# Оглавление

# Введение

Чтобы выдерживать серьёзную конкуренцию на рынке программного обеспечения (ПО), необходимо разрабатывать всё более сложные, расширяемые и поддерживаемые программы с минимальным количеством ошибок. Поэтому разработка не обходится без использования сторонних библиотек и фреймворков, тщательного продумывания архитектуры и используемых абстракций. Эти все инструменты вбирают в себя часть сложности и делают программу доступной для человеческого понимания, а значит, развития.

Однако не стоит забывать, что также немаловажную роль в успешности программного обеспечения играет выбор языка программирования (ЯП). Он способен существенно упростить разработку за счёт предоставления достаточного количества разумно организованных возможностей. Так, если ЯП реализует в себе популярные шаблоны программирования, у программиста не будет необходимости писать их вручную, рискуя допустить ошибку и чрезмерно усложнить программный интерфейс. Также ЯП может предоставлять удобные средства для построения абстракций и контроля за корректностью кода, например, за счёт системы типов.

В данной работе рассматривается добавление Self-типов в промышленный язык Kotlin, позволяющих облегчить реализацию некоторых популярных шаблонов программирования путём уменьшения количества необходимого кода, дополнительного типового контроля и упрощения программных интерфейсов. Для этого, во-первых, в качестве мотивации были приведены многочисленные примеры прикладных сценариев использования Self-типов, в которых другие языковые решения оказываются менее оптимальными. Во-вторых, были проанализированы теоретические результаты, из которых становятся понятны случаи нарушения безопасности системы с Self-типами. В третьих, были рассмотрены существующие академические и практические решения по обеспечению безопасности системы с Self-типами. В четвёртых, Self-типы были безопасным образом интегрированы в систему типов языка Kotlin на основании опыта других решений. И наконец, Self-типы были реализованы в компиляторе kotlinc, а реализация была протестирована.

# 1. Анализ предметной области

В данной главе будет дан обзор системы типов языка Kotlin, на прикладных примерах будут описаны некоторые ограничения её выразительности и доступные решения. После будет введено понятие Self-типа и показаны его преимущества как потенциального решения обозначенных проблем. В завершение будут приведены приложения Self-типов в других задачах.

Для нашего изложения не принципиальна аккуратная формализация понятий объекта, наследования, подтипизации, полиморфизма с ограничениями и д.р. Они будут даны прикладным образом как описание релевантных данной работе возможностей языка Kotlin. Однако некоторые формализмы будут введены в главе 2 для демонстрации особенностей Self-типов, которые могут приводить к небезопасности системы типов (опр. 1.3). В данной работе не ставится задачи формального доказательства безопасности, но встраивание в систему типов языка Kotlin новой возможности заведомо безопасным образом (в соответствии с существующими результатами, приведёнными в главе 3), покрывая релевантные случаи использования, данные в разделе 1.4.

## 1.1. Система типов языка Kotlin

**Kotlin** — это объектно-ориентированный (ОО) промышленный язык программирования, поддерживающий компиляцию в Java bytecode, JavaScript и нативный код [10].

**Определение 1.1. Система типов** — это гибко управляемый синтаксический метод доказательства отсутствия в программе определенных видов поведения при помощи классификации выражений языка по разновидностям вычисляемых ими значений [16].

**Определение 1.2.** Тип $B$ является **подтипом** типа $A$ ($A$ в таком случае — **надтип** $B$), если произвольное выражение типа $B$ может быть безопасно использовано в позиции, в которой ожидается выражение типа $A$ [12, 16]. Обозначение $B <: A$. Если два типа $C$ и $D$ не связаны отношением подтипизации, это обозначается так: $C \not<: \not:> D$. $C$ не подтип $D$: $C \not<: D$.

**Определение 1.3. Безопасная (sound) система типов** — всякая программа без приведений типов, в которой во время исполнения может возникнуть ошибка типизации, отвергается безопасной статической проверкой типов.

Система типов языка программирования Kotlin обладает следующими основными свойствами [1]:

1. Статическая — проверка типов происходит на этапе компиляции;
2. Gradually typed [20] — система типов может накладывать более слабые ограничения, делегируя проверку типов времени исполнения программы для лучшей поддержки взаимодействия с кодом целевой платформы (реализуется в Kotlin с помощью **flexible types**);
3. Flow [15] — типизация значений в программе может зависеть от графа потока управления (реализуется в Kotlin через механизм **smart-casts**, см. 1.1.2);
4. Null-безопасная — вводится две вселенные типов: типы, которые содержат значение `null`, **nullable** (обозначаются знаком вопроса в конце записи типа), и которые `null` не содержат, **not-null**;
5. Без небезопасных неявных типовых конверсий;
6. С поддержкой параметрического полиморфизма (опр. 1.8) с ограничениями (1.1.3);
7. С номинальной подтипизацией — один тип является подтипом (опр. 1.2) другого только в случае явного указания программиста в виде наследования (1.1.1) или вариантности типовых параметров (1.1.3);
8. С общими верхним и нижним типами относительно отношения подтипизации — тип `Nothing` является подтипом любого типа, а `Any?` — супертипом любого типа;
9. Безопасная (опр. 1.3), за исключением свойства (2).

### 1.1.1. Наследование и виртуальный полиморфизм

Система типов языка Kotlin является **номинативной**. Это значит, что отношение подтипизации нужно указывать явно и оно не следует из структуры значений типа. Основным способом задания отношения подтипизации в Kotlin является **наследование**.

С помощью наследования же в Kotlin реализуется **виртуальный полиморфизм**. Он заключается в том, что по ссылке базового типа происходит вызов метода наследника, если в действительность это ссылка на объект наследника.

```
1  open class Base {
2      fun method() { println("Base") }
3  }
4
5  class Derived : Base() {
6      override fun method() { println("Derived") }
7  }
8
9  fun test() {
10     val base: Base = Derived()
11     base.method() // Печатает "Derived"
12 }
```

### 1.1.2. Smart-casts

**Определение 1.4. Базовый блок** — непрерывная последовательность инструкций, всегда выполняющихся последовательно.

**Определение 1.5. Граф потока управления программы (CFG — control flow graph)** — граф, составленный по программе из её базовых блоков. На рисунке 1 приведён пример графа потока управления.

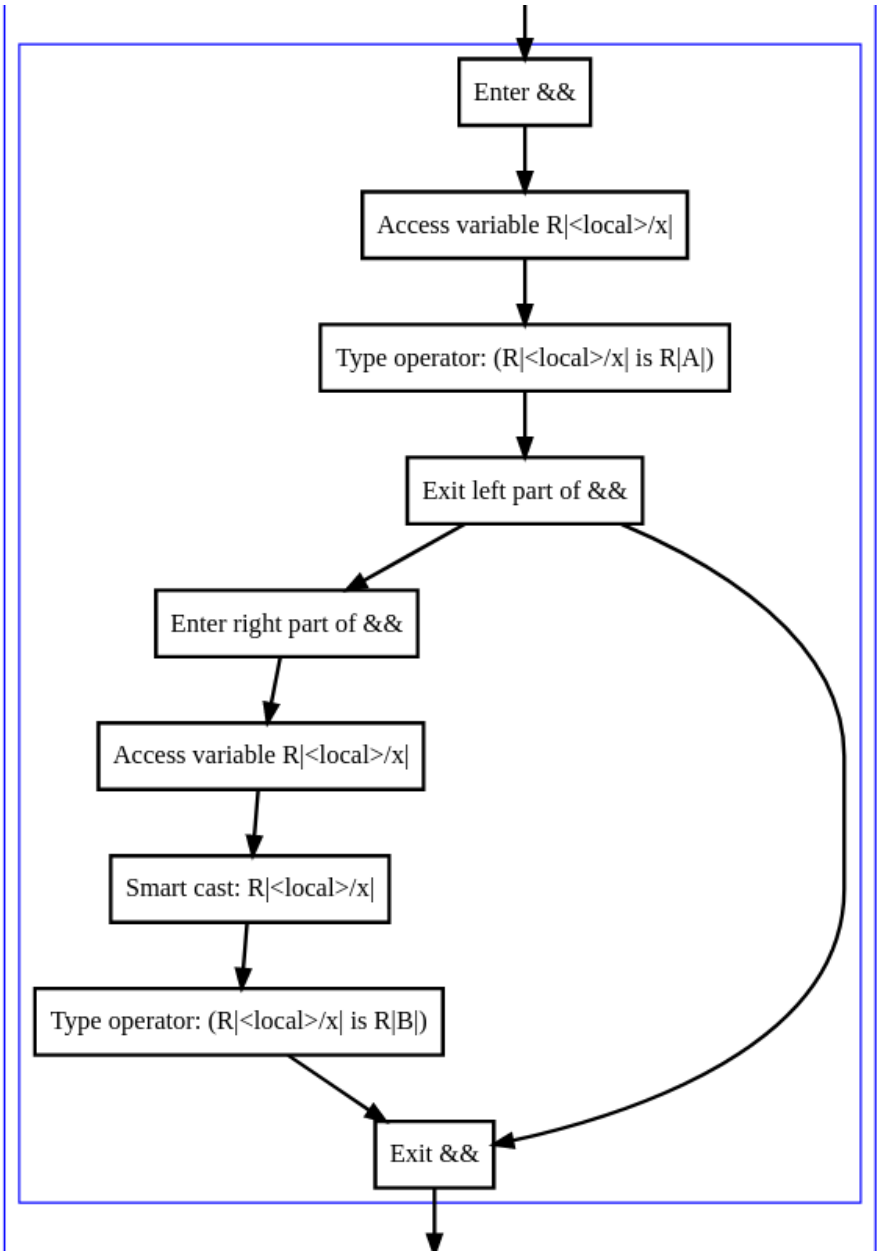


Рис. 1: Граф потока управления выражения `x is A && x is B`.

