Дизайн и разработка Self-типов для языка Kotlin

Стоян Андрей Сергеевич
научный руководитель: к. ф.-м. н. Москвин Денис Николаевич
научный консультант: Новожилов Дмитрий Павлович

Университет ИТМО
Разработка программного обеспечения/Software engineering

Санкт-Петербург 2023г.

Системы типов

- Система типов позволяет выявлять заведомо некорректные программы
- Проверка нетривиальных свойств программы — неразрешимая задача
- Анализ типов отвергает много корректных программ, чтобы оставаться разрешимым

Задачи дизайна безопасной системы типов для промышленного языка программирования

- Отвергнуть все некорректные программы (с точки зрения типов времени исполнения)
- Типизировать как можно больше корректных программ без потери практичности:
  - Сложность понимания
  - Многословность кода

Ограничение текущей системы типов Kotlin

Ограничение

Нет прямого способа написать тип получателя[a] (`this`'а) в декларации метода.

[a] *Получатель вызова* — объект, на котором вызывается метод. Пример: `a.f()`, `a` — получатель.

Нехватка точности анализа типов (корректный код не типизируется)

```
interface PCollection<E> {
    fun add(elem: E): PCollection<E> // нет способа написать тип специфичнее
}
fun test(list: PList<Int>) {
    list.add(42)        // : PCollection<Int> - базовый тип
        .listSpecific() // ошибка типизации
}
```

Существующие способы записи типа получателя

Типовой параметр с рекурсивным ограничением (Kotlin)

```
interface PCollection<E, S : PCollection<E, S>>
```

- Небезопасность: явное приведение типов: `this as S`
- Сложность: рекурсивное ограничение
- Многословность: дополнительный типовой параметр распространяется по всему коду

Переопределяющие методы с более специфичным возвращаемым типом (Kotlin)

- Ненадёжность: нет контроля компилятора, что такие методы не были забыты
- Многословность: переопределить каждый метод в каждом наследнике
- Ограниченность: только для возвращаемого типа

Ассоциированные типы (Scala)

- Небезопасность, ненадёжность, ограниченность
- Сложность использования и реализации [Path-dependent types, Odersky et al, 2014]

Self-типы

- Δ *Self-тип* — прямой способ записи типа получателя вызова (тип `this`'а)
- \+ Не имеет упомянутых ранее недостатков
- − Требует нетривиального языкового дизайна

Self-типы уточняют анализ типов

```
interface PCollection<out E> {
    fun add(value: E): Self
}
fun test(list: PList<Int>) {
    list.add(x)          // область видимости PList<Int>
        .listSpecific() // корректный код типизируется
}
```

Цель и задачи

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

Существующие реализации Self-типов

Академические результаты, характерные черты

- Кодирование в типизированном лямбда-исчислении
- Видны места потенциального нарушения безопасности системы типов
- Предлагаются наиболее гибкие решения, нередко в ущерб практичности

Прикладные языки, поддерживающие Self-типы

- × Python, TypeScript, Java с плагином Manifold — небезопасная реализация
- × Rust не поддерживает нетривиальные случаи использования Self-типов
- Swift — полноценная безопасная реализация Self-типов

Переписывание Self-типа

Пример: небезопасность покидания Self-типом контекста объекта

```
open class A {
    fun self(): Self = this
    fun unsafe(a: A): Self = a.self()
}

class B : A() {
    fun bOnly() {}
}

fun test(b: B) {
    val a = A()
    b.unsafe(a) // область видимости B
     .bOnly()   // ошибка исполнения
}
```

Академическое решение

- Self-тип аналогичен параметру рекурсивного типа $\mu Self.\, T[Self]$
- Для использования требуется применить правило развертки (unfold)

Решение для Kotlin

- Self-тип должен переписываться в тип получателя в его области видимости:

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

- Ограничение подтипизации:

$$\forall T.\, T \ll: Self,\, T \neq \bot,\, T \neq Self$$

Значения Self-типа: существующие решения

Академические решения

- Специальные виды методов с безопасной типизацией посторонних объектов Self-типом
  - Ненаследуемые методы[a] [Saito et al, 2009]
  - Виртуальные конструкторы[b] [Na et al, 2012]

