NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Mapping Kotlin IR to source code elements to visualize program analysis results

Efstratia I. Evangelinou

Supervisors: **Yannis Smaragdakis,** Professor UoA
George Fourtounis, Research Scientist

ATHENS

APRIL 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αντιστοίχιση Kotlin IR σε πηγαίο κώδικα για την οπτικοποίηση αποτελεσμάτων ανάλυσης προγραμμάτων

Ευστρατία Ι. Ευαγγελινού

Επιβλέποντες: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Μεταδιδακτορικός Ερευνητής

ΑΘΗΝΑ

ΑΠΡΙΛΙΟΣ 2024

BSc THESIS

Mapping Kotlin IR to source code elements to visualize program analysis results

Efstratia I. Evangelinou

S.N.: 1115201500038

SUPERVISORS: **Yannis Smaragdakis,** Professor UoA
George Fourtounis, Research Scientist

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αντιστοίχιση Kotlin IR σε πηγαίο κώδικα για την οπτικοποίηση αποτελεσμάτων ανάλυσης προγραμμάτων

Ευστρατία Ι. Ευαγγελινού

Α.Μ.: 1115201500038

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Γιάννης Σμαραγδάκης,** Καθηγητής ΕΚΠΑ
Γιώργος Φουρτούνης, Μεταδιδακτορικός Ερευνητής

ABSTRACT

This thesis implements support for mapping Kotlin source code elements to the corresponding (Java) bytecode that is emitted from the Kotlin compiler. This is achieved via a plug-in in the Kotlin compiler that intervenes during compilation and processes the intermediate representation (IR) to recover the bytecode produced in this phase in and its relation to the source code elements. The results of this plugin can be used to present results of bytecode static analysis over source code in an IDE.

SUBJECT AREA: Program analysis

KEYWORDS: static program analysis, intermediate representation, Kotlin compiler plug-ins

ΠΕΡΙΛΗΨΗ

Αυτή η πτυχιακή υλοποιεί την αντιστοίχηση των αντικειμένων του πηγαίου κώδικα με το αντίστοιχο (Java) bytecode που παράγεται από τον μεταγλωττιστή της Kotlin. Αυτό επιτυγχάνεται με τη δημιουργία ενός plugin για τον μεταγλωττιστή της Kotlin που επεμβαίνει κατά τη μεταγλώττιση και επεξεργάζεται την ενδιάμεση αναπαράσταση (IR) για να ανακτήσει το bytecode που κατασκευάζεται και τη σχέση του με τα στοιχεία του πηγαίου κώδικα. Τα αποτελέσματα αυτού του plugin μπορούν να χρησιμοποιηθούν για την παρουσίαση των αποτελεσμάτων της στατικής ανάλυσης του bytecode πάνω από τον πηγαίο κώδικα ολοκληρωμένο περιβάλλον ανάπτυξης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάλυση προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: στατική ανάλυση, ενδιάμεση αναπαράσταση, επεκτάσεις μεταγλωττιστή Kotlin

ACKNOWLEDGEMENTS

I want to express my gratitude to my supervisor, Prof. Yannis Smaragdakis, for giving me the opportunity to work on this incredibly interesting and educational subject, and for his guidance in approaching this thesis.

Additionally, I would like to thank my supervisor Dr. George Fourtounis for his contribution and guidance throughout the research and development process, as well as for his availability to discuss any problems that arose towards completing this thesis.

CONTENTS

1. INTRODUCTION **11**

2. Background(Kotlin compiler + clyze) **13**

2.1 Kotlin 13

2.2 Visitor Pattern 13

2.3 Kotlin IR 13

2.4 Kotlin Compiler Plugins 14

2.5 The clyze metadata model 16

3. Kotlin Compiler Plugin **17**

3.1 Intervene in Kotlin's compilation process 17

3.2 Output data generated 18

3.3 Doop 18

3.4 KotlinFileInfo 19

3.5 Classes 20

3.6 Methods 21

3.7 Variables 22

3.8 Fields 23

3.9 Method Invocations 23

3.10 Heap Allocation 24

3.11 Subtle problems that came up and future work 25

4. Handling source code that misses libraries **27**

5. CONCLUSIONS AND FUTURE WORK **28**

ABBREVIATIONS - ACRONYMS **29**

REFERENCES **30**

LIST OF FIGURES

2.1 IrElementVisitor 14
2.2 KotlinCompilerPluginSupportPlugin 15
2.3 CommandLineProcessor 15
2.4 ComponentRegistrar 15
2.5 IrGenerationExtention 16

3.1 Kotlin compiler's plugin class implementing IrGenerationExtension 17
3.2 Kotlin compiler's plugin class implementing IrElementVisitor 18
3.3 Example of code annotated with analysis results 19
3.4 KotlinFileInfo Class 20
3.5 The visitClass() function of IRvisitor 20
3.6 A typical kotlin function where the name is right after "fun" keyword 25
3.7 A kotlin function where type parameter and extension receiver parameter are declared between "fun" keyword and the function name 25
3.8 A class with a primary and a secondary constructor in Kotlin 26
3.9 An example of a static kotlin function that does not set the "isStatic" boolean field in function declaration 26
3.10 Heap allocations using constructor versus library function 26

PREFACE

This thesis aims to access the Kotlin compiler to maintain a source-to-bytecode correspondence and report results that cannot be easily traced back to the source code in the form of metadata. It was developed as my undergraduate thesis for the conclusion of my studies at the Department of Informatics and Telecommunications of the University of Athens.