**Определение 1.6. Тип-пересечение** — тип, являющийся подтипом одновременно всех типов из пересечения. Обозначение для пересечения типов $A$, $B$ и $C$: $A \ \& \ B \ \& \ C$.

Типы-пересечения являются примером типов, которые нельзя записать в Kotlin, то есть они не доступны программисту, но возникают в процессе вывода типов.

```
1  interface A {
2      fun aMethod() { /* ... */ }
3  }
4
5  interface B {
6      fun bMethod() { /* ... */ }
7  }
8
9  class C : A, B
10 class D : A, B
11
12 fun test(cond: Boolean) {
13     val intersection /* : A & B */ = if (b) C() else D()
14     intersection.aMethod() // ok
15     intersection.bMethod() // ok
16 }
```

**Определение 1.7. Smart-casts** — механизм уточнения типа за счёт информации, получаемой из графа потока управления программы.

Например, если в условной конструкции проверяется принадлежность значения переменной типу, то в теле доступна информация о том, что тип этой переменной включает в себя тип из условия. Это выражается с помощью типов-пересечений (опр. 1.6).

```
1  fun test(x: A) {
2      if (x is B) {
```

```
3           // x: A & B
4           x.bMethod()
5       }
6   }
```

### 1.1.3. Параметрический полиморфизм и вариантность

**Определение 1.8. Параметрический полиморфизм** — способность функции или класса работать с объектами различных типов путём абстрагирования реализации по **типовому параметру**. Тип, подставляемый вместо типового параметра — **типовой аргумент**.

**Определение 1.9. Параметрический полиморфизм с поддержкой ограничений** — параметрический полиморфизм, поддерживающий добавление ограничений на типы, которые можно использовать как типовые аргументы. В случае Kotlin, данным ограничением является требование на наличие у типа определённого супертипа.

**Определение 1.10. Вариантность** — механизм языков с параметрическим полиморфизмом, позволяющий дополнять отношение подтипизации между параметризованными типами, когда типовые аргументы — разные типы, связанные в свою очередь отношением подтипизации.

В Kotlin вариантность может сообщаться типовому параметру класса как на стороне декларации, так и в месте использования этого класса. В нашем изложении мы ограничимся только первым случаем, второй рассматривается аналогично.

**Определение 1.11.** Возвращаемый тип функции будем называть типом в **исходящей позиции**. Тип параметра функции будем называть типом во **входящей позиции**. Тип, выступающий ковариантным типовым аргументом типа во входящей позиции, будем называть типом во *входящей ковариантной позиции*. Аналогично для исходящей позиции и других вариантностей.

**Определение 1.12.** Типом в **положительной или ковариантной позиции** будем называть тип в исходящей позиции, исходящей ковариантной позиции или входящей контравариантной позиции[1]. Типом в **отрицательной или контравариантной позиции** будем называть тип во входящей позиции, во входящей ковариантной позиции и в исходящей контравариантной позиции.

**Определение 1.13. Инвариантный типовой параметр** — может использоваться в произвольных позициях деклараций методов класса. Не дополняет отношение подтипизации между параметризованными типами.

```
1  interface Inv<T> {
2      fun id(x: T): T
3  }
4
5  // ∀A, B. Inv⟨A⟩ ≮:≯: Inv⟨B⟩
6  fun test(b: Inv<B>) {
7      // Отношение подтипизации не задано
8      val a: Inv<A> = b // ошибка компиляции
9  }
```

**Определение 1.14. Ковариантный типовой параметр** — может использоваться в положительных позициях деклараций методов класса. Параметризованный тип образует такое же отношение подтипизации, что и типовые аргументы.

```
1  interface Out<out T> {
2      fun produce(): T
3  }
4
5  // B <: A ⟺ Out⟨B⟩ <: Out⟨A⟩
```

[1]Определение типовых позиций даётся с практической точки зрения и не в самом общем случае, однако этого достаточно для данного изложения.

```
6  fun test(b: Inv<B>) {
7      val a: Inv<A> = b
8  }
```

**Определение 1.15. Контравариантный типовой параметр** — может использоваться в отрицательных позициях деклараций методов класса. Параметризованный тип образует обратное отношение подтипизации относительно типовых аргументов.

```
1  interface In<in T> {
2      fun consume(x: T)
3  }
4
5  // B <: A ⟺ In⟨A⟩ <: In⟨B⟩
6  fun test(a: Inv<A>) {
7      val b: Inv<B> = b
8  }
```

### 1.1.4. Ресиверы в Kotlin

**Определение 1.16.** Каждая функция, объявленная как метод или функция-расширение имеет специальные параметры (один или более), называемые **ресиверами** [1]. Такие параметры доступны как `this` в теле функции.

Далее в зависимости от контекста будем произвольно называть ресиверами такие специальные параметры, а так же объекты, используемые как соответствующие этим параметрам аргументы. Например, в вызове `"a".plus("b")` строка `"a"` является объектом-ресивером для функции `plus`.

Ресиверы в Kotlin могут быть трёх видов: dispatch, extension и context-ресиверы.

**Определение 1.17. Dispatch-ресивер** — ресивер, по которому происходит виртуальная диспетчеризация вызова.

```
interface Base {
    fun greet()
}

class Derived : Base {
    override fun greet() {
        println("Hello")
    }
}

val b: Base = Derived();
// b - dispatch-ресивер вызова
b.greet() // Печатает "Hello"
```

**Определение 1.18. Extension-ресивер** — ресивер, который является лишь специальным аргументом функции, но по которому не происходит виртуальной диспетчеризации вызова.

```
fun String.twice() = this + this
// функционально аналогично определению
// fun twice(String s) = s + s

println("Y".twice()) // Печатает "YY"
```

**Определение 1.19. Context-ресивер** — ресивер, аналогичный extension-ресиверу (опр. 1.18), только для него не доступен синтаксис вызова функции явно через точку (не может быть явным).

В дальнейшем изложении если не уточняется вид ресивера, имеется в виду dispatch или extension ресивер.

### 1.1.5. Область видимости типа

**Определение 1.20. Область видимости типа** — множество функций, для которых объект данного типа может быть использован как ресивер.

Например, функция `plus` содержится в области видимости типа `String`:

$$(plus : String.(String) \to String) \in scope(String)$$

## 1.2. Описание проблемы и возможных решений

На данный момент система типов Kotlin недостаточно выразительна. Так, в ней нет способа непосредственно сослаться на тип времени исполнения объекта-ресивера, на котором вызывается функция. В то же время такая необходимость нередко возникает. И при этом существующие решения, как правило, требуют либо существенного усложнения программных интерфейсов, либо написания большого объёма подверженного ошибкам рутинного кода, а также зачастую ограничивают возможности по обобщению и переиспользованию кода. Многие из этих проблем решаются с помощью специальной поддержки в языке возможности записать тип времени исполнения ресивера функции.

В этом разделе мы рассмотрим некоторые наиболее распространённые случаи, на которых видна обозначенная нехватка выразительности системы типов Kotlin, а так же приведём возможные решения и укажем их недостатки.

### 1.2.1. Шаблон «Строитель»

Классическим примером, когда тип возвращаемой функции должен совпадать с типом ресивера, на котором она вызвана, является шаблон программирования «Строитель».

Допустим, что у нас есть базовый класс для любого транспортного средства и строитель для него:

```
1  open class Vehicle(val speed: Int)
2
```

```
3  open class VehicleBuilder {
4      protected lateinit var speed: Int
5
6      open fun setSpeed(speed: Int): VehicleBuilder {
7          this.speed = speed
8          return this
9      }
10
11     open fun build() = Vehicle(speed)
12 }
```

Заведём наследника — машину, имеющего своего строителя. Чтобы переиспользовать функциональность строителя базового класса, унаследуем строителя наследника от него. Однако возвращаемый тип метода `setSpeed` — всё ещё строитель базового класса, что не даёт возможность собирать объект машины в виде цепочки вызовов вида `builder.setBase(1).setDerived(24)`. Чтобы добиться такого синтаксиса, добавим переопределяющий метод с более специфичным возвращаемым типом и делегируем в нём вызов методу базового класса:

```
1  class Car(speed: Int, val make: String) : Vehicle(speed)
2
3  class CarBuilder : VehicleBuilder() {
4      private lateinit var make: String
5
6      override fun setSpeed(speed: Int): CarBuilder =
7          super.setSpeed(speed) as CarBuilder
8
9      fun setMake(make: String): CarBuilder {
10         this.make = make
11         return this
12     }
13
```

```
14      override fun build() = Car(speed, make)
15  }
16
17  fun testBuild(): Car = CarBuilder()
18      .setSpeed(1)      // : CarBuilder
19      .setMake("some") // : CarBuilder
20      .build()
```

Видим, что нам потребовалось написать ещё один метод с тривиальной реализацией и, более того, воспользоваться явным приведением типов.

Однако в Kotlin шаблон «Строитель» принято выражать с помощью анонимной функции с ресивером. Так отпадает необходимость в записывании цепочек вызовов аналогичных приведённой выше.

```
1  fun buildCar(block: CarBuilder.() -> Unit) =
2      CarBuilder().apply { block() }.build()
3
4  fun testBuild() = buildCar { x = 1; y = 2 }
```

Тем не менее заметим, что для реализации второго подхода класс строителя должен быть изменяемым.

