Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Анализ инициализированности в языке Kotlin

Студент гр. 3530901/80201 М.Л. Динмухаметов

Санкт-Петербург
2022

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Работа допущена к защите
зав. кафедрой

_____ В.М. Ицыксон

«____» _____ 2022 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Анализ инициализированности в языке Kotlin

по направлению 09.03.01 «Информатика и вычислительная техника»
по образовательной программе
09.03.01_02 «Технологии разработки программного обеспечения»

Выполнил студент гр. 3530901/80201 _____ М.Л. Динмухаметов

Научный руководитель,
ст. преп. _____ М.Х. Ахин

Консультант по нормоконтролю,
к. т. н., доц. _____ А.Г. Новопашенный

Санкт-Петербург
2022

РЕФЕРАТ

На 27 с.

БЕЗОПАСНАЯ ИНИЦИАЛИЗАЦИЯ, KOTLIN, СИСТЕМА ТИПОВ И ЭФФЕКТОВ

Тема выпускной квалификационной работы: «Анализ инициализированности в языке Kotlin».

Данная работа посвящена созданию анализа для нахождения ошибок инициализации в языке Kotlin на основе подхода представленного для Scala 3 [5] . В ходе данной работы решались следующие задачи:

- Анализ существующих решений по поиску ошибок инициализации в языках.
- Разработка анализа для нахождения не безопасной инициализации в языке Kotlin.
- Создание прототипа, реализующего данный подход.
- Оценка производительности и тестирование данного подхода.

Анализ основывается на подходе, который был представлен для языка Scala 3 [5] и модифицирует его для работы в языке Kotlin. Анализ основывается на системе типов и эффектов, и способен находить ряд проблем с инициализацией, которые существующий анализ находить не способен.

Должна ли она тут быть и нужно ли вообще упоминать скалу

THE ABSTRACT

27 pages

SAFE INITIALIZATION, KOTLIN, TYPE-AND-EFFECTS SYSTEM

TODO Hello

СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

self	*this* в языке Swift
freedom model	Freedom before commitment
NPE	NullPointerException — Исключение возникающее, при использовании переменной хранящей null

ВВЕДЕНИЕ

Написание безопасных и надежных программ является сложной задачей для программиста, некоторые ошибки можно легко найти на стадии компиляции, но не все. Так, обращение к еще не инициализированным полям является частой причиной возникновения ошибок в объектно-ориентированных языках программирования. Такие ошибки тяжело находить, а поведение программы может становиться не предсказуемым.

Целью данной работы является разработка анализа для нахождения ошибок инициализации для языка Kotlin. Также в этой работе создается прототип системы безопасной инициализации, и производится оценка его производительности.

Работа состоит из 6 разделов. Первый раздел посвящен проблеме. Включает в себя ряд примеров не безопасной инициализации и ряд обязательных требований без которых анализ будет не состоятельным. Второй раздел включает в себя обзор существующих решений в индустриальных языках и теоретических подходов к решению проблемы безопасной инициализации. Также в данном разделе выделяются достоинства и недостатки каждого из подходов. Третий раздел включает в себя постановку задачи. Четвертый раздел посвящен дизайну анализа инициализированности. В данном разделе выбирается подход на основе, которого будет создаваться безопасная инициализация для языка Kotlin. Также рассмотрены проблемы, которые пришлось решать, чтобы выбранный подход мог работать для языка Kotlin. В пятом разделе рассказывается про детали реализации прототипа и объясняются основные идеи анализа. Шестой раздел включает в себя тестирование написанного прототипа, а также оценку производительности и качества анализа. Рассказывается на каких проектах был запущен прототип и какие результаты он показал.

1. ПРОБЛЕМА

Возможно добавить определение инициализации в свифт буке не плохое

Проблема не полностью инициализированных объектов известна с появления первых объектно-ориентированных языков программирования, поэтому для данной проблемы существует множество решений, но все эти решения имеют свой набор плюсов и минусов.

Ошибки инициализации имеют достаточно простую природу — это доступ к еще не инициализированным полям, но хоть они и имеют достаточно простую природу, находить ошибки инициализации часто являются головной болью для программистов

Давайте рассмотрим ряд ошибок инициализации для лучшего понимания проблемы. В приведенном примере, если создать экземпляр класса *Hello*, то программа аварийно завершится с *NullPointerException*(NPE), так как во время инициализации свойства *nameLength*, свойство *name* еще не существует.