1. INTRODUCTION

In computer science, static program analysis refers to the process of reasoning about the behavior of computer programs solely based on their code, without executing them. This approach stands in contrast to dynamic program analysis, which is conducted during the program's execution to observe its behavior.

Static program analysis [1] is typically conducted at the source code level, providing valuable insights into the behavior and structure of a program. However, a significant challenge arises when access to the complete program is restricted. This issue often occurs when certain segments of the program's source code are inaccessible. A typical scenario is when a program relies on third-party binary libraries that are not accompanied by the corresponding source code. In such instances, conducting comprehensive static analysis becomes difficult, as the analysis tool lacks insight into the internal operations of these external dependencies. As a result, this can limit how deeply and accurately the analysis is done, possibly causing important parts of the program's functionality to be missed.

An alternative approach to overcome this issue is to conduct program analysis on programs represented at a lower level, such as binary/native code or intermediate representations such as Java bytecode or .NET Common Intermediate Language (CIL). While this method provides a more accurate reflection of the program's behavior as a whole and can capture additional properties, the outcomes are often less visually appealing. This is because low-level elements such as compiler-generated names and complex structures can complicate the reports generated by low-level static analyzers.

Doop [2], is a static analysis tool designed specifically for programs in the form of Java bytecode. It is a mature tool capable of reasoning about fundamental properties of programs and has been the basis of more complex applications [3], [4]. However, it's important to note that its results concern low-level bytecode elements rather than source code constructs. Consequently, these results may not be immediately applicable in contexts such as integrated development environments (IDEs).

To maintain the correspondence between the source code and bytecode, the doop-jcplugin was previously developed [5]. While this plugin proved beneficial, it encountered challenges due to its reliance on the undocumented and frequently evolving internals of the Java compiler (javac).

This present work aims to adopt a similar approach as the doop-jcplugin, but within the context of Kotlin. Kotlin, a JVM-based language that is targeted at replacing significant uses of the Java language (such as Android app development [6]), offers a well-documented and mature compiler framework. This framework is notably more conducive to integration with intermediate representation (IR) plugins, providing a more stable foundation for our analysis.

. The rest of this thesis is structured as follows:

- Chapter 2 gives necessary background about Kotlin and its compiler infrastructure, the visitor pattern that will be used, plus the code metadata model that we will use.
- Chapter 3 describes our implementation.

- Chapter 4 addresses a significant real-world concern: partial programs.
- Chapter 5 concludes this thesis and describes future work.

2. BACKGROUND(KOTLIN COMPILER + CLYZE)

2.1 Kotlin

Kotlin is an open-source, statically typed programming language that runs on the Java Virtual Machine (JVM) and can also be compiled to JavaScript or native code. It is supported developed by JetBrains, the creators of IntelliJ IDEA, and was first released in 2011. Kotlin is designed to be fully interoperable with Java, allowing both languages to coexist within the same project seamlessly. It is endorsed by Google as an official language for Android development.

One of Kotlin's key benefits it its multiplatform capabilities. By supporting multiplatform programming it reduces the needed time to write and maintain the same code in different platforms while retaining the flexibility and benefits of native programming.Overall,Kotlin is designed to be a more modern, concise, and expressive language compared to Java. It addresses some of Java's limitations and aims to make development more enjoyable and less error-prone.

2.2 Visitor Pattern

The Visitor pattern [7] is a software design patern that allows algorithms to be seperated from structures. This separation enables the addition of new functionalities to existing object structures without necessitating changes to the structure itself. Consequently, it becomes easier to incorporate new operations regularly and in a more organized manner, as they can be managed centrally in a single location.

A Visitor class can execute all relevant specializations of a virtual function by taking the instance reference as input and implementing the goal through double dispatch. By introducing a seperate Visitor object responsible for implementing operations to be executed on elements within an object structure, clients navigate through the object structure and subsequently invoke a dispatching operation that accepts a visitor on an element. This dispatching operation delegates the request to the Visitor object that has been accepted. Subsequently, the Visitor object executes the operation on the element. This approach enables the creation of new operations independently from the classes within an object structure, achieved by introducing new Visitor objects.

2.3 Kotlin IR

Kotlin IR, or Kotlin Intermediate Representation, is the new internal representation used by the Kotlin compiler during the parsing of Kotlin source files. This representation is used to transform the code and then it is translated by the compiler backend into platform-specific representations. This approach facilitates the sharing of lowerings across all compiler backends in Kotlin, enabling developers to create a single compiler plugin that functions universally across platforms, eliminating the need for individual transformations for each platform-specific representation.

Kotlin IR [8] is an abstract syntax tree for representing Kotlin code. Every parents, siblings, and children node in the Kotlin IR syntax tree implements an IrElement. Elements of the syntax tree represent things like modules, packages, files, classes, properties, functions, parameters, if statements, function invocations, and much more. To navigate this Ir tree the visitor patern can be used. Specifically IrElementVisitor interface is implemented to allow the navigation of Kotlin IR tree, with the appropriate function being called when visiting that type of element.

```
package org.jetbrains.kotlin.ir.visitors

public interface IrElementVisitorVoid : org.jetbrains.kotlin.ir.visitors.IrElementVisitor<kotlin.Unit, kotlin.Nothing?> {
    public open fun visitAnonymousInitializer(declaration: org.jetbrains.kotlin.ir.declarations.IrAnonymousInitializer): kotlin.Unit { /* compiled code */ }

    public open fun visitAnonymousInitializer(declaration: org.jetbrains.kotlin.ir.declarations.IrAnonymousInitializer, data: kotlin.Nothing?): kotlin.Unit { /* compiled co

    public open fun visitBlock(expression: org.jetbrains.kotlin.ir.expressions.IrBlock): kotlin.Unit { /* compiled code */ }

    public open fun visitBlock(expression: org.jetbrains.kotlin.ir.expressions.IrBlock, data: kotlin.Nothing?): kotlin.Unit { /* compiled code */ }

    public open fun visitBlockBody(body: org.jetbrains.kotlin.ir.expressions.IrBlockBody): kotlin.Unit { /* compiled code */ }

    public open fun visitBlockBody(body: org.jetbrains.kotlin.ir.expressions.IrBlockBody, data: kotlin.Nothing?): kotlin.Unit { /* compiled code */ }

    public open fun visitBody(body: org.jetbrains.kotlin.ir.expressions.IrBody): kotlin.Unit { /* compiled code */ }
```

