Компиляторные плагины

Модификация и анализ Compose

Что будет в докладе?

- Изучим особенности компиляторых плагинов
- Разберем код, генерируемый Compose Compiler
- Напишем свой компиляторный плагин

Асхар Айдаров

- Android разработчик в Core-команде ВКонтакте
- Менторил и преподавал в VK Education

@ack

@secundans
@int_ax

Направления работ с применением компиляторых плагинов

1. Тестирование
    - Подсветка рекомпозиции
    - Простановка testTag
    - Логирование причин рекомпозиции
2. Анализ кода
    - Анализ стабильности параметров Composable функций
3. Оптимизация
    - Удаление testTag
    - Удаление sourceInformation

Направления работ с применением компиляторых плагинов

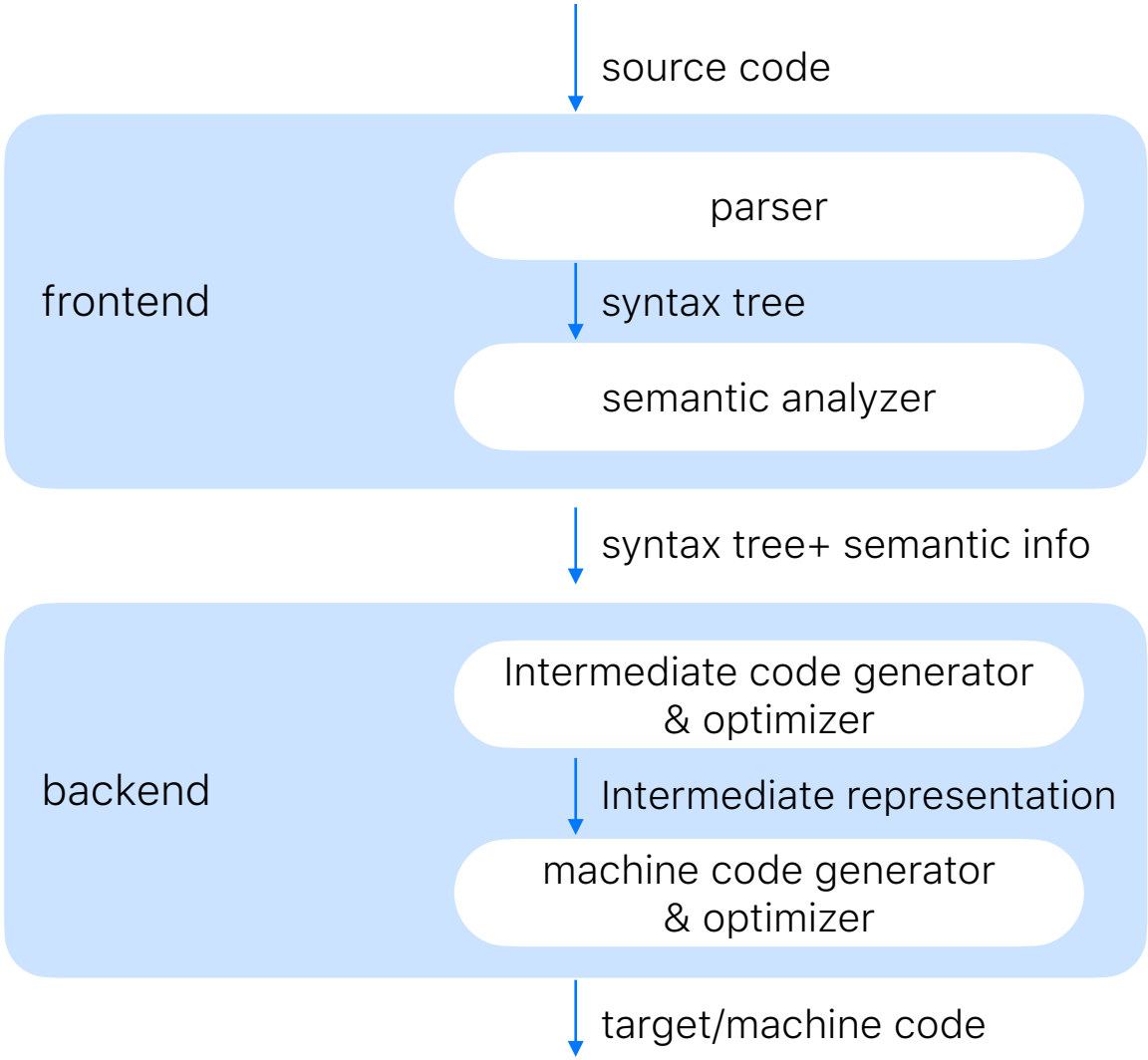
1. Тестирование
    - **Подсветка рекомпозиции**
    - Простановка testTag
    - Логирование причин рекомпозиции
2. Анализ кода
    - **Анализ стабильности параметров Composable функций**
3. Оптимизация
    - Удаление testTag
    - Удаление sourceInformation

План

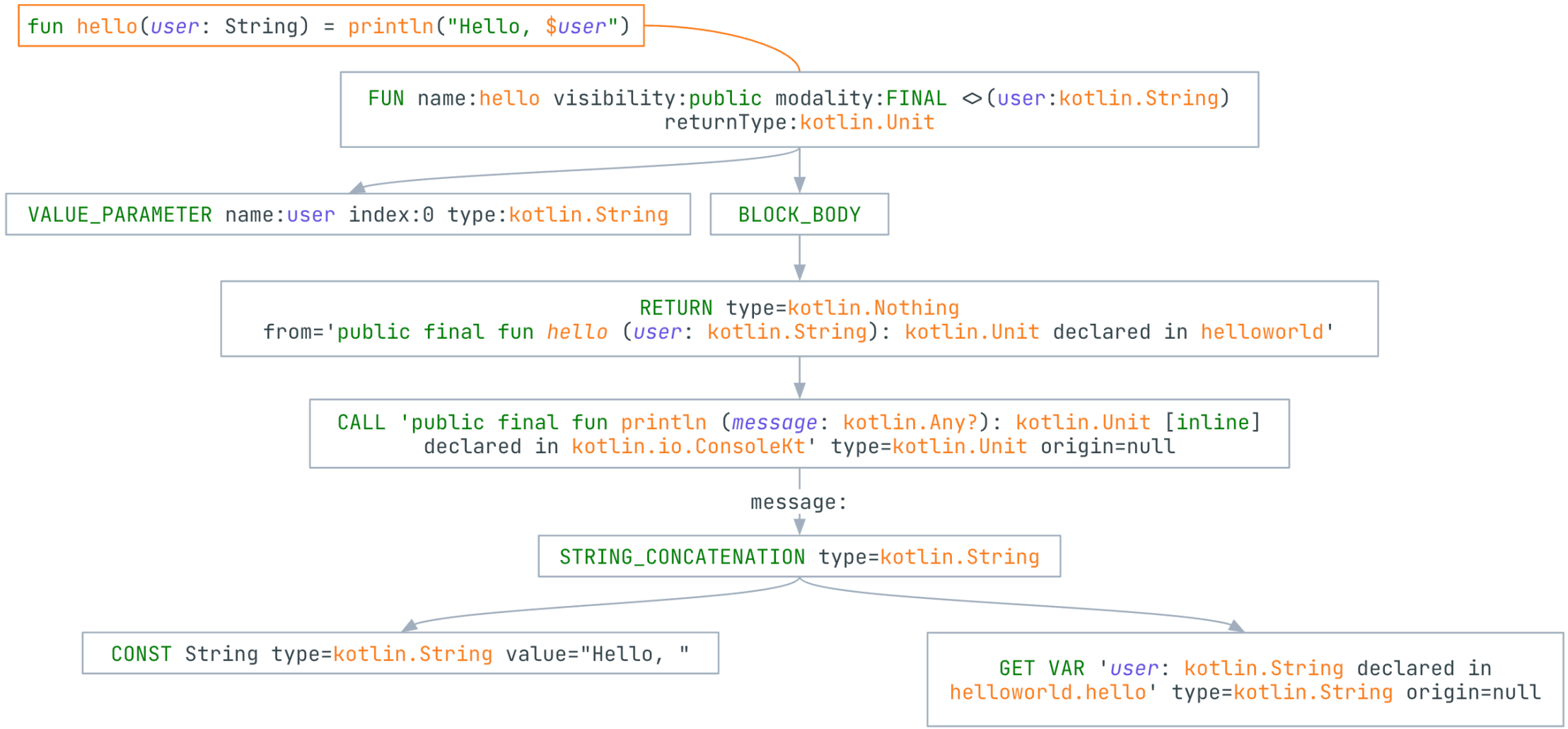
1. Устройство компилятора Kotlin
2. Как писать плагины
3. Подсветка рекомпозиции
4. Определение стабильности параметров функций

