NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

MASTER THESIS

Combining source code metadata and static analysis results via a compiler plug-in

Anastasios I. Antoniadis

SUPERVISOR: Yannis Smaragdakis, Professor, University of Athens

ATHENS

OCTOBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συνδυάζοντας μεταδεδομένα πηγαίου κώδικα και αποτελέσματα στατικής ανάλυσης μέσω μιας επέκτασης για το μεταγλωττιστή

Αναστάσιος Ι. Αντωνιάδης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Γιάννης Σμαραγδάκης, Καθηγητής, Πανεπιστήμιο Αθηνών

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

MASTER THESIS

Combining source code metadata and static analysis results via a compiler plug-in

Anastasios I. Antoniadis

October 2016

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Συνδυάζοντας μεταδεδομένα πηγαίου κώδικα και αποτελέσματα στατικής ανάλυσης μέσω μιας επέκτασης για το μεταγλωττιστή

Αναστάσιος Ι. Αντωνιάδης

Οκτώβριος 2016

ABSTRACT

As static program analysis techniques keep evolving, leading to higher precision and improved soundness, analysis results become all the more useful for consumption by code comprehension tools. However, performing static analysis on a low-level IR (such as bytecode) often leads to the loss of some source-code-level information. This thesis presents a Java 8 compiler plugin that introduces a new phase to the compilation process in order to generate metadata for classes, variables, fields, methods, method invocations, and heap allocations. This metadata can then be used to associate low-level analysis results with the source code of the program. The emphasis of the work is on leveraging the extensible architecture of the recent official Java compiler implementations. We show how each facility of the Oracle Java compiler plugin infrastructure operates and how it is employed in our system. As an application, we demonstrate how the enriched analysis results are presented on a web-based source code viewer.

SUBJECT AREA: Program Analysis

KEYWORDS: Points-to analysis, Java compiler plug-ins, Code comprehension tools

ΠΕΡΙΛΗΨΗ

Καθώς οι τεχνικές στατικής ανάλυσης προγραμμάτων εξελίσσονται, οδηγώντας σε μεγαλύτερη ακρίβεια και βελτιωμένη ορθότητα, τα αποτελέσματα των αναλύσεων γίνονται όλο και περισσότερο χρήσιμα για κατανάλωση από εργαλεία κατανόησης κώδικα. Ωστόσο, η εκτέλεση στατικής ανάλυσης σε ενδιάμεση γλώσσα οδηγεί σε απώλεια πληροφορίας επιπέδου πηγαίου κώδικα. Αυτή η διπλωματική εργασία παρουσιάζει μια προσθήκη για τον μεταγλωττιστή της Java 8 που εισάγει μία νέα φάση στη διαδικασία μεταγλώττισης προκειμένου να παράξει μεταδεδομένα για κλάσεις, μεταβλητές, πεδία, μεθόδους, κλήσεις μεθόδων, και δεσμεύσεις αντικειμένων στο σωρό. Αυτά τα μεταδεδομένα μπορούν να συσχετίσουν τα χαμηλού επιπέδου αποτελέσματα της ανάλυσης με τον πηγαίο κώδικα του προγραμμάτος. Η έμφαση αυτής της εργασίας δίνεται στην αξιοποίηση της επεκτάσιμης αρχιτεκτονικής των πρόσφατων υλοποιήσεων του επίσημου μεταγλωττιστή της Java. Παρουσιάζουμε πώς εκτελείται κάθε λειτουργία της υποδομής των προσθέτων του Oracle Java μεταγλωττιστή και πώς χρησιμοποιείται το σύστημά μας. Ως εφαρμογή, παρουσιάζουμε, πώς απεικονίζονται τα εμπλουτισμένα αποτελέσματα της ανάλυσης σε έναν διαδικτυακό εργαλείο προβολής πηγαίου κώδικα.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάλυση Προγραμμάτων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: “Δείχνει-σε” ανάλυση, Επεκτάσεις του μεταγλωττιστή της Java, Εργαλεία κατανόησης κώδικα

To sunshine, for always being there...

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Yannis Smaragdakis for providing his expertise and guidance, which were valuable for the preparation and completion of this work and most of all for his patience throughout the whole process. He has been very considerate during our collaboration and his work ethic has been admirable.

I would also like to thank my parents for their support and faith in me, and for bearing with me throughout all my struggles. Furthermore, I would like to thank my friends, Nikos, Themis, Lena, Ilianna, Thomas, Nasia, Nicole, Dimitris, Dimis, Despina, Christos, Giannos, Kostas, Irini to name a few, who also had to endure a lot while I was pursuing my masters degree and stood by me.

Finally, I would like to thank Kostas Saidis for his contributions to this work and our ongoing collaboration in order to improve this project and several others in our pipeline.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

As software size and complexity increase there is a growing need for more sophisticated and complex code comprehension and reviewing tools. Static analysis can be very useful for code comprehension as it provides information for all possible executions of a program.

This thesis presents *doop-jcplugin,* the prototype of a Java Compiler plug-in that combines static source code information generated by processing the abstract syntax tree (AST) and the symbol tables generated during the compilation phases of the Java compiler and points-to analysis information produced by analyzing the bytecode of the program. The aim of our plug-in is to preserve source code information and combine it with analysis results in order to make them presentable at the source code level.

We emphasize on pointer-analysis in particular, which is a category of static program analysis that evaluates where each variable of a program can 'point-to' for each possible execution of the code. The end product of the *doop-jcplugin* comes in the form of metadata that can be used to enrich analysis relations, such as where a variable may point to, which method may be called from a method invocation, and where a field of a particular object may point to. We also demonstrate the prototype of a web-browser-based source code viewer enhanced with the ability to present points-to and call-graph information by querying the post-processed metadata of our plugin after an analysis is completed.

