Context-sensitive Points-To Analysis

Comparing precision and scalability
Abstract

Points-to analysis is a static program analysis that tries to predict the dynamic behavior of programs without running them. It computes reference information by approximating for each pointer in the program a set of possible objects to which it could point to at runtime. In order to justify new analysis techniques, they need to be compared to the state of the art regarding their accuracy and efficiency. One of the main parameters influencing precision in points-to analysis is context-sensitivity that provides the analysis of each method separately for different contexts it was called on. The problem raised due to providing such a property to points-to analysis is decreasing of analysis scalability along with increasing memory consumption used during analysis process. The goal of this thesis is to present a comparison of precision and scalability of context-sensitive and context-insensitive analysis using three different points-to analysis techniques (Spark, Paddle, P2SSA) produced by two research groups.

This comparison provides basic trade-offs regarding scalability on the one hand and efficiency and accuracy on the other. This work was intended to involve previous research work in this field consequently to investigate and implement several specific metrics covering each type of analysis regardless context-sensitivity – Spark, Paddle and P2SSA. These three approaches for points-to analysis demonstrate the intended achievements of different research groups. Common output format enables to choose the most efficient type of analysis for particular purpose.

Keywords:
Points-To Analysis, Context-sensitivity, Spark, Paddle, P2SSA, Simulated execution, Binary Decision Diagrams, JNI, Soot, Analysis Time, Call Edge, Heap Object, Analysis Memory.
Acknowledgments

I owe my deepest gratitude to my supervisor Antonina Danylenko and Dr. Jonas Lundberg for their direction, assistance and guidance. In particular, Antonina’s recommendations were invaluable for my project improvement.

I wish also to thank Tobias Gutzmann for helping me with necessary information for this thesis.

Special thanks should be given to my student colleagues who helped me with their advices and assistance. Words alone should be given to Kateryna Kozlova who helped me in many ways.
Table of content

1. Introduction .................................................................................................................. 1
   1.1 Background .............................................................................................................. 1
   1.2 Tasks and goals ...................................................................................................... 2
   1.3 Motivation ............................................................................................................... 2
   1.4 Thesis outlines ...................................................................................................... 3
2. Theoretical background .............................................................................................. 4
   2.1 Program Analysis ................................................................................................. 4
   2.2 Points-To Analysis .............................................................................................. 4
   2.3 Points-To Analysis properties ............................................................................ 6
   2.4 Binary decision diagrams (BDD) ....................................................................... 6
   2.5 Points-to Analysis Techniques ........................................................................... 7
      2.5.1 Soot optimization framework ..................................................................... 7
      2.5.2 Spark ............................................................................................................. 8
      2.5.3 Paddle ........................................................................................................... 9
      2.5.4 P2SSA .......................................................................................................... 10
3. Implementation ........................................................................................................... 12
   3.1 Implementation Details ....................................................................................... 12
      3.1.1 Overview of using technologies ................................................................. 12
      3.1.2 JNI usage in BDD libraries ...................................................................... 12
      3.1.3 Soot application ......................................................................................... 13
      3.1.4 Spark ........................................................................................................... 14
      3.1.5 Paddle ........................................................................................................... 15
      3.1.6 P2SSA .......................................................................................................... 16
   3.2 Functional structure .............................................................................................. 17
4. Experiments ............................................................................................................... 19
   4.1 Metrics .................................................................................................................... 19
   4.2 Experimental setup .............................................................................................. 19
      4.2.1 Spark configuration .................................................................................... 20
      4.2.2 Paddle configuration .................................................................................. 20
      4.2.3 Extracting the information ....................................................................... 21
   4.3 Comparing Analysis ............................................................................................. 21
   4.4 Summary ............................................................................................................... 27
5. Conclusions and Future Work ................................................................................. 28
References ....................................................................................................................... 29
Appendices ...................................................................................................................... 30

Appendix A: The example of abstract object heap file................................................. 30
1. Introduction

In this chapter the problem background, the motivation of the task and goals will be discussed. The problem background and motivation also dive into discussion why it is always important to have an up-to-date overview of the state of the art in software analysis industry field. Goals and goals criteria are defined according to the formulation of the problem and the scope of the thesis task.