Устройство компилятора Kotlin

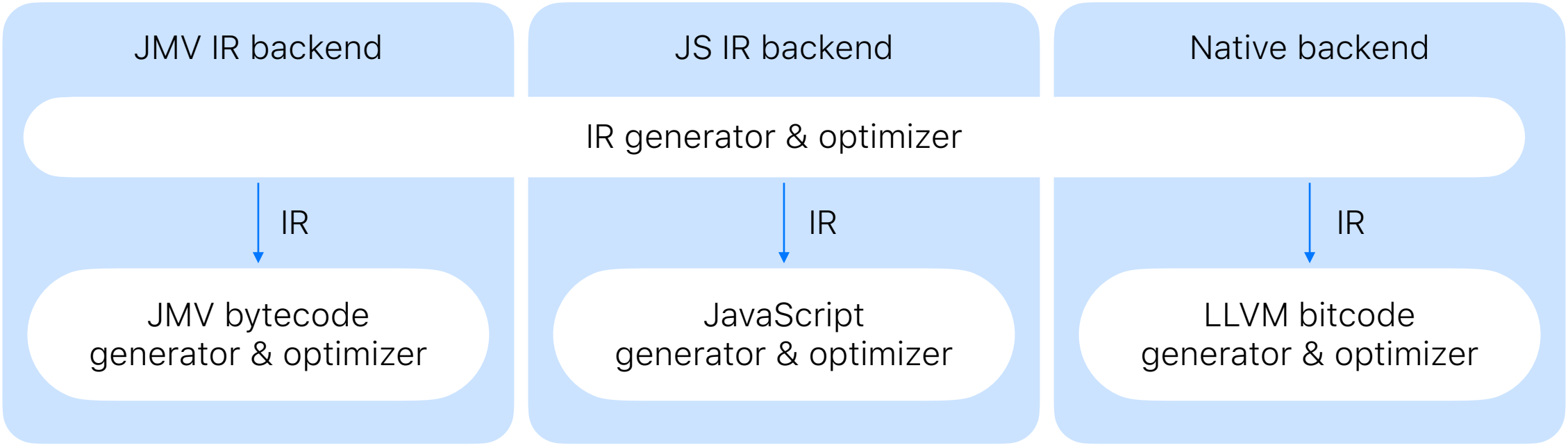
Kotlin Compiler



IR

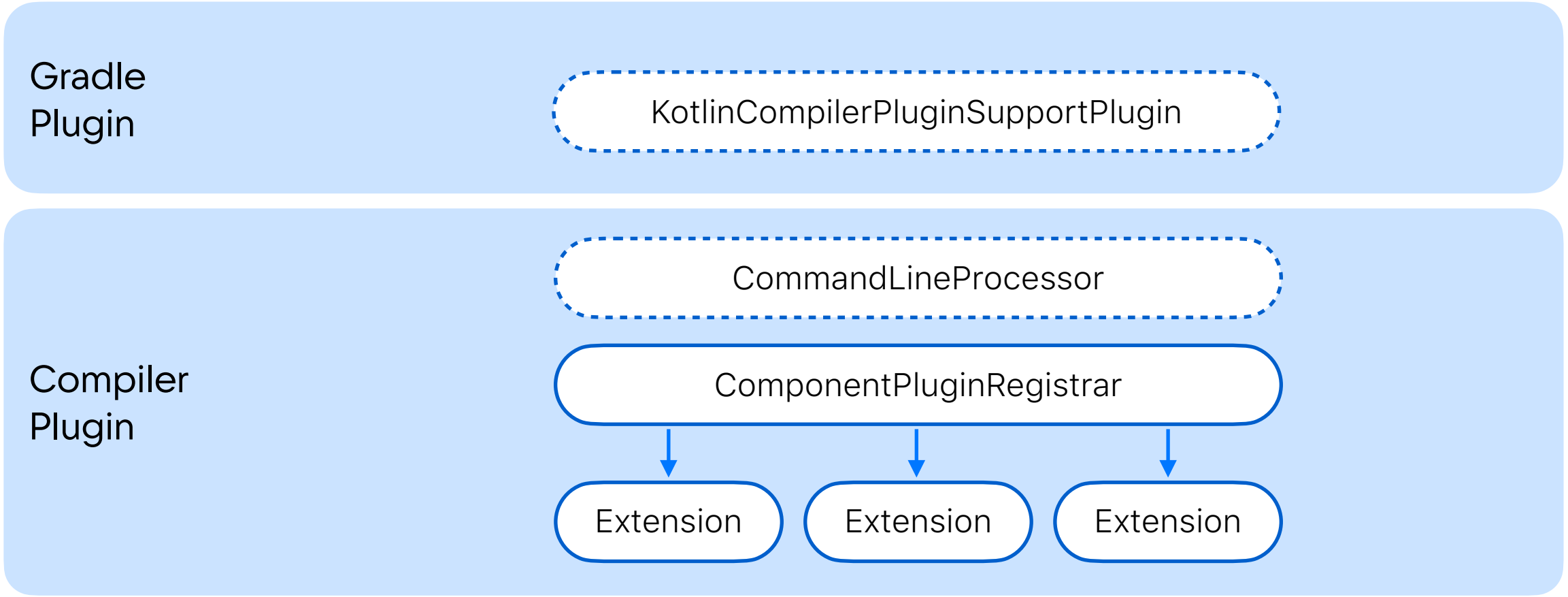

fun hello(user: String) = println("Hello, $user")
FUN name:hello visibility:public modality:FINAL <>(user:kotlin.String)
returnType:kotlin.Unit
VALUE_PARAMETER name:user index:0 type:kotlin.String
BLOCK_BODY
RETURN type=kotlin.Nothing
from='public final fun hello (user: kotlin.String): kotlin.Unit declared in helloworld'
CALL 'public final fun println (message: kotlin.Any?): kotlin.Unit [inline]
declared in kotlin.io.ConsoleKt' type=kotlin.Unit origin=null
message:
STRING_CONCATENATION type=kotlin.String
CONST String type=kotlin.String value="Hello, "
GET VAR 'user: kotlin.String declared in
helloworld.hello' type=kotlin.String origin=null

Backend


JMV IR backend
JS IR backend
Native backend
IR generator & optimizer
IR
IR
IR
JMV bytecode generator & optimizer
JavaScript generator & optimizer
LLVM bitcode generator & optimizer

Как писать плагины?

Структура


Gradle
Plugin
KotlinCompilerPluginSupportPlugin
Compiler
Plugin
CommandLineProcessor
ComponentPluginRegistrar
Extension
Extension
Extension

Компиляторный плагин в 4 шага

1. Создать модуль
2. Добавить ComplierPluginRegistrar
3. Зарегистрировать ComplierPluginRegistrar
4. Подключить модуль к проекту

Создать модуль

1 2 3 4

```
plugins {
    kotlin("jvm")
}

dependencies {
    compileOnly(kotlin("compiler-embeddable"))
}
```

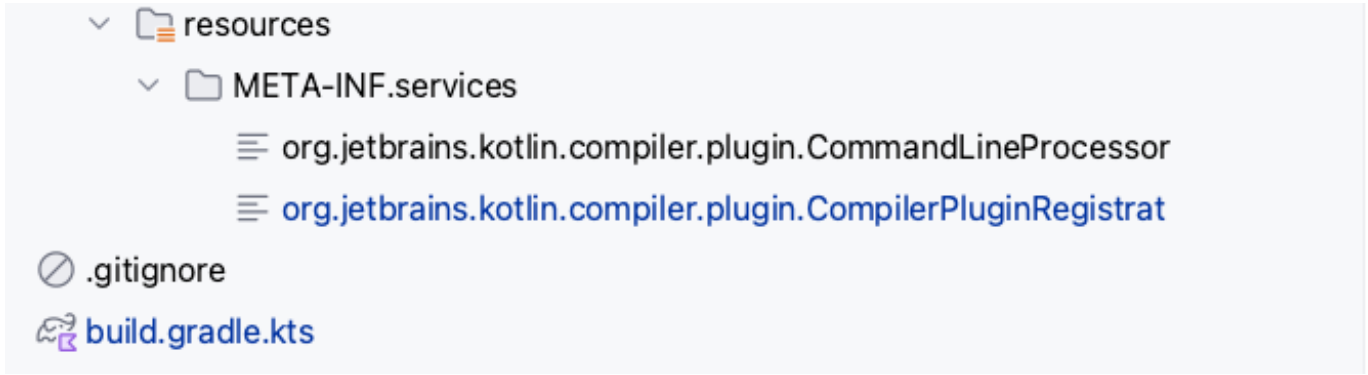
Добавить ComplierPluginRegistrar

1 2 3 4

```
@ExperimentalCompilerApi
class SamplePluginRegistrar : CompilerPluginRegistrar() {
    override val supportsK2: Boolean = true

    override fun ExtensionStorage.registerExtensions(configuration: CompilerConfiguration) {
        // register extensions
        IrGenerationExtension.registerExtension(SimpleIrGenerationExtension())
    }
}
```