Листинг 1.1. Пример не безопасной инициализации

```
1 class Hello {
2     fun foo() = name
3
4     val nameLength = foo().length
5     val name = "Alice"
6 }
```

Такую ошибку легко увидеть если код состоит из 6 строчек, но если вы разрабатываете большой и сложный класс, то это становится нетривиальной задачей

Следующий пример показывает, что ошибки инициализации могут появиться и при наследовании. Проблема заключается в том, что вначале идет инициализация класса родителя, в данном случае класс *A*. А во время его инициализации будет вызван переопределенный

метод *getName*, который обращается к еще не инициализированному свойству *c*. Оно будет определено только во время инициализации класса *B*.

Листинг 1.2. Пример не безопасной инициализации

```
1  open class A {
2      val a = "Hello"
3      val b = getName().length
4
5      open fun getName() = a
6  }
7
8  class B : A() {
9      val c = "World"
10
11     override fun getName() = c
12 }
```

Приведенный выше пример специфичен для языка Java, но он вызывает проблемы и в языке Kotlin. Давайте перепишем данный пример для демонстрации, того что в языке Kotlin данная ошибка может быть чуть менее очевидна программисту.

Листинг 1.3. Пример не безопасной инициализации

```
1  open class A1 {
2      open val a = "Hello"
3      val b = a.length
4  }
5
6  class B1 : A1() {
7      override val a = "World"
8  }
```

Кажется, что данный пример не должен приводить к *NPE*, так как *a* и *b* правильно инициализируются, после чего вызывается конструктор *B1*, и завершает создание объекта. Но в языке Kotlin данный пример приведет к *NPE*. Для поля автоматически генерируются компилятором геттеры и сеттеры , и во время инициализации класса *A1* будет вызван уже переопределённый геттер, который в свою очередь воз-

Не уверен что можно использовать такие терми-

вращает еще неинициализированное поле *a* класса *B1*. На самом деле любое практически любое переопределение членов класса родителя может привести к ошибкам. Эта проблема известна, как *проблема хрупкого базового класса* [2].

ИИИ?

Также проблемы инициализации могут нести в себе и внутренние классы. Рассмотрим данный пример.

Листинг 1.4. Пример не безопасной инициализации

```
1  class Outer {
2      val i = Inner()
3      val tag = "Hello"
4
5      inner class Inner {
6          val tagLength = tag.length
7      }
8  }
```

Проблема заключается в том, что внутренний класс может иметь доступ к *this* еще не достроенного внешнего класса. Так класс *Inner* имеет доступ к еще не инициализированному свойству *tag*.

Пример ниже показывает еще один пример не тривиальной ошибки инициализации.

Листинг 1.5. Пример не безопасной инициализации

```
1  class C {
2      val b = O.c
3
4      fun hello(): String = "Hello, World!"
5  }
6
7  object O {
8      val c = C()
9      val hello = c.b.hello()
10 }
```

В данном примере во время инициализации свойства *c*, создается экземпляр класса *C*. Он в свою очередь инициализирует свойство *b*, через свойство *c*, которое еще находится в стадии инициализации. В

данной работе не рассматривается проблема инициализации статических объектов, но они также могут быть источниками ошибок инициализации, и их поддержка, возможно, будет добавлена в следующих версиях анализа

Проблема ошибок инициализации создает трудности не только для программистов, но и для разработчиков компиляторов и дизайнеров языков программирования, так если в языке нет безопасной инициализации, то нельзя гарантировать неизменяемость или ненулевость [1]

нормальное слово

Не уверен, что хорошо ссылаться на личный блог, но там и правда неплохо все объяснено ссылка из статьи

Kotlin — это null-безопасный язык, и система типов языка Kotlin гарантирует, что *non-nullable* тип не может быть null.

Листинг 1.6. Пример не безопасной инициализации

```
1 class Message {
2     fun foo(): String = name
3
4     val hello: String = "Hello, " + foo()
5     val name: String = "Alice"
6 }
```

Если попробовать напечатать значение хранящееся в *hello*, то выведется *"Hello, null"*, хотя *name* не может быть null, потому что это гарантирует система типов языка Kotlin. Это делает ошибки инициализации в языке Kotlin менее очевидными и ожидаемые

1.1. Резюме

В данном разделе была рассмотрена причина возникновения ошибок инициализации. Они возникают при обращении к еще не инициализированным полям. А источником ошибок инициализации могут быть:

- методы

- внутренние классы
- наследование
- статические объекты
- итд

В языке Kotlin такие ошибки могут быть менее ожидаемыми, так как это null-безопасный язык. Также, отмечается, что ошибки инициализации это проблема не только для программистов, но и для дизайнеров языка.