1.1 Background

Software industry grows and evolves with enormous progress nowadays. This is the result of applying the research achievements to particular fields or sectors as in other industries. The software as the main product of the industry has not been easy to produce and maintain because of high level of human interaction over it. Enterprise level software has become a big trade-off between efficiency in particular business and probability of failure in certain operation. This leads to the procedure of preventing such failures in terms of better software testing and better software production. However, software development itself is not the only thing to rely on. The existing software should be analyzed in the right way in order to prevent the general issues and to make certain issues to be general. In analysis case it is necessary to avoid issue “puddles” for correct functionality — when poorly described feature becomes an issue. Of course, this has to do with software producers and software consumers who may mistake correct things in the software that are called false-positives. In general, such substitution could be performed by any of involved parties — “a misunderstood explanation means the error is ignored or, worse, transmuted into a false positive” [1].

However, the room for software analysis occurs when the software has strict requirements concerning the functional and non-functional aspects. Program analysis is the process of automatically analyzing the behavior of computer programs [2]. In general, it has two basic interpretations strongly connected to the means of analysis behavior. One of these is static analysis that takes into account only static representation of the program in terms of source code, byte code and low-level instructions. The other is a dynamic analysis that relies on program execution and run-time tracking for all properties of the given software. In this thesis we concentrate on the static program analysis.

The major application of the program analysis results is to allow compilers to generate code avoiding redundant computations, for example by reusing available results, managing already known values and getting rid of unreachable loops [3]. It is given by the set of cases in optimization logic when analysis results are provided.

Two main computational criteria — time and memory — have always been the aspects of careful observations. From the technical level they lead to only one thing — operational data and they depend on how efficiently this data is being manipulated. Taking into account one of the most popular programming paradigms (Object-Oriented Programming) it has to do with better memory management in objects that are being used in the particular software. This static code analysis technique, strictly concerning the pointers in memory, is called Points-To Analysis. Since Points-To Analysis requires a huge amount of free memory (comparing to other types of software analysis) it is already a challenge to get basic trade-off for precision and scalability. Of course, the software itself is different and software analysis depends on this difference in order to get the efficient and precise results.
Traditionally, the amount of data presented in the run-time memory had a direct impact on time for operations. However, in terms of Points-To Analysis there are also an algorithmic logic and data structures. They take place in all levels of analysis ending up in results. The algorithms depend on theoretical part of operations and time they take. In theory, presumed time for operations should be computed in regards to data and operation logic. By default, time complexity can be simply mapped from theory to practice, but in cases of additional options in Points-To Analysis (to be more precise in data manipulations and algorithms that are being used) there could be unpredictable amount of differences presented or discovered in time. Of course, there is no such difference in memory since well-designed data structures can simply serve as planned to omit leaks in memory and data-inconsistency.

In general, Points-To Analysis as a technique has a room for making experiments in time and memory. Moreover, the results gained from these experiments construct the basis for analysis improvement.

The Points-To Analysis technique implementations that exist nowadays are Spark, Paddle and Points-To SSA. Spark represents context-insensitive approach comparing with others and it is theoretically and practically faster due to low precision in results. Paddle is context-sensitive framework that supports several context abstractions (call site strings, object sensitivity and different algorithms implementations). Points-To SSA is context-sensitive analysis that supports few context abstractions as well. It uses Static Single Assignment form for program representation.

1.2 Tasks and goals
The main task of this thesis is to compare well known Points-To Analysis techniques and provide an extended overview of their precision and scalability.

This task has decent goals of configuring, executing and evaluating several types of Points-To Analysis that is Spark, Paddle and P2SSA. Basically, the evaluation of analysis implies the analysis comparison based on Heap Objects, Call Graph and Reachable Methods metrics. This will help to assess which type of analysis is more efficient in comparison to others.

The goal criteria are as follows:
- provide the same format for the output results from different analysis types in order to adapt metrics for data comparison and to achieve analysis type compliance;
- perform the analysis using different platforms with regards to context-sensitivity;
- compare the results in terms of memory and time consumption.

1.3 Motivation
The reason that Points-To Analysis has such big impact on software environment improvement lies in an approach for taking complete set of analysis results into account. Such set could also have a direct impact on software improvement process by static and dynamic suggestions in certain program code parts (software design patterns could be included). Comparing the given properties we can achieve not only the efficiency of software code optimization but also the efficiency of certain type of analysis. Comparison of different types of analysis makes a room for research since they have many properties to rely on and to make a decision of what to use in particular case. These properties are connected to Inter-procedural Analysis when all required data is collected while traversing
between functions (or functional points) in the program code in order to have a complete vision of static behavior of the program.