The rest of the thesis is organized as follows:

- In Chapter 2 we make an introduction to the Doop framework.
- In Chapter 3 we present the Java Compiler plug-in mechanism introduced in Java 8.
- In Chapter 4 we present our Java Compiler plug-in (*doop-jcplugin*), which extracts metadata by visiting the AST of a program and combining information found in the symbol tables produced by the compiler. We also present the data model of the extracted metadata.
- In Chapter 5 we present the analysis post-processing that combines the *doop-jcplugin* metadata with the analysis results and stores them in a database. We also provide a very brief summary of some of the most significant relations of points-to analysis in Datalog. In particular, we focus on analyses in the Doop [1] framework, whose analysis results are the target for our plug-in metadata.
- In Chapter 6 we present our web-based source code viewer as an application that consumes the enriched relations produced during the post-processing step, to demonstrate the analysis results on source-code level.
- In Chapter 7 we present our conclusions and future goals regarding this project.

2. JAVA 8 COMPILER PLUG-INS

2.1 Java 8 Compiler Plug-Ins Introduction

In this section we provide a brief introduction to the mechanics of the Java Compiler plug-in infrastructure. Java 8 introduced a new mechanism that allows the programmer to write plug-ins for the Java compiler (javac) [4]. A compiler plug-in enables the developer to introduce new phases to the compiler without making changes to the Java Compiler codebase, a process which would be very complex and time consuming. Instead, we can encapsulate new behavior in a Java compiler plug-in and distribute it for other people to use. For instance, Java Compiler plug-ins could be used to do the following:

- Add extra compile-time checks.
- Add code transformations.
- Perform customized analysis at the source code level.

2.2 Java Compiler Plug-in Architecture

A javac plug-in supports two methods, *getName()* and *call()*, which it inherits and implements from the *com.sun.source.util.Plugin* interface:

- The *getName()* method returns the name of the plug-in for identification purposes.
- The *call()* method is invoked by the Java Compiler with the current environment it is processing. The *call()* method gives access to the compiler functionalities through a *JavacTask* object which allows us to perform parsing, type checking, and compilation. Moreover, the *JavacTask* object lets us add our own task listeners i.e., instances of *TaskListener* to various events generated during compilation. This is done through the *addTaskListener()* method, which accepts a *TaskListener* object.

```
public interface Plugin {
    public String getName();

    public void call(JavacTask task, String[] pluginArgs);
}
```

Figure 2.1: The *com.sun.tools.Plugin* interface.

We also demonstrate some of the methods in the *com.sun.source.util.JavacTask* class.

```
public abstract class JavacTask implements CompilationTask {
    ...
    public abstract Iterable<? extends Element> analyze() throws
     ↪  IOException;
    public abstract Iterable<? extends JavaFileObject> generate() throws
     ↪  IOException;
    public abstract void addTaskListener(TaskListener taskListener);
    ...
}
```

Figure 2.2: The com.sun.tools.JavacTask class.

Each method of the *JavacTask* class represents a Java Compiler task (*parse*, *enter*, *analyze*, *generate*). We will discuss the Java Compiler tasks in detail in the next chapter. Finally, we present the methods in the *TaskListener* class.

```
public interface TaskListener {
    public void started(TaskEvent e);
    public void finished(TaskEvent e);
}
```

Figure 2.3: The *com.sun.tools.TaskListener* interface.

In order to create a Java Compiler plug-in, we need to implement the *com.sun.source.util.Plugin* interface, which provides the *getName()* and *call()* methods. The next step of the setup requires the addition of a *TaskListener* to *JavacTask*. We need to implement the *TaskListener* interface, which provides the *started(TaskEvent e)* and *finished(TaskEvent e)* methods. Whenever a *JavacTask* starts or finishes we can check for the kind of the *TaskEvent* instance triggered in order to identify the nature of the event and execute our own logic whenever a specific event is triggered. For instance, we can implement a *TaskListener* that runs our logic before or after a particular task of the Java Compiler is finished, in order to have access to the static type information produced by Java and residing in the symbol tables.

We will discuss Java Compiler Plug-Ins further in the next chapter where see we will the implementation of an actual javac plug-in.

3. DOOP

Doop [1] is a framework for pointer, or points-to, analysis of Java programs. It implements the variations of several different context-sensitive static analysis algorithms, written in Datalog.

From the Doop website:
”Doop builds on the idea of specifying pointer analysis algorithms declaratively, using Datalog: a logic-based language for defining (recursive) relations. Doop carries the declarative approach further than past work by describing the full end-to-end analysis in Datalog and optimizing aggressively through exposition of the representation of relations (for example indexing) to the Datalog language level. Doop uses the Datalog dialect and engine of LogicBlox.”

Among the advantages of Doop compared to alternative context-sensitive pointer analysis frameworks, is that Doop is faster and provides better scalability. Also, with comparable context-sensitivity features, Doop achieves better precision when handling some Java features (for example exceptions and reflection) than alternatives.

3.1 Fact Generation and Analysis Entity Ids

Doop, before running a pointer or points-to analysis, invokes Soot to generate either Jimple (**J**ava s**imple**) or Shimple (an **S**sa version of J**imple**) intermediate representations. Jimple is a typed 3-address IR(Internal Representation) suitable for performing optimizations; it only has 15 statements. Doop takes the Jimple code as input and generates the facts. The facts are then imported into a database with multiple tables, so the analysis rules can process them. Shimple is an SSA-version of Jimple; first Jimple is generated and then Soot applies a group of transformations to Jimple body to create Shimple.

The fact generation process is of great importance for our Java Compiler plug-in, since it defines how each entity of the analysis is represented in the database with a unique key. The rules of the analysis will produce new relations between these facts, so the Java Compiler needs to correctly produce metadata that will match the analysis facts.

In the following figures we present a small subset of the analysis facts. In the chapter we explain how the *doop-jcplugin* produces metadata matching the ids of the analysis entities.

```
...
Main.main/globalMap
Main.main/args
<typechecking.GlobalSymbolTableMaker: java.lang.String
 ↪  visit(syntaxtree.NodeList,java.lang.String)>/@this
<typechecking.GlobalSymbolTableMaker: java.lang.String
 ↪  visit(syntaxtree.PlusExpression,java.lang.String)>/@this
...
```

Figure 3.1: A subset of the facts produced for the variables of a program.