Зарегистрировать ComplierPluginRegistrar

1 2 3 4




```
1  com.sample.plugin.SamplePluginRegistrar
2
3
4
5
6
7
```

Подключить модуль к проекту

1 2 3 4

```
dependencies {
    kotlinCompilerPluginClasspath(project("sample-compiler-plugin"))
}
```

Extensions

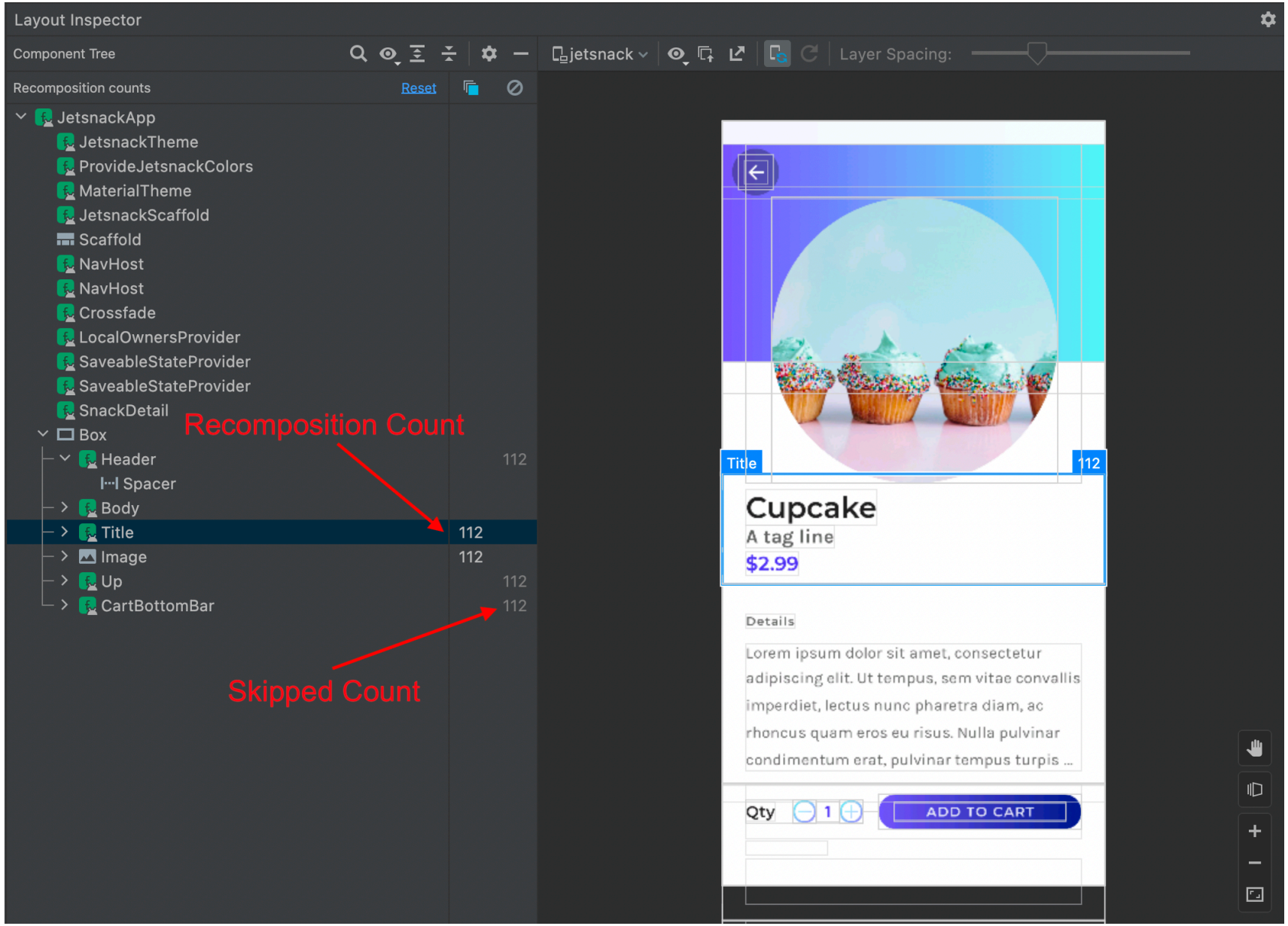
- IrGenerationExtension
- FirExtensionRegistrarAdapter
- ProcessSourcesBeforeCompilingExtension
- SyntheticJavaResolveExtension
- PackageFragmentProviderExtension
- TypeResolutionInterceptor
- ClassFileFactoryFinalizerExtension
- ExtraImportsProviderExtension
- SyntheticResolveExtension
- ScriptEvaluationExtension
- AnalysisHandlerExtension
- SyntheticScopeProviderExtension
- AssignResolutionAltererExtension
- DeclarationAttributeAltererExtension
- CandidateInterceptor
- JsSyntheticTranslateExtension
- PreprocessedVirtualFileFactoryExtension
- ClassBuilderInterceptorExtension
- CompilerConfigurationExtension
- ReplFactoryExtension
- ClassGeneratorExtension
- StorageComponentContainerContributor
- ShellExtension
- DescriptorSerializerPlugin
- ExpressionCodegenExtension
- AnalysisHandlerExtension
- UltraLightClassModifierExtension
- CollectAdditionalSourcesExtension
- TypeAttributeTranslatorExtension

Extensions

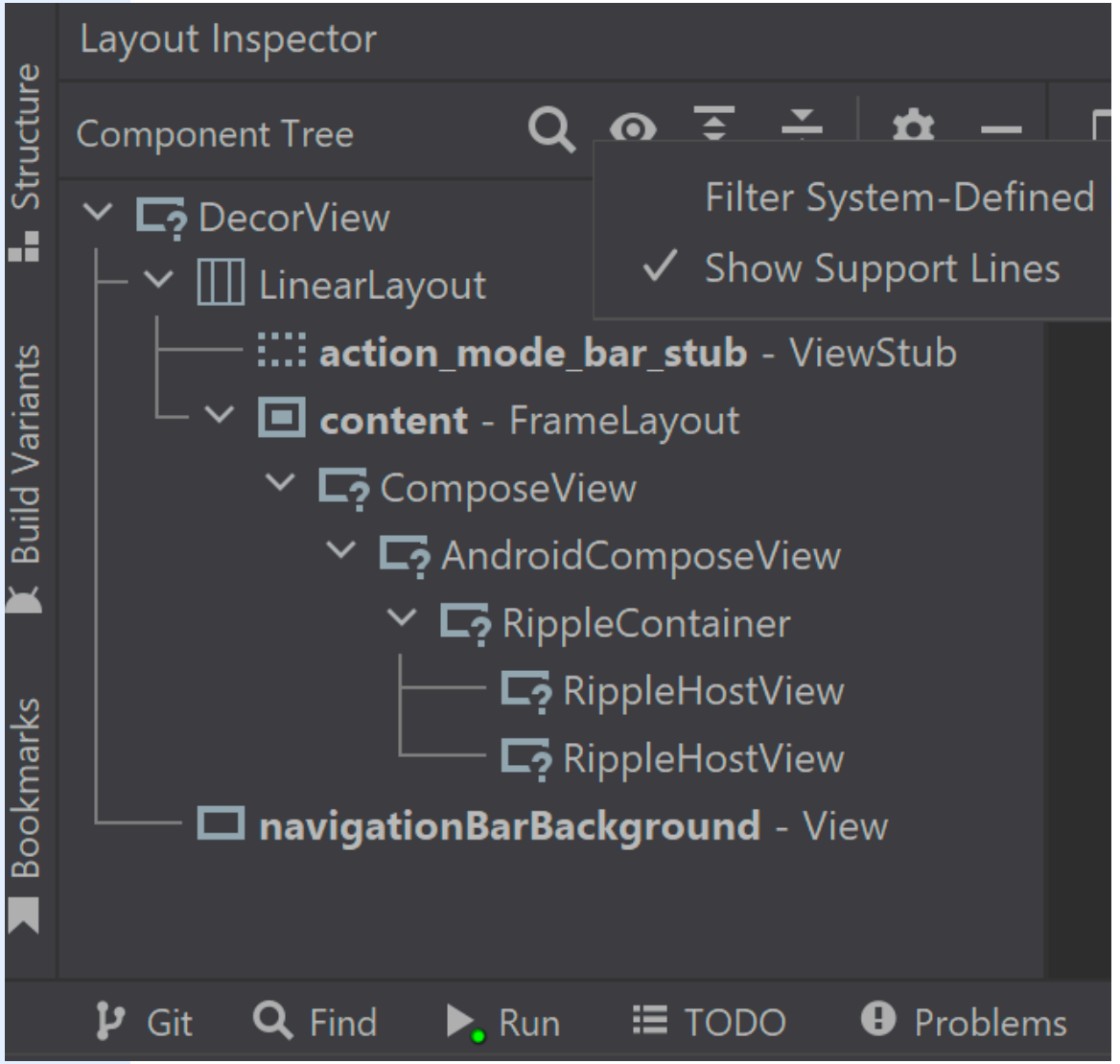
- IrGenerationExtension
- FirExtensionRegistrarAdapter