### 1.2.2. Неизменяемые объекты

Неизменяемые объекты играют значимую роль в современном программировании и являются основой распространённого функционального стиля программирования. Важной задачей при написании кода является сокращение количества сущностей, которыми приходится единовременно оперировать, так как кратковременная память человека ограничена [13]. Неизменяемость сущностей же позволяет не отслеживать постоянно их состояние (которое по определению не может измениться). Так, для получения каждой модификации необходимо создавать новый объект. Возможность же программировать таким образом обеспечивает развитие современных компьютеров и техник сборки мусора [11].

Поэтому использование неизменяемых объектов считается хорошей практикой, а значит, язык должен предоставлять достаточное количество инструментов для простого и безопасного программирования с их использованием. Однако при выстраивании иерархии неизменяемых объектов проблематичным оказывается использование метода базового класса, возвращающего объект ресивера или его копию:

```
1  sealed interface Data {
2      data class One(val a: Int) : Data
3      data class Two(val a: Int, val b: Int) : Data
4
5      fun update(a: Int): Data = when (this) {
6          is One -> One(a)
7          is Two -> Two(a, b)
8      }
9  }
10
11 fun test() {
12     val a = Data.Two(1, 2)
13     val b: Data.Two = a.update(a = 2) // : Data, ошибка
14 }
```

Как мы увидим далее (см. 1.2.3), на данный момент в Kotlin не существует простых и удобных способов добиться корректности типизации приведённого кода.

Такими объектами могут быть как простые неизменяемые структуры данных, так и, например, парсер-комбинаторы [9], содержащие в иерархии наследования различные методы порождения более сложных парсеров через комбинирование базовых друг с другом.

### 1.2.3. Персистентные коллекции

Важным частным случаем неизменяемых объектов являются персистентные коллекции, которые на каждую модификацию возвращают новую ссыл-

ку на модифицированную коллекцию, в то время как коллекция, доступная по старой ссылке, остаётся неизменной [14]. Рассмотрим различные потенциальные способы задания интерфейсов библиотеки персистентных коллекций, используя существующие возможности языка Kotlin, а так же укажем недостатки этих подходов.

### Типовые параметры с рекурсивными ограничениями

Во-первых, требуемого можно добиться с помощью ковариантных типовых параметров с рекурсивными ограничениями. Так, метод `add` должен вернуть новую коллекцию того же типа, но с добавленным элементом. Обозначим это так, что `add` возвращает типовой параметр, представляющий собой конкретного наследника этого интерфейса, на котором вызывается метод.

```
1  interface PCollection<out E, out S : PCollection<E, S>> {
2      fun add(value: @UnsafeVariance E): S
3  }
4
5  interface PList<out E, out S : PList<E, S>> : PCollection<E, S> {
6      fun listSpecific()
7  }
8
9  class PListImpl<out E> : PList<E, PListImpl<E>> {
10     /* ... */
11 }
12
13 fun <T, L> test(xs: L, x: T) where L : PList<T, L> {
14     xs.add(x) /* : L */ .listSpecific()
15 }
```

У данного подхода можно выделить следующие основные недостатки:

- Паттерн рекурсивного ограничения распространяется по всему коду.

- В случае определения метода в любом месте иерархии, кроме финального класса, требуется явное приведение типов: `this as S`.
- Типовой параметр `S` может быть только однажды фиксирован в иерархии и более не может быть уточнён в наследниках.
- Такой подход является нетривиальным для понимания, особенно начинающими разработчиками.

### Abstract override методы

Альтернативой может быть использование **abstract override методов**. Теперь дополнительный типовой параметр не требуется, однако в каждом наследнике нужно добавить переопределяющий метод с более специфичным возвращаемым типом.

```
1  interface PCollection<out E> {
2      fun add(value: @UnsafeVariance E): PCollection<E>
3  }
4
5  interface PList<out E> : PCollection<E> {
6      abstract override fun add(value: @UnsafeVariance E): PList<E>
7      fun listSpecific()
8  }
9
10 class PListImpl<out E> : PList<E> {
11     /* ... */
12 }
13
14 fun <T> test(xs: PList<T>, x: T) {
15     xs.add(x) /* : PList<T> */ .listSpecific()
16 }
```

Видим, что теперь паттерн рекурсивного ограничения исчез, однако добавилась вспомогательная декларация. У такого подхода можно выделить

следующие недостатки:

- Существенное количество рутинного кода: подобные декларации с более специфичными возвращаемыми типами требуется написать в теле каждого наследника.
- Нет контроля со стороны компилятора, что программист не забыл добавить подобные декларации во всех наследников при добавлении нового метода в базовый класс, что особенно чувствительно для разработчиков библиотек.
- Этот подход имеет место только для возвращаемых типов, типы параметров методов же должны совпадать по правилу переопределения функций базовых классов.

Несмотря на обширный список недостатков, подход с abstract override методами используется, например, в библиотеке персистентных коллекций языка Kotlin `kotlinx.collection.immutable`[2].

### Ассоциированные типы

**Определение 1.21. Ассоциированные типы** [2] — это возможность объявить в базовом классе тип, определение которого будет дано только в реализации.

Ассоциированные типы присутствуют в большом количестве языков без наследования, реализующих специальный полиморфизм через классы типов или трейты, например, Haskell и Rust. Однако среди языков с наследованием ассоциированные типы встречаются не часто, например, в Swift и Scala.

Рассмотрим пример использования ассоциированных типов для интерфейсов персистентных коллекций на языке Scala, где ассоциированные типы носят название **abstract type members**[3].

[2] `https://github.com/Kotlin/kotlinx.collections.immutable`
[3] `https://docs.scala-lang.org/tour/abstract-type-members.html`

```
1  trait PCollection[T] {
2    type S
3    def add(x: T): S
4  }
5
6  trait PList[T] extends PCollection[T] {
7    type S <: PList[T]
8    def listSpecific: Unit
9  }
10
11 class PListImpl[T] extends PList[T] {
12   type S = PListImpl[T]
13   override def add(x: T): PListImpl[T] = // ...
14   override def listSpecific: Unit = // ...
15 }
```

Обозначим основные недостатки данного подхода:

- Заметим, что аналогично подходу с abstract override методами в каждом абстрактном наследнике требуется уточнить ассоциированный тип.
- Также нет контроля со стороны компилятора, что это утонение не было забыто.
- Определение ассоциированного типа должно быть дано либо в абстрактном классе, либо в первом не абстрактном классе в иерархии. Дальнейшее уточнение ассоциированного типа ниже по иерархии наследования невозможно.
- В реализациях методов, находящихся выше по иерархии наследования, чем определение ассоциированного типа, требуется явное приведение типов.

Более того, реализация ассоциированных типов является очень нетривиальной задачей, если позволять вызывать методы, содержащие в сигнатуре

ассоциированный тип, на экзистенциальном типе (см. 2.4), так как нельзя полагаться, что тип, ассоциированный с одним значением, равен типу, ассоциированному с другим. Поэтому приходится отслеживать, откуда пришло значение ассоциированного типа, и вместо установления подтипизации между двумя типами, сравнивать пути получения значений (значение может глубоко вложено как поле в композиции объектов). Как правило, языки запрещают использовать такие методы на объектах экзистенциальных типов (Swift, Rust), а это почти все типы в OO языках, что существенно снижает выразительность данного подхода.

Также стоит отметить, что на момент написания данной работы запроса от пользователей на добавление ассоциированных типов в Kotlin нет.

## 1.3. Self-типы

Как мы увидели выше, на данный момент в Kotlin не существует удобного способа сослаться на тип времени исполнения объекта-ресивера. Тем не менее у этой проблемы существует каноническое решение — Self-типы.

**Определение 1.22. Self-тип** — тип ресивера (опр. 1.16), на котором вызывается функция (тип `this`'а).

Self-типы встречаются так же как **рекурсивные типы** и **This-типы** [17]. А под названием Self-типы может подразумеваться нечто совершенно другое, как, например, в языке Scala[4].

Пусть у нас есть базовый класс, в котором объявлен метод с `Self` в качестве возвращаемого типа. Тогда при вызове этого метода базового класса на объекте наследника, мы сможем воспользоваться результатом как объектом типа наследника. Вернёмся к примеру интерфейсов библиотеки персистентных коллекций (стр. 17 и 18). Легко видеть, что вариант с Self-типами лишен перечисленных ранее недостатков.

```
1  interface PCollection<out E> {
2      fun add(value: @UnsafeVariance E): Self
```

[4] https://docs.scala-lang.org/tour/self-types.html

```
3  }
4
5  interface PList<out E> : PCollection<E> {
6      fun listSpecific()
7  }
8
9  class PListImpl<out E> : PList<E> {
10     /* ... */
11 }
12
13 fun <T> test(xs: PList<T>, x: T) {
14     xs.add(x) /* : PList<T> */ .listSpecific()
15 }
```

Как было показано выше, Self-типы могут быть эмулированы с помощью дополнительных явных приведений типов и существенного количества рутинного кода. Отсутствие поддержки Self-типов в языке и сложность их эмуляции может в некоторых случаях заставлять принимать менее удачные решения при разработке, выбирая подходы, не требующие Self-типов. Таким образом, учитывая многочисленные запросы сообщества[5][6] и предоставляемое Self-типами удобство во многих ситуациях, было решено провести эксперимент по введению Self-типов в типовую систему языка Kotlin.