Figure 3.1 presents some of the variable facts. The variables *args* and *globalMap* are a parameter and a local variable respectively in the *main* method of class *Main* on the default package. Since there is only one *main* method defined in class *Main* the method name is enough to distinguish the method. The same is not true for the next two variables, *this* defined in two overloaded *visit* methods. In order to distinguish them we need the full method signature, along with the declaring class. <typechecking.GlobalSymbolTableMaker: java.lang.String visit(syntaxtree.PrimaryExpression,java.lang.String)>

```
...
<Main: void main(java.lang.String[])>
<typechecking.GlobalSymbolTableMaker: java.lang.String
 ↪  visit(syntaxtree.NodeList,java.lang.String)>
<typechecking.GlobalSymbolTableMaker: java.lang.String
 ↪  visit(syntaxtree.PlusExpression,java.lang.String)>
...
```

Figure 3.2: A subset of the facts produced for the methods of a program.

Figure 3.2 presents some of the method facts. All methods are represented by their full signature in Doop.

```
...
<MiniJavaParser.JJCalls: int arg>
<MiniJavaParser: Token jj_scanpos>
...
```

Figure 3.3: A subset of the facts produced for the fields of a program.

Figure 3.3 presents some of the field facts. All methods are represented by their full signature in Doop. The field signature consists of the fully qualified name of the field's declaring class, followed by ":", followed by the field's type and the field's name.

We formalize these conventions in the next chapter, where we present the necessary transformations to match the Doop entity ids.

4. DOOP-JCPLUGIN

4.1 Introduction - Java Compiler Compilation Phases

The *doop-jcplugin* aims to extract static type information metadata given the source code of a program that can be compiled successfully. Before we proceed to present our own compilation phase we need to explain the default compilation phases of the Java Compiler.

The compilation of a program in the Java compiler consists of three phases:

- *Parse and Enter*: The compiler reads the files explicitly specified on the command line, parses them into ASTs, and enters externally visible definitions to the compiler's symbol tables.
- *Annotation Processing*: The compiler calls the annotation processors and restarts the compilation if new source files have been generated.
- *Analyze and Generate*: The compiler analyzes the ASTs and translates them into class files. If there are references to additional classes, then:
  - If they are found on class files, the class files will be read to determine the definitions in that class.
  - If they are found on source files, these files are added to the compilation process. The files will pass the parse and enter phase and then will be added to a *TO-DO* list for the *Analyze and Generate* phase.

4.2 Introducing our own compilation phase

Our own compilation phase comes into play twice, performing two passes on the program AST. The first pass over the AST is executed before the *Generate* phase starts for a file. This pass only registers all the symbol declarations (classes, methods, fields, variables) in that file. The second pass of the AST is performed after the *Generate* phase finishes and registers all the occurrences of symbols in the code, that is, read/write occurrences for variables and fields, and also registers the invocations of methods and heap allocation, i.e., constructor invocations that are called after the creation of new objects.

Successful compilation is a strict requirement since Java Compiler will terminate if an error occurs, thus not reaching the *Generate* phase of the compilation process. In order to extract the metadata we need to access the symbol tables and also resolve external references, therefore we need all the previous phases to have finished. In particular, the source code elements we are interested in for the time being are classes, variables, fields, methods, heap allocations and method invocations.

4.3 Invoking our own compilation phase

In this section we first describe how the *doop-jcplugin* is invoked before and after the *Generate* task for each compilation unit and then we discuss the internals of the plug-in procedures, the conventions it follows in order to produce metadata which can be used by Doop and the caveats of trying to produce metadata which need to match an internal representation produced by processing the Java bytecode.

The requirement for a Java Compiler plug-in in order to implement a web-based source code viewer is the loss of information at bytecode level. The bytecode can only maintain line number information about variables, methods, etc, while in our case we need the exact coordinates at which a source code symbol begins and ends. This information can be preserved by exploring the symbol tables produced by a Java compiler, and the Java 8 Compiler provides the proper infrastructure in order to implement a new compilation phase that extracts information about every symbol as we traverse the AST of the program with customized visitors.

```
public class DoopJcPlugin implements Plugin {
    @Override
    public String getName() {
        return "DoopJcPlugin";
    }

    @Override
    public void init(JavacTask task, String... args) {
        ...
        Task.addTaskListener(new DoopJcPluginTaskListener());
    }
}
```

Figure 4.1: The DoopJcPlugin class implementing the Plugin interface.

As we already explained in the previous chapter, our *DoopJcPlugin* class (Figure 4.1) must implement the *Plugin* interface. We also add a *DoopJcPluginTaskListener* object to the current task being processed.

```
class DoopJcPluginTaskListener implements TaskListener {
    private final Reporter reporter;
    ...
    @Override
    public void started(TaskEvent arg0) {
        if (arg0.getKind().equals(GENERATE)) {
            if (!processedFiles.contains(arg0.getSourceFile().getName())) {
                /*
                Get the AST root for this source code file.
                */
                JCTree treeRoot = (JCTree) arg0.getCompilationUnit();
                String sourceFileName = getName(arg0.getSourceFile().getName());
                String sourceFilePath = (arg0.getCompilationUnit().getPackageName() + "."
                 ↪   + sourceFileName).replaceFirst("^null\\.", "");

                LineMap lineMap = arg0.getCompilationUnit().getLineMap();
                SourceFileReport report = new SourceFileReport();
                /* Declaration scanner pass */
                treeRoot.accept(new DeclarationScanner(sourceFilePath, lineMap,
                 ↪   varSymbolMap, report));
                fileMap.put(arg0.getSourceFile().getName(), report);
            }
        }
    }
    @Override
    public void finished(TaskEvent arg0) {
        if (arg0.getKind().equals(GENERATE)) {
            if (!processedFiles.contains(arg0.getSourceFile().getName())) {
                processedFiles.add(arg0.getSourceFile().getName());
                /*
                * Get the AST root for this source code file.
                */
                JCTree treeRoot = (JCTree) arg0.getCompilationUnit();
                String sourceFileName = getName(arg0.getSourceFile().getName());
                String sourceFilePath = (arg0.getCompilationUnit().getPackageName() + "."
                 ↪   + sourceFileName).replaceFirst("^null\\.", "");

                LineMap lineMap = arg0.getCompilationUnit().getLineMap();
                /* Deep scanner pass */
                treeRoot.accept(new DeepScanner(sourceFilePath, lineMap, varSymbolMap,
                 ↪   fileMap.get(arg0.getSourceFile().getName())));
            }
        }
    }
}
```

Figure 4.2: The DoopJcPluginTaskListener class implementing the *TaskListener* interface.