It is very important to have a strict overview of existing analysis techniques, moreover correct comparison may light up all bottlenecks in certain analysis implementation that will lead to better and more precise analysis results in the future. Such comparison will also help with necessary decision for using certain technique while solving certain problem or type of a problem.

1.4 Thesis outlines
The thesis begins with introduction Chapter 1 that includes problem definition, motivation and goals. Chapter 2 briefly describes the program analysis theory and points of interest in Points-To analysis — context-insensitive Spark Points-to Analysis, context-sensitive Paddle and P2SSA Points-To Analysis. We explain how they use Soot — Java byte code framework in order to get access to binary representation of the program. In Chapter 3, we describe configuration and usage of Spark, Paddle and P2SSA among with related technologies. We present functional structure of the tool that extracts necessary analysis information. We make an outlook to specific techniques that are necessary to be used. In Chapter 4, we define metrics, make experiments and present the results of Spark, Paddle and P2SSA comparison going into details of common metrics’ differences. Finally, in Chapter 5, we make a conclusion of this work with highlights on theory that was used in thesis practice.
2. Theoretical background

In this chapter the theory behind software analysis and Points-To analysis will be discussed. We focus on techniques, technologies and data structures being used for extracting, storing and processing the analysis data.

2.1 Program Analysis

Program analysis is the process of understanding the program behavior. It is used either to analyze the program correctness or to perform program (together with environment) optimization. On the one hand, it can be applied as the basic technique used in optimizing compilers and other programming tools for obtaining information about the possible states that program can reach during execution without actual execution (static program analysis) providing any input data [4]. On the other hand, it is the way of monitoring and dealing with run-time behavior of software under typical and non-typical tests (dynamic program analysis). Keeping in mind first case when actual program source code is transformed into byte code or low-level code it is natural to apply different techniques that will improve program execution in the future. It is important to know that such transformations should be efficient enough to save time spent in finding and applying these transformations and making them general in the future. This has been commonly described for Inter-procedural analysis of truly Object-Oriented programming languages (Java, for instance) when analyzing tools goes through each statement of the functional points (functions and procedures) and builds en execution sequence out of it. As the result it provides a basic understanding of the behavior without actual execution but in certain cases it is still hard to presume the behavior due to programming language-specific features like reflection or dynamic class loading. In case of Object-Oriented languages any part of the execution sequence depends on objects and objects’ fields. It is also essential to understand that not only the separate objects are the only point of interest but the scope of their usage or “context” is also significant. Improvements based on these analysis results — have been a common way of justifying the compiler optimization. Analysis results have recently found their application in software engineering tools that help developers understand, maintain and verify programs [5]. It enables to maintain the software much better in general from both sides — internally (program code) and externally (program environment code).

2.2 Points-To Analysis

The fundamental challenge of dealing with the objects in inter-procedural program analysis is the way how sets of memory location pointers could be represented. Traditionally, this representation was meant to encode the pairs of variables which could point to the same memory location [6]. It had two major problems — the size of these sets could be in power of 4 bigger comparing to the number of variables and this sets did not give any other information about the variables they contained - their sizes, types and any other additional info. In order to solve these problems Pointer Analysis or Points-To Analysis was introduced [7]. Points-To Analysis separates program memory (memory which is in use by this program) into blocks of data and computes a set of such blocks (objects, for instance) that each variable can point to. As stated before, not only separated pointer information makes it possible to reach efficiency in common way but also the scope or context in which this
pointer information is gathered and stored, tracking the flow of pointer information gathering.

There are two most common algorithms of construction points-to sets while traversing the program code. One of them is Andersen’s algorithm [8] when each node of the pointer assignment graph (being constructed on the fly) is represented by single variable. The other is Steensgaard’s algorithm [8] when each node is represented by set of variables connected by single edge. Consider the following Example 2.1 in Java programming language:

```java
Object a,b,c,d;
a = new Object(); //A object
b = new Object(); //B object
c = new Object(); //C object
d = new Object(); //D object
a = b;
c = d;
a = c;
```

Example 2.1: Java code example to show difference between Andersen and Steensgaard algorithms.

Points-to set in Andersen case should be as next:
- a → b
- a → c
- c → d

And pointer assignment graph for this example is presented on Figure 2.1.

![Figure 2.1: Example of Andersen algorithm graph](image)

Steensgaard case has another approach on combining of variables, points-to set is next:
- a → b
- a → c
- b → d
- c → d

Steensgaard’s single approach on edges will give next results presented on Figure 2.2.