## 1.4. Приложения Self-типов

Self-типы могут быть полезны для реализации ещё нескольких популярных подходов и шаблонов программирования. Перечислим некоторые из них в этом разделе.

### 1.4.1. Рекурсивные структуры данных

Self-типы нужны и для реализации рекурсивных структур данных, узлы которых могут состоять в иерархии наследования. Допустим, что у нас

[5] https://discuss.kotlinlang.org/t/self-types/371
[6] https://discuss.kotlinlang.org/t/this-type/1421

есть класс, представляющий вершину дерева, и у него есть наследник, представляющий вершину с дополнительной функциональностью. В таком случае Self-типы могут контролировать гомогенность типов вершин, а также позволяют вызывать методы наследника на результатах методов базового класса.

```
1  abstract class Node<out T>(
2      val value: T, val children: List<Self>
3  )
4
5  class BetterNode<out T>(
6      value: T, children: List<Self> = emptyList()
7  ) : Node<T>(value, children) {
8      fun betterSpecific() = println(value)
9  }
10
11 fun test() {
12     val betterTree = BetterNode(value = 2, children =
13         listOf<BetterNode<Int>>(
14             BetterNode(1, listOf(BetterNode(0))),
15             BetterNode(4, listOf(BetterNode(3), BetterNode(5)))))
16     betterTree.children
17         .flatMap { it.children }
18         .forEach { it.betterSpecific() } // Печатает "0 3 5"
19 }
```

### 1.4.2. Шаблон «Абстрактная фабрика»

В случае если требуется типизированным образом получать по элементу, созданному с помощью фабрики, её саму, Self-типы оказываются полезны в исходящей ковариантной позиции.

```
abstract class Element<out F : Factory>(val factory: F)

interface Factory {
    fun create(): Element<Self>
}

abstract class SpecificFactory : Factory {
    abstract fun doSpecific()
}

fun <F : SpecificFactory> test(element: Element<F>) {
    entity.factory.doSpecific()
}
```

### 1.4.3. Шаблон «Наблюдатель»

С помощью Self-типов можно абстрагировать логику хранения и оповещения наблюдателей. В этом примере Self-тип используется во входящей контравариантной, то есть в положительной (ковариантной), позиции (опр. 1.11).

```
abstract class AbstractObservable {
    private val observers = mutableListOf<(Self) -> Unit>()

    fun observe(observer: (Self) -> Unit) {
        observers += observer
    }

    private fun notifyObservers() {
        observers.forEach { observer ->
            observer(this)
        }
```

```
12      }
13  }
14
15  class Entity : AbstractObservable {
16      var color: Color = Color.Purple
17          set(new: Color) {
18              field = new
19              notifyObservers()
20          }
21  }
22
23  fun observer(entity: Entity) {
24      println("New: ${it.color}")
25  }
26
27  fun test() {
28      val entity = Entity()
29      entity.observe(::observer)
30      entity.color = Color.Blue // Печатает "New: Color.Blue"
31  }
```

### 1.4.4. Алгебры

Ещё одним классическим применением Self-типов является кодирование алгебраических структур. Так, интерфейс полугруппы мог бы выглядеть следующим образом.

```
1  interface Semigroup {
2      infix fun add(other: Self): Self
3  }
```

Однако уже, например, моноид с одним константным символом таким образом уже задать не получится, так как любой метод всегда принимает

как минимум один аргумент (**this**). Более того, как мы увидели выше в 1.3, использовать Self-типы во входящей позиции небезопасно.

В современном Kotlin существует другой механизм, который больше подходит для описания алгебр, — контекстные ресиверы (опр. 1.19).

```
interface Semigroup<S> {
    infix fun S.add(other: S): S
}

interface Monoid<M> : Semigroup<M> {
    val empty: M
}

object StringMonoid : Monoid<String> {
    override val empty: String = ""
    override fun String.add(other: String): String = this + other
}

context(Monoid<T>)
fun <T> concat(vararg xs: T): T =
    xs.fold(empty) { acc, x -> acc add a }

fun test() {
    with(StringMonoid) {
        println(concat("a", "bc", "d")) // Печатает "abcd"
    }
}
```

## 1.5. Цель и задачи работы

Таким образом, можем сформулировать цель и задачи данной работы.

**Цель:** Разработать дизайн Self-типов для языка Kotlin на основании опыта других языков и реализовать поддержку Self-типов в компиляторе kotlinc.

**Задачи:**

1. Выделить особенности существующих решений: возможные значения Self-типа, допустимые позиции Self-типа и меры по обеспечению безопасности системы типов;
2. Интегрировать Self-типы в типовую систему языка Kotlin на основании проведённого анализа решений;
3. Реализовать поддержку Self-типов в компиляторе kotlinc.

# 2. Формализация Self-типов

В этой главе мы рассмотрим некоторое формальное представление типов объектов и наследования. Это, во-первых, позволит понять, с какими сложностями может быть связано добавление Self-типов в язык. Во-вторых, даст возможность далее описать существующие решения обнаруженных проблем (глава 3), многие из которых оснащены доказательствами безопасности предлагаемых систем типов. И наконец, мы сможем, быть более уверенными в интеграции Self-типов в Kotlin (глава 4), благодаря выстраиванию соотношения технических решений с формальными верифицированными построениями.

Существует множество различных попыток формализации объектно ориентированного программирования как в рамках лямбда-исчисления, так и в виде отдельных исчислений [16]. Однако выбор конкретного формализма становится важен лишь в случае намерения производить формальные рассуждения, что не является задачей данной работы. Мы не будем перегружать описание строгостью, цель данной главы — продемонстрировать идею построения такого рода формализмов.

## 2.1. Записи в $\lambda$-исчислении

Возьмём классическое просто типизированное лямбда-исчисление (STLC [16]) и расширим его сперва записями.

**Терм записи** имеет вид

$$\{x_1 = v_1, \ldots, x_n = v_n\}$$

где $x_i$ — идентификаторы полей, а $v_i$ — термы, значения полей.

**Терм доступа к полю** имеет вид $t.x_i$, где $t$ — терм, а $x_i$ — идентификатор поля, к которому производится доступ.

**Тип терма записи имеет вид**

$$\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$$

где $x_i$ — идентификаторы полей, а $\sigma_i$ — типы значений полей.

**Правило (структурной) подтипизации для типов записей** записывается следующим образом:

$$\frac{\sigma_1 <: \tau_1 \quad \ldots \quad \sigma_k <: \tau_k}{\{x_1 : \sigma_1, \ldots, x_k : \sigma_k, \ldots, x_n : \sigma_n\} <: \{x_1 : \tau_1, \ldots, x_k : \tau_n\}} \tag{1}$$

В формальной модели мы будем оперировать структурной подтипизацией вместо номинативной, так как нас интересует внутренняя структура объектов и возможность использовать один объект вместо другого (см. опр. 1.2).

Существуют различные кодировки типов объектов в лямбда-исчислении. Большинство из них так или иначе опираются на типы записей, имея различие в том, принимают ли методы явно ссылку на объект, или уже содержат её в замыкании. В нашем изложении это не играет принципиальной роли, поэтому для краткости записи будем предполагать второе. Тогда объект точки, содержащий координату и мутирующий метод перемещения можно протипизировать следующим образом:

$$\{x : int, move : int \to unit\} \tag{2}$$

Будем считать тип **типом объекта наследника**, если он содержит все те же метки базового типа с не менее специфичными соответствующими типами, а также, может быть, новые метки.

## 2.2. Рекурсивные типы

Объект точки в предыдущем примере был изменяемым. Тип чисто функционального объекта точки мог бы выглядеть следующим образом (добавим так же метод сравнения на равенство, пригодится в дальнейшем):

$$P = \{x : int, move : int \to P, eq : P \to bool\}$$

Но поскольку мы выбрали структурную типизацию, мы не можем в правой части использовать то же имя — так мы задаём бесконечное типовое дерево.

Чтобы получить конечный тип, используется $\mu$-нотация:

$$P = \mu t.\, \{x : int, move : int \to t, eq : t \to bool\} \tag{3}$$

Воспользоваться значением $\mu$-типа можно с помощью **правила развёртки** [8]:

$$\frac{U = \mu t.\, T \quad \Delta \vdash o : U}{\Delta \vdash unfold(o) : [t \mapsto U]T}\ \text{Unfold} \tag{4}$$

Так, чтобы получить смещённую точку относительно точки $p : P$ требуется применить правило Unfold:

$$unfold\ p : \{x : int, move : int \to P, eq : P \to bool\} \tag{5}$$

$$(unfold\ p).move\ 42 : P \tag{6}$$

Нетрудно видеть, что связанные в $\mu$-нотации типы — это не что иное, как Self-типы.

Мы пользуемся вариантом **изорекурсивных типов** (с явным применением развёртки через $unfold$) вместо **эквирекурсивных** из-за большей наглядности и простоты в реализации [16].

## 2.3. Подтипизация рекурсивных типов

Классическое определение подтипизации между изорекурсивными типами было предложено в языке Amber [7]:

$$\frac{t \neq u \quad t, u \notin freeVars(\Delta) \quad \Delta, t <: u \vdash \tau <: \sigma}{\Delta \vdash \mu t.\, \tau <: \mu u.\, \sigma}\ \text{Amber} \tag{7}$$

Возьмём объект-наследник $p'$ точки $p : P$ (3), содержащий метод вычисления расстояния до другой точки. Он будет иметь следующий тип:

$$P' = \mu t.\, \{x : int, move : int \to t, eq : t \to bool, dist : t \to int\} \tag{8}$$

Однако нетрудно видеть, что $p' : P'$ является наследником $p : P$, но не является его подтипом [8]. Это связано с тем, что в методе $eq$ рекурсивный тип присутствует в контравариантной позиции (слева от функциональной

стрелки), а значит, его тип в $p'$ будет супертипом типа $eq$ в первом объекте, а не подтипом, как того требует правило (7).