Тестирование

Количество рекомпозиций


Layout Inspector
Component Tree
Recomposition counts
Reset
JetsnackApp
JetsnackTheme
ProvideJetsnackColors
MaterialTheme
JetsnackScaffold
Scaffold
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
SnackDetail
Box
Header 112
Spacer
Body
Title 112
Image 112
Up 112
CartBottomBar 112
Recomposition Count
Skipped Count
jetsnack
Layer Spacing:
Title
112
Cupcake
A tag line
$2.99
Details
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut tempus, sem vitae convallis imperdiet, lectus nunc pharetra diam, ac rhoncus quam eros eu risus. Nulla pulvinar condimentum erat, pulvinar tempus turpis ...
Qty
1
ADD TO CART

Что не так с LayoutInspector?



Какие проблемы с RecomposeCounter?

- Необходимость прописывать
- Спам в логи
- Сложность обнаружения аномалий

```
@Composable
inline fun RecomposeCounter(name: String) {
    val ref = remember { Ref() }
    SideEffect { ref.count++ }
    Log.d("Recomposes", "${ref.count}\n$name")
}
```

Идея с подстветкой

Как будем прокидывать?

```
Greeting("Android", modifier)

Greeting("Android", Modifier)

Greeting("Android", Modifier.fillMaxSize())

Greeting("Android")
```

Как будем прокидывать?

```
Greeting("Android", modifier.then(HighlighterModifier))

Greeting("Android", Modifier.then(HighlighterModifier))

Greeting("Android", Modifier.fillMaxSize().then(HighlighterModifier))

Greeting("Android", DefaultModifier.then(HighlighterModifier))
```

Как будем прокидывать?

```
Greeting("Android", modifier.then(HighlighterModifier))

Greeting("Android", Modifier.then(HighlighterModifier))

Greeting("Android", Modifier.fillMaxSize().then(HighlighterModifier))

Greeting("Android", DefaultModifier.then(HighlighterModifier))
```

```
class RecomposeHighlighterIrGeneration : IrGenerationExtension {

    override fun generate(moduleFragment: IrModuleFragment, pluginContext: IrPluginContext) =
        moduleFragment.transformChildrenVoid(RecomposeHighlighterTransformer(pluginContext))
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {

    private val highlighterModifier by lazy {
        pluginContext.referenceClass(ClassId(
            FqName("com.vk.recompose.highlighter"),
            Name.identifier("HighlighterModifier")
        ))
    }

    private val thenFunction by lazy {
        pluginContext.referenceFunctions(CallableId(
            ClassId(
              FqName("androidx.compose.ui"),
              Name.identifier("Modifier")
            ),
            Name.identifier("then"))
        ).firstOrNull()?.owner?.symbol
    }
    ...
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {

    private val highlighterModifier by lazy {
        pluginContext.referenceClass(ClassId(
            FqName("com.vk.recompose.highlighter"),
            Name.identifier("HighlighterModifier")
        ))
    }

    private val thenFunction by lazy {
        pluginContext.referenceFunctions(CallableId(
            ClassId(
              FqName("androidx.compose.ui"),
              Name.identifier("Modifier")
            ),
            Name.identifier("then"))
        ).firstOrNull()?.owner?.symbol
    }
    ...
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {
    ...

    override fun visitCall(expression: IrCall): IrExpression {
        val thenFunction = thenFunction
        val highlighterModifier = highlighterModifier

        if (thenFunction != null
            && highlighterModifier != null
            && expression.symbol.owner.isComposable()
        ) {
            expression.transformModifierArguments(
                highlighterModifier,
                thenFunction
            )
        }

        return super.visitCall(expression)
    }
    ...
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {
    ...

    override fun visitCall(expression: IrCall): IrExpression {
        val thenFunction = thenFunction
        val highlighterModifier = highlighterModifier

        if (thenFunction != null
            && highlighterModifier != null
            && expression.symbol.owner.isComposable()
        ) {
            expression.transformModifierArguments(
                highlighterModifier,
                thenFunction
            )
        }

        return super.visitCall(expression)
    }
    ...
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {
    ...

    override fun visitCall(expression: IrCall): IrExpression {
        val thenFunction = thenFunction
        val highlighterModifier = highlighterModifier

        if (thenFunction != null
            && highlighterModifier != null
            && expression.symbol.owner.isComposable()
        ) {
            expression.transformModifierArguments(
                highlighterModifier,
                thenFunction
            )
        }

        return super.visitCall(expression)
    }
    ...
}
```

```
private val Composable = FqName("androidx.compose.runtime.Composable")

private fun IrFunction.isComposable(): Boolean {
    return hasAnnotation(Composable)
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {
    ...

     override fun visitCall(expression: IrCall): IrExpression {
         val thenFunction = thenFunction
         val highlighterModifier = highlighterModifier

         if (thenFunction != null
             && highlighterModifier != null
             && expression.symbol.owner.isComposable()
         ) {
             expression.transformModifierArguments(
                 highlighterModifier,
                 thenFunction
             )
         }

         return super.visitCall(expression)
    }
    ...
}
```

```
private val Composable = FqName("androidx.compose.runtime.Composable")

private fun IrFunction.isComposable(): Boolean {
    return hasAnnotation(Composable)
}
```

```
internal class RecomposeHighlighterTransformer(
    private val pluginContext: IrPluginContext
) : IrElementTransformerVoid() {
    ...

     override fun visitCall(expression: IrCall): IrExpression {
         val thenFunction = thenFunction
         val highlighterModifier = highlighterModifier

         if (thenFunction != null
             && highlighterModifier != null
             && expression.symbol.owner.isComposable()
         ) {
             expression.transformModifierArguments(
                 highlighterModifier,
                 thenFunction
             )
         }

         return super.visitCall(expression)
    }
    ...
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrType.isComposeModifier(): Boolean =
    classFqName?.asString() == "androidx.compose.ui.Modifier"
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
Greeting("Android", Modifier.fillMaxSize())
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

Greeting("Android", Modifier)

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

Greeting("Android", modifier)

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

```
private fun IrCall.transformModifierArguments(
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
) {
    for (index in 0..<valueArgumentsCount) {
        val parameter = symbol.owner.valueParameters[index]