Figure 2.1: IrElementVisitor

In more detail, IrElement is the base interface for all IR elements, and all the functions in IrElementVisitor eventually delegate to this visitElement function. This visitElement function is the only function which does not have a default implementation, it helps enable recursion to children of each element. The IR element visitor also provides the accept function that is goint to call the appropriate visitor function. This polymorphic behavior allows a caller to properly handle any IrElement simply by calling element.accept(visitor, data). This behavior of calling the appropriate visitor function is consistent across all element implementations. The acceptChildren function is also implemented to call the accept function on each child IrElement it contains. This means that by calling element.acceptChildren(visitor, data) the visitor will properly handle all children of the element. This leads us to how recursion works with the visitor pattern, making sure all the elements are visited.

2.4 Kotlin Compiler Plugins

In this segment, we present a concise overview of the Kotlin Compiler plugin infrastructure. A compiler plugin is a capability of kotlinc that empowers developers to execute code during the compilation process, generating Java bytecode or LLVM IR. This functionality allows for the alteration of functions or classes, offering the capability to address new classes of metaprogramming problems.

A Kotlin compiler plugin [9] is seperated into two distinct modules: the Gradle module, serving as the entry point from Gradle, and the Kotlin module, containing the primary logic. The Gradle module serves as an entry point that allows configuration options via gradle extensions. Additionally, it provides Kotlin subplugin options and defines the compiler plugin's unique internal key.

```
interface KotlinCompilerPluginSupportPlugin : Plugin<Project> {
    override fun apply(target: Project) = Unit

    fun isApplicable(kotlinCompilation: KotlinCompilation<*>): Boolean

        Configures the compiler plugin to be incorporated into a specific kotlinCompilation. This function
        is only called on kotlinCompilations approved by isApplicable. The Provider returned from this
        function may never get queried if the compilation is avoided in the current build.

    fun applyToCompilation(
            kotlinCompilation: KotlinCompilation<*>
    ): Provider<List<SubpluginOption>>

    fun getCompilerPluginId(): String
    fun getPluginArtifact(): SubpluginArtifact
    fun getPluginArtifactForNative(): SubpluginArtifact? = null
}
```

Figure 2.2: KotlinCompilerPluginSupportPlugin

The Kotlin module consists of the command-line processor, the component registrar, and, in the final stage, the compiler extensions that are invoked. Within the command-line processor, kotlinc is called, and the arguments are passed through the pipeline.

```
package org.jetbrains.kotlin.compiler.plugin

public interface CommandLineProcessor {
    public abstract val pluginId: kotlin.String

    public abstract val pluginOptions: kotlin.collections.Collection<org.jetbrains.kotlin.compiler.plugin.AbstractCliOption>
```

Figure 2.3: CommandLineProcessor

Following this, the component registrar reads these keys and registers the compiler extensions that are going to be called in the plugin. The component registrar supports various types of extensions, and many widely-used Kotlin plugins require multiple extensions.

```
package org.jetbrains.kotlin.compiler.plugin

public interface ComponentRegistrar {
    public companion object {
        public final val PLUGIN_COMPONENT_REGISTRARS: org.jetbrains.kotlin.config.CompilerConfigurationKey<kotlin.collections.MutableList<org.jetbrains.kotlin.compiler.plugin.ComponentRegistrar>>
    }

    public abstract fun registerProjectComponents(project: org.jetbrains.kotlin.com.intellij.mock.MockProject, configuration: org.jetbrains.kotlin.config.CompilerConfiguration): kotlin.Unit
}
```

Figure 2.4: ComponentRegistrar

In this particular case, the compiler extension to be called is IrGenerationExtension, designed for accessing and modifying the Intermediate Representation (IR) tree.

```
package org.jetbrains.kotlin.backend.common.extensions

public interface IrGenerationExtension : org.jetbrains.kotlin.ir.linkage.IrDeserializer.IrLinkerExtension {
    public companion object : org.jetbrains.kotlin.extensions.ProjectExtensionDescriptor<org.jetbrains.kotlin.backend.common.extensions.IrGenerationExtension> {
    }

    public abstract fun generate(moduleFragment: org.jetbrains.kotlin.ir.declarations.IrModuleFragment, pluginContext: org.jetbrains.kotlin.backend.common.extensions.IrPluginContext): kotlin.Unit
}
```

Figure 2.5: IrGenerationExtention

2.5 The clyze metadata model

In this section we are going to introduce the clyze metadata model [10] , which is a library model that can be used for applications such as code navigation, high-level code structure analysis, or IDE integration. This tool outputs a JSON mapping of source code elements that correspond to low-level entities. Those elements are classes, methods, fields, variables, heap allocations and method invocations, for each of these elements a unique symbolid is created and static type information metadata is extracted. Some of this information provided cannot be easily traced back to the source code and that creates the need for a source-to-bytecode relationship.