Таким образом, Self-тип безопасно использовать только в ковариантных позициях декларациях методов класса (опр. 1.12). Поэтому в языках с наследованием требуется предпринимать специальные меры по обеспечению безопасности системы типов, так как наследник, как правило, автоматически считается подтипом. Заметим, что для языков без наследования (таких, как Rust, Haskell и т.д.) обозначенная проблема не актуальна.

## 2.4. Экзистенциальные типы

Экзистенциальные типы используются как механизм построения абстракций путём сокрытия деталей конкретного типа или всего типа целиком. Как правило, в ОО языках если переменная имеет статический тип $U$, то во время исполнения её значение может иметь тип любого подтипа $U$ (**неточные типы, inexact types**). В нотации экзистенциальных типов тип времени исполнения можно записать следующим образом:

$$\exists X <: U.\, X$$

Иногда вместо отношения $<:$ используются какие-то другие, нас будут интересовать бинарные асимметричные отношения, обозначим их как $\blacktriangleleft$.

Вспомним, что типом объекта мы считаем рекурсивный тип записи, то есть тип $U$ может иметь вид $U = \mu x.\, T$. Чтобы воспользоваться таким объектом требуется определить правило развёртки для экзистенциального типа с рекурсивным ограничением. Сделаем это консистентно обычному правилу развёртки (4):

$$\frac{U = \mu t.\, T \quad U' = \exists X \blacktriangleleft U.\, X \quad \Delta \vdash o : U'}{\Delta \vdash unfold(o) : [t \mapsto U']T} \text{ Unfold-Ex} \tag{9}$$

# 3. Анализ существующих реализаций Self-типов

Self-типы привлекают много внимания со стороны исследователей объектно ориентированного программирования, так как возникающее несоответствие наследования и подтипизации является вызовом для любого формализма [8]. Также, как было показано в разделе 1.4, Self-типы имеют множество полезных приложений. Поэтому не удивительно, что Self-типы уже присутствуют во многих промышленных языках программирования.

В этой главе мы рассмотрим различные существующие подходы к реализации Self-типов в объектно ориентированных языках, чтобы выбрать для Kotlin наиболее удачные и, вместе с тем, безопасные решения. Нас будут в первую очередь интересовать возможные значения Self-типа, позиции, в которых тот или иной язык позволяет использовать Self-типы, а также на меры, предпринимаемые языком для сохранения безопасности системы типов (опр. 1.3).

## 3.1. Академические подходы

Основным вопросом, занимающим исследователей, является возможность безопасного использования Self-типов в контравариантных позициях, а именно — в **бинарных методах** (методы, принимающие один параметр Self-типа). Для этого было придумано множество подходов, рассмотрим основные из них.

Как было показано, если Self-тип встречается в методах класса в контравариантных позициях, то наследник не является подтипом [8]. Но существует другое отношение, отличное от подтипизации, которое тем не менее сохранятся. Это отношение называют, например, **matching** [4]. Если тип $A$ находится в отношении match с $B$ ($A \trianglelefteq B$), то все сообщения, которые можно послать объекту типа $B$, можно послать и объекту типа $A$. Даже разрабатывались объектно ориентированные языки, полностью основанные на этом отношении, не содержащие подтипизации [6]. Однако matching — существенно менее интуитивно понятное отношение, чем подтипизация, что приводило к разработке других отношений [17].

Другим подходом является введение отдельной вселенной **точных типов (exact types)**, статический тип которых совпадает с типом значения во время исполнения (в противоположность неточным типам из 2.4). Бинарные методы разрешается вызывать только на точных типах, что фактически запрещает динамическую диспетчеризацию бинарных методов [5]. Этот подход требует дополнения системы типов выводом точных типов. Расширения множества допустимых программ можно добиться, вводя в язык вспомогательные возможности, такие как локальное уточнение [18], точные типовые параметры, именованные wildcard'ы, конструкция утверждения точности [17].

Также существует некоторое количество подходов по созданию новых объектов Self-типа. Сложность в том, чтобы гарантировать на этапе компиляции, что тип создаваемого объекта всегда является подтипом типа времени исполнения ресивера. Для этого вводят **ненаследуемые методы** (которые обязаны быть переопределены в каждом наследнике) [18], а так же **виртуальные конструкторы** [17].

## 3.2. TypeScript

В языке TypeScript [3] Self-типы присутствуют под названием «this-типы»[7]. У таких типов единственных обитатель — `this`.

Разработчики TypeScript не ставят задачи поддержки безопасности системы типов языка и this-типы можно использовать в произвольной позиции. Поэтому нетрудно написать код, например, с this-типом в контравариантной позиции, который проходит проверку типов, но при этом получающий без явных приведений типов значение `undefined`.

```
1  class Box {
2    content: string = "";
3    sameAs(other: this): boolean {
4      return other.content === this.content;
5    }
```

[7] https://www.typescriptlang.org/docs/handbook/2/classes.html#this-types

```
6  }
7
8  class DerivedBox extends Box {
9    otherContent: string = "?";
10   sameAs(other: this): boolean {
11     if (other.otherContent === undefined) {
12       console.log("Система типов TS небезопасна")
13     }
14     return other.otherContent === this.otherContent;
15   }
16 }
17
18 const base = new Box();
19 const derived = new DerivedBox();
20
21 function test(x: Box): boolean {
22   return x.sameAs(base)
23 }
24
25 test(derived) // Печатает: "Система типов TS небезопасна"
```

## 3.3. Python

Python [19] — изначально динамически типизированный язык программирования, в который последовательно добавляются возможности статической типизации. В том числе в версию языка 3.11 были введены Self-типы[8].

Также авторы дизайн-предложения Self-типов для Python приводят[8] интересную статистику, согласно которой паттерн типового параметра с рекурсивным ограничением (отчасти заменяемый Self-типами, как мы увидели выше в разделе 1.2.3) встречается в количестве 40% случаев от использования других популярных типов — `dict` и `Callable`.

Стратегия реализации Self-типов в Python заключается в превращении

[8] `https://peps.python.org/pep-0673/`

их обратно в `TypeVar` с рекурсивным ограничением[8].

Дизайн Self-типов в Python сходен с дизайном в TypeScript. Только значение `self` населяет Self-тип, его можно использовать в произвольной позиции без гарантий безопасности со стороны системы типов.

## 3.4. Java Manifold

Плагин Manifold к компилятору Java позволяет[9] проаннотировать тип аннотацией `@Self`.

В простой исходящей позиции аннотация `@Self` позволяет обойтись без abstract override методов (см. 1.2.3) и дополнительных приведений типов:

```
1  class VehicleBuilder {
2      /* ... */
3      public @Self VehicleBuilder withWheels(int wheels) {
4          _wheels = wheels;
5          return this;
6      }
7  }
8
9  class AirplaneBuilder extends VehicleBuilder {
10     /* ... */
11 }
12
13  Airplane airplane = new AirplaneBuilder()
14     .withWheels(2) // Возвращает AirplaneBuilder
15     .withWings(1)
```

Во входящей позиции тоже можно использовать аннотацию `@Self`. Это позволяет иметь некоторые дополнительные типовые ограничения:

[9] https://github.com/manifold-systems/manifold/tree/master/manifold-deps-parent/manifold-ext#the-self-type-with-self

```
class A {
    public boolean equals(@Self Object obj) {
        /* ... */
    }
}

A a = new A();
a.equals("строчка вместо объекта типа A"); // Ошибка компиляции
```

Однако эти ограничения легко обойти. Так, следующий код уже не отвергается системой типов. Поэтому классическая проверка `isinstance` в методе `equals` всё ещё необходима.

```
Object obj = a;
obj.equals("строчка вместо объекта типа A");
```

`@Self` можно безопасно использовать и в исходящей ковариантной позиции, например, для реализации рекурсивной структуры данных. Однако для входящей позиции всё ещё требуется проверка времени исполнения.

```
public class Node {
    private List<Node> children;

    public List<@Self Node> getChildren() {
        return children;
    }

    public void addChild(@Self Node child) {
        checkAssignable(this, child); // Необходима проверка
        children.add(child);
    }
}
```

```
13
14 public class MyNode extends Node {
15     /* ... */
16 }
17
18 MyNode myNode = findMyNode();
19 List<MyNode> = myNode.getChildren();
```

Manifold предоставляет возможность писать функции-расширения в Java, и в связи с этим, `@Self` может ссылаться на тип ресивера функции-расширения.

```
1 public static <K,V> @Self Map<K,V> add(
2         @This Map<K,V> thiz, K key, V value) {
3     thiz.put(key, value);
4     return thiz;
5 }
6
7 HashMap<String, String> map = new HashMap<>()
8     .add("nick", "grouper")
9     .add("miles", "amberjack");
```

## 3.5. Swift

Swift [21] — компилируемый промышленный язык программирования со строгой статической типизацией. Имеет безопасную поддержку Self-типов[10].

В классах Self-тип можно использовать только в простой исходящей позиции, значением Self-типа может быть только `self`.