![Figure 2.2: Example of Steensgaard algorithm graph](image)
Such large sets of pointers have always been a tricky question when it comes to their storage. Common data structure should be adopted to fulfill the efficiency in faster writable and readable aspects. In order to experiment with more precise program analyses there is one called Binary Decision Diagram (BDD).

2.3 Points-To Analysis properties
There are several properties of Points-To Analysis – context sensitivity, flow sensitivity and field sensitivity.

Context sensitivity has a most significant impact on analysis precision due to separate treatment for each method in the program. In practice, each time analysis considers new method call it creates a new structure that represents unique method scope in the memory. It treats all read/write operations inside this method in scope of this context and makes this information available later for its processing.

Flow sensitivity identifies the order of operations that might be taken into account for producing the analysis results. In theory, it should increase the precision of the analysis but it could be useful in certain situations – for example when some read/write operations have side effects and order of operations is a top priority for assumptions.

Field sensitivity treats each instance field as a global variable. In scope of object representation of the given program – each unique field of certain object will have a separate room (“a set”) for all read/write operations for it.

2.4 Binary decision diagrams (BDD)
BDD has been a very attractive way of representation state sets in model checking area when it was used to check (quickly read) large state spaces. Recent research showed that “BDDs are an effective representation of collections of large sets in inter-procedural program analyses, and their use facilitates the development of and experimentation with new, precise, efficient analyses” [5].

BDD is a directed acyclic graph with values 0 and 1 in the nodes. It has two terminal nodes (with values 0 and 1 respectively) and any amount of non-terminal nodes. Each non-terminal has value of 0 or 1 and has two successors. In order to verify that given binary vector (object in practical case) is in the given set - it gets the binary representation and traverse the graph depending on the node values ending either with 0 (set does not contain this vector) or with 1 (set does contain this vector).

Referring to the example above (Example 2.1) we have identified the following points-to sets:

\[
\begin{align*}
\text{Pt (a)} & \rightarrow A, B, C, D \\
\text{Pt (b)} & \rightarrow B \\
\text{Pt (c)} & \rightarrow C, D \\
\text{Pt (d)} & \rightarrow D
\end{align*}
\]

The common set when each variable paired with each object that it might point to will be represented as \{(a,A),(a,B),(a,C),(a,D),(b,B),(c,C),(c,D),(d,D)\}

If we encode each pair of variable (a=00, b=01, c=10, d=11) and corresponding object (A=00, B=01, C=10, D=11) the resulting set will be as follows:
In such way points-to sets are using BDDs which are interpreted in the graph in Points-To analyses tools.

2.5 Points-to Analysis Techniques

In this sub chapter we discuss detailed specifications of Points-To Analysis implementations and tools that are commonly used.

2.5.1 Soot optimization framework

In order to start static program analysis and get an access to program code or byte code there is a widely known framework. It is called Soot [9]. Soot is a Java optimization framework that provides an easy access to Java program byte code and contains various tools for its manipulation.

One of the most advanced features of Soot is Intermediate Representation (IR) [9]. IR is a way of providing byte code access. In other words, it is an interface. Keeping in mind that all program logic is written in method statements goes that IR plays major role in presenting such statements in different abstract level which definitely helps in different analyses.

Soot distribution comes with 4 IRs — Baf, Jimple, Shimple and Grimp. Baf is a streamlined (stack-based) access to method body. Jimple is a typed 3-address byte code representation with optimization purpose in general. Shimple is SSA (Single Static Assignment form) version of Jimple, it rewrites Jimple code (similar to text representation of byte code but with temporary variables defined) in order to make each variable to be assigned only once. Grimp again transforms Jimple to provide the easiest way for disassembling and reverted code analysis [10].

As a framework Soot has its own object representation of the given program code. This object representation exists in data structures of different code levels:

- Scene — it is a main container for every aspect that Soot can provide. It contains a main class (starting point for analysed program), application classes (other classes in analysed program) and also various information in regards to Inter-procedural analysis like call graph, pointer information, reachable methods, etc;
- SootClass — main representation of the class in the analysed program. It contains a list of SootMethods and SootFields and provides an extended information about Java class type and class hierarchy;
- SootMethod — basic implementation for method in Object-oriented programming for Java with list of statements (Body), method parameters and return type (if any);
- SootField — represents field of a class;
- Body — taking into account different Intermediate Representations Soot provides different method body interfaces (namely BafBody, JimpleBody and so on) in order to make support of all of them. In JimpleBody case, for instance, it provides a list of statements in Jimple way with specific typed representation for Java syntax.