– Усложняют язык: новая сущность

– Непросто использовать: рутинный код и нетривиальные правила

[a] Не могут быть унаследованы — требуют переопределение в каждом наследнике

[b] Могут быть унаследованы по сложным правилам

Решения в прикладных языках

- Только объект получатель (`this`) имеет Self-тип

  ⇒ Нельзя создавать новые объекты Self-типа

  – Необходимо для важных приложений (например, персистентные коллекции)

Значения Self-типа: решение для Kotlin

Условие безопасной замены типа C на Self

`C::class isSubtypeOf this.getClass()` должно выполняться статически

Правило значений Self-типа для Kotlin

1. Класс C должен быть финальным
2. Тип `this` либо равен C, либо включает C после smart-cast
3. Тип C объявлен в том же модуле, в котором создаётся объект

\+ Простота использования

− Ограниченная поддержка открытых классов

Пример: персистентный список

```
class PListImpl<E> : PList<E> {
    override fun add(elem: E): Self {
        val newList = Cons(elem, list)
        return PListImpl(newList)
    }
}
```

Возможные позиции Self-типа: ковариантные позиции

Ковариантные позиции — значение передаётся вызывающему коду

- `fun add(elem: E): Self`
- `fun onClick(observer: (Self) -> Unit)`

Академический результат

- Self-типы можно безопасно использовать в ковариантных позициях [Cook et al, 1989]

Решение в Swift

- Нет поддержки вариантности типовых параметров
- Полностью разрешена только позиция возвращаемого типа

Решение для Kotlin

- Есть поддержка вариантности типовых параметров
- Допустимо использовать Self-типы во всех ковариантных позициях

Контравариантные позиции Self-типа: существующие решения

Контравариантные позиции — значение передаётся коду метода

- Δ *Сложные методы* — методы, содержащие Self-тип в контравариантной позиции
- `fun combine(other: Self): Self`

Академические результаты

- В классе есть сложный метод ⇒ наследник не является подтипом [Cook et al, 1989]
  - Отношение matching'а вместо подтипизации [Bruce et al, 1996]
  - Разделение точных и экзистенциальных типов, локальное уточнение [Saito et al, 2009]
  - Именованные wildcard'ы и точные типовые параметры [Na et al, 2012]
- – Специальные правила ограничивают использование сложных методов
- – Массивные изменения в системе типов для больших возможностей

Решение в Swift

- Запрещено делать виртуальный вызов сложного метода

Контравариантные позиции Self-типа: решение для Kotlin

- Для поддержки сложных методов запрещают виртуальную диспетчеризацию на них

⇒ Можно использовать статическую, эмулируя классы типов в Kotlin

⇒ Другая возможность языка покрывает сценарии использования сложных методов

Эмуляция трейтов с помощью контекстов языка Kotlin

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

context(Semigroup<S>)
fun <S> combineAll(xs: Iterable<S>): S =
    xs.reduce { acc, x -> acc.combine(x) }
```

Решение для Kotlin

Запрет на использование Self-типов в контравариантных позициях.

Реализованная функциональность

Ковариантные позиции Self-типа

```
interface Button {
    fun onClick(observer: (Self) -> Unit)
}
checkBox.onClick { print(it.isChecked) }
```

Типизация посторонних объектов Self-типом

```
class PListImpl<E> : PList<E> {
    override fun add(elem: E): Self = PListImpl(/*...*/)
}
```

Реализована проверка на использование Self-типов в недопустимых позициях

