DOI: 10.15514/ISPRAS-2021-33(6)-5

Kotlin с точки зрения разработчика статического анализатора

[1,2] *В.О. Афанасьев, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *С.А. Поляков, ORCID: 0000-0002-8542-8035 <inly@ispras.ru>*
[1] *А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Национальный исследовательский университет Высшая школа экономики, 101000, Россия, г. Москва, ул. Мясницкая, д. 20*
[3] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. В статье описывается статический анализатор для поиска ошибок и анализа метрик и отношений в программах на языке Kotlin. Анализатор был реализован с помощью расширения инструмента *Svace*, разрабатываемого в ИСП РАН. В статье описываются проблемы, с которыми мы столкнулись в ходе выполнения работы, и предложенные методы их решения, а также экспериментальные результаты полученного анализатора. Инструмент умеет не только анализировать программы на языке Kotlin, но также поддерживает анализ смешанных проектов, использующих языки Java и Kotlin. Надеемся, что статья будет полезна разработчикам статических анализаторов, а также тем, кто проектирует новые языки программирования.

Ключевые слова: статический анализ; поиск ошибок; анализ метрик; уязвимости; Kotlin; JVM; байткод

Для цитирования: Афанасьев В.О., Поляков С.А., Бородин А.Е., Белеванцев А.А. Kotlin с точки зрения разработчика статического анализатора. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 67-82. DOI: 10.15514/ISPRAS–2021–33(6)–5

Kotlin from the perspective of a static analyzer developer

[1,2] *V.O. Afanasyev, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *Polyakov S.A., ORCID: 0000-0002-8542-8035 <inly@ispras.ru>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1,3] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *National Research University Higher School of Economics, 20, Myasnitskaya Str., Moscow, 101000, Russian Federation*
[3] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*

Abstract. The paper describes a static analysis for finding defects and computing metrics for programs written in the Kotlin language. The analysis is implemented in the Svace static analyzer developed at ISP RAS. The paper focuses on the problems we met during implementation, the approaches we used to solve them, and the experimental results for the tool we have built. The analyzer supports not only Kotlin analysis, but is also

capable of analyzing mixed projects that use both Java and Kotlin languages. We hope that the paper might be useful to static analysis developers and language designers.

Keywords: static analysis; search for defects; vulnerabilities; Kotlin; JVM; bytecode

For citation: Afanasyev V.O., Polyakov S.A., Borodin A.E., Belevantsev A.A. Kotlin from the perspective of a static analyzer developer. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 67-82 (in Russian). DOI: 10.15514/ISPRAS–2020–33(6)–5

1. Введение

Kotlin – относительно молодой язык, разрабатываемый компанией JetBrains [1]. Язык является статически-типизированным и поддерживает парадигмы объектно-ориентированного и функционального программирования. При разработке языка особое внимание уделялось типобезопасности. Язык использует Java Virtual Machine. В мае 2017 года компания Google сообщила, что язык Kotlin будет стандартным языком для разработки ОС Android и приложений для неё наравне с языком Java [2].

Исходный код на языке Kotlin может быть скомпилирован в три различных варианта промежуточного представления: JVM-байткод [3] при использовании платформы Kotlin/JVM, код на JavaScript при использовании Kotlin/JS, LLVM-биткод при компиляции с использованием Kotlin/Native. При этом надо отметить, что один и тот же код может успешно компилироваться под одну платформу, но не под другую, если этот код использует платформо-зависимые библиотеки или API. Так, например, мобильное приложения для Android не может быть полностью собрано при помощи инструментария Kotlin/Native.

В данной работе мы рассматриваем только компиляцию для JVM. Отметим, что в данном случае код, написанный на языке Java, может вызываться из кода на языке Kotlin и наоборот.

Язык Kotlin спроектирован таким образом, чтобы исключить возможность возникновения многих ошибочных ситуаций в коде программ. В частности, для исключения ошибки «разыменование нулевого указателя» [4] система типов языка Kotlin поддерживает два вида типов: те, что могут принимать значение null (nullable references), и те, что не могут (non-nullable references). При этом разыменование значения null может произойти только в следующих случаях:

- явное небезопасное разыменование nullable-объекта при помощи операторов !!$ и *as*;
- передача объекта в какой-либо метод в процессе его конструирования до инициализации всех полей (leaking this);
- небезопасное взаимодействие с Java-кодом.

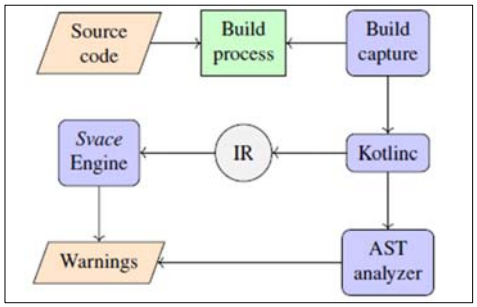

Рис. 1. Схема анализа
Fig. 1. Analysis scheme

Для языка доступно несколько видов легковесных анализаторов (линтеров): detekt [5] и ktlint [6]. Тем не менее, нам не известно о существующих статических анализаторах, выполняющих

глубокий межпроцедурный анализ. Мы решили восполнить этот пробел и добавить поддержку анализа языка Kotlin в инструмент статического анализа *Svace* [7, 8].

Среди языков, поддерживаемых инструментом *Svace*, находится язык Java. Анализатор использует байткод JVM как промежуточное представление для анализа. Поэтому реализация статического анализатора на основе байткода JVM для языка Kotlin представлялась несложной задачей. В данной статье мы опишем проблемы, с которыми столкнулись.