2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Обращение еще к неинициализированным полям является проблемой не только для языка Kotlin, но и для многих других языков. В индустриальных языках существует два классических подхода для решения данной проблемы.

Первый способ — это запретить программисту писать некоторый код на стадии инициализации, зато гарантировать ему безопасную инициализацию. Данный подход использует язык Swift.

Второй способ — это не отслеживать ошибки инициализации, но разрешать программисту писать практически любой код. Такой подход используют такие языки как Kotlin, Scala 2, Java и многие другие. Данное решение хорошо тем, что он дает программисту достаточное количество свободы, и не заставляет его пытаться обойти ограничения, которые вводит первый подход.

Мб примерчик на Swift, для инициализации циклических структур данных с нулями итд

На самом деле существует и третий подход, он рассмотрен в секции 2.3 с языком Dart. Это скорее способ смягчить данную проблему, а не решить ее полностью

В данном разделе обозреваются существующие решения проблемы ~~не~~ безопасной инициализации.

2.1. Безопасная инициализация в языке Kotlin

В языке Kotlin нет безопасной инициализации, но есть ряд не больших проверок.

В интегрированной среде разработки IntelliJ IDEA, существуют средства, которые анализируют код и могут сообщать о неэффективном или небезопасном коде. Одно из таких средств умеет анализировать код на языке Kotlin и находить проблемы связанные с проблемой

ие хочу
их на-
зывать
инспек-
циями,
чтобы
ие объ-
яснять

утекшего this. Данный анализ достаточно слабый и чтобы его обмануть достаточно обернуть проблемный код в вызов метода.

ссылочк
на глу-
хих или
код

Листинг 2.1. Пример не безопасной инициализации

```
1  open class A3 {
2      open val a = "Hello"
3      val b = foo()
4
5      fun foo() = a.length
6  }
7
8  class B3 : A3() {
9      override val a = "World"
10 }
```

Если для кода приведенного в листинге 1.2 пользователь получит предупреждение в IntelliJ IDEA, то для кода примера выше предупреждения не будет, но данные коды одинаково не безопасны. Таким образом данный анализ может найти только прямые обращения к свойствам и методам, которые можно переопределить, во время построения объекта. Более того данный анализ не находит ошибки инициализации, и если переопределение безопасно как здесь, он все равно будет показывать предупреждение.

Листинг 2.2. Пример не безопасной инициализации

```
1  open class A4 {
2      val a = "Hello"
3      val b = foo().length
4
5      open fun foo() = a
6  }
7
8  class B4 : A4() {
9      override fun foo() = "World"
10 }
```

Данный код не имеет ошибок инициализации, но программист все равно будет получать ошибки инициализации. Да, это не большая проблема, потому что такой код не самый безопасный, но иногда приходится

его писать.

Также, в языке Kotlin есть анализ, который проверяет, что все свойства были инициализированы и, что нет доступа к еще неинициализированным свойствам, но данный анализ имеет ту же проблему, что и предыдущий, если обернуть проблему в вызов метода анализ не сможет ее найти. Например, код из листинга 1.1 компилируется и не предупреждает об использовании свойства *name* до его инициализации.

Таким образом способы обеспечения безопасной инициализации реализованные в данный момент, не могут в полной мере, найти все ошибки инициализации.

2.2. Инициализации в языке Swift

В языке Swift по сути решена проблема с ошибками инициализации. Решение достаточно радикальное, но оно хорошо справляется с данными ошибками. В языке Swift существует две стадии инициализации, во время первой стадии запрещено вызывать методы, обращаться к свойствам и использовать *self* (аналог *this*) Она длится пока не будут инициализированы все свойства. То есть получить доступ к еще не инициализированному свойству нельзя. Так, естественным образом решается проблема не безопасной инициализации.

Данный подход имеет одно не очень приятное свойство, для написания некоторого кода программисту иногда приходится жертвовать неизменяемостью и null-безопасностью. Например, для создания циклической структуры данных из примера ниже, свойство *child* должно быть изменяемое и иметь nullable тип. Это не очень хорошо, так как такой код тяжелее поддерживать и больше вероятность ошибиться.