```
class Semigroup {
    fun combine(other: Self) // ошибка компиляции
}
```

Результаты

1. Проанализированы существующие реализации Self-типов в других языках на предмет их возможностей и мер по обеспечению безопасности системы с Self-типами
2. Self-типы интегрированы[1] в типовую систему языка Kotlin с опорой на существующие решения
3. Прототип системы с Self-типами реализован[2] в компиляторе kotlinc

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

Содержание дополнительных слайдов

- Материалы
- Kotlin
- Существующие решения
- Сценарии использования
- Решения в других языках
- Формализация
- Интеграция Self-типов в типовую систему Kotlin
- Реализация прототипа

Материалы

1. YouTrack: Запрос на добавление Self-типов в Kotlin
2. Cook, William R., Walter Hill, and Peter S. Canning. 1989. «Inheritance is not subtyping». Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. https://cs.rice.edu/~javaplt/papers/Inheritance.pdf
3. Sukyoung Ryu. 2016. «ThisType for Object-Oriented Languages: From Theory to Practice». ACM Trans. Program. Lang. Syst. 38, 3, Article 8 (May 2016), 66 pages. https://dl.acm.org/doi/10.1145/2888392
4. Self-типы как плагин для Java
5. Swift: Self-типы
6. Python: Self-типы PEP
7. TypeScript: this-типы

Ресиверы (получатели) в Kotlin

(Dispatch) Ресивер функции

Специальный параметр функции, по которому происходит виртуальная диспетчеризация вызова к реализации наследника. Доступен как `this` в её теле.

- Например, в вызове `"a".plus("b")` строка `"a"` является ресивером для функции `plus`

Область видимости типа

Множество функций, для которых объект данного типа может быть использован как ресивер.

- Функция `plus` содержится в области видимости типа `String`:
  `(plus: String.(String) -> String)` $\in$ `scope(String)`

Вариантность

Вариантность типового параметра

- Определят, в каких позициях можно использовать этот типовой параметр
- Задаёт отношение подтипизации между параметризованными типами

Инвариантные типовые параметры: `interface Inv<T> { fun id(x: T): T }`

- Можно использовать в произвольных позициях в декларациях методов
- Не устанавливает отношения подтипизации: `Inv<B> !<:> Inv<A>`

Ковариантные типовые параметры: `interface Out<out T> { fun produce(): T }`

- Можно использовать в исходящих позициях в декларациях методов
- Устанавливает прямое отношение подтипизации: `B <: A => Out<B> <: Out<A>`

Контравариантные типовые параметры: `interface In<in T> { fun accept(x: T) }`

- Можно использовать во входящих позициях в декларациях методов
- Устанавливает обратное отношение подтипизации: `B <: A => In<B> :> In<A>`

Дополнительный типовой параметр

Пример: персистентные коллекции с дополнительным типовым параметром

```
interface PCollection<out E, out S : PCollection<E, S>> {
    fun add(value: E): S
}
interface PList<out E, out S : PList<E, S>> : PCollection<E, S> {
    fun listSpecific()
}
fun <L : PList<Int, L>> test(xs: L) {
    xs.add(42) /* : L */ .listSpecific()
}
```

Недостатки

- Паттерн рекурсивного ограничения распространяется по всему коду
- Требуется явное приведение типов: `this as S`
- Однажды зафиксированный типовой аргумент `S` не может быть уточнён в наследниках

Добавление abstract override методов

Пример: персистентные коллекции с abstract override методами

```
interface PCollection<out E> {
    fun add(value: E): PCollection<E>
}
interface PList<out E> : PCollection<E> {
    abstract override fun add(value: E): PList<E>
    fun listSpecific()
}
fun test(xs: PList<Int>) {
    xs.add(42) /* : PList<Int> */ .listSpecific()
}
```

Недостатки

- Много рутинного кода: переопределить каждый метод в каждом наследнике
- Нет контроля компилятора, что abstract override методы не были забыты
- Работает только для возвращаемого типа (типы параметров обязаны совпадать)

Рекурсивные структуры данных

Также Self-типы помогают строить рекурсивные структуры данных из вершин одного типа:

```
// исходящая контравариантная позиция
abstract class Node<out T>(val value: T, val children: List<Self>)
class BetterNode<out T>(value: T, children: List<Self> = emptyList()) :
    Node<T>(value, children) {
    fun betterSpecific() = println(value)
}

fun test() {
    val betterTree = BetterNode(value = 2, children =
        listOf<BetterNode<Int>>(
            BetterNode(1, listOf(BetterNode(0))),
            BetterNode(4, listOf(BetterNode(3), BetterNode(5)))))
    betterTree.children
        .flatMap { it.children }
        .forEach { it.betterSpecific() } // Печатает "0 3 5"
}
```

Шаблон «Абстрактная фабрика»

Пусть требуется по элементу типизируемым образом получить породившую его фабрику.

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

interface Factory {
    fun create(): Element<Self> // ковариантная исходящая позиция
}

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

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

Шаблон «Наблюдатель»

Абстрагируем логику регистрации и нотификации наблюдателей:

```
abstract class AbstractObservable {
    private val observers =
        mutableListOf<(Self) -> Unit>()
    // контравариантная входная позиция
    fun observe(
        observer: (Self) -> Unit
    ) {
        observers += observer
    }
    private fun notifyObservers() {
        observers.forEach { observer ->
            observer(this)
        }
    }
}
```