Comparing all different IRs of Soot — Jimple is a most important when it comes to pointer analysis. First of all translation from byte code to Jimple is done by simply moving instructions to statements without types, then introducing temporary variables to cover internal logic and then being optimized with removing unused assignments and variables. After all manipulations Jimple code is definitely optimized (comparing to ordinary Java
byte code instructions) to process it with program analysis. From internal prospective and development such manipulations make Jimple to be between Java byte code and Java source code with high readability options. Jimple is a perfect background of implementing memory efficient analyses based on it — Spark and Paddle.

2.5.2 Spark

Spark is a Points-To Analysis framework with context-insensitive approach as basis. The core component of Spark is pointer analysis engine [6]. This engine is designed to use Soot as the source of code Intermediate Representation. The integration between Soot and Spark is presented on Figure 2.3 [6]. This integration happens in “Other Client Analyses” process where Spark takes action as consumer of Intermediate Representation code. The processes showed below Pointer Analysis Engine are internal and they help to optimize the initial byte-code with trimming, annotating and side-effects resolving.

The Pointer Analysis Engine contains three main parts in regards to stages in analysis — pointer assignment graph builder, pointer assignment graph simplifier, points-to sets propagator, which uses this graph to optimize and present resulting points-to sets by the help of internally designed classes. The basic optimization for pointer assignment graph in the second step is done by analyzing and merging (identified by fast union-find algorithm) nodes that have the same points-to sets. In regards to nodes merging there is a huge work should be done in modifying all successors and predecessors in the certain node. However, Spark uses various optimization algorithms in its simplifier and application of different techniques on all parts of Spark itself (builder, simplifier or propagator) helps to achieve yet precise and efficient points-to analysis.

After final propagation Spark uses its own infrastructure in order to present resulting points-to sets. There are four implementations for points-to sets exist in Spark distribution. All of them are designed to implement or extend the basic PointsToSetInternal class — Hash Set, Sorted Array Set, Bit Set, Hybrid Set. User-defined collections with objects of such class provide a general usage of Spark in comparison of various points-to analyses.
2.5.3 Paddle
There exists a common opinion that context-sensitive approach in program analyses would provide the most precise and accurate results, comparing to context-insensitive ones. Nevertheless, there is a trade-off between analysis efficiency and memory and time consumption. Paddle is context-sensitive analysis framework that uses Soot for accessing the byte code [11]. It provides several options in various aspects of context abstraction, object sensitivity and optimization algorithms. It uses BDDs in order to store information
about context and introduces several modifications to internal classes comparing with Spark:

- ContextAllocNode (Spark’s AllocNode);
- BDDPointsToSet (Spark’s PointsToSetInternal);
- PaddleField.

Like Spark — Paddle is based on 4 types of statements when analyzing byte code. These are allocation (when new object of certain type is created), simple assignment (when variable is assigned to another variable and moves its object into set), field store (when object is assigned to certain field of another object), field load (when field of an object is assigned to another object). Obviously, to get that kind of statements — they can only be generated by one of the Intermediate Representations of Soot — Paddle uses Jimple for that purposes.

In order to apply BDDs for more efficient program analyses, Paddle uses Java language extension Jedd with its initial purpose of having database-like relations between internal classes supporting BDDs. In addition, it is the only way of integration for Paddle and libraries with BDDs implementation — BuDDy, CUDD, JavaBDD, SableJBDD [5].

The core of Jedd implementation is the theory of relations between objects and how these relations are encoded using BDDs. Relations have several types of entities — physical domain, attribute, numberers. These all help to propagate BDDs (using Propagator class) within the scope of Paddle options - context sensitivity, pre-jimplify.

In general, there are several key contributions of Paddle on the way of achieving more precise and efficient analyses [5]:

- On-the-fly call graph construction;
- BDD-based prerequisite and client analyses;
- Reducing the cost of encoding prerequisite analysis results in BDDs;
- Parameterized context sensitivity;
- Modular design.

Together with Jedd — Paddle is a significant framework in the scope of context-sensitivity (among with context-insensitivity) and its options make it possible to achieve precise analysis results. It is also beneficial because of constant call graph and points-to set optimization by various algorithms for BDDs.