На рис. 1 показана схема анализа. Анализ можно разделить на два важных этапа: контролируемая сборка проекта с генерацией промежуточного представления программы и статический анализ получившегося представления. Эти этапы будут описаны в разд. 4 и 6 соответственно.

2. Перехват сборки

Преимуществом анализатора *Svace* является поддержка автоматического анализа кода: участие пользователя в нем сведено к минимуму. Для анализа проекта необходимо запустить утилиту перехвата сборки, подав на вход оригинальную команду сборки:

```
svace build make.
```

После этого запускается анализ с помощью команды:

```
svace analyze.
```

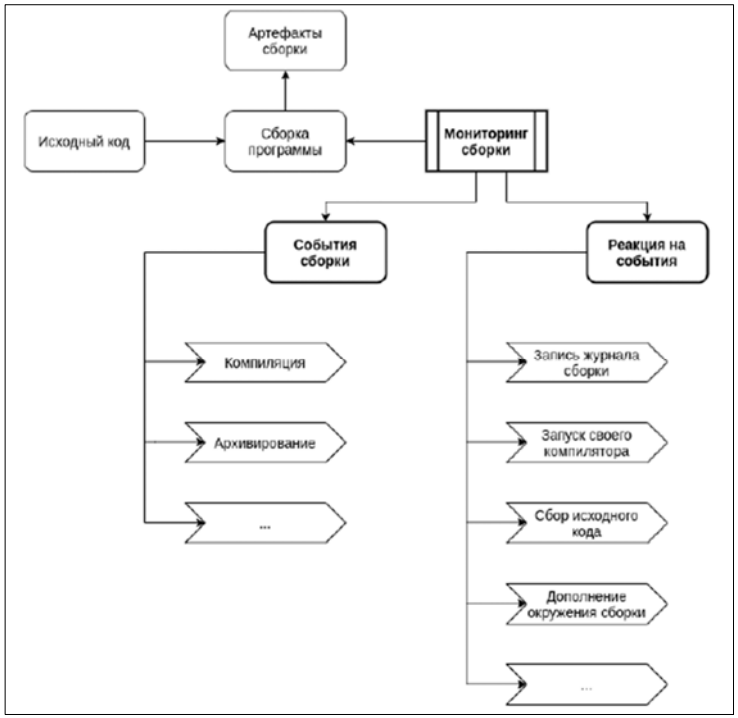

Рис. 2. Устройство контролируемой сборки
Fig. 2. Build capturing structure

Назовём *контролируемой сборкой* процесс запуска оригинальной сборки с отслеживанием интересующих нас процессов. Контролируемая сборка необходима для извлечения информации о том, как именно предполагалось компилировать тот или иной файл проекта.

В частности, для языка Kotlin необходимо правильно задать пути к JAR-библиотекам, пути к директориям с исходными файлами на языке Java, пути к плагинам компилятора, версию языка и т.п.

Svace проводит мониторинг процесса сборки без его искажения и отслеживает выполнение определенных действий – событий сборки: запуск JVM, команды компиляции и т.д. Также в процессе контролируемой сборки проводится первый этап трансляции исходного кода проекта в промежуточное представление *Svace*.

Устройство контролируемой сборки [9] в анализаторе *Svace* представлено на рис. 2.

На каждое событие сборки анализатор *Svace* реагирует специальным образом, собирая необходимую для анализа информацию. Для событий сборки проектов на языке Kotlin реализованы следующие реакции:

- добавление специального Java-агента [10] в параметры запуска виртуальной машины;
- запуск собственного компилятора Kotlin}
- сбор исходных кодов и библиотек.

Компилятор языка Kotlin представляет из себя JAR-библиотеку и вызывается через программный интерфейс. API компилятора используется такими инструментами автоматической сборки, как Gradle, Maven и Ant. Бинарный файл *kotlinc*, поставляемый вместе с компилятором, является bash-скриптом для запуска компилятора на ОС семейства Linux. Для запуска компилятора на ОС семейства Windows используется bat-скрипт *kotlinc.bat*. Оба скрипта используют API компилятора для его запуска. Таким образом, в процессе сборки любого проекта на языке Kotlin происходит не запуск компилятора в виде процесса, а вызов некоторого метода в виртуальной машине Java. Анализатор реагирует на запуск виртуальной машины Java (событие сборки) передачей виртуальной машине пути к библиотеке с Java-агентом для перехвата Kotlin-компиляций, выполняемых через программный интерфейс.

В общем случае Java-агент – это JAR-библиотека, которой виртуальная машина сообщает о загрузке какого-либо класса и позволяет изменить этот класс. В случае анализатора *Svace*, если переданный класс реализует интерфейс *org.jetbrains.kotlin.cli.common.CLITool*, то метод *exec* в этом классе будет проинструментирован таким образом, что при его вызове будет дополнительно вызван специальный метод *interceptKotlinc*, которому в качестве параметров будут переданы необходимые данные о компиляции. Данный метод запускает специальный *dummy*-процесс, параметрами которого являются все собранные данные о компиляции.

Основная сложность данной реакции заключается в том, что нет официального и задокументированного API компилятора.

Для построения промежуточного представления мы используем модифицированный компилятор Kotlin (собственный компилятор), который запускается в ответ на событие «запуск dummy-процесса». В модифицированном компиляторе отключены оптимизации, которые могут затруднить анализ, и реализовано сохранение дополнительной информации, о чём будет подробнее написано в разд. 3.