```
class Entity : AbstractObservable {
    var color: Color = Color.Purple
        set(new: Color) {
            field = new
            notifyObservers()
        }
}
fun observer(entity: Entity) {
    println("New: ${it.color}")
}
fun test() {
    val entity = Entity()
    entity.observe(::observer)
    // Печатает "New: Color.Blue"
    entity.color = Color.Blue
}
```

Алгебры

Наследник не подтип, если рекурсивный тип стоит в контравариантной позиции[3].

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

От использования рекурсивного типа можно отказаться с помощью контекстных ресиверов:

```
interface Semigroup<S> {
    infix fun S.add(other: S): S
}
interface Monoid<M> : Semigroup<M> {
    val empty: M
}
context(Monoid<T>)
fun <T> concat(vararg xs: T): T =
    xs.fold(empty) { acc, x -> acc add a }
```

[3] [Inheritance is not subtyping, Cook et al, 1989]

Ассоциированные типы (abstract type members)

- Поддерживаются в Scala[a] и Swift
- Аналогично рекурсивным дженерикам, но без зашумления клиентского кода
- Сложны в использовании и реализации: отслеживание по какому пути получено значение ассоциированного типа[b]
- Запроса от пользователей на поддержку ассоциированных типов в Kotlin нет

```
trait PCollection[T] {
  type S
  def add(x: T): S
}

trait PList[T] extends PCollection[T] {
  type S <: PList[T]
  def listSpecific: Unit
}

class PListImpl[T] extends PList[T] {
  type S = PListImpl[T]
  override def add(x: T): PListImpl[T] = // ..
  override def listSpecific: Unit = // ...
}
```

[a] https://docs.scala-lang.org/tour/abstract-type-members.html

[b] [Odersky et al, 2014]

Self-типы как частный случай ассоциированных типов

Ассоциированные типы всё равно требуют доработки для сценариев Self-типов

- Ассоциированному типу приходится вручную уточнять границу в каждом наследнике
- Требуется явное приведение типа `this` к ассоциированному типу
- Новые значения по умолчанию не принадлежат ассоциированному типу (пока он не зафиксирован)
- Ассоциированный тип однажды должен быть зафиксирован в иерархии наследования и не может быть уточнён ниже без изменения базового класса

Преимущества реализации Self-типов через ассоциированные

- Автоматически возможна нетривиальная поддержка контравариантных позиций
  - Локальное уточнение: `node.next.insert(node)` [Saito et al, 2009]

TypeScript: небезопасность системы типов