The *doop-jcplugin* is invoked very straightforwardly. The *DoopJcPluginTaskListener* (Figure 4.2) object we create implements both the *started()* and the *finished()* method of the *TaskListener* interface, since our aim is to invoke our own compilation phase before and after the *Generate* task of the compiler. The *finished()* method of the *DoopJcPluginTaskListener* will be called every time an event indicating the start of a compiler task is triggered. The *finished()* method of the *DoopJcPluginTaskListener* will be called every time an event indicating the end of a task is triggered. In our case these events are the start and the end of *Generate* task for each compilation unit (source file).

At the start of the *Generate* phase, we execute the first part of our own logic. To accomplish that we need to perform the following steps.

First, we need to get the *LineMap* of the compilation unit. This *LineMap* is an interface which maps the position of a symbol as found in the symbol table to actual lines and columns in the source file.

Afterwards, we call the *accept()* method of the root node of the AST passing as argument our *SymbolScanner* visitor [3] with the source file name and the *LineMap* object as arguments. From here on, our own compilation phase begins. The *SymbolScanner* visitor visits all the declarations of symbols and registers them in our own structures.

The second part of our compilation phase executes after the end of the *Generate* task for a compilation unit. Once again we use the *LineMap* in order to map symbol positions to actual lines and columns in the source code. We then invoke our second visitor, the *DeepScanner* visitor, which associates symbol occurrences with symbol declarations and also registers method invocations and heap allocations.

In the upcoming sections of this chapter we will focus on the use of the internal API to get information from the symbol tables and on the transformation logic necessary to conform to the conventions enforced by Doop in order to produce metadata which can be easily matched with Doop analysis results.

4.4 Output Data Model

After a successful invocation, the *doop-jcplugin* produces a report with all the found classes, variables, fields, methods, the occurrences of fields or variables within a method body, method invocations and heap allocations. Before proceeding to explain how exactly each of these symbols is handled we present the data model describing the relationships between these symbols. Each symbol is represented in the *doop-jcplugin* as a Java object with a unique id. The rest of the fields of each Java object are all related metadata extracted from the symbol tables of the Java compiler. However, we also need to represent relationships between program symbols. To accomplish this, we use points-to-parent references where necessary. Below we present the set of rules describing these relationships:

- A field object refers to its declaring class object using the id of that class object.
- A method object refers to its declaring class object.
- A variable object refers to its declaring method object.

- A field or variable occurrence refers the corresponding variable or field object.
- A method invocation object refers to the invoking method object.
- A heap allocation object refers to the allocating method object.

The following code snippets demonstrate the Java classes we created in order to represent the source code elements as Java objects.

```
@EqualsAndHashCode
public abstract class ItemImpl implements Item, GroovyObject {
    private static final String ID_FIELD = "id";

    public ItemImpl() {
        CallSite[] var1 = $getCallSiteArray();
        MetaClass var2 = this.$getStaticMetaClass();
        this.metaClass = var2;
    }
    ...
}
```

Figure 4.3: The abstract class *ItemImpl* implementing Item and GroovyObject.

All the elements in our data model are subtypes of the *ItemImpl* class. Objects of type *ItemImpl* represent items that will be stored in our NoSQL database. They also implement *toJSON()* and *fromJSON()* methods, since the Java objects need to be converted to JSON objects [2] before we save them to our storage.

```
public abstract class Element extends ItemImpl {
    private String id;
    ...
    public String getId() {
        return this.id;
    }

    public void setId(String var1) {
        this.id = var1;
    }
}
```

Figure 4.4: The *Element* class.

The *Element* class implements *ItemImpl* and is the parent class of all the source elements. The id field is used to identify each element uniquely.

```
public class Class extends Symbol {
    private String name;
    private String packageName;
    private boolean isInterface;
    private boolean isEnum;
    private boolean isInner;
    private boolean isAnonymous;
    ...
}
```

Figure 4.5: The *Class* class representing class declarations.

The *Class* class represents the declarations. Its fields hold information about the name of the class, the package name of the class, whether the class is an interface, since classes and interfaces are represented by the same symbol in the Java Compiler, whether it is an Enum, an inner class, or an anonymous class.

```
public class Position implements GroovyObject {
    private long startLine;
    private long startColumn;
    private long endLine;
    private long endColumn;
    ...
}
```

Figure 4.6: The *Position* class representing the source code position of a symbol.

The *Position* classes represents the position of the symbol, keeping its start line, start column, end line, and end column.

```
public class Variable extends Symbol {
    private String name;
    private String doopName;
    private String type;
    private String declaringMethodID;
    private boolean isLocal;
    private boolean isParameter;
    ...
}

public class Field extends Symbol {
    private String signature;
    private String type;
    private String declaringClassID;
    private boolean isStatic;
    ...
}
```

Figure 4.7: The *Variable* and *Field* classes representing class and field declarations.

The *Variable* (Figure 4.7) represents variable declarations. Its fields include the name of the of the variable, the name of the variable entity in Doop, the type of the variable, its declaring method id and whether the variable is a local variable or a parameter. The *Field* class (Figure 4.7) represents field declarations. Its fields include the name of the signature of the field, the type of the field, the id of the declaring class and whether the field is static or not.

```
public class Occurrence extends Symbol {
    private String symbolID;
    private OccurrenceType occurrenceType;
    ...
}

public enum OccurrenceType implements GroovyObject {
    READ,
    WRITE,
    IMPORT,
    EXTEND;
    ...
}
```

Figure 4.8: The *Occurrence* class representing read and write occurrences of variables and fields.