ссылочка куда-нибудь

Листинг 2.3. Циклическая структура данных в языке Swift

```
1 class Parent { // hello
2   var child: Child? = nil
3   let tag = "Parent"
```

```
4     init() {
5       child = Child(p: self)
6     }
7   }
8
9   class Child {
10    let parent: Parent
11    init(p: Parent) {
12      parent = p
13    }
14  }
```

2.3. Инициализации в Dart

Если хватит времени рассмотреть инициализацию в Dart

2.4. Маскирующие типы

Одним из многообещающих подходов является решение основывающееся на маскирующих типах [3]. Данный подход представляет чувствительную к потоку систему типов и эффектов для безопасной инициализации. Маскирующий тип $T \backslash f$ значит, что на данной стадии инициализации типа T нельзя получить доступ к полю f. В работе вводятся эффекты, которые передают информацию о маске между вызовами методов.

Листинг 2.4. Маскирующие типы

```
1  Point(int x, int y) effect * -> Point.sub
2  void display() effect {} -> {}
```

В данном случае конструктор *Point* может быть вызван, когда ни одно поле еще не инициализировано, это означает *. При этом после построения получается объект с маской, у которого все поля класса *Point* были инициализированы, а все поля подклассов еще не инициализированы. Эффект для метода *display* переносит информацию, что

данный метод может быть вызван только если все поля были инициализированы, и возвращает тоже полностью инициализированный объект.

Данный подход обладает набором приятных свойств. Он поддерживает абстрактные классы, циклические структуры данных, внутренние классы, typestate полиморфизм, а также может быть расширен на другие сложные конструкции. Но у подхода есть и ряд серьезных недостатков. Так, анализ поддерживает typestate полиморфизм, но в нем нельзя выразить, что метод можно использовать при любом статусе инициализированности. Вторым серьезным минусом данного подхода является, обширный дополнительный синтаксис. Авторы статьи утверждают, что можно сделать систему вывода, но в данной и следующих статьях такой возможности не представлено.

2.5. Freedom before commitment

В работе «Freedom before commitment»(далее freedom model) [4] представляется нечувствительную к потоку систему типов и эффектов для безопасной инициализации. В данной работе объекты делятся на два вида свободные и зафиксированные. Свободные объекты — это объекты, обращение к полям, которых может давать null. Зафиксированные объекты — это полностью инициализированные объекты обращение к которым полностью безопасно. Выделяется четыре приятных свойства данного анализа:

- Модульность: Классы проверяются по отдельности.
- Надежность: Если объект имеет зафиксированный статус, то использование данного объекта гарантировано безопасно
- Выразительность: Данный подход поддерживает сложный код, как инициализация циклических структур данных
- Простота: Подход не требует большего числа аннотаций в отличие от маскирующих типов из раздела 2.4

Но отмечается и ряд недостатков, так в данной системе не реализована система вывода типов, а значит программистам придется самим помечать объекты в своем коде. Также, проблемой подхода является не возможность использования уже инициализированных полей во время конструирования объекта. Авторы статьи предлагают использовать анализ потока данных для того, чтобы находить уже инициализированные поля, но не реализуют такой возможности. Более серьёзной проблемой с точки зрения анализа является плохая поддержка typestate полиморфизма.

Листинг 2.5. Проблема с typestate полиморфизмом в подходе

```
1    class Parent {
2        val child = Child(this)    // недоинициализированное значение
3        val tag: Int = child.tag   // ошибка
4    }
5    class Child(parent: Parent @free) {
6        val tag = 10
7    }
```

Так, для кода выше была бы выдана ошибка инициализации, так как значения *child* недоинициализировано, а значит оно свободное. Следовательно, нельзя обращаться к его методам.

Данный подход один из первых сделал анализ, достаточно простым и выразительным, чтобы его можно было использовать в разработке. При этом подход может иметь ряд дополнений, которые позволят сделать анализ более безопасным и простым.

2.6. Безопасная инициализация для языка Scala 3

Развитием идей представленных в работе freedom model, является система типов и эффектов для безопасной инициализации представленная для языка Scala 3 [5]. В данной работе тип несет информацию об уже инициализированных полях(Ω), и вводятся три дополнительных подтипа:

- Холодный — объект не инициализирован.
- Теплый — все поля объекта инициализированы, но он может достигать холодных значений.
- Горячий — объект полностью инициализирован.

Таким образом C^{Ω} обозначает, что для класса C инициализированы поля Ω. А запись C @*cold* означает, что ни одно свойство класса не было инициализировано. Такой подход позволил реализовать полноценный typestate полиморфизм, который не мог быть реализован во freedom model. А также позволил обращаться к уже инициализированным полям, при этом без использования анализа потока данных.