Язык Kotlin активно развивается и имеет множество версий. При этом обратная совместимость версий компилятора [11] поддерживается не в полном объеме.

На данный момент актуальной версией является версия 1.5.31.

Собственный компилятор Kotlin в *Svace* основан на версии 1.5.10. По этой причине, и поскольку пользовательский проект может использовать любую доступную версию компилятора, опции, с которыми был запущен оригинальный компилятор, необходимо переработать перед запуском собственного компилятора. Например, опция *–language-version*

1.2 запрещена для использования, начиная с компилятора версии 1.5.10. Данную опцию необходимо исключить из списка опций для запуска собственного компилятора.

Для компилятора Kotlin существует набор стандартных плагинов, а также пользователь может создавать собственные плагины. Например, стандартный плагин kapt [12] реализует процессор аннотаций. Версия плагина должна быть совместима с версией используемого компилятора. По этой причине при запуске собственного компилятора необходимо использовать собственные стандартные плагины. Таким образом, опцию *-Xplugin*, используемую для передачи путей к JAR-библиотекам с плагинами компилятора также необходимо переработать. Пути к известным стандартным плагинам должны быть заменены на пути к собственным плагинам компилятора из дистрибутива *Svace*. Если при сборке проекта используются нестандартные плагины компилятора, несовместимые с компилятором версии 1.5.10, успешный анализ таких проектов не гарантируется.

Файлы с исходным кодом используются для показа предупреждений анализа. Такие файлы могут быть удалены или перемещены в процессе сборки, поэтому сохранить их следует немедленно при обработке соответствующего события сборки. Для сохранения исходных файлов было необходимо реализовать в собственном компиляторе механизм, помечающий такие файлы.

Библиотеки в Kotlin, так же, как и в Java, принято распространять в виде JAR-библиотек. Эти библиотеки представляют собой запакованный байткод и используются анализатором для увеличения точности анализа. Используемые JAR-библиотеки также являются артефактами сборки.

3. Генерация промежуточного представления

Компиляторы Java и Kotlin на вход получают исходный код, производят синтаксический разбор и создают абстрактное синтаксическое дерево. Результатом работы компиляторов является байткод JVM.

По сравнению с Java язык Kotlin имеет значительно больше синтаксического сахара, что приводит к следующим проблемам:

- многие детали оригинальной программы теряются на уровне байткода;
- байткод содержит конструкции, которые являются результатом трансляции конструкций языка Kotlin и не были написаны непосредственно программистом.

Поясним обе эти проблемы. Анализатор *Svace* осуществляет поиск дефектов в исходном коде. Критерием выдачи предупреждения является то, что код надо поправить. При этом ошибка не обязательно будет проявляться во время выполнения. Например, это может быть бесполезное сравнение либо код в функции, которую никто не вызывает.

Так как не все свойства языка сохраняются при трансляции в байткод, не все ошибки могут быть найдены. Частичным решением этой проблемы являются детекторы на основе абстрактного синтаксического дерева (АСД). Но эти детекторы имеют свои хорошо известные ограничения. Поэтому часть дефектов, детали которых отсутствуют в байткоде и которые сложны для АСД, не может быть найдена.

Другой проблемой, связанной с трансляцией, является генерация бесполезного либо недостижимого кода. Компилятор генерирует его для множества конструкций языка Kotlin. Чтобы не выдавать бесполезные предупреждения, мы помечаем такие конструкции, как «сгенерированные компилятором». Например, из-за того, что в JVM отсутствуют булевский тип (он заменяется численным типом) и соответствующая для него инструкция отрицания, оператор отрицания в исходном коде часто раскрывается в ветвление. Отметим, что данная проблема актуальна и для Java кода [13].

На рис. 3 представлен пример такой генерации кода. Инструкции, соответствующие байтам 5, 6, 9, 10, 13 в байткоде, образуют ветвление, которое неявно присутствует в исходном коде

благодаря использованию оператора отрицания. Но поскольку ранее в условном выражении значение переменной *x* было проверено на истинность, одна из ветвей такого кода является недостижимой, о чём и сообщит статический анализатор.

```
1: fun foo(x: Boolean) {          0:  iload_1
2:     if (x) {                   1:  ifeq 17
3:       // Unreachable code      4:  aload_0
4:       // warning is incorrect  5:  iload_1
5:       smth(!x)                 6:  ifne 13
6:     }                          9:  iconst_1
7: }                              10: goto 14
                                  13: iconst_0
                                  14: invokevirtual smth:(Z)V
```

Рис. 3. Пример с оператором отрицания
Fig. 3. Negation operator example

```
1: var err = 0                    0: iconst_0
2: try {                          1: istore_2
3: return smth()                  2: aload_0
4: } catch(e: RuntimeException) { 3: invokevirtual smth:()I
5: err = 1                        6: istore_3
6: } finally {                    7: iload_2
7: if (err == 1) {                8: iconst_1
8: report()                       9: if_icmpne 16
9: }                              12: aload_0
10: finish()                      13: invokevirtual report:()V
11: }                             16: aload_0
12: return err                    17: invokevirtual finish:()V
                                  20: iload_3
                                  21: ireturn
                                  22: astore_3
                                  23: iconst_1
                                  24: istore_2
                                  25: iload_2
                                  26: iconst_1
                                  27: if_icmpne 34
                                  30: aload_0
                                  31: invokevirtual report:()V
                                  34: aload_0
                                  35: invokevirtual finish:()V
                                  38: goto 59
                                  41: astore 4
                                  43: iload_2
                                  44: iconst_1
                                  45: if_icmpne 52
                                  48: aload_0
                                  49: invokevirtual report:()V
                                  52: aload_0
                                  53: invokevirtual finish:()V
                                  56: aload 4
                                  58: athrow
                                  59: iload_2
                                  60: ireturn
```

Рис. 4. Пример с try-catch-finally
Fig. 4. Try-catch-finally example

Чтобы избежать выдачи ложных предупреждений, подобный код был помечен при помощи специальной аннотации, которая сообщает статическом анализатору о том, что такой код сгенерирован компилятором[1].