        if (parameter.isVararg || !parameter.type.isComposeModifier()) continue

        val expression = getValueArgument(index) ?: parameter.defaultValue?.expression

        if (expression is IrCall || expression is IrGetObjectValue || expression is IrGetValue) {
            val modifiedArgumentExpression = createHighlightedModifierArgument(
                parameter,
                expression,
                highlighterModifier,
                thenFunction
            )
            putValueArgument(index, modifiedArgumentExpression)
        }
    }
}
```

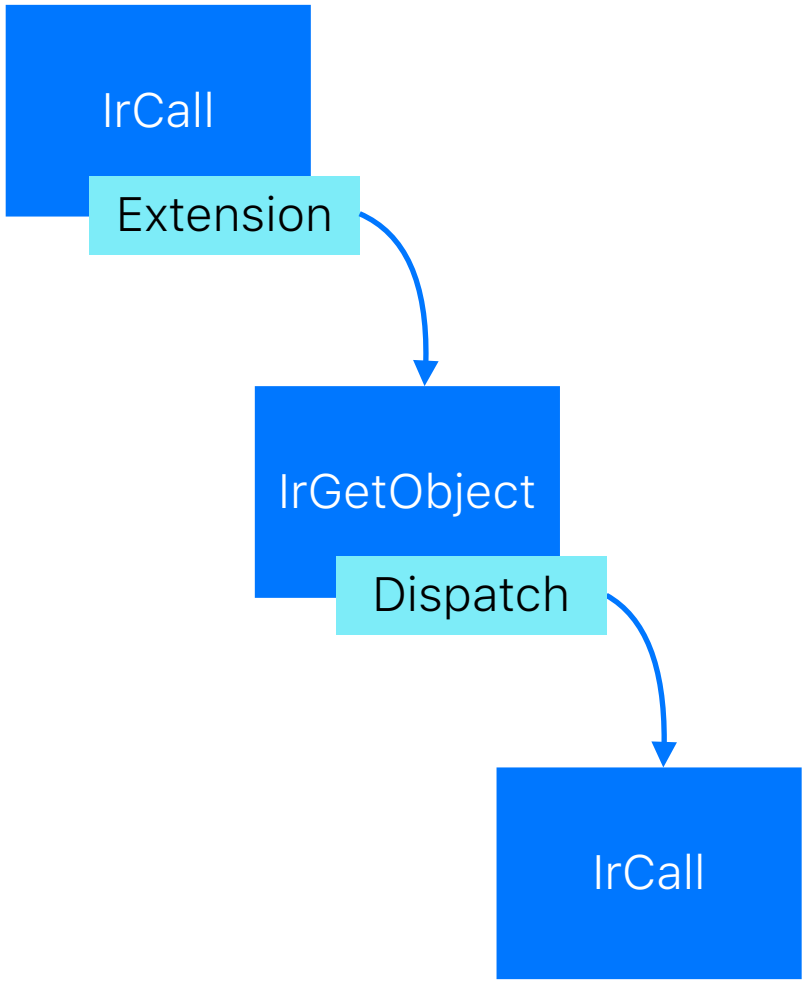
```
private fun createHighlightedModifierArgument(
    parameter: IrValueParameter,
    argumentExpression: IrExpression,
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
): IrExpression {

    val highlighterObject = IrGetObjectValueImpl(
        startOffset = UNDEFINED_OFFSET,
        endOffset = UNDEFINED_OFFSET,
        type = parameter.type,
        symbol = highlighterModifier
    )

    val thenCall = IrCallImpl(
        startOffset = UNDEFINED_OFFSET,
        endOffset = UNDEFINED_OFFSET,
        type = parameter.type,
        symbol = thenFunction,
        typeArgumentsCount = 0,
        valueArgumentsCount = 1,
    )
    thenCall.putValueArgument(0, highlighterObject)
    thenCall.dispatchReceiver = argumentExpression
    return thenCall
}
```

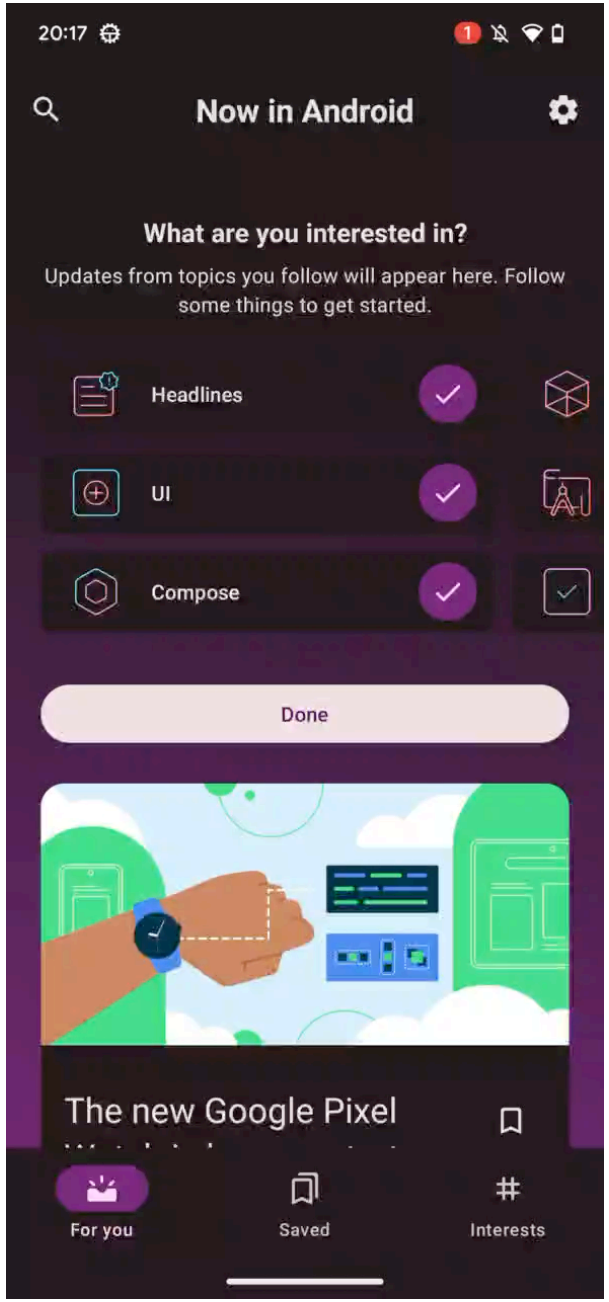
```
private fun createHighlightedModifierArgument(
    parameter: IrValueParameter,
    argumentExpression: IrExpression,
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
): IrExpression {

    val highlighterObject = IrGetObjectValueImpl(
        startOffset = UNDEFINED_OFFSET,
        endOffset = UNDEFINED_OFFSET,
        type = parameter.type,
        symbol = highlighterModifier
    )

    val thenCall = IrCallImpl(
        startOffset = UNDEFINED_OFFSET,
        endOffset = UNDEFINED_OFFSET,
        type = parameter.type,
        symbol = thenFunction,
        typeArgumentsCount = 0,
        valueArgumentsCount = 1,
    )
    thenCall.putValueArgument(0, highlighterObject)
    thenCall.dispatchReceiver = argumentExpression
    return thenCall
}
```

```
private fun createHighlightedModifierArgument(
    parameter: IrValueParameter,
    argumentExpression: IrExpression,
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
): IrExpression {

    val highlighterObject...
    val thenCall...

    thenCall.putValueArgument(0, highlighterObject)
    thenCall.dispatchReceiver = argumentExpression
    return thenCall
}
```

```
private fun createHighlightedModifierArgument(
    parameter: IrValueParameter,
    argumentExpression: IrExpression,
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
): IrExpression {

    val highlighterObject...
    val thenCall...

    thenCall.putValueArgument(0, highlighterObject)
    thenCall.dispatchReceiver = argumentExpression
    return thenCall
}
```

```
private fun createHighlightedModifierArgument(
    parameter: IrValueParameter,
    argumentExpression: IrExpression,
    highlighterModifier: IrClassSymbol,
    thenFunction: IrSimpleFunctionSymbol
): IrExpression {

    val highlighterObject...
    val thenCall...

    thenCall.putValueArgument(0, highlighterObject)
    thenCall.dispatchReceiver = argumentExpression
    return thenCall
}
```




20:17
Now in Android
What are you interested in?
Updates from topics you follow will appear here. Follow some things to get started.
Headlines
UI
Compose
Done
The new Google Pixel
For you
Saved
Interests

Или нет?

```
Greeting("Android", modifier.then(HighlighterModifier))

Greeting("Android", Modifier.then(HighlighterModifier))

Greeting("Android", Modifier.fillMaxSize().then(HighlighterModifier))

Greeting("Android", DefaultModifier.then(HighlighterModifier))
```

Не работает

Порядок имеет значение

```
dependencies {
    kotlinCompilerPluginClasspath(project(":compiler-plugin1"))
    kotlinCompilerPluginClasspath(project(":compiler-plugin2"))
    kotlinCompilerPluginClasspath(project(":compiler-plugin3"))
}
```

Compose ломает

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

Compose ломает

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    composer: Composer,
    changed: Int,
    default: Int
) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (default and 0b0001 !== 0) {
        dirty = dirty or 0b0110
    } else if (changed and 0b1110 === 0) {
        dirty = dirty or if (composer.changed(modifier)) 0b0100 else 0b0010
    }
    if (default and 0b0010 !== 0) {
        dirty = dirty or 0b00110000
    } else if (changed and 0b01110000 === 0) {
        dirty = dirty or if (composer.changed(name)) 0b00100000 else 0b00010000
    }
    if (dirty and 0b01011011 !== 0b00010010 || !composer.skipping) {
        if (default and 0b0001 !== 0) {
            modifier = Modifier.Companion
        }
        if (default and 0b0010 !== 0) {
            name = "Android"
        }
        Text("Hello $name!", modifier, composer, 0b1110 and dirty, 0)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope {
        composer: Composer, force: Int ->
        Greeting(modifier, name, composer, updateChangedFlags(changed or 0b0001), default)
    }
}
```