Специальный полиморфизм поддерживается в Swift через механизм протоколов. Протоколы аналогичны Java интерфейсам, но не позволяют делать реализации методов по умолчанию. В протоколах Self-тип можно использовать в произвольной позиции.

[10] https://docs.swift.org/swift-book/documentation/the-swift-programming-language/types/#Self-Type

При реализации протокола классом, все Self-типы в позициях, отличных от простой исходящей должны быть заменены на тип текущего класса:

```
protocol P {
    func produce() -> Self
    func consume(_ x: Self)
}

class C: P {
    func produce() -> Self { return self }
    func consume(_ x: C) {}
}
```

Протоколы можно использовать как ограничения типовых параметров, тогда Self-тип подменяется на этот типовой параметр. Аналогично для `some Protocol` параметров, так как для каждого из них генерируется типовой параметр с соответствующим ограничением.

```
func testConstrait<T: P>(_ x: T) {
    x.consume(/* value of type T is expected */)
}
```

Также протоколы могут быть границами экзистенциальных типов: `any Protocol`. В таком случае вызов метода с Self-типом в позиции, отличной от простой исходящей запрещён:

```
func testAny(_ x: any P) {
    // error: member 'consume' cannot be used on value of type
    // 'any P'; consider using a generic constraint instead
    x.consume(/* ... */)
}
```

Self-типы в функциях-расширениях ссылаются на расширяемый тип.

# 4. Интеграция Self-типов в типовую систему языка Kotlin

Ранее в главе 2 были рассмотрены соображения, указывающие на потенциальную небезопасность неаккуратной реализации Self-типов. Далее в главе 3 были приведены различные существующие решения по безопасному внедрению Self-типов в языки. И теперь мы готовы к тому, чтобы интегрировать Self-типы в Kotlin, учитывая его специфику, и используя заведомо безопасные практики.

Ключевой задачей является сохранение безопасности системы типов вместе с поддержкой наибольшего количества полезных приложений Self-типов, описанных ранее в разделах 1.2 и 1.4. Так, сначала будут рассмотрены различные аспекты интеграции Self-типов в типовую систему языка Kotlin, а также предложены решения на основании опыта других языков. И наконец, будет описана экспериментальная реализация Self-типов в компиляторе kotlinc.

## 4.1. Синтаксис Self-типов

Чтобы добавить какой-либо синтаксис в язык, нужно сперва тщательно убедиться, что обратная совместимость исходных кодов не будет нарушена. Если же нарушение неизбежно, то разработать политику по миграции кодовых баз пользователей на новые версии языка. Подобные исследования не являются задачей данной работы. Поэтому мы будем вводить синтаксические конструкции вольным образом по необходимости. Так, в первую очередь будем считать, что идентификатор `Self` введён как ключевое слово языка.

## 4.2. Граница Self-типа

**Определение 4.1. Границей Self-типа** назовём наиболее общий тип ресивера, на котором может быть вызван соответствующий метод.

Границы будут помогать нам далее различать, откуда пришел тот или

иной Self-тип. Чтобы понять, например, является ли Self-тип подтипом другого типа (раздел 4.4). Во введённой ранее теории той же цели служила структура типа записи вместе с правилом (1).

Граница Self-типа совпадает с типом ресивера текущей декларации метода. В примере ниже для `Self` границей является `Base`, обозначение $Self(Base)$:

```
1  interface Base {
2      fun base(): Self = this
3  }
4
5  class Defived : Base {
6      fun derived() { /* ... */ }
7  }
8
9  fun test(d: Derived) = d.base() /* : Derived */ .derived()
```

Границей является либо ближайший dispatch-ресивер, либо ближайший extension-ресивер, если dispatch-ресивера нет. При использовании Self-типа в контексте с ресиверами обоих видов не очевидно, к какому именно он относится, поэтому этот случай использования имеет смысл запретить. Границей всегда является not-null тип (см. 1.1) для удобства дальнейших рассуждений. Так, в следующем примере границей Self-типа в декларации `dispatch` является `C`, `topLevel` — `B`, и в `nullable` — тоже `B`.

```
1  class C {
2      fun dispatch(): Self /* (C) */ = this
3      fun B.extension(): Self /* ошибка */ = this
4  }
5
6  fun B.topLevel(): Self /* (B) */ = this
7  fun B?.nullable(): Self? /* (B) */ = this // : Self(B)?
```

### 4.2.1. Указание границы Self-типа

Естественным образом возникает вопрос, можно ли вручную задавать границу Self-типа. Например, в синтаксисе меток, аналогичным меткам ссылок на ресиверы (`this@functionOrClass`).

```
class C {
    fun A.foo(): Self@C /* (C) */ {
        fun B.bar(): Self@C /* (C) */ = this@C
        fun B.baz(): Self@foo /* (A) */ = this@foo
        return B().bar() /* : Self(C) */
    }

    fun bar(c: C): C {
        with(c) {
            return A().foo() /* : C */
        }
    }
}
```

Другим применением меток Self-типа может быть использование их во вложенных классах:

```
class Outer {
    inner class Inner {
        fun getOuter(): Self@Outer = this@Outer
    }

    fun out(): Out<Self> = object : Out<Self> {
        override fun produce(): Self@Outer = this@Outer
    }
}
```

Однако не известны приложения явного указания границы, кроме случая функции `out` из примера при наличии поддержки Self-типов в позиции типового аргумента (позиции будут рассмотрены далее в 4.6).

### 4.2.2. Область видимости Self-типа

После смены типа ресивера $C$ внутри класса на $Self(C)$, введя Self-типы, должна быть возможность продолжать вызывать на **`this`** все те же методы, что и раньше. Поэтому область видимости Self-типа должна, хотя бы за вычетом методов с Self-типами ($selfs(C)$), совпадать с областью видимости его границы:

$$scope(C) \setminus selfs(C) = scope(Self(C)) \setminus selfs(C)$$

## 4.3. Материализация Self-типа

Чтобы воспользоваться значениями и методами, которые содержатся в записи с рекурсивным типом (см. 2.2), требуется применить правило развёртки рекурсивного типа (4) и (9), как было показано на примере (6).

Если мы позволим вызывать методы с Self-типами без каких-либо предварительных преобразований, это приведёт к небезопасности системы типов, так как Self-тип (связанный $\mu$-нотацией типовой параметр в нашей аналогии 2.2) окажется вне своего контекста и будет обозначать тип времени исполнения ресивера уже другого объекта:

```
1  abstract class A {
2      fun self(): Self = this // Имеет Self-тип по определению
3      fun unsafe(a: A): Self = a.self() // Пусть получили Self-тип
4  }
5
6  class B : A() {
7      fun bOnly() {}
8  }
9
```

```
10  fun test(a: A, b: B) {
11      b.unsafe(a) /* область видимости типа B */ .bOnly() // ошибка
12  }
```

Поэтому, чтобы пользоваться методами с Self-типом нам требуется ввести аналог правила развёртки.

**Определение 4.2. Материализация Self-типа** — подмена Self-типа в сигнатуре метода на тип ресивера в области видимости типа этого ресивера (опр. 1.20).

Материализация $Self(A)$ может происходить в произвольного подтипа типа границы $A$, так для примера выше $Self(A)$ материализуется в сам $A$ в его области видимости:

$$(self : A.() \to \boxed{\text{A}}) \in scope(A)$$

И если $A \not<: Self(A)$, этот код будет отвергаться системой типов, так как $a.self() : A$ (аккуратно отношение подтипизации для Self-типов будет введено позже в разделе 4.4).

Заметим, что в области видимости типа ресивера, `this`, (то есть Self-типа, например, $Self(B)$), Self-тип декларации ($Self(A)$) материализуется в $Self(B)$. Так как Self-тип не покидает контекста того же объекта, его значение безопасно использовать в позициях, где ожидается Self-тип. В то же время поскольку материализация может уточнить границу, следующий корректный код будет типизироваться:

```
1  abstract class A {
2      fun self(): Self = this
3      fun safeA(): Self /* (A) */ = this.self() // : Self(A)
4  }
5
6  class B : A() {
7      fun bOnly() {}
```

```
8       fun safeB(): Self /* (B) */ = this.self() // : Self(B)
9   }
10
11  fun test(b: B) {
12      b.safeA().bOnly()
13      b.safeB().bOnly()
14  }
```

Введённое правило материализации можно сопоставить правилу развёртки следующим образом. Будем считать Self-тип аналогом точного типа (так как он ссылается на рекурсивный тип записи), тогда развёртка для него выполняется по правилу (4), и вместо рекурсивной ссылки $t$ в типе записи появляется тот же точный тип, то есть Self-тип. В то же время будем считать обычный статически известный тип $C$ экзистенциальным типом $\exists X <: C.\, X$ (см. 2.4), поэтому для него будет работать правило (9), подставляющее экзистенциальный тип вместо рекурсивной ссылки.

## 4.4. Подтипизация Self-типов

Ранее в разделе 4.3 мы увидели, некоторые свойства относительно подтипизации, которыми должен обладать Self-тип. Приведём остальные.

1. $B <: A \Rightarrow Self(B) <: Self(A)$, так как все значения типа $Self(B)$ неизбежно содержатся среди значений $Self(A)$;
2. $B <: A \Rightarrow Self(B) <: A$, чтобы код с **this** оставался типизируемым;
3. $Nothing <: Self(A)$ и $Self(A) <: Any$;
4. $B \not<: Self(A)$, если $B \neq Nothing \wedge B \neq Self(\widehat{B})$.