Ещё одна проблема генерации промежуточного представления связана с трансляцией finally-блоков. Компилятор языка Kotlin при генерации конструкций try-catch-finally генерирует дублирующийся код, соответствующий коду из finally-блока, и добавляет его после кода из блока try, после кода каждого из блоков catch и перед каждой инструкцией *break*, *continue* и *return*, которые могут совершить выход из конструкции try-catch-finally. Дублирование finally-блоков неудобно тем, что исходный код программы перестаёт взаимнооднозначно отображаться на байткод – каждой строке исходного кода может соответствовать сразу несколько инструкций из промежуточного представления. Следовательно, сгенерированный байткод может содержать пути, которые будут недостижимы, но при этом исполнение программы может проходить через соответствующие точки исходного кода. На рис. 4 представлен пример, взятый из [13]. Если код, представленный в try-блоке, завершится без исключений, то значение переменной *err* будет равно нулю, и условное выражение из finally-блока будет бессмысленно, о чём и сообщит статический анализатор. Чтобы избежать подобных ложных срабатываний и сгенерировать код из finally-блока лишь единожды, было использовано решение, применённое ранее при модификации компилятора javac, с использованием инструкций *jsr* и *ret* [13].

Следующей проблемой в генерации промежуточного представления компилятором языка Kotlin является генерация специальных intrinsic-вызовов для проверок значений на *null*. Например, при изменении типа значения с nullable в non-nullable при помощи операторов !! и *as* генерируется intrinsic-вызов *kotlin.jvm.internal.Intrinsics.checkNotNull*. Поведение этой функции неизвестно статическому анализатору, поэтому такой вызов заменяется на вызов специальной функции, которая сообщает анализатору, что происходит разыменование, и после этого объект не может принимать значение *null*. Все другие intrinsic-вызовы, генерируемые компилятором Kotlin, не имеют какого-либо значения для статического анализа, поэтому генерация таких вызовов была отключена при помощи соответствующих опций компилятора.

В языке Kotlin в отличии от Java иногда допустима перегрузка по возвращаемому значению, если компилятор может вывести тип возвращаемого значения из контекста. Например, такое возможно из-за более умной трансляции generic-типов параметров функций. В листинге 1 приведены примеры кода, идентичного в языках Java и Kotlin, но компилятор языка Java выдаёт ошибку при компиляции такого кода, а код на языке Kotlin может быть корректно скомпилирован при помощи kotlinc.

```
1: interface Example {
2:    Integer smth(List<Integer> l);
3:    String smth(List<String> l);
4: }
```

```
1: interface Example {
2:    fun smth(l: List<Int>): Int
3:    fun smth(l: List<String>): String
4: }
```

Листинг 1. Пример с перегрузкой по возвращаемому значению
Listing 1. Return value overload example

Такие перегрузки возможны благодаря тому, что в JVM в сигнатуру метода входит и тип возвращаемого значения метода, из-за чего методы с одинаковыми именами и типами параметров могут быть различены по типу возвращаемого значения. Компилятор языка Kotlin частично использует эту возможность JVM.

[1] Фактически, весь код генерируется компилятором. В данном случае инструкция ветвления появляется только в байткоде, а в исходном коде нет соответствующих инструкций ветвления.

Для поддержки такой возможности в анализаторе *Svace* пришлось исправить вид сигнатуры методов таким образом, чтобы в нём присутствовал и тип возвращаемого значения.

Существенным отличием JVM-байткода, сгенерированного компилятором Kotlin, от байткода, сгенерированного компилятором Java, является наличие большого числа синтетических функций, которые не представлены явно в исходном коде. К примеру, одним из нововведений языка Kotlin являются функции с параметрами по умолчанию, которые отсутствуют в языке Java. Для поддержки таких функций в байткоде генерируется специальная функция с суффиксом *$default*, в которую, помимо аргументов исходной функции, передаются как минимум два дополнительных аргумента, по которым вычисляется, какой из параметров принимает значение по умолчанию.

Так как такие синтетические функции скрываются при отладке, для них генерируются очень мало отладочной информации – например, зачастую отсутствует такая информация, как названия, типы и индексы локальных переменных и параметров. Наличие подобного рода функций существенно ухудшает понятность и полезность предупреждений, выдаваемых статическим анализатором. Чтобы улучшить качество выдаваемых предупреждений, компилятор kotlinc был модифицирован таким образом, чтобы для функций генерировалось больше отладочной информации: для многих выражений, в частности многострочных, была улучшена генерация атрибута *LineNumberTable*, хранящего взаимное соответствие строк исходного кода с инструкциями байткода; все локальные переменные и параметры, в том числе синтетические, добавляются в атрибут *LocalVariableTable* с корректными именами, индексами и типами. Тем не менее, наличие подобных синтетических функций всё ещё может создавать некоторые проблемы, поэтому в будущем нам представляется возможным изменение генерации промежуточного представления с целью полного или частичного отключения генерации этих функций.