Compose ломает

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier
}
```

```
@Composable
fun Greeting(
    ...
    default: Int
) {
    ...
    if (dirty and 0b01011011 !== 0b00010010 || !composer.skipping) {
        if (default and 0b0001 !== 0) {
            modifier = Modifier.Companion
        }
        if (default and 0b0010 !== 0) {
            name = "Android"
        }
        Text("Hello $name!", modifier, composer, 0b1110 and dirty, 0)
    } else {
        composer.skipToGroupEnd()
    }
    ...
}
```

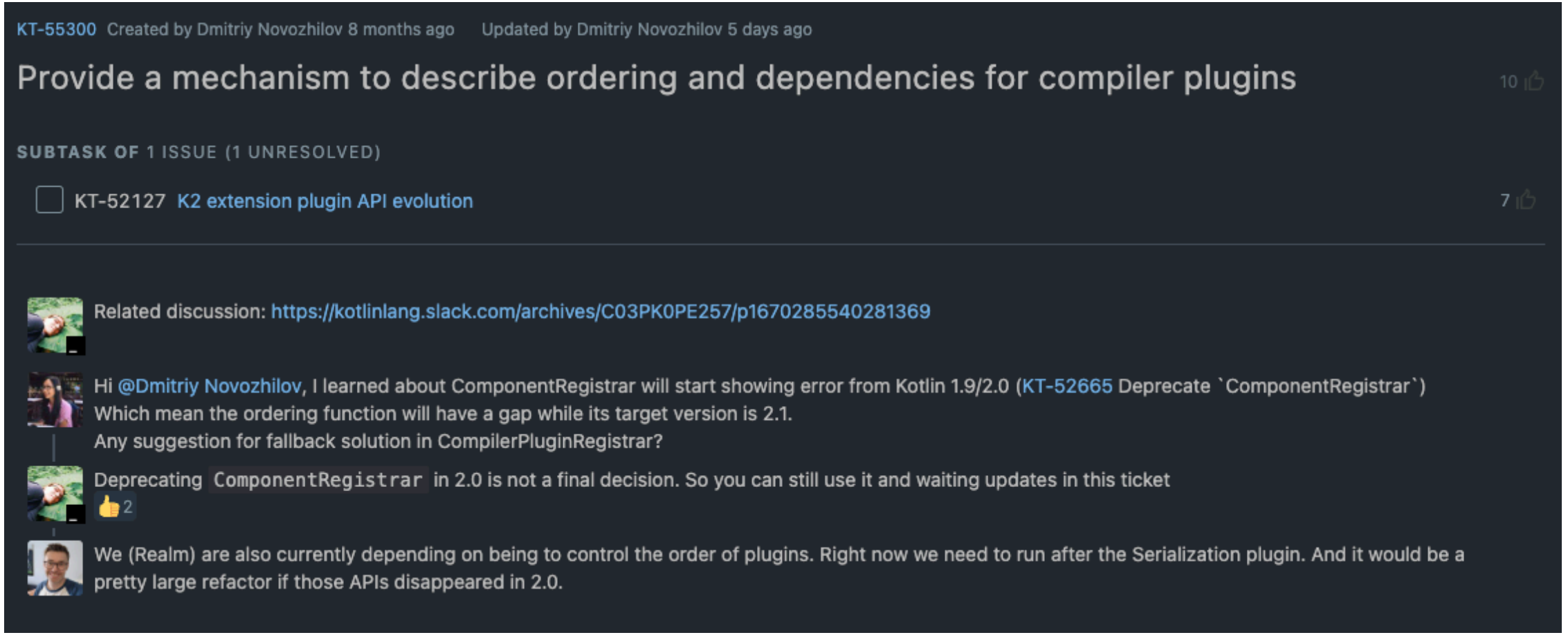
Меняем порядок плагинов

```
@ExperimentalCompilerApi
class RecomposeHighlighterPluginRegistrar : CompilerPluginRegistrar() {

    override fun ExtensionStorage.registerExtensions(configuration: CompilerConfiguration) {
        IrGenerationExtension.registerExtension(RecomposeHighlighterIrGeneration())
    }

    override val supportsK2: Boolean = true
}
```

Меняем порядок плагинов

```
@ExperimentalCompilerApi
class RecomposeHighlighterComponentRegistrar : ~~ComponentRegistrar~~ {
class RecomposeHighlighterPluginRegistrar : CompilerPluginRegistrar() {

    override fun registerProjectComponents(
        project: MockProject,
        configuration: CompilerConfiguration
    ) {
        project.extensionArea.getExtensionPoint(IrGenerationExtension.extensionPointName)
            .~~registerExtension~~(RecomposeHighlighterIrGeneration(), LoadingOrder.FIRST)
    }

    override fun ExtensionStorage.registerExtensions(configuration: CompilerConfiguration) {
        IrGenerationExtension.registerExtension(RecomposeHighlighterIrGeneration())
    }

    override val supportsK2: Boolean = true
}
```

Это безопасно!



https://youtrack.jetbrains.com/issue/KT-55300

Анализ кода

Rules & Guidelines

- Compose Guidelines
  - compose-api-guidelines
  - compose-component-api-guidelines
- X(formely Twitter) Rules

Как проверять стабильность параметров функции и на что это влияет?

Если параметр стабильный

```
@Composable
fun WithStable(stable: String) {
    Text(text = stable)
}
```

Если параметр стабильный

```
@Composable
fun WithStable(stable: String) {
    Text(text = stable)
}
```

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed and 0b1110 === 0) {
        dirty = dirty or if (composer.changed(stable)) 0b0100 else 0b0010
    }
    if (dirty and 0b1011 !== 0b0010 || !composer.skipping) {
        Text(stable, null, composer, 0b1110 and dirty, 0b0010)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or 0b0001))
    }
}
```

Если параметр стабильный

```
@Composable
fun WithStable(stable: String) {
    Text(text = stable)
}
```

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed and 0b1110 === 0) {
        dirty = dirty or if (composer.changed(stable)) 0b0100 else 0b0010
    }
    if (dirty and 0b1011 !== 0b0010 || !composer.skipping) {
        Text(stable, null, composer, 0b1110 and dirty, 0b0010)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or 0b0001))
    }
}
```

Если параметр стабильный

```
@Composable
fun WithStable(stable: String) {
    Text(text = stable)
}
```

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

- *ForceRecompose* - флаг принудительной рекомпозиции

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

- *Uncertain* - изменение аргумента неопределено
- *Different* - аргумент не изменился
- *Same* - аргумент не изменился с предыдущей рекомпозиции

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

- *Same* - аргумент не изменился с предыдущей рекомпозиции
- *Static* - аргумент статичен и не будет меняться в runtime
- composer.skipping - состояние Compose, в котором игнорируется пропуск рекомпозиции функции
- *MaskConstant* - вспомогательная маска, генерируемая компилятором
- *DefaultModifier* - флаг того, что используется Modifier по умолчанию

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

Перезапускаемая?

Пропускаемая?

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

Перезапускаемая? ✅

Пропускаемая?

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Если параметр стабильный

Перезапускаемая? ✅

Пропускаемая? ✅

```
@Composable
fun WithStable(stable: String, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (changed === Uncertain) {
        dirty = dirty or if (composer.changed(stable)) Different else Same
    }
    if ((dirty !== Same && dirty !== Static) || !composer.skipping) {
        Text(stable, null, composer, MaskConstant and dirty, DefaultModifier)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithStable(stable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

А если нестабильный?

```
@Composable
fun WithUnstable(unstable: Any) {
    Text(text = unstable.toString())
}
```

А если нестабильный?

```
@Composable
fun WithUnstable(unstable: Any) {
    Text(text = unstable.toString())
}
```

```
@Composable
fun WithUnstable(unstable: Any, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)

    Text(unstable.toString(), null, composer, 0, DefaultModifier)

    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithUnstable(unstable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

А если нестабильный?

Перезапускаемая?

Пропускаемая?

```
@Composable
fun WithUnstable(unstable: Any, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)

    Text(unstable.toString(), null, composer, 0, DefaultModifier)

    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithUnstable(unstable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

А если нестабильный?