Правило (4) не позволяет использовать в позиции, где ожидается Self-тип, посторонний объект, отличный от ресивера, чем обеспечивает безопасность системы типов.

Приведённые правила стандартным образом дополняются для работы с nullable значениями: $B <: A \Rightarrow B <: A?, B? <: A?$.

Тогда множество непосредственных супертипов Self-типа ($ST$, supertypes) будет таким:

$$A \in ST(B) \Rightarrow ST\left(Self(B)\right) = \{B, Self(A), Self(B)?\}$$

Из него следует правило вычисления ближайшего общего супертипа ($CST$, common supertype):

1. $CST\left(Self(C1), Self(C2)\right) \sim Self(CST(C1, C2))$;
2. $CST\left(Self(B), A\right) \sim CST(B, A)$, если $A \neq Self(\widehat{A})$.

Правила проверки переопределения метода (`override`) требуют, чтобы типы его аргументов совпадали, а возвращаемый тип был подтипом возвращаемого типа соответствующего метода базового класса. Таким образом, требуется дополнить эти правила так, чтобы два Self-типа, даже с различными границами, всегда считались равными. Это даёт возможность переопределять в наследниках методы с Self-типами во входящих позициях.

## 4.5. Создание новых объектов Self-типа

Ранее мы подразумевали, что существует лишь одно значение Self-типа — `this` (или `self`, как его ещё называют). Однако для многих важных приложений, например персистентных коллекций, требуется уметь создавать новые объекты Self-типа. Оказывается, во многих случаях это делать небезопасно. Так, присвоение Self-типа новым объектам открытых классов[11] и в открытых классах недопустимо:

```
1  open class A {
2      // Создание объекта открытого класса
3      fun newOfOpenA(): Self = A()
4      // Создание объекта в открытом классе
5      fun newOtherQ(): Self = Q()
```

[11] **Открытый класс** — класс, которой можно использовать как базовый при объявлении других классов.

```
6  }
7
8  class Q : A() { fun qOnly() {} }
9  class P : A() { fun pOnly() {} }
10
11 fun test(q: Q, p: P) {
12     q.newOfOpenA() // область видимости типа Q для A
13       .qOnly()     // ошибка
14     p.newOther()   // область видимости типа P для Q
15       .pOnly()     // ошибка
16 }
```

Проблемы возникают из-за того, что при материализации Self-тип может стать произвольным наследником своей границы (или другим Self-типом с границей-наследником). Поэтому, чтобы безопасно протипизировать новый объект `C(..)` Self-типом, необходимо добиться, чтобы его тип всегда был подтипом типа ресивера времени исполнения[12]:

```
1  C::class isSubtypeOf this@decl.getClass()
```

Среди промышленных языков, описанных в главе 3, ни один не поддерживает какие-либо значения Self-типа, кроме ссылки на ресивер (**this** или **self**). Однако в литературе (см. 3.1) приводятся ряд способов безопасно протипизировать новое значение Self-типом.

Первый способ — ненаследуемые методы [18]. **Ненаследуемый метод** — это метод, который должен быть переопределён в каждом наследнике. Утверждается, что в таком методе безопасно типизировать новые объекты объемлющего класса Self-типом. Действительно, поскольку метод заведомо переопределён, виртуальная диспетчеризация всегда приводит к вызову метода наследника, создающему объект того же типа времени исполнения, на котором метод был вызван. Поэтому необходимое условие типизации нового значения Self-типом, данное выше, заведомо выполняется.

[12]Пусть `this@decl` — ссылка на ресивер ближайшей декларации.

Второй способ — виртуальные конструкторы [17]. **Виртуальный конструктор** — это метод, возвращающий Self-тип, который вместо явного вызова конструктора класса (что может приводить к небезопасности, описанной выше) использует конструкцию вида `new This()`, создающую объект объект нужного наследника. Этот подход, в отличие от ненаследуемых методов, позволяет наследовать реализацию виртуального конструктора в наследнике и переиспользовать его код. Однако для этого требуется наложить большое количество специфических правил. А именно: виртуальный конструктор может быть только один; если виртуальный конструктор в наследнике имеет отличные от виртуального конструктора базового класса параметры, то он скрывает виртуальный конструктор базового класса, и все методы, вызывающие его, должны быть переопределены для нового конструктора; и др.

Нетрудно видеть, что оба подхода требуют довольно масштабных нововведений в языке в виде нового вида методов и сложным образом организованных правил для них. Это проблематично с точки зрения как реализации в компиляторе, так и дизайна языка, и, судя по всему, несоразмерно пользе и количеству сценариев использования. Поэтому мы приведём упрощённое правило, позволяющее типизировать новые объекты Self-типом, которое, тем не менее, не закрывает возможность реализации обоих подходов в дальнейшем.

Для того чтобы протипизировать создание нового объекта `C(..)` Self-типом, необходимо выполнение следующих условий:

1. Класс `C` должен быть финальным[13];
2. Тип `this@decl`[12] либо равен `C`, либо включает `C` в типе-пересечении (опр. 1.6) после smart-cast (опр. 1.7);
3. Тип `C` объявлен в том же модуле, в котором создаётся объект.

Правила (1) и (2) гарантируют выполнение необходимого условия безопасности типизации нового значения Self-типом. Действительно, так как `C`

[13]**Финальный класс** — класс, для которого нельзя объявлять наследников.

совпадает со статическим типом ресивера (2) и у этого типа не может быть подтипов из-за финальности (1), то `C` всегда равен типу времени исполнения ресивера, а значит тривиально является подтипом самого себя.

То, что `C` может включаться в статический тип пересечения (2), позволяет создавать новые объекты Self-типа не только в последнем классе иерархии наследования (однако это ограничение всё ещё делает предложенный метод менее мощным, чем академические, описанные выше, где создавать новые объекты Self-типа позволено на произвольном уровне иерархии наследования):

```
1  sealed interface Data {
2      class One(val a: Int) : Data
3      class Two(val a: Int, val b: Int) : Data
4
5      fun update(a: Int): Self =
6          when (this) {
7              is One -> One(a)     // : Self(One)
8              is Two -> Two(a, b) // : Self(Two)
9          } // : Self(Data) no CST
10 }
```

Последнее ограничение (3) предотвращает возникновение несовместимости исходных кодов при открытии класса. Это важно, так как до сих пор (до Kotlin версии 1.8.21) открытие класса не может нарушить обратной совместимости. Рассмотрим следующий пример, в нём открытие класса `One` из примера выше делает код некомпилируемым, если декларация `One` находится в другом модуле — это нарушение совместимости исходных кодов:

```
1  operator fun Data.plus(d: Int): Self =
2      when (this) {
3          is One -> One(a + d) // : Self, пока One финальный
4          is Two -> Two(a + d, b + d)
5      }
```

## 4.6. Позиции Self-типа

Как мы поняли ранее (см. 2.3), не во всех позициях безопасно использовать Self-типы. Так, в ковариантных позициях Self-типы можно использовать без опасений. Языки с безопасными системами этого, как правило, всё равно не позволяют, кроме как в простой исходящей позиции, ввиду отсутствия поддержки вариантности, как, например, Swift (см. 3.5). В то же время Self-типы в контравариантных позициях мешают наследнику быть подтипом [8], и языки вынуждены предпринимать те или иные специальные меры.

Однако если Self-тип относится не к dispatch ресиверу (опр. 1.17), а к extension ресиверу (опр 1.18), то проблемы наследования становятся не актуальны. Этот случай будет рассмотрен в 4.6.3.

### 4.6.1. Ковариантные позиции Self-типа

Общая суть ковариантных позиций состоит в том, что через них функция предоставляет некоторый объект вызывающему коду. В этом смысле не важно, как значение Self-типа предоставляется наружу, с помощью `return obj` или `observer(obj)`, если Self-тип находится во входящей контравариантной позиции (опр. 1.11), как продемонстрировано в разделе 1.4.3. В любом случае нужно только гарантировать, что значение Self-типа всегда можно использовать как значение того типа, в который этот Self-тип может быть материализован на вызывающей стороне.

Так, можно универсальным образом рассуждать о безопасности Self-типов в ковариантной позиции (рассматриваем вызов метода снаружи класса, материализация в Self-тип внутри рассматривается аналогично):

1. База: тип времени исполнения как ресивера (`this`), так и нового объекта Self-типа, всегда является подтипом того типа, в который соответствующий Self-тип может быть материализован (см. 4.5).

2. Переход: по предположению индукции, объект-ресивер — корректный объект своего статического типа, тогда передаваемый из функции вызывающей стороне объект Self-типа корректен относительно произвольной материализации (в том числе, если возвращается `this`), а значит,

получаемое вызывающей стороной значение Self-типа является допустимым значением своего типа после материализации.

Поскольку Kotlin поддерживает вариантность (1.1.3), а также существует немало приложений у Self-типов в ковариантных позициях (см. 1.4), то есть смысл поддержать использование Self-типы в этих позициях.

### 4.6.2. Контравариантные позиции Self-типа

Теперь рассмотрим контравариантные позиции, а именно самый простой случай — бинарные методы. Мы уже ранее приводили пример кода на TypeScript (см. 3.2), который использовал this-тип в контравариантной позиции, что приводило к небезопасности системы типов (возникновению `undefined` в корректно типизированной программе). Рассмотрим различные подходы по обеспечению безопасности контравариантной позиции, приведённые в главе 3, применительно к Kotlin и оценим их применимость.

Каждый приведённый подход так или иначе выделяет отдельно методы с Self-типами и накладывает на них особенные ограничения.