The *Occurrence* class represents read and write occurrences of variables and fields, import occurrences for packages and extend occurrences for classes.

```
public class Method extends Symbol {
    private String name;
    private String declaringClassID;
    private String returnType;
    private String doopSignature;
    private String doopCompactName;
    private String[] params;
    private String[] paramTypes;
    private boolean isStatic;
    ...
}
```

Figure 4.9: The *Method* class representing method declarations.

The *Method* class (Figure 4.9) represents method declarations. Its fields include the method name, the declaring class id, the rerturn type of the method, the method signature in Doop, the compact name of the method in Doop, the parameters of the method and their types, and whether the method is static or not.

```
public class MethodInvocation extends Symbol {
    private String doopID;
    private String invokingMethodID;
    ...
}
```

Figure 4.10: The *MethodInvocation* class representing method invocations.

```
public class HeapAllocation extends Symbol {
    private String type;
    private String allocatingMethodID;
    private String doopID;
    ...
}
```

Figure 4.11: The *HeapAllocation* class representing heap allocations.

4.5 Classes

In order to record all the found classes we have to override the *visitClassDef(JCClassDecl tree)* method of the *TreeScanner* interface. All the nodes of the AST that refer to actual symbols in the symbol table contain a field with signature *sym* and type *{*}Symbol* which is a subtype of the abstract *Symbol* class.

In the case of an object of type *JCClassDecl*, the field with signature *sym* is an instance of the *ClassSymbol* class. For a class declaration the necessary metadata is the position of the declaration (starting and ending coordinates), the source file name, the full package name, and whether the class is an interface, enum, inner or an anonymous class.

The *Symbol* class provides several methods among which we can find all the methods that return information about the characteristics of a type. For instance, whether it is an enum, an interface, an anonymous class, or an inner class. However most of these methods are not unique to the *ClassSymbol* class since they are part of the *Symbol* class, which is an unexpected design choice. The following snippet shows the implementations of some of the methods we need to use and some others like *isConstructor()* that are too specific to be part of the *Symbol* class.

Figure 4.12 demonstrates how we create an object of type *Class*, which represents a class declaration in our data model. It also shows how the *LineMap* instance provided to the visitor is used to get the source code coordinates of the class declaration.

```
/*
* Visit class declaration AST node.
*/
@Override
public void visitClassDef(JCClassDecl tree) {
    Map<String, Integer> methodNamesMap;

    Position position = new Position(lineMap.getLineNumber(tree.pos),
     ↪  lineMap.getColumnNumber(tree.pos), lineMap.getColumnNumber(tree.pos) +
     ↪  tree.sym.name.toString().length());

    currentClass = new Class(position, sourceFileName,      tree.sym.name.toString(),
     ↪  tree.sym.packge().fullname.toString(), tree.sym.isInterface(),
     ↪  tree.sym.isEnum(), tree.sym.isStatic(), tree.sym.isInner(),
     ↪  tree.sym.isAnonymous());
    ...
}
```

Figure 4.13: The *visitClassDef()* method.

Finally, we keep track of all the methods in order to identify overloaded methods. We will explain why this is necessary in the upcoming sections that concern methods and method invocations.

```
public abstract class Symbol extends AnnoConstruct implements Element {
    public int kind;
    public long flags_field;
    public Name name;
    public Type type;
    public Symbol owner;
    public Symbol.Completer completer;
    public Type erasure_field;
    protected SymbolMetadata metadata;
    ...
    public boolean isInterface() {
        return (this.flags() & 512L) != 0L;
    }

    public boolean isPrivate() {
        return (this.flags_field & 7L) == 2L;
    }

    public boolean isEnum() {
        return (this.flags() & 16384L) != 0L;
    }

    public boolean isLocal() {
       return (this.owner.kind & 20) != 0 || this.owner.kind == 2 && this.owner.isLocal();
    }

    public boolean isAnonymous() {
        return this.name.isEmpty();
    }

    public boolean isConstructor() {
        return this.name == this.name.table.names.init;
    }

    public Name getQualifiedName() {
        return this.name;
    }

    public boolean isInner() {
        return kind == 2 && type.getEnclosingType().hasTag(TypeTag.CLASS);
    }
}
```

Figure 4.12: The *Symbol* abstract class.

4.6 Variables and Fields

We classify variables and fields in the same category because their declarations are visited by the same method, *visitVarDef(JCVariableDecl tree)* (Figure 4.14). The *sym* field of the tree node instance *tree* is of type *VariableSymbol* both for variables and fields. The first step is to distinguish fields and variables, since we will handle them differently. This section also introduces the first transformation rules in order to follow the Doop conventions.

The format for field signatures consists of the fully qualified name of the class followed by the field name. The rest of the information we need is:

- the position of the field (mapped from the *pos* field of the tree node instance and adding the length of the field name to get the ending column),
- the name of the field,
- the field signature as represented textually in Doop,
- the type of the field, and
- the id of the declaring class of the field, since each field has to reference its declaring class and whether the field is static or not.

The format for a variable consists of (a) the fully qualified compact name of the declaring method of the variable in case the method is not overloaded and (b) the full method signature in case the method is overloaded, since we need to distinguish methods of the same class with the same name. This is the reason we had to identify overloaded methods earlier, while visiting the class declaration. The rest of the metadata for a variable are the following:

- the position of the variable,
- the name of the variable in the source code,
- the name of the variable in Doop textual representation,
- the type of the variable
- the id of the variable's declaring method, since each variable has to reference its declaring method, and
- whether the variable is a local variable or a parameter.