Значительную сложность добавило наличие в данном языке встраиваемых (inline) функций, которые очень часто используются; в частности, большое количество функций стандартной библиотеки являются встраиваемыми. В отличие от языка C++, где наличие ключевого слова *inline* является только подсказкой для компилятора и может быть проигнорировано, язык Kotlin не позволяет полностью отключить такое поведение. Рассмотрим пример, представленный в листинге 2. Оператор *return*, вложенный в лямбда-выражение, передаваемое в функцию *map*, относится к внешней функции *sumOrNull*. Если в списке встретится строка, не являющаяся целым числом, то функция завершится и вернёт *null*. Такие *return*, которые находятся внутри лямбда-выражения, но завершают внешнюю функцию, называются нелокальными (non-local). Заметим, что функция *map* является встраиваемой. Если бы функция *map* и передаваемое внутрь неё лямбда-выражение не были встроены в место вызова, то такое поведение оператора *return* было бы невозможным, поскольку компилятор языка Kotlin не поддерживает нелокальные *return* внутри лямбда-выражений, передаваемых в обычные, не встраиваемые функции.

```
1: fun sumOrNull(strings: List<String>): Int? {
2:   return strings.map {
3:     val x = it.toIntOrNull()
4:     if (x == null) return null
5:   x
6:  }.sum()
7: }
```

Листинг 2. Пример с встраиваемыми функциями
Listing 2. Inline functions example

Наличие встраиваемых функций в коде существенно влияет на качество статического анализа. Так как зачастую встраивание функций создаёт код, являющийся недостижимым или излишним, то использование таких функций в анализируемом коде будет создавать большое количество ложных предупреждений. Листинг 3 иллюстрирует такое ложное срабатывание. Используемая функция *substring* принимает non-nullable параметры, поэтому

в месте вызова неявно генерируется проверка передаваемых аргументов на *null.substring* – это функция-расширение, поэтому в качестве её аргументов также передаётся объект, на котором вызывается эта функция. Так как функция вызывается два раза на одном и том же объекте, то его проверка на значение *null* будет осуществлена дважды. Следовательно, вторая проверка будет излишней, о чём и сообщит статический анализатор. В данном случае количество выданных ложных предупреждений было уменьшено благодаря изменениям в генерации intrinsic-вызовов, которые были описаны выше.

```
1: fun String.duplicateBefore(position: Int): String {
2:    return substring(0, position)
3:       + substring(0, position)
4: }
```

Листинг 3. Пример с встраиваемыми функциями из стандартной библиотеки
Listing 3. Inline functions from standard library example

В некоторых случаях наличие встраиваемых функций в коде может, наоборот, улучшать результаты анализа, так как становится проще понимать эффект применения таких функций в месте вызова. Например, стандартная библиотека языка Kotlin имеет большое количество функций высшего порядка, которые позволяют работать с коллекциями. Поскольку в инструменте *Svace* не реализовано моделирование таких функций, то при отключённом встраивании таких функций было бы сложнее понять, какой эффект они оказывают на результирующую коллекцию.

Также нужно отметить, что даже частичное отключение встраиваемых функций может существенно повлиять на время и результаты анализа. Отключение встраивания для функций означало бы, что для каждого лямбда-выражения, используемого при вызове, генерировался бы анонимный класс с виртуальным методом *invoke*, позволяющим выполнить данное лямбда-выражение. А из-за того, что использование встраиваемых функций в Kotlin является широко распространённой практикой, число анонимных классов и виртуальных функций, генерируемых компилятором, существенно возрастёт.

4. Анализ

Инструмент *Svace* использует анализ на основе резюме. На вход движку анализа подаются модули, представляющие из себя модифицированный байткод. Анализатор читает эти модули, строит граф вызовов, и начинает обход функций поочерёдно начиная с листьев графа вызовов. Вызываемые функции посещаются до вызывающих.

Каждая функция анализируется только один раз. После анализа создаётся резюме, которое описывает поведение функции, интересное для анализатора. При анализе инструкции вызова функции используется только её резюме, которое транслируется в контекст вызова.

Анализ отдельной функции является потоково-чувствительным. Анализ отличает различные поля структур и отдельные элементы массивов. Кроме этого, анализ имеет чувствительность к путям, то есть способен отличать отдельные пути, проходящие через граф потока управления. Для определения невыполнимых путей используется SMT-решатель. Подробнее про анализ отдельной функции можно прочитать в [14].

Мы реализовали детекторы для следующих видов ошибок:

- утечки ресурсов;
- использование ресурса после освобождения;
- разыменования нулевых указателей;
- недостижимый код;
- деление на ноль;
- отсутствие проверки кода возврата библиотечных функций;
- переполнение буфера данными из внешних источников.

5. Совместный анализ Kotlin и Java

Результат компиляции исходного кода Kotlin в байткод можно запаковать в JAR-библиотеку и использовать её методы в проекте, написанном на Java, передавая компилятору путь к библиотеке.

Кроме этого, разработчики языка Kotlin реализовали возможность вызывать методы, исходный код которых написан на Java, причём не только с помощью передачи пути до соответствующей JAR-библиотеки, но и с помощью передачи путей до исходного Java-кода, где эти методы определены. Таким образом, существует возможность разрабатывать проекты, в которых используются Kotlin и Java одновременно, более того, в которых существует циклическая зависимость по коду.