1. Swift позволяет использовать Self-типы в контравариантных позициях в протоколах, но требует подменять их на конкретный тип класса при реализации протокола этим классом (см. 3.5). В отличие от Kotlin в Swift нет возможности писать реализации по умолчанию методам в протоколах/интерфейсах, в Kotlin специально для методов с Self-типами это пришлось бы запретить.

2. Также, как правило, вводится новая вселенная точных типов, на которых можно вызывать методы с Self-типами в контравариантных позициях. Что требует довольно масштабных изменений в системе типов языка как с точки зрения дополнительных конструкций, так и новых механизмов вывода точных типов [17].

3. В противном случае приходится границей Self-типа в контравариантной позиции назначать самый базовой тип (и соответствующую область видимости), метод которого переопределяется данным. Однако такая

область видимости, как правило, не позволяет написать никакой полезной нетривиальной реализации (как в примере 1.4.4).

Поскольку все решения ограничивают виртуальную диспетчеризацию вызова бинарного метода, имеет смысл полностью от неё отказаться в пользу статической. Такое решение уже рассматривалось в 1.4.4, где выбор делался в пользу использования контекстных ресиверов языка Kotlin, которые покрывают подобные сценарии использования. Аналогично для других контравариантных позиций.

Таким образом, поддержка Self-типов, соответствующих dispatch-ресиверам, в контравариантных позициях выглядит нецелесообразной.

### 4.6.3. Позиции Self-типа в функциях-расширениях

Поскольку для функций-расширений не актуальны проблемы наследования, Self-тип в них можно использовать в произвольной позиции подобно инвариантному типовому параметру:

```
1  fun A.f(x: Self, p: Out<Self>, c: In<Self>): Self {
2      c.consume(p.produce()); return x
3  }
4  // аналогично
5  fun <T : A> T.f(x: T, p: Out<T>, c: In<T>): T {
6      c.consume(p.produce()); return x
7  }
```

Границей может быть типовой параметр или инстанциированный типовой конструктор:

```
1  fun <T> T.f(x: T, y: Self): Pair<Self, Self> =
2      Pair(x /* ошибка: T⩽:Self(T) */, y /* ok */)
3  fun <T> List<T>.shuffle(): Self { /* ... */ }
```

В то же время, как обсуждалось выше (4.2), граница всегда not-null тип:

```
1  fun A?.f(x: Self): Self? = this?.doSomething(x)
2  // аналогично следующему
3  fun <T : A> T?.f(x: T): T? = this?.doSomething(x)
```

## 4.7. Экспериментальная реализация Self-типов в компиляторе kotlinc

В рамках данной работы была разработана экспериментальная реализация Self-типов в компиляторе kotlinc версии K2.

Компилятор K2 имеет двухчастную архитектуру. **Frontend** занимается семантическим анализом программы, дополняя специальное промежуточное представление программы **FIR** (frontend intermediate representation) различной информацией, в том числе происходящей из анализа типов: вывода типов и проверки типизируемости. **Backend** занимается генерацией кода под различные целевые платформы, а также — оптимизациями. При передаче программы с frontend на backend модель программы, которой пользуется frontend (FIR) преобразуется в модель для backend'а (**IR**).

Поскольку целью данной работы является развитие системы типов языка Kotlin, основные изменения были произведены во frontend и в модуле конвертации FIR в IR.

Как уже было сказано выше (4.1), исследование синтаксиса Self-типов не входит в задачи данной работы, поэтому Self-тип введён как ключевое слово языка. А точнее, при конвертации результата парсинга программы в FIR, вершина дерева, соответствующая типу с именем Self, подменяется на специальную вершину Self-типа. Для прототипа этого оказывается достаточно.

Далее правила системы типов Kotlin были дополнены для работы с Self-типами описанным выше образом. А именно: правило проверки подтипизации, вычисления ближайших супертипов типа и вычисления ближайшего общего супертипа двух типов.

Ключевым в реализации является модификация сервисов областей видимости таким образом, чтобы они производили материализацию Self-типа в тип ресивера. **Сервисы областей видимости** — это набор сервисов ком-

пилятора, которые производят поиск деклараций, доступных для вызова на конкретном типа (см. 1.20). Чтобы произвести материализацию, требуется генерировать синтетические декларации, в которых вместо Self-типа подставлен тип ресивера. Это происходит аналогично подстановке типовых аргументов в параметризованных функциях.

И наконец, при преобразовании FIR в IR Self-тип подменяется на его границу, а так же метаинформацию, чтобы сохранить сведение о том, что это изначально был Self-тип (для использования этой декларации в другом Kotlin-модуле, например). Соответственно из Java Self-типы видны просто как их границы.

Полученная реализация была протестирована на предмет недопуска некорректных программ с Self-типами: использующих ссылку на посторонний объект, где ожидается Self-тип; некорректно создающих новый объект; использующих Self-тип в небезопасной позиции. А так же была протестирована компилируемость и работоспособность различных сценариев использования Self-типов.

# Заключение

В рамках данной работы был разработан дизайн Self-типов для языка Kotlin на основе опыта других языков, а также была реализована поддержка Self-типов в компиляторе kotlinc. Для этого были решены следующие задачи:

1. Выделены особенности существующих реализаций Self-типов в других языках:
   (a) Какие значения можно типизировать Self-типом;
   (b) В каких позициях разрешено использовать Self-тип;
   (c) Каковы меры по обеспечению безопасности системы типов.
2. Self-типы интегрированы[14] в типовую систему языка Kotlin:
   (a) Введено понятие материализации Self-типа;
   (b) Прописаны правила подтипизации, вычисления супертипов;
   (c) Разработаны условия типизации новых объектов Self-типом;
   (d) Указаны безопасные позиции использования Self-типа.
3. Поддержка Self-типов реализована[15] в компиляторе kotlinc:
   (a) Добавлен новый вид типов — Self-типы;
   (b) Сервисы областей видимости расширены функциональностью материализации Self-типов;
   (c) Полученная реализация протестирована на предмет покрытия необходимых сценариев использования, а так же недопуска небезопасного кода с Self-типами.

Приведённый дизайн Self-типов естественным образом дополняет систему типов языка Kotlin, расширяя её выразительные возможности на целый ряд полезных приложений. А предложенная реализация органично встраивается в архитектуру компилятора kotlinc. Таким образом, можно ожидать появления Self-типов в последующих версиях языка Kotlin.

[14] `https://github.com/winter-yuki/kotlin-self-types`
[15] `https://github.com/winter-yuki/kotlin/tree/self-types`

# Список литературы


[1] Akhin Marat, Belyaev Mikhail. Kotlin language specification. — 2021.

[2] Associated types with class / Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow // Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 2005. — P. 1–13.

[3] Bierman Gavin, Abadi Martín, Torgersen Mads. Understanding typescript // ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28 / Springer. — 2014. — P. 257–281.

[4] Bruce Kim B. Safe type checking in a statically-typed object-oriented programming language // Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1993. — P. 285–298.

[5] Bruce Kim B. Increasing Java's expressiveness with ThisType and match-bounded polymorphism // Available on the author's web page6. — 1997.

[6] Bruce Kim B, Petersen Leaf, Fiech Adrian. Subtyping is not a good "match" for object-oriented languages // ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11 / Springer. — 1997. — P. 104–127.

[7] Cardelli Luca. Amber // Combinators and Functional Programming Languages: Thirteenth Spring School of the LITP Val d'Ajol, France, May 6–10, 1985 Proceedings. — Springer, 2005. — P. 21–47.

[8] Cook William R, Hill Walter, Canning Peter S. Inheritance is not subtyping // Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1989. — P. 125–135.

[9] Hutton Graham, Meijer Erik. Monadic parser combinators. — 1996.

[10] Jemerov Dmitry, Isakova Svetlana. Kotlin in action. — Simon and Schuster, 2017.

[11] Jones Richard, Hosking Antony, Moss Eliot. The garbage collection handbook: the art of automatic memory management. — CRC Press, 2016.

[12] Liskov Barbara. Keynote address-data abstraction and hierarchy // Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum). — 1987. — P. 17–34.

[13] Lisman John E, Idiart Marco AP. Storage of 7±2 short-term memories in oscillatory subcycles // Science. — 1995. — Vol. 267, no. 5203. — P. 1512–1515.

[14] Okasaki Chris. Purely functional data structures. — Cambridge University Press, 1999.

[15] Pearce David J. A calculus for constraint-based flow typing // Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs. — 2013. — P. 1–7.

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

[17] Ryu Sukyoung. ThisType for object-oriented languages: From theory to practice // ACM Transactions on Programming Languages and Systems (TOPLAS). — 2016. — Vol. 38, no. 3. — P. 1–66.

[18] Saito Chieri, Igarashi Atsushi. Matching ThisType to subtyping // Proceedings of the 2009 ACM symposium on Applied Computing. — 2009. — P. 1851–1858.

[19] Sanner Michel F et al. Python: a programming language for software integration and development // J Mol Graph Model. — 1999. — Vol. 17, no. 1. — P. 57–61.

[20] Siek Jeremy, Taha Walid. Gradual typing for objects // ECOOP 2007–Object-Oriented Programming: 21st European Conference, Berlin,

Germany, July 30-August 3, 2007. Proceedings 21 / Springer. — 2007. — P. 2–27.

[21] The swift programming language / James Goodwill, Wesley Matlock, James Goodwill, Wesley Matlock // Beginning Swift Games Development for iOS. — 2015. — P. 219–244.