```
class Box {
  sameAs(other: this): boolean { /* ... */ }
}
class DerivedBox extends Box {
  otherContent: string = "?";
  sameAs(other: this): boolean {
    if (other.otherContent === undefined) {
      console.log("broken")
    }
    /* ... */
  }
}
const base = new Box();
const derived = new DerivedBox();
function test(x: Box): boolean { return x.sameAs(base) }
test(derived) // Печатает "broken"
```

Swift

- Существует полная поддержка Self-типов для простой исходящей позиции
- Для методов классов Self-тип доступен только для исходящей позиции
- Если декларация метода протокола[4] содержит ассоциированный тип или Self-тип не в простой исходящей позиции:
  - Запрещено вызывать такие методы виртуально на `any Protocol`
  - Можно вызывать, если протокол является ограничением типового параметра и на `some Protocol`
  - Реализующий класс обязан заменить такие вхождения Self-типа на себя
  - Ситуация становится аналогична языкам без наследования
- Ассоциированные типы так же позволяют эмулировать Self-типы, но на один уровень иерархии и с дополнительными приведениями типа
- Self-типы в расширениях ссылаются на расширяемый тип

[4] Протоколы — механизм специального полиморфизма как трейты или классы типов

Формализация Self-типов — рекурсивные типы

Тип объекта формализуется в расширенном λ-исчислении через тип записи.

Правило подтипизации для типов записей

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

Пример: тип объекта одномерной точки

$$T = \{x : int, equal : T \to bool\}$$
$$T = \mu t.\, \{x : int, equal : t \to bool\}$$

Правило подтипизации рекурсивных типов

$$\frac{\Gamma, s <: t \vdash \sigma[s] <: \tau[t]}{\Gamma \vdash \mu s.\, \sigma[s] <: \mu t.\, \tau[t]}$$ Amber rule

Наследник не является подтипом [Inheritance is not subtyping, Cook at al, 1989]

$$T = \mu t.\, \{x : int, equal : t \to bool\}$$
$$T' = \mu t.\, \{x : int, equal : t \to bool, dist : int\}$$

Рекурсивный тип входит в тип *equal* в контравариантной позиции, значит, T' не подтип T.

Традиционная подтипизация [Ryu et al, 2016]

Metavariables:

t, u	type variable	τ, σ	::=	$t \mid \rho \mid \gamma \mid \tau \to \tau$	type
ρ	primitive type	γ	::=	$\mu t.\{l_i\colon \tau_i\ ^{i\in 1..n}\}$	record type
l	label				

Type variables: $\boxed{vars(\Delta)}$

$vars(\{t_i \le u_i\ ^{i\in 1..n}\}) = \{t_i, u_i\ ^{i\in 1..n}\}$

Traditional subtyping: $\boxed{\Delta \vdash \tau \le \sigma \text{ where } \Delta ::= \{t_i \le u_i\ ^{i\in 1..n}\}}$

[TS-PRIM]
$$\frac{}{\Delta \vdash \rho \le \rho}$$

[TS-FUNC]
$$\frac{\Delta \vdash \sigma \le \tau \qquad \Delta \vdash \tau' \le \sigma'}{\Delta \vdash \tau \to \tau' \ \le\ \sigma \to \sigma'}$$

[TS-RECORD]
$$\frac{\forall i \in 1..n\colon\ \Delta \vdash \tau_i \le \sigma_i \quad n \ge 0,\ m \ge 0}{\Delta \vdash \{l_i\colon \tau_i\ ^{i\in 1..n+m}\} \ \le\ \{l_i\colon \sigma_i\ ^{i\in 1..n}\}}$$

[TS-AMBER1]
$$\frac{t \le u \in \Delta}{\Delta \vdash t \le u}$$

[TS-AMBER2]
$$\frac{t \ne u \qquad t, u \notin vars(\Delta) \qquad \Delta \cup \{t \le u\} \vdash \tau \le \sigma}{\Delta \vdash \mu t.\tau \ \le\ \mu u.\sigma}$$

Пересмотренная подтипизация [Ryu et al, 2016]

Metavariables:

$$
\begin{array}{lll}
\tau, \sigma & ::= & t \mid \rho \mid \alpha \mid \gamma \mid \tau \to \tau \qquad \text{type (revised)} \\
\alpha, \beta & ::= & \mu t.\{l_i\colon \tau_i{}^{i \in 1..n}\} \qquad \text{record type (revised)} \\
\gamma & ::= & \exists s \lhd \alpha.s \qquad \text{existential record type (new)}
\end{array}
$$

Specializing: $\boxed{\Delta \vdash \alpha \lhd \beta \text{ where } \Delta ::= \{t_i{}^{i \in 1..n}\}}$

[SPECIALIZING]

$$\frac{n \geq 0,\ m \geq 0 \qquad t \notin \Delta \qquad \forall i \in 1..n\colon\ \Delta \cup \{t\} \vdash \tau_i <\colon \sigma_i}{\Delta \vdash \mu t.\{l_i\colon \tau_i{}^{i \in 1..n+m}\} \lhd \mu t.\{l_i\colon \sigma_i{}^{i \in 1..n}\}}$$

Revised subtyping: $\boxed{\Delta \vdash \tau <\colon \sigma \text{ where } \Delta ::= \{t_i{}^{i \in 1..n}\}}$

[RS-PRIM]

$$\frac{}{\Delta \vdash \rho <\colon \rho}$$

[RS-FUNC]

$$\frac{\Delta \vdash \sigma <\colon \tau \qquad \Delta \vdash \tau' <\colon \sigma'}{\Delta \vdash \tau \to \tau' <\colon \sigma \to \sigma'}$$

[RS-TVAR]

$$\frac{t \in \Delta}{\Delta \vdash t <\colon t}$$

[RS-RTOR]

$$\frac{}{\Delta \vdash \alpha <\colon \alpha}$$

[RS-RTOS]

$$\frac{\Delta \vdash \alpha \lhd \beta}{\Delta \vdash \alpha <\colon \exists s \lhd \beta.s}$$

[RS-STOS]

$$\frac{\Delta \vdash \alpha \lhd \beta}{\Delta \vdash \exists s \lhd \alpha.s <\colon \exists s \lhd \beta.s}$$

Соответствие правил для Kotlin с [Ryu et al, 2016]

- Self-тип — это точный тип $\mu t.\, \{\overline{l_i : \tau_i}\}$ (exact type)
- А любой другой — экзистенциальный тип $\exists s \triangleleft \alpha.\, s$ (inexact type)

Подтипизация

Точный тип может быть подтипом экзистенциального, но не наоборот:

$$\frac{\Delta \vdash \alpha \triangleleft \beta}{\Delta \vdash \alpha <: \exists s \triangleleft \beta.\, s}\ \text{RS-RtoS} \qquad \frac{\Delta \vdash A <: B}{\Delta \vdash \mathit{Self}(A) <: B}\ \text{Self-NoSelf}$$

Переписывание Self-типа

Переписывание сходно с одним шагом развёртки изорекурсивного типа:

$$\frac{U = \mu t.\, T \quad \Delta \vdash o : U}{\Delta \vdash \mathit{unfold}(o) : [t \mapsto U]\, T}\ \text{Unfold-Self} \qquad \frac{U = \mu t.\, T \quad U' = \exists s \triangleleft U.\, s \quad \Delta \vdash o : U'}{\Delta \vdash \mathit{unfold}(o) : [t \mapsto U']\, T}\ \text{Unfold-NoSelf}$$

Правила подтипизации для Self-типа

Граница Self-типа

Тип B является границей Self-типа (обозначение $\underline{Self}(B)$), если B — наиболее общий тип получателя, на котором этот метод может быть вызван. Совпадает с типом текущего класса.

Правила подтипизации для Self-типа

1. $B <: A \iff Self(B) <: Self(A)$ для возможности переопределения методов с `Self`
2. $B <: A \iff Self(B) <: A$, чтобы код с `this` оставался типизируемым
3. $Nothing <: Self(A)$ и $Self(A) <: Any$
4. $B \not<: Self(A)$, если B не подходит под правила (1) и (3)

Безопасность присваиваний

1. `this`, ссылающийся на получатель C, имеет тип $Self(C)$
2. Правило (4) не позволяет использовать посторонний объект в качестве $Self$

Некорректное создание новых объектов

Для реализации персистентных и иммутабельных структур данных нужно иметь возможность создавать новый объект Self-типа.

В общем случае небезопасно

- Создавать объект открытого класса
- Создавать объект другого класса