Перезапускаемая? ✅

Пропускаемая?

```
@Composable
fun WithUnstable(unstable: Any, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)

    Text(unstable.toString(), null, composer, 0, DefaultModifier)

    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithUnstable(unstable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

А если нестабильный?

Перезапускаемая? ✅

Пропускаемая? ❌

```
@Composable
fun WithUnstable(unstable: Any, composer: Composer, changed: Int) {
    val composer = composer.startRestartGroup(74386374)

    Text(unstable.toString(), null, composer, 0, DefaultModifier)

    composer.endRestartGroup()?.updateScope { composer: Composer, force: Int ->
        WithUnstable(unstable, composer, updateChangedFlags(changed or ForceRecompose))
    }
}
```

Суть задачи

Найти перезапускаемые функции и определить почему они непропускаемые

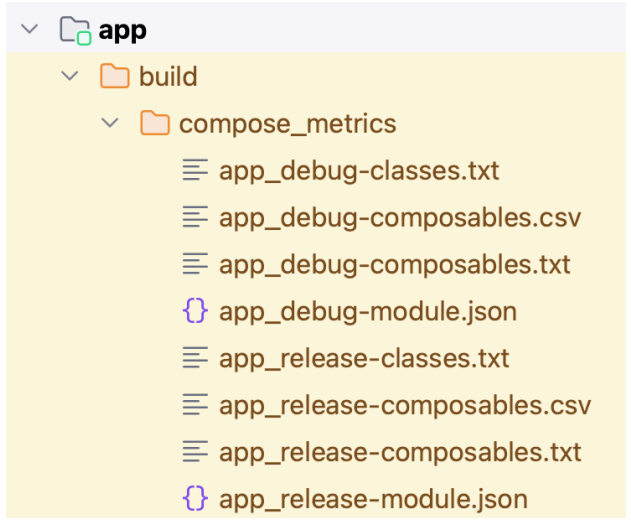
Решение из коробки

```
kotlinOptions {
    val metricsPath = "${project.buildDir.absolutePath}/compose_metrics"
    freeCompilerArgs += listOf(
        "-P",
        "plugin:androidx.compose.compiler.plugins.kotlin:reportsDestination=$metricsPath",
        "-P",
        "plugin:androidx.compose.compiler.plugins.kotlin:metricsDestination=$metricsPath"
    )
}
```

https://developer.android.com/jetpack/compose/performance/stability/diagnose#compose-compiler
https://github.com/androidx/androidx/blob/androidx-main/compose/compiler/design/compiler-metrics.md

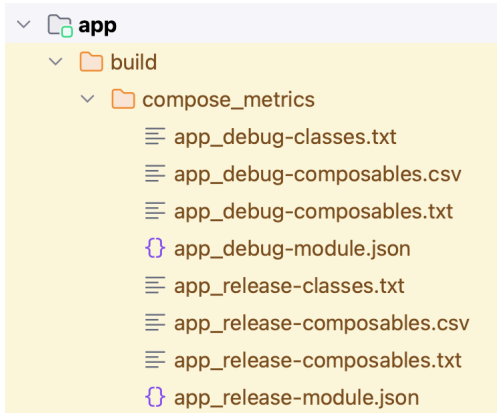
Решение из коробки

```
kotlinOptions {
    val metricsPath = "${project.buildDir.absolutePath}/compose_metrics"
    freeCompilerArgs += listOf(
        "-P",
        "plugin:androidx.compose.compiler.plugins.kotlin:reportsDestination=$metricsPath",
        "-P",
        "plugin:androidx.compose.compiler.plugins.kotlin:metricsDestination=$metricsPath"
    )
}
```



https://developer.android.com/jetpack/compose/performance/stability/diagnose#compose-compiler
https://github.com/androidx/androidx/blob/androidx-main/compose/compiler/design/compiler-metrics.md

Решение из коробки




```
restartable fun WithUnStable(
  unstable unstable: Any
)
restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun WithStable(
  stable stable: String
)
```

https://developer.android.com/jetpack/compose/performance/stability/diagnose#compose-compiler
https://github.com/androidx/androidx/blob/androidx-main/compose/compiler/design/compiler-metrics.md

Какие минусы?

- Отчет build папке
- Необходимость сборки модуля без кэша

```
./gradlew compileKotlin --rerun-tasks
```

А что если проверять метрики на CI?

~~А что если проверять метрики на CI?~~

А может компиляторый плагин?

Какие функции могут быть перезапускаемыми?

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- **Не пустая функция**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- **Содержит Composer в аргументах**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- Содержит Composer в аргументах
- **Не inline функция**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- Содержит Composer в аргументах
- Не inline функция
- **Не лямбда**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- Содержит Composer в аргументах
- Не inline функция
- Не лямбда
- **Возвращает Unit**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- Содержит Composer в аргументах
- Не inline функция
- Не лямбда
- Возвращает Unit
- **Не содержит аннотаций @NonRestartableComposable и @ExplicitGroupsComposable**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть перезапускаемыми?

- Не пустая функция
- Содержит Composer в аргументах
- Не inline функция
- Не лямбда
- Возвращает Unit
- Не содержит аннотаций @NonRestartableComposable и @ExplicitGroupsComposable
- **Не делегирует вызов в другую @Composable функцию**

```
fun IrFunction.isRestartable(): Boolean = when {
    body == null || this !is IrSimpleFunction -> false
    composerParam() == null -> false
    isInline -> false
    isLambda -> false
    origin == IrDeclarationOrigin.LOCAL_FUNCTION_FOR_LAMBDA -> false
    isLocal && parentClassOrNull?.origin != JvmLoweredDeclarationOrigin.LAMBDA_IMPL -> false
    !returnType.isUnit() -> false
    hasAnnotation(NonRestartableComposable) -> false
    hasAnnotation(ExplicitGroupsComposable) -> false
    else -> !isComposableDelegatedAccessor
}
```

Какие функции могут быть пропускаемыми?

Перезапускаемые функции, у которых:

1. Все параметры стабильные
2. Нестабильные параметры имеют значение по умолчанию
3. Нестабильные параметры не используются

Определение стабильности

Stability.kt

```
private fun stabilityOf(
    classifier: IrClassifierSymbol,
    substitutions: Map<IrTypeParameterSymbol, IrTypeArgument>,
    currentlyAnalyzing: Set<IrClassifierSymbol>
): Stability {...}

private fun stabilityOf(
    argument: IrTypeArgument,
    substitutions: Map<IrTypeParameterSymbol, IrTypeArgument>,
    currentlyAnalyzing: Set<IrClassifierSymbol>
): Stability {...}

private fun stabilityOf(
    type: IrType,
    substitutions: Map<IrTypeParameterSymbol, IrTypeArgument>,
    currentlyAnalyzing: Set<IrClassifierSymbol>
): Stability {...}
```

androidx.compose.compiler.plugins.kotlin.analysis.Stability.kt

Определение отсутствия значения по умолчанию

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

Определение отсутствия значения по умолчанию

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    composer: Composer,
    changed: Int,
    default: Int
) {
    val composer = composer.startRestartGroup(74386374)
    var dirty = changed
    if (default and 0b0001 !== 0) {
        dirty = dirty or 0b0110
    } else if (changed and 0b1110 === 0) {
        dirty = dirty or if (composer.changed(modifier)) 0b0100 else 0b0010
    }
    if (default and 0b0010 !== 0) {
        dirty = dirty or 0b00110000
    } else if (changed and 0b01110000 === 0) {
        dirty = dirty or if (composer.changed(name)) 0b00100000 else 0b00010000
    }
    if (dirty and 0b01011011 !== 0b00010010 || !composer.skipping) {
        if (default and 0b0001 !== 0) {
            modifier = Modifier.Companion
        }
        if (default and 0b0010 !== 0) {
            name = "Android"
        }
        Text("Hello $name!", modifier, composer, 0b1110 and dirty, 0)
    } else {
        composer.skipToGroupEnd()
    }
    composer.endRestartGroup()?.updateScope {
        composer: Composer, force: Int ->
        Greeting(modifier, name, composer, updateChangedFlags(changed or 0b0001), default)
    }
}
```

Определение отсутствия значения по умолчанию

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    default: Int,
    ...
) {
    ...
    if (...) {
        ...
        if (default and 0b0001 !== 0) {
            modifier = Modifier.Companion
        }
        if (default and 0b0010 !== 0) {
            name = "Android"
        }
        Text("Hello $name!", modifier,...)
    } else {
        ...
    }
}
```

Определение отсутствия значения по умолчанию

Значение по умолчанию отсутствует, если количество **set** меньше **1**

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    default: Int,
    ...
) {
    ...
    if (...) {
        ...
        if (default and 0b0001 !== 0) {
            modifier = Modifier.Companion
        }
        if (default and 0b0010 !== 0) {
            name = "Android"
        }
        Text("Hello $name!", modifier,...)
    } else {
        ...
    }
}
```

