, эквивалентное телу функции *better_f3* из раздела V.

VIII. Эксперименты

Тестирование производительности при необходимости осуществляется на псевдослучайных целых числах от 0 до 10^9 и на псевдослучайных числах с плавающей точкой от 0.0 до 1000000.0. Каждый из этих тестов включает в себя 1000000 итераций тестируемых действий. Компиляция тестов производительности осуществляется с флагом *-opt*, который включает оптимизации компилятора Kotlin/Native и набор оптимизаций *-O3* на стороне LLVM. Для тестирования используется существующий в Kotlin/Native модуль измерения производительности. Подсчет запаковок производится отдельно, так как инструментирование функций может повлиять на цикл оптимизаций и исказить результаты.

Результаты измерения эффективности построения специализаций приведены в Таблице I. В тестах *Matrix2* и *Matrix3* осуществляется создание обобщенных матриц размером 2×2 и 3×3, параметризованных целыми числами, и вычисление сумм их элементов. В тесте *ListMin* осуществляется создание обобщенного списка на базе обобщенного массива *Array<T>*, параметризованного целыми числами, и применение к нему обобщенной функции высшего порядка *reduce*, которая в данном случае вычисляет минимум. В тесте *SatisfiesAll* осуществляется последовательное применение предикатов делимости на 2, 3 и 5 к набору чисел. Во всех этих тестах обобщенные объявления аннотированы *@Specialized/@SpecializedClass* и предоставляют специализацию для типа *Int*.

Стоит поподробнее остановиться на тестах *Mapper1*, *Mapper2* и *Mapper3*. Каждый тест включает в себя обобщенный функциональный интерфейс *Mapper<T>*, предоставляющий функцию *map: (T)* $\rightarrow$ *T*, и три обобщенных класса-реализации: два простых и один составной, *CompositeMapper*, принимающий две произвольные реализации *Mapper* (*mapperFirst* и *mapperSecond*) и осуществляющий их композицию:

```
fun map(value: T) = secondMapper.map(↩
    firstMapper.map(value))
```

Суть тестов заключается в применении функций описанных классов к различным числам. В тесте *Mapper1* все четыре объявления предоставляют специализацию для типа *Int*, и функции применяются к целым числам, поэтому при включенной специализации запаковки не генерируются. Тест *Mapper2* отличается от *Mapper1* тем, что функции применяются к значениям типа *Double*, но, так как специализация для типа *Double* не предоставляется, вызываются обобщенные версии классов, и компилятор генерирует запаковки.

Тест *Mapper3* также похож на *Mapper1*, но в нем класс *CompositeMapper* предоставляет специализацию только для типа *Double*, а остальные классы предоставляют специализацию только для типа *Int*. Если

Таблица I: Оценка эффективности построения специализаций

Тест	**Без спец-ии**	**Со спец-ей**	**Ускорение, раз**
Matrix2	961 $\pm 5.6ms$	189 $\pm 1.3ms$	5.1
Matrix3	1976 $\pm 11.9ms$	241 $\pm 0.9ms$	8.2
ListMin	2261 $\pm 23.8ms$	572 $\pm 5.8ms$	3.95
SatisfiesAll	609 $\pm 11.2ms$	249 $\pm 1.4ms$	2.45
Mapper1	884 $\pm 1.9ms$	505 $\pm 2.8ms$	1.75
Mapper2	901 $\pm 11.3ms$	888 $\pm 2.4ms$	n/a
Mapper3	894 $\pm 6.1ms$	1017 $\pm 3.0ms$	0.88
Время компиляции	35.5 s	38.9 s	+9.6%
Размер кода	10367.58 Kb	10363.04 Kb	-0.04%

Тест	**Запаковок без спец-ии**	**Запаковок со спец-ей**
Matrix2	4000000	0
Matrix3	9000000	0
ListMin	9000000	0
SatisfiesAll	3333332	0
Mapper1	2000000	0
Mapper2	2000000	2000000
Mapper3	2000000	3000000

явно выделить запаковки и распаковки, то исполнение функции *CompositeMapper.map* будет иметь вид:

```
fun map(value: T) = box<
    secondMapper.map(
        unbox<box<
            firstMapper.map(unbox<value>)
        >>
    )
>
```

Таким образом, вызов функции *CompositeMapper.map* требует одну запаковку аргумента и две запаковки значений внутри самой функции. При этом все запаковки различны, и алгоритм удаления повторных запаковок не уменьшит их количество.

Заметим, что размер кода немного уменьшился при компилировании со специализацией. Это вызвано тем, что и на стороне Kotlin/Native, и на стороне LLVM производится операция удаления мертвого кода (Dead Code Elimination, DCE). При включенной специализации тесты производительности используют специализированные версии классов и функций, поэтому обобщенные версии удаляются, а размеры специализированных версий меньше, чем размеры оригиналов.

В Таблице II приведен результат проверки эффективности удаления повторных запаковок для функции *putIfAbsent* из раздела IV. Результат показывает, что в некоторых случаях алгоритм действительно позволяет убрать излишние запаковки, привнесенные специализацией.