As we explained, this approach provides us only with the metadata regarding field and variable declarations. We also need to track all occurrences of each field and variable in the source code. In order to accomplish this, we created two more visitors, the *DefOccurrenceVisitor*, which records all the writes to a field or variable and *UseOccurrenceVisitor*, which records all the reads of a field or variable. We distinguish the reads and writes based on the AST information — for instance occurrences in the left-hand-side of an assignment are writes to a variable or field, while the right-hand-side occurrences are reads. This however, can be a very complicated process since we need to ensure both that we report the correct occurrence type for each occurrence and that we don not miss any occurrence by accident due to not scanning a certain branch of the AST with the appropriate visitor. For instance, in an assignment like the following:

```
a.b.c.d.e = f.g;
```

- The occurrence of *a* is a read of the variable a.
- The occurrences of *b*, *c* and *d* are reads of the corresponding fields.
- The occurrence of *e* is a write to the field *e*.
- The occurrence of *f* is a read of the variable *f*.
- The occurrence of *g* is a read of the field *g*.

4.7 Methods

We handle method declarations by overriding the *visitMethodDef(JCMethodDecl tree)* method of *TreeScanner*. Constructors are handled differently than all other methods since their method name is always init. The metadata we record are the following:

- the position of the method declaration,
- the source file name where the method declaration is found,
- the name of the method,
- the id of the declaring class of the method since a method must reference its declaring class,
- the compact name of the method in Doop textual format,
- the full signature of the method in Doop textual format,
- the return type of the method,
- the parameter list of the method,
- the list of parameter types of the method,
- whether the method is static or not.

```
/** Visit variable declaration AST node. **/
@Override
public void visitVarDef(JCVariableDecl tree) {
    if (tree.sym.getKind().toString().equals("FIELD")) {
        Position position = new Position(this.lineMap.getLineNumber(tree.pos),
         ↪  this.lineMap.getColumnNumber(tree.pos),
         ↪  this.lineMap.getColumnNumber(tree.pos +
         ↪  tree.sym.getQualifiedName().toString().length()));

        String fieldName = tree.sym.name.toString();
        String fieldSignature = this.doopReprBuilder.buildDoopFieldSignature(tree.sym);
        Field field = new Field(position, sourceFileName, fieldName, fieldSignature,
         ↪  tree.sym.type.toString(), currentClass.getId(), tree.sym.isStatic());

        SourceFileReport.fieldList.add(field);
        varSymbolMap.put(tree.sym.hashCode(), field);
    }
    else {
        Position position = new Position(this.lineMap.getLineNumber(tree.pos),
         ↪  this.lineMap.getColumnNumber(tree.pos),
         ↪  this.lineMap.getColumnNumber(tree.pos + tree.sym.name.toString().length()));

        String varNameInDoop;
        if { (this.methodNamesPerClassMap.get(tree.sym.enclClass())
                .get(curMethSym.getQualifiedName().toString()) > 1)
            varNameInDoop = this.doopReprBuilder.buildDoopVarName(curMethDoopSig,
             ↪  tree.sym.getQualifiedName().toString());
        }
        else {
            varNameInDoop = this.doopReprBuilder.buildDoopVarName(curMethCompactName,
             ↪  tree.sym.getQualifiedName().toString());

            Variable variable = new Variable(position, sourceFileName,
             ↪  tree.sym.name.toString(), varNameInDoop, tree.sym.type.toString(),
             ↪  currentMethod.getId(), isLocal, isParameter);

            SourceFileReport.variableList.add(variable);
            varSymbolMap.put(tree.sym.hashCode(), variable);
        }
            ...
    }
}
```

Figure 4.14: The *visitVarDef()* method for variable and field declarations.

4.8 Method Invocations

The deep-scanner visitor is responsbile for scanning the method body of each method declaration in order to find heap allocations (i.e., constructor/special method invocations) and normal method invocations (virtual or static). If the invoked method is not overloaded, the invocation text representation will consist of its compact name followed by a slash and a counter *n*, which indicates that this is the $(n+1)^{th}$ invocation of this particular method within the current method body. The same rule applies for overloaded methods with the exception that we use the full method signature as we did in the case of method declarations. In order to keep track of method invocations, we keep a map of *MethodSymbol* objects to the number of invocations found for each symbol.

Constructors are handled slightly differently, since there is only one counter for all the different constructor invocations within the current method body and we have to increase it every time we encounter a constructor invocation.

4.9 Heap Allocations

Whenever the deep scanner visitor encounters a constructor invocation, we also need to record an abstract heap allocation object of the class whose constructor was invoked. The heap allocation id consists of the allocating method signature, followed by “new”, followed by the fully qualified type of the allocation, a slash and a counter *n*, which indicates the $(n+1)^{th}$ occurrence of this particular heap allocation of type T in the current method body. We also record the position, the type of the heap allocation and the current method id, since a heap allocation must reference the allocating method.

In Figure 4.15 we present the functions that perform the transformation in order to produce entity ids for Doop().

4.10 Caveats

The main difficulties we have encountered in this project are related to three different parameters.

- The process of identifying the correct representation of ids by inspecting the IR produced by the bytecode and covering corner cases, such as Java idioms, erased generics, different transformation rules for inner and anonymous classes is very difficult. This has led to mismatches between analysis entity ids and plug-in metadata object ids leading to loss of information. For instance any *ArrayList<T>* type where *T* is a type will appear as *ArrayList* where its type occurs in the bytecode since *T* will be erased to *Object*. Another example is the naming scheme for nested classes. Instead of *com.anantoni.demo.Klass.NestedKlass* we need to convert all the dots af-

$$gen\_msig(\langle R, m, P\rangle) = R + \text{"}\llcorner\!\lrcorner\text{"} + m + \text{"("} + P + \text{")"}$$

$$gen\_m(\langle R, m, P\rangle) = m$$

$$\textit{gen\_variable\_id(C, M, V, m)} = \begin{cases} C + \text{"."} + gen\_m(M) + \text{"/"} + V & \text{m is not overloaded} \\ C + \text{":"} + gen\_msig(M) + \text{"/"} + V & \text{m is overloaded} \end{cases}$$