Определение используемости

```
@Composable
fun Greeting(
    modifier: Modifier = Modifier,
    name: String = "Android"
) {
    Text(text = "Hello $name!", modifier = modifier)
}
```

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    ...
) {
    ...
    if (...) {
        ...
        Text("Hello $name!", modifier,...)
    } else {
        ...
    }
    composer.endRestartGroup()?.updateScope { ... ->
        Greeting(modifier, name,...)
    }
}
```

Определение используемости

Используется, если количество **get** больше **1**

```
@Composable
fun Greeting(
    modifier: Modifier,
    name: String,
    ...
) {
    ...
    if (...) {
        ...
        Text("Hello $name!", modifier,...)
    } else {
        ...
    }
    composer.endRestartGroup()?.updateScope { ... ->
        Greeting(modifier, name,...)
    }
}
```

Глубокая вложенность

```
@Composable
fun SimpleComposable(input: String) {

    @Composable
    fun LocalComposable() {

        val lambdaComposable = @Composable {

            val anonymousComposable = object : @Composable () -> Unit {

                @Composable
                override fun invoke() {
                    Text(text = input)
                }
            }
        }
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    override fun visitFunction(declaration: IrFunction) {
        super.visitFunction(declaration)
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        currentFunctionInfo = FunctionInfo(
            function = declaration,
            parent = currentFunctionInfo
        )

        super.visitFunction(declaration)

        val functionInfo = currentFunctionInfo
        currentFunctionInfo = currentFunctionInfo?.parent
        ...
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        currentFunctionInfo = FunctionInfo(
            function = declaration,
            parent = currentFunctionInfo
        )

        super.visitFunction(declaration)

        val functionInfo = currentFunctionInfo
        currentFunctionInfo = currentFunctionInfo?.parent
        ...
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        currentFunctionInfo = FunctionInfo(
            function = declaration,
            parent = currentFunctionInfo
        )

        super.visitFunction(declaration)

        val functionInfo = currentFunctionInfo
        currentFunctionInfo = currentFunctionInfo?.parent
        ...
    }
}
```

```
class FunctionInfo(
    val function: IrFunction,
    val parent: FunctionInfo? = null
) {
    ...
}
```

```
class FunctionInfo(
    val function: IrFunction,
    val parent: FunctionInfo? = null
) {
    ...
}
```

```
class FunctionInfo(
    val function: IrFunction,
    val parent: FunctionInfo? = null
) {

    private var notGeneratedValueParametersCount: Int = ...

    ...

    val trackingParams = buildList {
        function.extensionReceiverParameter?.let(::add)
        addAll(function.valueParameters.take(notGeneratedValueParametersCount))
        function.dispatchReceiverParameter?.let(::add)
    }

    val paramsGetCount = IntArray(trackingParams.size) { 0 }
    val paramsSetCount = IntArray(trackingParams.size) { 0 }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        ...
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        ...
    }

    override fun visitSetValue(expression: IrSetValue) {
        iterateOverParamsInFunctionChain(expression) { info, index ->
            ++info.paramsSetCount[index]
        }
        super.visitSetValue(expression)
    }

    override fun visitGetValue(expression: IrGetValue) {
        iterateOverParamsInFunctionChain(expression) { info, index ->
            ++info.paramsGetCount[index]
        }
        super.visitGetValue(expression)
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
        ...
    }

    override fun visitSetValue(expression: IrSetValue) {
        ...
    }

    override fun visitGetValue(expression: IrGetValue) {
        ...
    }
}
```

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      if (functionInfo?.function?.isRestartable() != true) return
      ...
}
```

Функция
непропускаемая,
если параметр:

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

Функция непропускаемая, если параметр:

- Не стабилен

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

Функция непропускаемая, если параметр:

- Не стабилен
- Не имеет значения по умолчанию

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

Функция непропускаемая, если параметр:

- Не стабилен
- Не имеет значения по умолчанию
- Используется

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

```kotlin
fun IrType.isFromCompose(): Boolean {
    return classFqName?.asString()?.startsWith("androidx.compose") == true
}
```

Функция непропускаемая, если параметр:

- Не стабилен
- Не имеет значения по умолчанию
- Используется
- Не из Compose

```kotlin
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

Функция непропускаемая, если параметр:

- Не стабилен
- Не имеет значения по умолчанию
- Используется
- Не из Compose

```
class SkippableCheckerIrElementVisitor(
    ...
) : IrElementVisitorVoid {

    private var currentFunctionInfo: FunctionInfo? = null

    override fun visitFunction(declaration: IrFunction) {
      ...
      val unstableParams = mutableSetOf<IrValueParameter>()
      functionInfo.trackingParams.forEachIndexed { paramIndex, param ->

        val stability = stabilityOf(param.varargElementType ?: param.type)
        val isUnstable = stability.knownUnstable()
        val isRequired = functionInfo.paramsSetCount[paramIndex] < 1
        val isUsed = functionInfo.paramsGetCount[paramIndex] > 1
        val isNotFromCompose = !param.type.isFromCompose()

        if (isUnstable && isRequired && isUsed && isNotFromCompose) {
            unstableParams += param
        }
      }
      val isNotSkippable = unstableParams.isNotEmpty()
      ...
}
```

В итоге

```
e: SKIPPABILITY CHECK IS NOT PASSED

These functions are not skippable because in them unstable parameters are used.
Please fix or add @Suppress("NonSkippableComposable")
How to fix:
1. Check parameters in list below. Maybe you fogot about @Stable or @Immutable annotation?
2. If you cannot make your parameter stable, try to declare default value (only if it is semantically required)
3. If your function is little(just using as proxy) and is not root function, try to add @NonRestartableComposable annotation (like LaunchedEffect, Image and etc).
4. If your function is root function and it doesn't read state, try to add @NonRestartableComposable annotation (like Surface and etc).

P.S. For 3 and 4 - When you store <this> reference to class instance and use it to access some parameters, your function cannot be skippable.
     If you cannot fix add "@Suppress("NonSkippableComposable")" to save restartable opportunity.
     For more information read: https://github.com/androidx/androidx/blob/androidx-main/compose/compiler/design/compiler-metrics.md#functions-that-are-restartable-but-not-skippable
```

```
Functions with unstable paramaters:
com.google.samples.apps.nowinandroid.core.ui.TrackJank:
     (name=keys, type=Parameter, class=kotlin.Array, reason=Unstable(type Any doesn't has @StabilityInferred))
     (name=reportMetric, type=Parameter, class=kotlin.coroutines.SuspendFunction2, reason=Unstable(type SuspendFunction2 doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.TrackDisposableJank:
     (name=keys, type=Parameter, class=kotlin.Array, reason=Unstable(type Any doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.NewsFeedContentPreview:
     (name=userNewsResources, type=Parameter, class=kotlin.collections.List, reason=Unstable(type List doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.NewsResourceCardExpanded:
     (name=userNewsResource, type=Parameter, class=com.google.samples.apps.nowinandroid.core.model.data.UserNewsResource, reason=Unstable(type UserNewsResource doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.NewsResourceMetaData:
     (name=publishDate, type=Parameter, class=kotlinx.datetime.Instant, reason=Unstable(type Instant doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.NewsResourceTopics:
     (name=topics, type=Parameter, class=kotlin.collections.List, reason=Unstable(type List doesn't has @StabilityInferred))
com.google.samples.apps.nowinandroid.core.ui.ExpandedNewsResourcePreview:
     (name=userNewsResources, type=Parameter, class=kotlin.collections.List, reason=Unstable(type List doesn't has @StabilityInferred))
```

Всё ли было хорошо?

Инкрементальная компиляция!

https://t.me/int_ax/35
https://android-review.googlesource.com/c/platform/frameworks/support/+/2759289
https://android-review.googlesource.com/c/platform/frameworks/support/+/2766125

А можно было раньше предупредить?

Frontend Plugin?

Frontend Plugin?

- Инвестиция в будущее
- Поддержка в IDEA из коробки
- Сильно распиарен
- Google уже переписал Android LInt и часть Compose

Frontend Plugin?

Frontend Plugin

IDEA Plugin?

IDEA Plugin

```
@Composable
private fun SimpleComposable(
    value: Int,
    value2: Int
) {
    Text(text = value.toString())
    Text(text = value2.toString())
}
```

Выводы

Выводы

- Плагины писать не так сложно
- Есть задачи, где их можно применить
- Устройство Compose Compiler следует изучать
- Compose Compiler все еще имеет проблемы и большой потенциал для развития

Будем ВКонтакте!