Ошибка в проекте может проявляться на пути, проходящим через Java и Kotlin код. Листинг [4] содержит пример такой ошибки. Метод *KotlinPartKt.test* вызывает метод *JavaPart.getBuggyString*, который вернет *null*, поскольку в качестве фактического параметра передана строка *bug*. Далее результат вызова метода разыменовывается, что приведет к *NullPointerException*.

```
// file: src/main/kotlin/kotlinPart.kt
1: package svace.test
2:
3: fun test(): Unit {
4:   val s: String = JavaPart.getBuggyString("bug")
5:   println(s.substring(s.lastIndexOf('/')))
6: }
7:
8: fun getBuggyString(b: String): String? {
9:   return if (b == "bug") null else "$b␣is␣OK"
10: }
...
// file: src/main/java/svace/test/JavaPart.java
1: package svace.test;
2:
3: public class JavaPart {
4:   private void test() {
5:     String s = KotlinPartKt.getBuggyString("bug");
6:     System.out.println(s.substring(s.lastIndexOf('/')));
7:   }
8:
9:   public static String getBuggyString(String b) {
10:    if (b.equals("bug")) return null;
11:    return b + "␣is␣OK";
12:  }
13: }
```

Листинг 4. Пример, для анализа которого необходим совместный анализа Kotlin и Java
Listing 4. Cross language analysis example

Заметим, что в определении метода *JavaPart.getBuggyString* возвращаемое значение не имеет аннотации *Nullable*. Приведенный пример также демонстрирует один из случаев, когда, несмотря на null-безопасную систему типов языка, в программе на Kotlin произойдёт разыменование нулевого указателя. В Java-части проекта метод *JavaPart.test* вызывает метод **KotlinPartKt.getBuggyString**, возвращающий *null*, который будет затем разыменован.

Соответственно, для поиска таких ошибок от анализатора требуется анализ кода для обоих языков с учётом зависимостей.

В анализаторе *Svace* для этого был разработан специальный режим работы, в котором строится общий граф вызовов для целого проекта и учитываются зависимости между Kotlin и Java кодом. Затем на общем графе вызовов проводится полноценный анализ.

Анализ для приведённого листинга 4 выдаёт следующие предупреждения:

1.
   - Pointer 's' returned from function 'KotlinPartKt.getBuggyString' at JavaPart.java:5 may be null, and it is dereferenced at JavaPart.java:6.
   - Variable 's' is dereferenced at JavaPart.java:6
   - Null assign at kotlinPart.kt:9
2.
   - Pointer 's' returned from function 'JavaPart.getBuggyString' at kotlinPart.kt:4 may be null, and it is dereferenced at kotlinPart.kt:5.
   - Variable 's' is dereferenced at kotlinPart.kt:5
   - Assign null at JavaPart.java:10

6. Анализ на основе абстрактного синтаксического дерева

Компилятор Kotlin предоставляет АСД, информацию о типах переменных, иерархии классов, а также константах времени компиляции. Более того, компилятор имеет встроенный анализатор АСД. Подобные анализаторы позволяют находить опечатки в исходном коде. Нами были реализованы детекторы для обнаружения следующих дефектов:

- сравнение вместо присваивания, то есть использование оператора равенства, не влияющего на выполнение программы;
- повторяющиеся условия, то есть дублирование условий в операторе с несколькими условиями;
- некорректные границы интервала: при создании интервала вида *a..b*, такого, что $a > b$, в Kotlin создаётся пустой интервал
- вызов метода *next()* в реализации метода *hasNext()* в классе, реализующем интерфейс *Iterator*.

7. Спецификации

Спецификации представляют собой код на анализируемом языке, который добавляется в анализатор. В спецификациях используются вызовы специальных функций, которые имеют особое значение для анализатора. Функции из спецификаций анализируются, и далее их резюме используется вместо резюме оригинальных функций проекта. Спецификации позволяют решить две проблемы:

- добавить семантику для библиотечных функций, код которых отсутствует;
- сообщить анализатору детали поведения функции, которые не могут быть выведены средствами статического анализа.

В листинге 5 представлен пример спецификации, а также код, дефект в котором анализатор обнаружит только при наличии соответствующей спецификации. Рассмотрим пример подробнее. В методе *testTainted* значение *num* вычисляется с помощью вызова метода *toInt* объекта типа *String*. Затем значение num используется в качестве индекса доступа к массиву.

Как правило, функции преобразования строк в число используются для данных из внешних источников. Мы используем эвристику, заключающуюся в том, что результат таких функций необходимо проверить. Это позволяет упростить анализ и не проверять, что строка получена из внешнего источника. Использование целых чисел из внешнего источника как индекс доступа к массиву может привести к возникновению исключения. Если эти данные

действительно из внешних источников, то злоумышленник сможет контролировать такое поведение. В отличие от языка C, здесь это не является уязвимостью, но всё ещё остаётся ошибкой в программе.

```
// specification source file
1: package kotlin.text
2:
3: import ru.isp.svace.sf.*;
4:
5: public fun String.toInt(): Int {
6:   val res = SpecFunc.sf_get_some_int() as Int
7:   SpecFunc.sf_set_tainted_int(res)
8:   return res
9: }
...
// test source file
1: fun testTainted(number: String) {
2:   val num: Int = number.toInt()
3:   val x: IntArray = intArrayOf(1, 2, 3)
4:   print(x[num])
5: }
```

Листинг 5. Пример спецификации
Listing 5. Specification example

Функция *SpecFunc.sf_set_tainted_int* сообщает анализатору, что её параметр получен из внешнего источника и требует проверки. Далее это свойство распространяется анализатором. И на 4 строке будет выдано сообщение об ошибке, так как индекс доступа к массиву может лежать за границами, заданными его размером.