3. KOTLIN COMPILER PLUGIN

3.1 Intervene in Kotlin's compilation process

To access information that is only available within the compiler, we must initiate our own compilation process by supplying the appropriate arguments to kotlinc before triggering the compilation. The fields we are interested in setting in the KotlinCompilation class are:

- List of source files to be compiled
- Boolean value of `useIr` set to true to ensure the compiler uses the IR backend, since we are overriding the IR element visitor.
- Compiler plugin in the form of the component registrar that registers the `IrGenerationExtension` to enable navigation of Kotlin IR.
- Flag indicating whether to inherit the classpath of the current program, provided for testing purposes.
- Classpath needed to be made available to compilation to resolve dependencies.

After the component registrar sets the correct extension, the IrGenerationExtension has been extended to override its generate function.

```
class TemplateIrGenerationExtension(
  private val outputPath: String
) : IrGenerationExtension {
  override fun generate(moduleFragment: IrModuleFragment, pluginContext: IrPluginContext) {
    val artifact = moduleFragment.name.asString()
    println("outputPath=$outputPath, artifact=$artifact")
    moduleFragment.accept(IrVisitor(outputPath, artifact), data: null)
  }
}
```

Figure 3.1: Kotlin compiler's plugin class implementing IrGenerationExtension

This overridden function creates an instance of IrVisitor, passing the output path where the results will be placed as an argument. Upon the creation of the IrVisitor (Figure 3.2), it initializes the stacks that will later be used to keep track of the class or function that contains an element. Additionally, it creates an instance of the printer that will be used to print the metadata collected after visiting each file. This function also utilizes an instance of IrModuleFragment to accept the IrVisitor.

Since the IrGenerationExtension has been registered within the component registrar, this instance will be invoked during compilation when IR code needs to be generated. This allows us to access information from the compilation process and collect metadata as required.

```
class IrVisitor(private val outputPath: String, private val artifact: String): IrElementVisitorVoid {
  private var configuration = Configuration(Printer( output: true))
  private var fileInfo: KotlinFileInfo? =null
  private var functionStack=Stack<elements.Function>()
  private var classStack=Stack<Class>()

  override fun visitElement(element: IrElement) {
    element.acceptChildren( visitor: this,  data: null)
  }
```

Figure 3.2: Kotlin compiler's plugin class implementing IrElementVisitor

3.2 Output data generated

The Kotlin IR plugin is designed to provide information in the form of metadata for the elements discovered in the source code. The elements that the plugin reports metadata for are:

- Declaration of classes
- Declaration of functions
- Declaration of fields
- Declaration of variables
- Method invocations
- Heap allocations

For each of those elements, we implement a class that extends the corresponding class from the metadata-model library. This information is gathered by overriding the IR element visitor functions to visit those elements. For each element encountered, we create an instance of the respective class we have implemented to process and store the necessary metadata. After the program execution, we utilize the metadata-model's FileReporter class to save the reported metadata to .json files. One file is generated for each visited file. Each .json file contains the metadata collected for the respective file during the execution of the program.

3.3 Doop

The plugin generates metadata intended for use by Doop [11] to produce analysis results. Doop consumes this metadata via the –sarif option [12] to produce results in SARIF format. THe SARIF outputs can then be integrated by installing the SARIF Viewer plugin [13] in compatible environments such as Visual Studio Code. Consequently, this provides the option to view the intermediate code annotated with the analysis results.