$$\textit{gen\_method\_invo\_id(M, i, } m_I\textit{)} = \begin{cases} gen\_m(M) + \text{"/"} + m_I + \text{"/"} + i & \text{m is not overloaded} \\ gen\_msig(M) + \text{"/"} + m_I + \text{"/"} + i & \text{m is overloaded} \end{cases}$$

where i indicates the i-th invocation of a method named m_I in M, starting from 0

$$\textit{gen\_heap\_alloc\_id(M, C, j)} = \begin{cases} gen\_m(M) + \text{"/new"} + C + \text{"/"} + j & \text{m is not overloaded} \\ gen\_msig(M) + \text{"/new"} + C + \text{"/"} + j & \text{m is overloaded} \end{cases}$$

where j indicates the j-th heap allocation of C in M, starting from 0

Symbol	Description	Type
R	fully qualified return type	*String*
m	method name	*String*
P	fully qualified parameter types list	*String*
M	method signature	$\langle R, m, P\rangle$
C	fully qualified class name	*String*
V	variable name	*String*
I	method invocation	*String*

Figure 4.15: The transformation functions for variables, method invocations and heap allocations. We use $+$ to show string concatenation. The symbol $\llcorner\!\lrcorner$ stands for the space character.

ter the first class to ".", thus generating the *com.anantoni.demo.Klass$NestedKlass* string representation.

- As we have already explained, identifying all read and write occurrences correctly without missing any of them is also a very complicated process.
- Identifying the exact beginning and ending columns of a symbol is also not trivial and sometimes requires the traversal of more than one node in order to identify syntactic sugar and correctly calculate the full length of the symbol in the source code. For instance, declaring the type of a variable as *java.util.ArrayList* instead of *ArrayList* will lead to the same information, since the import on the latter case is just syntactic sugar. The symbol tables will be identical in both cases in terms of type information, but the two programs will have different ASTs, which we need to handle accordingly.

5. POST PROCESSING OF THE ANALYSIS

The *doop-jcplugin* produces metadata ready to be consumed by a post-processing step and matched with Doop analysis results. The ids generated by the *doop-jcplugin* need to be identical to the ones identifying analysis entities, so that by querying the analysis database for relations of interest we can match the ids of symbols produced by the *doop-jcplugin* with entity ids in the database and produce enriched relations.

As we have already explained Doop performs points-to analysis in Datalog so the initial relations we enrich are all points-to information related.

- The *VarPointsTo(var, heap)* relation representing the heap allocations a variable *var* may point to. By matching the string representation value of *var* and *heap* with the variable ids and heap allocation ids produced by the plug-in, we manage to enrich both parts of the relation with the corresponding *Variable* and *HeapAllocation* metadata objects and produce a source code presentable *VarPointsTo* relation.

- The *CallGraphEdge(invocation, method)* relation representing the method *method* that may be called by a method invocation, *invocation*. For this relation we compare the method ids and invocation ids generated by the *doop-jcplugin* with the values of *method* and *invocation* respectively, in order to match method analysis entities with *Method* metadata objects and method invocation analysis entities with *MethodInvocation* metadata objects.

- The *InstanceFieldPointsTo(baseheap, sig, heap)* relation, representing the heap allocations *heap* that the field *sig* of an object *baseheap* may point to. For this relation we match the value *baseheap* with the corresponding *HeapAllocation* object of type *T*, then we match the value of field signature *sig* with the *Field* object referring to the *Class* object representing type *T*. In order to complete the enriched relation, we match all the values of *?heap* with the correspoding *HeapAllocation* object.

By processing the results of the analysis we create objects that represent the analysis relations and refer to the plugin metadata associated with the entities of each relation.

```
class VarPointsTo extends SymbolRelation {

    String variableID;
    List<String> heapAllocationIDSet;
    ...
}
```

Figure 5.1: The *VarPointsTo* class representing *VarPointsTo* objects.

The field *variableID* of the class *VarPointsTo* is a *String* indicating the id of the *Variable* metadata object for a particular variable entity of the relation. The set of *Strings heapAllocationIDSet* contains the ids of the *HeapAllocation* metadata objects for the heap allocation entities of the *VarPointsTo* relation, one for each heap allocation the variable may point to.

```
class CallGraphEdge extends SymbolRelation {

    String methodInvocationID;
    List<String> methodIDSet;
    ...
}
```

Figure 5.2: The *CallGraphEdge* class representing CallGraphEdgeRelations.

The field *methodInvocationID* of the class *MethodInvocation* is a *String* indicating the id of the *MethodInvocation* metadata object for the method entity of the relation and the set of *Strings methodIDSet* contains the ids of the *Method* metadata objects for the method entities of the *CallGraphEdge* relation, one for each method that may be called by the invocation.

```
class InstanceFieldPointsTo extends SymbolRelation {

    String fieldID;
    String baseHeapAllocationID;
    List<String> heapAllocationIDSet;
    ...
}
```

Figure 5.3: The *InstanceFieldPointsTo* class representing InstanceFieldPointsTo relations.

The field *fieldID* of the class *InstanceFieldPointsTo* is a *String* indicating the id of the *Field* metadata object for the field entity of the *InstanceFieldPointsTo* relation. The *baseHeapAllocationID* field is a *String* indicating the instance of the object whose field may point to other objects. The set of *Strings heapAllocationIDSet* field contains the ids of the *HeapAllocation* metadata objects for the heap allocation entities of the relation, one for each heap allocation that the field of the specified object may point to.