Помимо спецификаций, добавленных разработчиками анализатора, пользователь может добавлять свои собственные спецификации для каждого анализируемого проекта.

8. Анализ метрик и отношений

В инструменте *Svace* содержится отдельный компонент, служащий для определения сущностей программы, их метрик, связей между ними. Сущностями обычно являются методы, глобальные переменные, поля, классы, файлы и каталоги; связями --- случаи чтения или записи одной сущности другой, вызова, включения, наследования и пр.; метриками --- количественные характеристики сущностей или групп связей. Более подробно об устройстве такого анализа для языков C/C++ можно прочесть в статье [15].

Для языка Java подсчет метрик и отношений ведется схожим с поиском дефектов способом: сначала выполняется контролируемая сборка для запуска собственного компилятора Java, который вычисляет часть метрик, могущих быть определенными только в момент разбора программы с полным доступом к исходному коду; затем запускается анализ, который считывает Java-байткод и специальные аннотации в нем, содержащие вычисленные компилятором метрики, и выполняет окончательный анализ, как агрегируя уже посчитанные метрики для всей программы, так и считая часть метрик по байткоду.

Учитывая, что язык Kotlin компилируется в байткод Java, было принято решение повторно использовать имеющийся анализатор Java-байткода и доработать собственный компилятор Kotlin для подсчета тех же метрик, что для Java также вычисляются в компиляторе. Задача такого подсчета внутри компилятора имеет инженерный характер. Отметим две интересные особенности. Во-первых, в отличие от компилятора javac абстрактные синтаксические деревья компилятора Kotlin полностью сохраняют всю информацию о лексемах, которые были использованы при построении данного дерева, и тем самым модификация лексического анализа не требуется: нужные данные можно получить на поздних этапах компиляции. Во-вторых, при обновлении компилятора Kotlin до версии 1.5 оказалось, что кодогенерация для

старых версий языка (1.4 и ниже) выполняется в компиляторе из одного вида внутреннего представления (АСД-деревьев), а кодогенерация для версии 1.5 выполняется из полностью другого вида деревьев. Такое решение разработчиков Kotlin можно охарактеризовать как до некоторой степени странное; для нас это означало необходимость рефакторинга кода записи аннотаций с метриками, чтобы его можно было вызывать из всех компонентов кодогенерации для обеих версий языка.

В целом, как и в случае поиска дефектов, удалось успешно использовать анализатор байткода Java для вычисления метрик также и для Kotlin; теперь возможен и совместный анализ программ на Java и Kotlin, например, вызовы между Java и Kotlin частями успешно распознаются. Компиляторная часть анализа метрик нуждается в дальнейшей доработке для полноценной поддержки случаев генерации синтаксического сахара и исключения сгенерированного компилятором кода наподобие того, что уже было описано для случая поиска дефектов.

9. Результаты

Для оценки качества и производительности разработанного анализатора был выбран проект компилятора Kotlin [16]. Выбор данного проекта обусловлен несколькими причинами. Во-первых, это самый крупный проект с исходным кодом на языке Kotlin. Проект содержит 2423 тысячи строк Kotlin кода и 1093 тысячи строк Java кода. Во-вторых, в проекте одновременно используется и Kotlin, и Java, следовательно, мы можем протестировать анализатор в режиме совместного анализа двух языков. Наконец, мы полагаем, что в проекте, над которым работают разработчики языка, присутствует наибольшее разнообразие языковых конструкций и минимальное количество дефектов, что является вызовом для статического анализатора, нацеленного на выдачу минимального числа ложных предупреждений в процентном соотношении от общего числа предупреждений.

Оригинальная сборка проекта длится в среднем 17 минут[2], контролируемая сборка длится 87 минут. Таким образом, оригинальная сборка замедляется приблизительно в 5 раз в случае контролируемой сборки для последующего проведения анализа с помощью *Svace*. Время анализа проекта в режиме совместного анализа Kotlin и Java кода составляет 19 минут.

Оценка качества анализа – нетривиальная задача. Разметка предупреждений, выданных анализатором, – достаточно трудоёмкий процесс. Более того, анализатор находится в фазе активной разработки, и множество выдаваемых предупреждений постоянно изменяется. На данный момент размечена лишь часть актуальных предупреждений на проекте (41% от общего числа предупреждений), хотя в общей сложности было размечено более тысячи предупреждений, большая часть которых в настоящий момент не выдается в результате проделанной работы по подавлению ложных предупреждений.

Табл. 1. Оценка качества результатов анализа
Table 1. Quality evaluation of analysis results

Группа детекторов	Истинные предупреждения, %
Разыменование нуля	30
Утечка ресурсов	43
Целочисленное переполнение	89
Недостижимый код	44

Несмотря на то, что наша команда дорабатывала некоторые модули компилятора, код большой части проекта нами мало изучен. По этой причине мы не утверждаем, что все

[2] Данный и последующие замеры времени проводились на вычислительной машине с характеристиками: 8 cores 2.4GHz, 64RAM. Среднее значение – это среднее арифметическое в серии из нескольких замеров.

предупреждения размечены корректно, в особенности предупреждения, выданные межпроцедурными детекторами.

Статистика для некоторых групп детекторов приведена в табл. 1. Для оценки качества группы детекторов будем вычислять процент истинных предупреждений от общего числа размеченных предупреждений в данной группе.

Полученные результаты являются неудовлетворительными для нас в данный момент – целевое минимальное значение процента истинных предупреждений составляет 70%. Мы будем продолжать работы для достижения приемлемого качества. Но поскольку выбранный проект – это компилятор, в котором используются сложные конструкции, и над ним работают квалифицированные разработчики, такой результат мы оцениваем, как адекватный на данном этапе разработки анализатора.