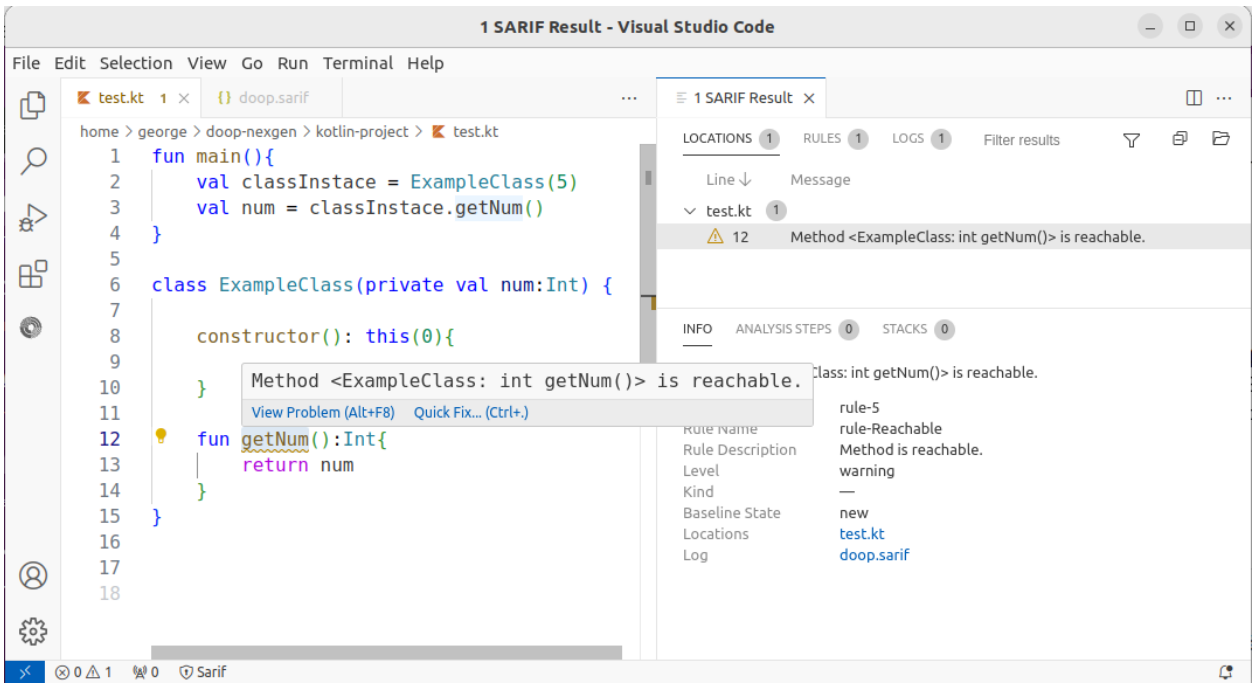

Figure 3.3: Example of code annotated with analysis results

3.4 KotlinFileInfo

To collect all the necessary metadata for the tracked elements, additional information is required, which is available within the visitFile method. We override this method and instantiate a KotlinFileInfo object, which extends FileInfo provided by metadata-model. This KotlinFileInfo class is designed to store pertinent information that is populated within the visitFile function. It serves as a field within the IR visitor, accessible from all functions that are invoked.

The KotlinFileInfo class is initialized with several important attributes: the filename, the contents of the file, the package name, and the field fileEntry, which is an interface within the IrFile declaration, containing useful tools. Within this class, methods are implemented to provide elements with necessary information. A pivotal function of the KotlinFileInfo class is accessing the contents of the source file, enabling it to determine the position of an element name within the sources. This capability is particularly valuable when, during the invocation of a visitor function, only the position of the entire declaration is available.

To accomplish this, in most cases the class initiates a search from the beginning of the declaration's position to identify the keyword used for declaring the element. Subsequently, it proceeds to locate the beginning of the name from that point onward. By obtaining a starting offset and the name of an element, it can determine the coordinates of the offset using Kotlin's fileEntry interface and return them as the correct position. If the name is absent, indicating that the element might be compiler-generated and not present in the sources, the position fields are all set to -1. Additionally, the flag indicating whether this element exists in the sources can be adjusted accordingly.

Element information is also stored within KotlinFileInfo, as it encapsulates an instance of the class JvmMetadata, implemented in metadata-model. This JvmMetadata class is utilized to aggregate all elements, categorized by their type. By the conclusion of the visitFile function, all elements have been visited, and KotlinFileInfo is ready to provide the metadata to be added to FileReporter which created the output of the plug in as mentioned

in the previous section.

```
class KotlinFileInfo(packageName :String, inputName :String, private val fileEntry : IrFileEntry, source :String):
        FileInfo(packageName,inputName,fileEntry.toString(),source,JvmMetadata()) {

    fun getPosition(startOffset: Int, name: String): Position {
        if(source.substring(startOffset,startOffset+name.length) != name) return Position( startLine: -1, endLine: -1, startColumn: -1, endColumn: -1)
        val info = fileEntry.getSourceRangeInfo(startOffset, startOffset+name.length)
        return Position(
                info.startLineNumber.toLong(), info.endLineNumber.toLong(), info.startColumnNumber.toLong(),
                info.endColumnNumber.toLong()
        )
    }
```

Figure 3.4: KotlinFileInfo Class

3.5 Classes

In order to gather all necessary information for each class encountered in the sources, IR-visitor overrides the visitClass method of the IrElementVisitorVoid interface. This method receives a parameter of type IrClass, containing all essential information regarding the class declaration. Upon visiting each class, a new instance of the class ”Class” is initialized with all the relevant fields and appended to the classStack at the beginning of the visitor call, ensuring its availability for elements within the class. Subsequently, before the visitor call concludes, all the elements within the class are visited and the class instance can be removed from the stack, as it is no longer required.

```
override fun visitClass(declaration: IrClass) {
  declaration.kind
  val classElement=Class(declaration, fileInfo!!)
  fileInfo!!.elements.jvmClasses.add(classElement)
  classStack.push(classElement)
  super.visitClass(declaration)
  classStack.pop()
}
```

Figure 3.5: The visitClass() function of IRvisitor

For every object of this type the name, the packagename and the source filename, saved in KotlinFileInfo, are set in coresponding fields of the class created to report this declaration. For each class declaration a unique symbol id is generated by appending the class name to the end of the name for ir serialiation of the parent declaration of this class. The parent declaration could be the package name, a function or a class, in the case of a nested class. This name recursively contains the parent name of each parent, starting with the package name. Metadata collected also include the class’s position, defined by its starting and ending coordinates, stores the location of the class name within the source file. The KotlinFileInfo class is employed to determine the precise coordinates because it allows access to the source file contents, enabling the exclusion of whitespaces and the declaration keyword indicating whether it’s a class, enum, interface, or object. However, calculating the position for anonymous classes is an exception, as they lack a name and the location of keyword ”object” is stored instead in this case.

Athother important class information reported is the kind of the corresponding class that could represent wether the class is an interface or an enum. Other included attributes indicate if the class is an inner class, anonymoys or static. Since kotlin does not have the static keyword that java uses, the static field is filled with the information of wether a class is a companion object or not. This is the kotlin's equivalent to creating a static method in Java. In advance the metadata provided for a class declaration also include characteristics indicating whether the visibility specified for the class is public, private, or protected. Information indicating an inheritance modifier, which denotes the modality of the class, is included if it determines that the class is either abstract or final.