```
open class A {
    fun newOfOpenA(): Self = A()
    fun newOtherQ(): Self = Q()
}

class Q : A() { fun qOnly() {} }
class P : A() { fun pOnly() {} }

fun test(q: Q, p: P) {
    q.newOfOpenA() // scope типа Q для A
     .qOnly()      // ошибка
    p.newOther()   // scope типа P для Q
     .pOnly()      // ошибка
}
```

Безопасное создание новых объектов типа `Self(C)`

Правило

- Класс `C` должен быть финальным
- Тип `this`[a] либо равен `C`, либо включает `C` после smart-cast
- Тип `C` объявлен в том же модуле, в котором создаётся объект[b]

[a] Получатель текущей декларации (с типом границы Self-типа)

[b] Иначе открытие класса нарушало совместимость исходных кодов

```
sealed interface Data {
    data class One(var a: Int) : Data

    data class Two(
        var a: Int, var b: Int
    ) : Data

    fun copy(): Self = when (this) {
        is One -> One(a)    //: Self(One)
        is Two -> Two(a, b) //: Self(Two)
    } // : Self(Data)
}
```

Self-типы для функций-расширений

- Для функций-расширений не актуальны проблемы наследования

$\Rightarrow$ Self-тип в них можно использовать в произвольной позиции подобно инвариантному типовому параметру

```
fun A.f(x: Self, p: Out<Self>, c: In<Self>): Self {
    c.consume(p.produce()); return x
}