В результате авторам удалось создать подход, который почти ничем не уступает подходу freedom model. Более того превосходит его решая проблемы, которые были не решены в изначальном подходе. Так, анализ обладает всеми четырьмя приятными свойствами freedom model: модульность, надежность, выразительность, простота. Но помимо того, что данный подход также не требует большого количества аннотаций, авторы предоставляют для него еще и автоматический вывод почти всех аннотаций. Этот подход является и более выразительным, так как авторы прошлого подхода даже не рассматривают такие конструкции как: лямбды, интерфейсы с реализацией, внутренние классы. Большим плюсом данного подхода является, то что он реализован для языка Scala 3.

Минусами данного подхода являются:

- Не высокая производительность анализа, авторы статьи заявляют, что анализ увеличивает время компиляции на 20%.
- Не реализована аннотации *@cold* для аргументов конструктора, но авторы обходят данный момент в реализации, выводя *@cold* хотя бы для *this*.

2.7. Резюме

В данном разделе были рассмотрены различные подходы для реализации безопасной инициализации. Так, язык Swift инициализация делится на две стадии. Во время первой запрещается вызывать методы, обращаться к полям и использовать *self*. Таким образом избегается доступ к еще не инициализированным полям. Маскирующие типы, freedom model и безопасная инициализация для языка Scala 3 пытаются создать менее радикальный подход, при этом данные подходы стараются минимально ограничивать программиста в способах инициализации.

3. ПОСТАНОВКА ЗАДАЧИ

Kotlin молодой язык, но при этом на нем написано уже много кода, поэтому чтобы не нарушать обратную совместимость, нельзя изменить

TODO Не очень понимаю что здесь нужно написать. Вроде про то что проблема есть в котлине и какими способами ее можно решить мы уже обсудили в предыдущем разделе

После прочтения раздела 1 может показаться, что достаточно запретить использование не до конца сконструированных значений, иначе говоря запретить использование *this* в конструкторе, но это не так.

Листинг 3.1. Пример не безопасной инициализации

```
1 class Parent {
2     val tag = "Hello"
3 }
4
5 class Child(parent: Parent) {
6     val tag = parent.tag
7 }
```

4. ДИЗАЙН

Придумать нормальное название разделу

4.1. Выбор подхода для создания безопасной инициализации в языке Kotlin

TODO Говорим какой подход выбрали почему не выбрали другие итд

4.2. Проблемы подхода для языка Kotlin

TODO Рассказ про проблемы которые существуют для применения решения из скалы и их решение

4.3. Резюме

TODO Резюме про выбранный подход и набор проблем

5. СОЗДАНИЕ ПРОТОТИПА

Раздел может быть различного размера для начала опишем минимум

5.1. Система типов и эффектов

TODO Рассказываем про итоговую систему типов и эффектов, что она сильно не изменилась и взята из скалы

5.2. Интеграция в компилятор

TODO Рассказываем про то как это все взаимодействует с компилятором

5.3. Резюме

TODO Кратко как это работает в компиляторе

6. ТЕСТИРОВАНИЕ И ОЦЕНКА РЕЗУЛЬТАТОВ

6.1. Тестирование

TODO Тестирование на чем запускались итд

6.2. Оценка результатов

TODO Производительность и количество ворнингов

6.3. Резюме

TODO Что получилось

ЗАКЛЮЧЕНИЕ

TODO Ну заключение

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Duffy Joe. On partially-constructed objects [Электронный ресурс]. — URL: http://joeduffyblog.com/2010/06/27/on-partiallyconstructed-objects/ (дата обращения: 18.05.2022).
2. Mikhajlov Leonid, Sekerinski Emil. A study of the fragile base class problem // ECOOP'98 — Object-Oriented Programming / Ed. by Eric Jul. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. — P. 355–382.
3. Qi Xin, Myers Andrew C. Masked Types for Sound Object Initialization // SIGPLAN Not. — 2009. — jan. — Vol. 44, no. 1. — P. 53–65. — URL: https://doi.org/10.1145/1594834.1480890.
4. Summers Alexander J., Mueller Peter. Freedom before Commitment: A Lightweight Type System for Object Initialisation // SIGPLAN Not. — 2011. — oct. — Vol. 46, no. 10. — P. 1013–1032. — URL: https://doi.org/10.1145/2076021.2048142.
5. A Type-and-Effect System for Object Initialization / Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis et al. // Proc. ACM Program. Lang. — 2020. — nov. — Vol. 4, no. OOPSLA. — 28 p. — URL: https://doi.org/10.1145/3428243.

Приложение

ЛИСТИНГИ

Листинг 1. Код на Java

```
class Hello {
    fun foo() = name

    val nameLength = foo().length
    val name = "Alice"
}
```