3.6 Methods

For methods the process is very similar to the one for classes. The visitFunction method is overridden and by IrElementVIsitorVoid and methods are saved to functionStack in the form of Funtion class that is created to hold relevant to every method information. Constructors in kotlin are visited by the same visitor with other functions so those are also processed the same way, constructors can be distinguished by their name that is always <init>.

Function objects have certain fields in common with class objects. These shared fields encompass the name of the corresponding declaration, the filename, and information indicating the existence of the item in the source code. In adition, metadata regarding a function's visibility or modality is configured in the same manner as described above for classes. Like other elements, functions also possess a unique symbol ID. In this case, the ID begins with the parent's name for its serialization, followed by details regarding the return type, as well as the parameters of the function along with their respective types.

Additionally, functions are equipped with specific metadata exclusively reported for them. This metadata outlines the function's return type, along with information regarding its parameters and their corresponding types. Furthermore, there exists a field indicating whether the function is native, which is equivalent to the ”external” keyword in Kotlin, and is set accordingly. Moreover, supplementary information is provided for functions, including whether they are synchronized, synthetic, or static.

As for all other elements, the position of functions in the source file is also retained. To determine this position, showing the function name, typically we locate the name of the function by skipping the whitespace after the keyword ”fun.” However, there are exceptions to this rule. If the function has extension receiver parameters or type parameters, these are placed after the keyword and before the function name. In such cases, we locate the name after the position of these factors.

Furthermore, for functions, their outer position is set, comprising the starting position where the method definition begins and the ending position where the definition ends. This provides a comprehensive understanding of the function's placement and structure within the source code file.

As previously mentioned, constructors are also visited by the same visitor and handled in

the same manner as functions. However, they possess unique characteristics, particularly in terms of their name and position. Unlike other functions, the source code position of constructors cannot be determined in the same manner because they do not have a name in their function declaration.

In Kotlin, constructors are divided into two categories: primary constructors and secondary constructors. Primary constructors are declared in the class header, following the class name and optional type parameters. In this case, the position of the constructor is set to match the position of the class, and the outer position is also determined by the class's start and end offsets.

Secondary constructors, on the other hand, are declared using the keyword ”constructor,” so their position is indicated by this keyword instead of a name. The outer position is calculated similarly to any other function.

The function stack used to save all the functions also serves for anonymous initializers, which are analogous to instance initializer blocks. These initializers aren't reported with the other metadata; instead, they solely exist in the stack. This arrangement enables the identification of elements inside these blocks, and the inIIB (in Instance Initializer Block) value can be set to true.In this scenario, the only values that need to be set are the declaring class ID, which is passed to the other elements, and the symbol ID, which is set to an empty string. Subsequent sections will elucidate how these stack elements are utilized.

In addition to tracking metadata, the Function class also serves another purpose. It contains two maps: one for monitoring heap allocations and another for method invocations that take place within the function. These maps store the symbol ID of the called function or the created item and keep track of how many times this item occurs within the function. This functionality provides insights into memory allocation and method invocation patterns within the function, aiding in performance analysis and optimization efforts.

3.7 Variables

Another type of metadata we track are variables. Similarly to other elements, a class is generated to manage variables. This class is instantiated upon encountering a variable declaration, either through overriding the ”visitVariableDeclaration” method or when visiting a function parameter via the ”visitValueParameter” declaration within the IR Element visitor. In both scenarios, essential details such as the variable's name, source file name, existence in the source code, and variable type are stored. A specific boolean field indicates whether the variable acts as a parameter in a function. The sole other attribute that holds significance in determining whether it is a parameter is the source code position, which saves the coordinates of the element's name. Parameters typically aren't declared using the ”var” or ”val” keyword, so this aspect must be disregarded only if it exists between the start and end offsets of the current declaration.

Other information that we take from the compiler for variables are a boolean value that shows if the variable is local, the declaring method id of the function that cointains this variable, retained by the function class instance that is on top of the function stack and the symbol id of the variable declaration that is created by adding variable's type and name at the end of the declaring method id.

The final piece of information required for variables is whether they reside within a function or an instance initializer block (IIB). In Kotlin, an IIB is a block declared within a class using the keyword ”init” followed by curly braces. The code contained within this block becomes part of the primary constructor, enabling it to execute when the primary constructor is invoked, as primary constructors cannot contain executable code. Additionally, the code within all initializer blocks is executed when secondary constructors are invoked, before their body is executed.

Instance initializer blocks present a unique scenario in the Kotlin IR plugin because the entities they encompass typically belong to a function as their parent declaration. As mentioned in the section on functions, the Kotlin compiler traverses instance initializer blocks using the function ”visitAnonymousInitializer”. To discern whether an element is located within an initializer block rather than a function, we examine whether the symbol ID in the metadata saved for the parent function of the element is an empty string. In such cases, the declaring method ID is also set to an empty string since the element does not belong to a method. This condition also impacts the symbol ID, which utilizes the symbol ID of the declaring class instead of the method, preserving the id of the parent element in the start of the symbol ID.

3.8 Fields

Fields are distinct from variables and are handled separately within the system. They are handled by a distinct method of their element visitor, and a distinct class named ”Field” is utilized to process and retain the pertinent metadata. Identification of field elements relies on their unique symbol IDs, formed by combining the declaring class ID with the field’s type and name. Supplementary metadata necessary for this data type encompasses:

- the name of the field
- the name of the source file that contains it
- the kotlin type of the field
- the symbol id of the element
- a boolean value indicating whether the field is present in the source code
- details regarding whether the field is static
- the position of the field in the source code
- the id of the declaring class which contains it

3.9 Method Invocations

The plugin also monitors every method invoked in the source code by overriding both”visitCall” and ”visitConstructorCall” functions of the visitor, since constructor invocations are visited by a different method. As a symbol in this source code method invocation has source filename, the name of the called method, and a flag indicating whether it exists in the sources. Additionally, the position of this element is reported, displaying either the name

of the function being called or the class name if the function is a constructor. Furthermore, the class dedicated to method invocations stores information about the types of parameters received by the invoced function and the return type of the function that is being called.