// аналогично

fun <T : A> T.f(x: T, p: Out<T>, c: In<T>): T {
    c.consume(p.produce()); return x
}
```

Способы проверки безопасности обновления системы типов

× Формализация фрагмента языка в виде модельного исчисления
- \+ Можно получить верифицированные доказательства свойств
- – Чрезвычайно трудоёмко
- – Под вопросом репрезентативность модельного языка

× Реализовать в компиляторе и подвергнуть массовому тестированию
- \+ Реализация будет корректной с высокой вероятностью
- – Требуются развитые методы генерации семантически корректных программ
- – Требуется примерно полная реализация возможности в компиляторе

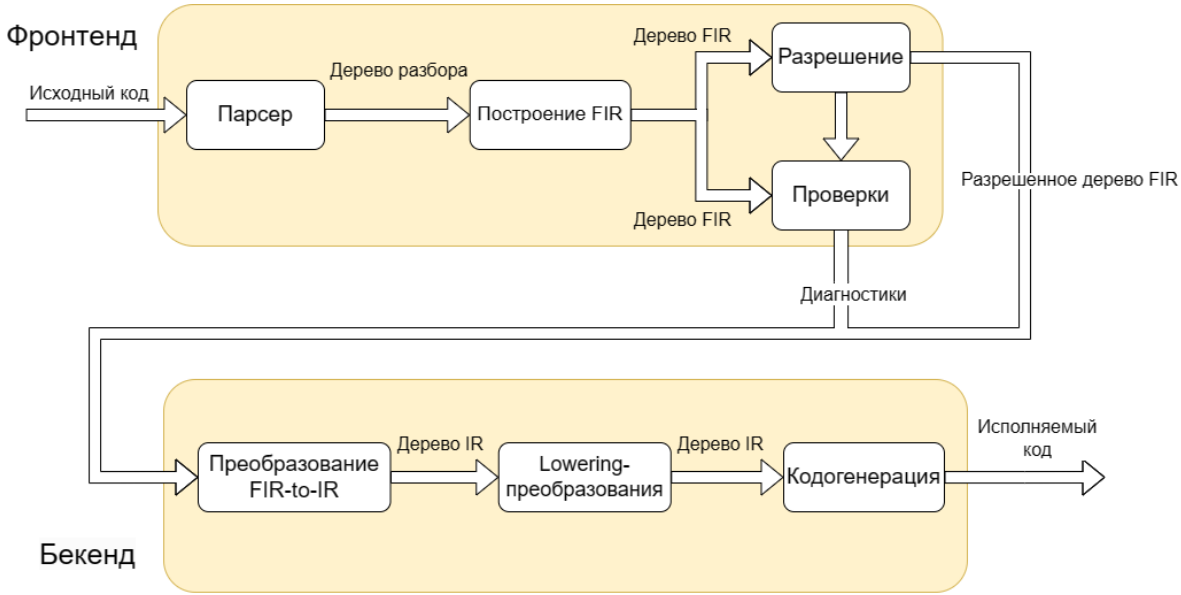
✓ Исследовать существующие решения и реализовывать заведомо безопасным образом
- \+ Априорно не требуется реализация в компиляторе
- \+ Фокус на реальном языке и его отличиях от других
- – Ничего нельзя со всей определённостью гарантировать
- ⇒ Хорошо подходит для первого приближения дизайна новой возможности системы типов

Безопасность переписывания

1. Значение Self-типа безопасно относительно переписывания (строили правила соответствующим образом)
2. Если получатель имеет тип $A \neq Self$, `Self` переписывается в A, а $A \ll: Self(A)$
3. Если получатель имеет тип `Self(B)`, то `Self(A)` декларации переписывается в `Self(B)`, и известно, что значение получателя безопасно, переписываемое значение безопасно

```
abstract class A {
    fun self(): Self = this
    fun unsafeSelf(a: A): Self/*(A)*/ =
        a    /* : A       */ .self() // : A , ошибка компиляции
    fun safeSelf():       Self/*(A)*/ =
        this /* : Self(A) */ .self() // : Self(A) , ok, Self(A) <: Self(A)
}
```

Архитектура компилятора Kotlin (K2)



Детали реализации Self-типов в компиляторе kotlinc

1. Идентификатор типа `Self` введён как ключевое слово языка
2. Введён новый вид типов — Self-типы
3. Правила системы типов Kotlin доопределены для работы с Self-типами:
   - Правило подтипизации
   - Правило определения непосредственных супертипов
   - Правило вычисления ближайших общих супертипов
4. Реализована область видимости[5], переписывающая Self-тип
   - Для каждого места вызова метода с Self-типом генерируется синтетическая декларация
   - В будущем потребуется разработать более эффективную реализацию
5. Реализовано преобразование, подменяющее Self-тип на его границу и метаинформацию при переходе к промежуточному представлению бекенда компилятора (IR)
6. Полученная реализация протестирована:
   - Корректный код с Self-типами допускается системой типов
   - Небезопасный код отвергается системой типов

[5] Области видимости — сервисы компилятора, возвращающие множество функций, для которых объект данного типа может быть использован как получатель: $(plus : String.(String) \to String) \in scope(String)$