Среди причин выдачи большого процента ложных предупреждений можно выделить значительный объём синтаксического сахара в исходном коде. Во многих случаях, описанных в разд. 3, мы смогли решить проблемы, возникающие в процессе генерации кода для таких конструкций. Однако работы в этом направлений ещё не закончены. Следующей причиной, на наш взгляд, является обширная стандартная библиотека Kotlin, которую еще предстоит описать с помощью механизма спецификаций, описанного в разд. 7. Последней известной нам причиной является тот факт, что в Kotlin-проектах активно используется парадигма функционального программирования. Для высокого качества анализа таких проектов от статического анализатора требуется построение графа вызовов с учётом девиртуализации. В *Svace* используются базовые алгоритмы девиртуализации, которые мы планируем совершенствовать в будущем.

10. Заключение

В статье была описана реализация анализа программ на языке Kotlin с помощью статического анализатора *Svace*. Основным принципом поддержки анализа Kotlin являлся анализ JVM-байткода, генерируемого компилятором Kotlin, так как имеющийся анализатор уже содержал поддержку анализа байткода для языка Java.

Процесс адаптации инструмента для анализа потребовал поддержки контролируемой сборки через перехват интерфейсов компиляции Kotlin, доработки компилятора Kotlin для построения адекватного для статического анализа внутреннего представления, а также доработки существующих детекторов для языка Java и создания некоторых новых детекторов, в том числе детекторов для анализа АСД-уровня. Полученные результаты приемлемы с учетом высокого качества анализируемого тестового кода, но требуются дальнейшие работы для достижения стандартного для *Svace* уровня в 70% истинных срабатываний.

Список литературы / References

[1] JetBrains s.r.o. URL: https://www.jetbrains.com, accessed October 6, 2021.
[2] P. Miller. Google is adding Kotlin as an official programming language for Android development. URL: https://www.theverge.com/2017/5/17/15654988/google-jet-brains-kotlin-programming-language-android-development-io-2017, accessed October 6, 2021.
[3] The Java Virtual Machine Specification. Java SE 8 Edition. URL: https://docs.oracle.com/javase/specs/jvms/se8/html, accessed October 6, 2021.
[4] C.A.R. Hoare. Null references: the billion dollar mistake. Presentation at QCon, 2009-08-25. URL: https://www. infoq.com/presentations/null-references-the-billion-dollar-mistake-tony-hoare, accessed October 6, 2021.
[5] Detekt analyzer. URL: https://detekt.github.io/detekt, accessed October 6, 202.
[6] An anti-bikeshedding Kotlin linter with built-in formatter. URL: https://ktlint.github.io, accessed October 6, 2021.
[7] В.П. Иванников, А.А. Белеванцев и др. Статический анализатор Svace для поиска дефектов в

исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250 / V.P. Ivannikov, A.A. Belevantsev et al. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7.

[8] А. Е. Бородин, А. А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134 / A.E. Borodin, A.A. Belevantsev. A static analysis tool Svace as a collection of analyzers with various complexity levels. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

[9] А.А. Белеванцев, А.О. Избышев, Д.М. Журихин. Организация контролируемой сборки в статическом анализаторе svace. Системный администратор, вып. 7-8, 2017 г., стр. 135-139 / A.A. Belevantsev, A.O. Izbyshev, D.M. Zhurikhin. Monitoring program builds for Svace static analyzer. System Administrator, issues 7-8, 2017, pp/ 135-139 (in Russian).

[10] Package java.lang.instrument. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html, accessed October 6, 2021.

[11] Kotlin Evolution. URL: https://kotlinlang.org/docs/kotlin-evolution.html, accessed October 6, 2021.

[12] Using kapt. URL: https://kotlinlang.org/docs/kapt.html, accessed October 6, 2021.

[13] А.П. Меркулов, С.А. Поляков, А.А. Белеванцев. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74 / A.P. Merkulov, S.A. Polyakov, A.A. Belevantsev. Supporting Java programming in the Svace static analyzer. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 3, 2017. pp. 57-74 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-5.

[14] А.Е. Бородин, И.А. Дудина. Внутрипроцедурный анализ для поиска ошибок на основе символьного выполнения. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 87-100 / A.E. Borodin, I.A. Dudina. Symbolic Execution Based Intra-Procedural Analysis for Search for Defects. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 87-100 (in Russian). DOI: 10.15514/ISPRAS-2020-32(6)-7

[15] А.А. Белеванцев, Е.А. Велесевич. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 53-64 / A.A. Belevantsev, E.A. Velesevich. Analyzing C/C++ Code Entities and Relations for Program Understanding. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 2, 2015, pp. 53-64 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-4.

[16] Kotlin compiler project. URL: https://github.com/JetBrains/kotlin, accessed October 19, 2021.

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ – студент бакалавриата факультета компьютерных наук НИУ ВШЭ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM языки.

Vitaly Olegovich AFANASYEV – undergraduate student at the Faculty of Computer Science, NRU HSE, employee of ISP RAS. Research interests: compiler technologies, static analysis, JVM languages.

Сергей Андреевич ПОЛЯКОВ – младший научный сотрудник. Сфера научных интересов: статический анализ, параллелизм, JVM языки.

Sergey Andreevich POLYAKOV – researcher. Research interests: static analysis, concurrency, JVM languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – PhD, researcher. Research interests: static analysis for finding errors in source code.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr.Sc., Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.