In Kotlin, we encounter two distinct types of functions: standalone functions, which exist independently of any object, and extension functions, that are called on an object. Extension functions are widely used in Kotlin, including within Kotlin's standard library. It's essential to note that for extension functions, the type of the receiver object is reported by the plugin as the target type of a method invocation. However, in the case of standalone functions, this field remains an empty string.

Method invocations also possess a symbol ID, generated based on the ID of its parent element, which may be a function or a class when the invocation occurs within an Instance Initialization Block (IIB). Subsequently, this parent ID is followed by the specification of the target type and name of the invoked function. Finally, a numerical value is appended, representing the count of invocations of this particular method within the parent function or block. This count is calculated internally within the class responsible for documenting function declarations.

The symbol id of the function where the method invocation is located is also included as the invoking method ID. This information is obtained by passing the top function on the stack to the constructor of the class designed to report method invocations. The last information needed for this element is a boolean field set of whether this exists inside the init block of a class, and this process is handled in the same manner as variables. This affects symbol IDs and InvokingMethodIDs as elaborated in the chapter addressing variable metadata.

3.10 Heap Allocation

Heap allocations are another type of metadata that are being tracked and reported within Kotlin's IR plugin. Heap allocations occur when memory is dynamically allocated on the heap for objects during runtime. This happens every time a constructor is called because memory is allocated on the heap to store the object that is being created. As mentioned in the previous section, when extending Kotlin's IR element visitor, constructor calls are intercepted and tracked as method invocations. Since heap space is allocated in this case, a new heap allocation element is also generated within the visitConstructorCall function.

Metadata reported for heap allocations are very similar to those reported for method invocations, since hoth types use the same type of expression to take this data from and also report common elements. Specifically the metadata we record are the following:

- the position of the heap allocation in the source code
- the name of the source file that contains it
- a boolean value indicating whether the element exists in sources
- A boolean value indicating whether the constructor is invoked within an IIB
- the unique symbol ID for heap allocation is generated by combining the ID of the allocated type with the symbol ID of the parent element, followed by a counter that

keeps track of how many times this specific type has been allocated inside.

- the allocating type id is the type of the element that is being allocated
- the allocating method id contains the symbol id of the function that allocates this item
- a boolean value representing whether heap allocation is made for an array

3.11 Subtle problems that came up and future work

Throughout this process, certain metadata information proved challenging to locate within the compiler. One notable example of a subtle issue that presented difficulty was determining the precise position for each element, ensuring it accurately pointed to the name of the declaration or expression. Many cases were addressed by comprehending the potential keywords that could be located between the keyword used to define the element and its name, thus enabling the identification of the name's position.

```
fun main() {
    val num=1
}
```

Figure 3.6: A typical kotlin function where the name is right after ”fun” keyword

```
fun <T> T.print() {
    print(this)
}
```

Figure 3.7: A kotlin function where type parameter and extension receiver parameter are declared between ”fun” keyword and the function name

Another challenging aspect to handle was determining the position and outer position of the primary constructor. It was decided that the position should indicate the name of the class since the primary constructor's arguments are declared after it. Additionally, unlike secondary constructors, the primary constructor does not have a body for the outer position to point to. In the IR element visitor, when encountering a primary constructor, the position of the function reflects the declaration of the class instead. Therefore, a decision was made to maintain the position of the class as the outer position for the primary constructor, aligning with what the compiler considers as its position.

Filing the field of whether an element is static poses a challenge in Kotlin due to the absence of an explicit ”static” keyword, as seen in Java. In Kotlin, companion objects serve as the equivalent to static classes in Java. Additionally, Kotlin provides the @JvmStatic annotation, which can be used inside companion objects to create static fields. However,

```
class exampleClass(var A:String){

    constructor() : this("example"){

    }
```

Figure 3.8: A class with a primary and a secondary constructor in Kotlin

distinguishing static elements solely based on syntax or keywords is not straightforward in Kotlin compared to languages like Java. However, within the IrFunction and IrField declarations, there exists a boolean value indicating whether a function or a field is static. This information is utilized by the plugin. Nonetheless, it's not guaranteed that this covers all cases of elements that would be considered static.

```
class C {
    companion object {
        @JvmStatic fun callStatic() {}
        fun callNonStatic() {}
    }
}
```

Figure 3.9: An example of a static kotlin function that does not set the "isStatic" boolean field in function declaration

Lastly, another detail to be noted is that in Kotlin, arrays, for example, can be instantiated using the class constructor, like any other class. However, they can also be created by a number of functions when those functions are called. In this case, the plugin will not record the heap allocation because it treats these array creation functions as any other function. It is not straightforward to identify which Kotlin functions might involve heap allocations to include them in the recording process. Therefore, it remains as future work to devise a method to also record heap allocations from functions that are used for creating objects like arrays. This would involve further analysis and possibly extending the plugin's capabilities to detect such allocations.

```
fun main(){
    // creating array using constructor
    var Array1 = Array<Int>(5){0}
    // creating array using the library function arrayOf()
    val Array2 = arrayOf(1, 2, 3)
}
```

Figure 3.10: Heap allocations using constructor versus library function

4. HANDLING SOURCE CODE THAT MISSES LIBRARIES

To utilize the Kotlin plugin, a main function is employed. This function leverages ”kotlinc” to initiate the compilation phase, providing the desired sources for analysis alongside the Kotlin IR plugin as arguments. The main function accepts input sources in the form of a .zip file, a .jar file, or a directory containing Kotlin files for which we aim to generate metadata.