В Таблице III приведен результат проверки эффек-

Таблица II: Оценка эффективности применения оптимизаций для функции *putIfAbsent*

	Время работы	**Кол-во запаковок**
Без оптимизаций	2342 $\pm 16.8ms$	10000
Со специализацией	2917 $\pm 21.4ms$	20000
Со специализацией + удалением повторных запаковок	2389 $\pm 17.1ms$	10000

Таблица III: Оценка эффективности удаления избыточных проверок на *null*

Тест	**Без удаления**	**С удалением**	**Ускорение, раз**
Elvis1	336 $\pm 10.5ms$	139 $\pm 2.1ms$	2.42
Elvis2	354 $\pm 10.2ms$	142 $\pm 3.1ms$	2.49
Elvis3	344 $\pm 12.5ms$	145 $\pm 2.1ms$	2.37
Размер кода	10390.57 Kb	10386.57 Kb	-0.04%

Тест	**Запаковок без удаления**	**Запаковок с удалением**
Elvis1	1000000	0
Elvis2	1000000	0
Elvis3	870574	0

тивности удаления лишних проверок на *null*. Тесты *Elvis1*, *Elvis2* и *Elvis3* соответствуют функциям *f1*, *f2* и *f3* из раздела V. На некоторых итерациях теста *Elvis3* при вызове функции ей передается значение, равное *null*. В таких случаях запаковка не совершается, поэтому в данном тесте количество запаковок изначально меньше, чем в двух других.

IX. Заключение

В ходе работы была реализована метрика, позволяющая вычислить точное количество запаковок, осущественных во время исполнения тестовых функций. С помощью данной метрики было найдено несколько источников излишних запаковок, возникающих в тестах производительности, рассмотрены возможные способы их устранения. Реализованы следующие оптимизации: специализация классов и функций по явным пользовательским аннотациям, поиск и удаление повторных запаковок неизменяемых переменных, удаление избыточных запаковок при проверке на *null*. Выполнено измерение производительности набора программ до оптимизаций и после. Поддержана трансляция обобщенных классов и функций в существующие специализированные версии во время специализации. Реализованы некоторые специализации для функциональных интерфейсов.

При проведении экспериментов было установлено, что специализация может увеличить количество запаковок, и среди них не будет повторных (как в примере с тестом *Mapper3*). Проверка гипотезы, что такие ситуации преимущественно связаны с использованием специализированных классов и функций в неспециализированных обобщенных объявлениях, и исследование способов решения этой проблемы является интересным направлением дальнейших исследований. В частности, можно рассмотреть вариант неявной специализации подобных обобщенных объявлений и вариант поддержки подобного кода со стороны фронтенда компилятора и IDE в виде предупреждений и инспекций.

Сопутствующей работой является расширение количества специализируемых классов и функций, проверка компилируемости и сохранения семантики на все более сложных конструкциях, а также специализация коллекций стандартной библиотеки.

Второе направление исследований заключается в улучшении алгоритма удаления повторных запаковок, а именно поддержке циклов и изменяемых переменных.

Наконец, некоторые предложенные оптимизации, такие как специализация классов и функций, не являются специфичными для Kotlin/Native, и их можно реализовать и для других бэкендов.

Ссылка на репозиторий: https://github.com/gascogen/kotlin-native/tree/opt-boxing.

Список литературы

[1] Alfred V. Aho, Monica S. Lam, Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools, Second Edition.* Addison-Wesley, 2007.

[2] Flemming Nielson, Hanne R. Nielson, Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, 1999.

[3] Andrew Kennedy, Don Syme. *Design and implementation of generics for the .NET Common language runtime.* ACM SIGPLAN Notices, Vol.36, No.5. - 2001. - С. 1-12.

[4] Iulian Dragos, Martin Odersky. *Compiling generics through user-directed type specialization.* Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems. - 2009. - С. 42-47.

[5] Yuji Chiba. *Redundant Boxing Elimination by a Dynamic Compiler for Java.* Proceedings of the 5th international symposium on Principles and practice of programming in Java. – 2007. – С. 215-220.

[6] Д. Жарков. *Оптимизация байткода, генерируемого компилятором Kotlin.* 2015. Презентация: https://wiki.compscicenter.ru/images/a/a6/Zharkov_master2015.pdf.

An analysis and optimization of primitive type boxings in Kotlin/Native

Alexander Kuznetsov
ITMO University
Saint-Petersburg, Russia

***Abstract*—Direct translation from primitive types in Kotlin to primitive types in LLVM IR — targeted intermediate representation of Kotlin/Native backend — is not always possible. In these cases compiler must generate box operations to get wrapped variables from variables of primitive types, or unbox operations to perform an inverse tranformation, which affects code performance and memory consumption. In this paper we will review some sources of boxings performed by Kotlin/Native and suggest several ways of their possible elimination. Obtained results indicate the efficiency of implemented optimizations.**

***Index Terms*—primitive types, boxing, specialization, static analysis.**