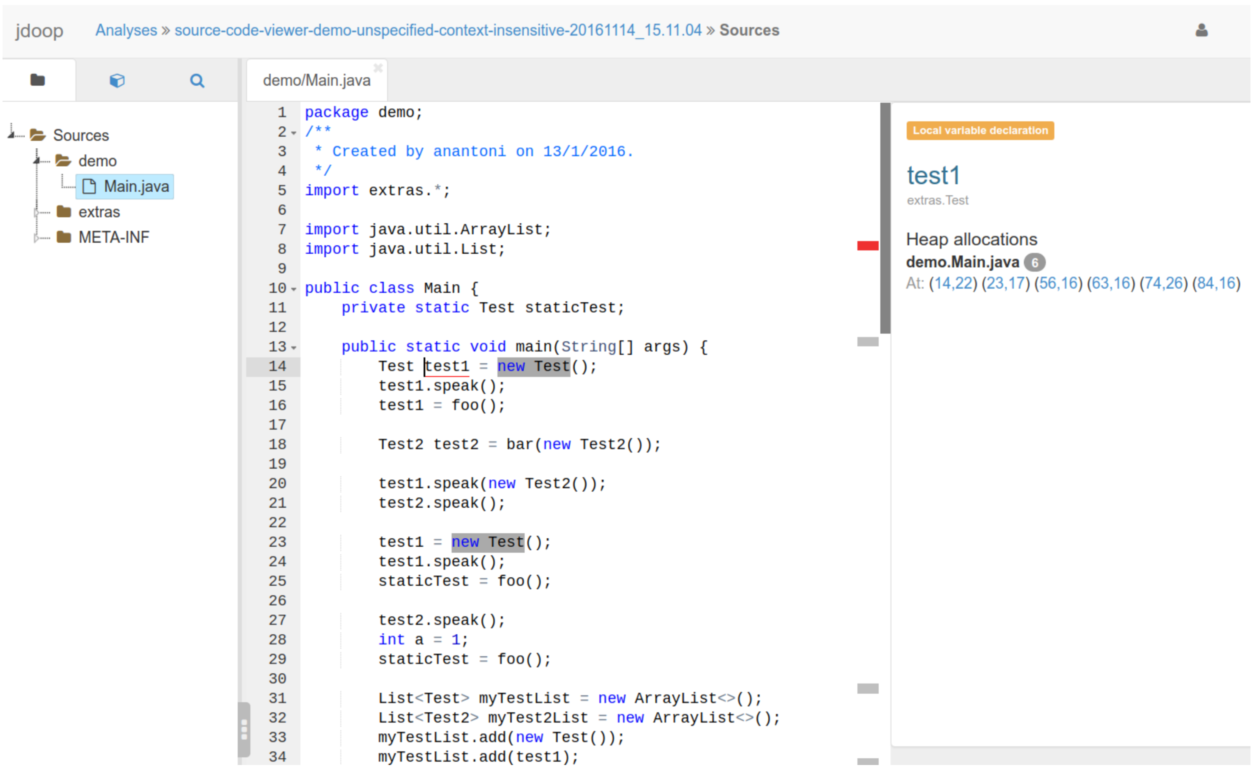
Finally, the results can be stored in a database and queried by a code comprehension tool in order to demonstrate the analysis results on the source code.

6. DISCUSSION AND APPLICATION

In this chapter we present a source code viewer as an application that consumes our enriched relations to demonstrate analysis results on the source code. Our source code viewer is a read-only web-based editor, on which the user can view the source code of a program. Moreover, the user can click on any position within the browser and our database will be queried about the symbol that may be residing in those coordinates. If there is indeed a symbol there, depending on whether it is a class, field, method, variable, method invocation or heap allocation, we present any information we have in the database for that symbol. For symbols that are variables, fields or methods, in particular, we execute a second query to our database which will return the enriched *VarPointsTo*, *CallGraphEdge* or *FieldPointsTo* objects. Thereupon, we can represent the analysis results for these relations on the source code since the enriched relations contain the coordinates of all the symbols associated.

Figure 6.1 demonstrates the result of a click to a variable occurrence in the source code viewer. The occurrence of variable *test1* in line 46 which the user clicked, is underlined, along with its declaration in line 14. The heap allocations that *test1* may point to are highlighted in lines 14, 23, 56, 63, 74, 84 — we demonstrate the highlighted heap allocations up to line 34 due to space limitations.

Currently, our *doop-jcplugin* project consists of 1,464 lines of Java code. Utilizing the extensible architecture of the Java Compiler it was much easier to create our own compilation phase rather than having to intervene to the source code of the Java Compiler itself, which spans over 130,000 lines of code. Our approach is much simpler and cleaner, and at the same time fully modular. Anyone can use the *doop-jcplugin* as long as they have the Java 8 Compiler on their system.


jdoop
Analyses » source-code-viewer-demo-unspecified-context-insensitive-20161114_15.11.04 » Sources
demo/Main.java
Sources
demo
Main.java
extras
META-INF

```
package demo;
/**
 * Created by anantoni on 13/1/2016.
 */
import extras.*;

import java.util.ArrayList;
import java.util.List;

public class Main {
    private static Test staticTest;

    public static void main(String[] args) {
        Test test1 = new Test();
        test1.speak();
        test1 = foo();

        Test2 test2 = bar(new Test2());

        test1.speak(new Test2());
        test2.speak();

        test1 = new Test();
        test1.speak();
        staticTest = foo();

        test2.speak();
        int a = 1;
        staticTest = foo();

        List<Test> myTestList = new ArrayList<>();
        List<Test2> myTest2List = new ArrayList<>();
        myTestList.add(new Test());
        myTestList.add(test1);
```

Local variable declaration

test1
extras.Test

Heap allocations
demo.Main.java 6
At: (14,22) (23,17) (56,16) (63,16) (74,26) (84,16)

Figure 6.1: A screenshot of the *VarPointsTo* relation presented on the source code

7. CONCLUSIONS AND FUTURE WORK

We presented a Java Compiler plug-in that produces source-code metadata in order to match them with static analysis results and enrich them. This is an ongoing project and in the future we aim to be able to produce metadata for every possible valid Java program. Another step forward for the *doop-jcplugin* will be the generation of metadata for complex expressions instead of just symbols. We also showcased the simplicity of utilizing the new Java Compiler mechanism for plug-ins and we presented a fair amount of the obtainable information during the compilation phases.

BIBLIOGRAPHY

[1] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[2] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, October 2015.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Raoul-Grabriel Urma and Jonathan Gibbons. Java Compiler Plug-ins in Java 8. *Oracle Java Magazine*, January/February 2013.