The primary requirement for using this plugin effectively is to have Kotlin sources that compile successfully. However, this condition can be problematic, especially when analyzing a single file extracted from a larger project. In such instances, the extracted file may depend on additional files that are not included, which could result in compilation errors. This absence of essential files would make it challenging for the compiler to resolve references for elements contained in those missing files. As a result, compilation would fail, preventing the plugin from being executed.

For instance, to analyze a Java library, for which there is both bytecode (JAR file) and source code available, we must run the Kotlin compiler on the sources, augmented by all code dependencies (the Java ”classpath”). Since these dependencies may not be readily available, compilation, and thus source-to-bytecode mapping, will fail.

For instance, to analyze a Kotlin library, for which there is both bytecode (JAR file) and source code available, we must run the Kotlin compiler on the sources, augmented by all code dependencies (the Java ”classpath”). Since these dependencies may not be readily available, compilation, and thus source-to-bytecode mapping, will fail.

The solution to resolve this potential issue involves including a tool called jphantom [14], which is utilized for Java program complementation. This tool can be enabled by passing the appropriate argument during execution.

When jphantom is enabled, the bytecode provided as a JAR file argument is enhanced by creating a new JAR file. This new JAR file includes dummy implementations for every phantom class identified in the original sources. Phantom classes refer to classes that are referenced in the sources but lack a corresponding definition. The generated phantom classes in the produced jar are designed to incorporate all missing fields and methods referenced in the sources intended for compilation. Additionally, they include a supertype that respects every type constraint that was identified. By employing this tool, the previously missing referenced elements are now generated, enabling the successful compilation and analysis of the code.

5. CONCLUSIONS AND FUTURE WORK

In this thesis, we introduced a Kotlin Compiler plugin designed to generate metadata with the aim of preserving a source-to-bytecode correspondence, facilitating static analysis with tools like Doop. While this plugin represents an initial effort to produce metadata for fundamental Kotlin elements, there is ample opportunity for future expansion. In future work the plugin's capabilities could be enhanced to report metadata for every element within a valid Kotlin program, ensuring that no source code information is overlooked.

Moving forward, the next steps for this plugin involve addressing and resolving the subtle issues highlighted in this thesis, thus improving its effectiveness and accuracy. Additionally, the functionality can be extended to include reporting on string constants and the usage of elements within the code, similar to what the doop-jcplugin already accomplishes.

Ultimately, this plugin serves as a demonstration of how Kotlin plugins can harness Kotlin Intermediate Representation (IR) to access vital information within the compiler. Throughout this thesis, we have provided insights into the workings of this IR and offered guidance on creating a basic Kotlin plugin.

ABBREVIATIONS - ACRONYMS

IR	Intermediate Representation
IIB	Instance Initializer Block

BIBLIOGRAPHY

[1] Wikipedia contributors. *Static Program Analysis*. https://en.wikipedia.org/wiki/Static_program_analysis. Accessed: April 13, 2024.

[2] Martin Bravenboer and Yannis Smaragdakis. “Strictly declarative specification of sophisticated points-to analyses”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 243–262. ISSN: 0362-1340. DOI: 10.1145/1639949.1640108. URL: https://doi.org/10.1145/1639949.1640108.

[3] Neville Grech and Yannis Smaragdakis. “P/Taint: unified points-to and taint analysis”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133926. URL: https://doi.org/10.1145/3133926.

[4] Anastasios Antoniadis et al. “Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 794–807. ISBN: 9781450376136. DOI: 10.1145/3385412.3386026. URL: https://doi.org/10.1145/3385412.3386026.

[5] Anastasios Antoniadis. *Combining source code metadata and static analysis results via a compiler plug-in*. 2016. URL: https://yanniss.github.io/theses/antoniadis2.pdf.

[6] Android Developers. *Kotlin Programming Language Documentation*. Google, 2022. URL: https://developer.android.com/kotlin.

[7] *Visitor Pattern — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Visitor_pattern.

[8] Brian Norman. *Writing Your Second Compiler Plugin - Part 3*. URL: https://blog.bnorm.dev/writing-your-second-compiler-plugin-part-3.

[9] JetBrains. *Writing Your First Kotlin Compiler Plugin*. https://speakerdeck.com/kevinmost/writing your-first-kotlin-compiler-plugin. Presentation slides. JetBrains, 2018.

[10] *Clyze metadata-model*. URL: https://github.com/clyze/metadata-model.

[11] *Using Doop with Visual Studio Code*. URL: https://bitbucket.org/yanniss/doop/src/master/.

[12] *Static Analysis Results Interchange Format (SARIF) Version 2.1.0 Plus Errata 01*. URL: https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[13] *SARIF Viewer for Visual Studio Code*. URL: https://marketplace.visualstudio.com/items?itemName=MS-SarifVSCode.sarif-viewer.

[14] George Balatsouras and Yannis Smaragdakis. “Class hierarchy complementation: soundly completing a partial type graph”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 515–532. ISBN: 9781450323741. DOI: 10.1145/2509136.2509530. URL: https://doi.org/10.1145/2509136.2509530.