### 2.5.4 P2SSA

P2SSA stands for Points-To SSA-based analysis approach. SSA is a Static Single Assignment form is type of Intermediate Representation of the program when each variable is assigned only once. The several usages, typically assignments, are split into versions.

P2SSA introduces a new way of considering context-sensitivity as precise model for program analysis. It uses SSA-based program view-base where each method is represented by a method graph [15]. In this graph – operations are presented as nodes. Defining and using operations for unique local variables are marked as edges between particular nodes. Finally, a method graph looks like a semantic abstraction of logic for that particular method within the SSA form. Despite of edges that are strictly represent operations over variables there is a model of dependencies between different memory operations [15]. On the other hand, graph nodes have properties (attributes in SSA form context) that may store additional information for certain node (for example – node that represents allocation of new object may store the name of the class this object is created from). According to the way the method statements are presented in the method graph this SSA based analysis is
flow sensitive that means that the analysis processes each statement in the order that appears in the program code. Moreover, analysis itself is a simulated execution mechanism that acts similarly to execution environment — simulates method processing on different execution depths starting from main class of the program and finishing the analysis when the execution queue is finished or any exit point is reached. This makes simulated execution more precise than many other analyses using classic flow-insensitive approach with Points-To assignment graph [15].

Points-to SSA analysis is a context-sensitive analysis, moreover, it introduces new context definition called This-sensitivity. The analyses developed earlier — Spark and Paddle — use Insensitive and Object-sensitive approaches respectively. The main difference in definitions for contexts is in the technique that is being used under their classification — for example CallSite technique groups all calls from one call site within one context. This-sensitivity relies on calls that use unique points-to set as separate context. This means that two calls from different abstract objects potentially having the same points-to sets will be presented in the single context.

Points-To SSA analysis depends on Soot byte-code framework and uses Shimple Intermediate Representation for SSA graph construction. This helps to use exactly one model (method graph) for different contexts within the same method distinguishing them by node-to-values map attached to each context [15]. In theory, this-sensitivity for contexts together with simulated execution should give more precise results comparing to other classic context-sensitive approaches.
3. Implementation

Chapter discusses the implementation of key technologies. We make a description of practical aspect in using technologies for several analysis implementations focused by this thesis.

3.1 Implementation Details

In this sub-chapter we dive into the details of platforms, tools and technical analysis implementations that being used in tool for extracting, presenting and comparing analysis results.

3.1.1 Overview of using technologies

Java is used for all analysis techniques and implementations in this thesis. Binary Decision Diagram implementation, required by Paddle, uses Java Native Interface (JNI) for faster data processing through algorithms written in native code.

Java has been a valuable platform in academic world due to its openness and support from many industry players past decade. It is a standard for research practice and experiments. It is also an enterprise basis for many applications in different businesses nowadays. Since it is the one of the first platforms supporting cross-platform approach — it has a room for improving on execution environment level. The principles of running applications on Java platform by means of byte code instead of interpretation approach make this room even bigger — that is why Java is a perfect example of experimenting with program analysis.

3.1.2 JNI usage in BDD libraries

In context of using Java as a major platform for analysis purposes there is a layer between managed code and native processor commands. This layer is intended to extend Java applications with calls to native procedures on operating system level or with complex time-consuming and memory-consuming operations (due to the fact that the JVM layer is obviously slower than operating system layer but rather “safe”). Taking into account omitting the “safe” aspect of code executions by simplifying memory operations and handling the control of it — this is a significant solution for moving logic from complex and memory-consuming algorithms onto this layer. It is called Java Native Interface. The general infrastructure of Java Native Interface is presented on the Figure 3.1 [13].

![Figure 3.1: JNI architecture](image-url)
Of course, this is not a perfect solution when it comes to portability that Java offers [14]. Such improvement from performance aspect requires native implementations of all logic and algorithms for concrete operating system. This is not the only thing we can care about — another important part that not all JVM realizations behave similarly on major operating systems in case of JNI due to difference in their architecture. This comes to another problem that “safe” code from the first view can “destabilize the entire JVM in ways that are very difficult to reproduce and debug” [14]. But from software analysis point of view — all such issues could be ignored since logic in this particular case on that level does not require specific features of JNI, only simple native calls with simple input and output data.

3.1.3 Soot application

All analysis approaches performed in this work use Soot byte-code framework. On the implementation level Soot exists as separate library in compiled or source distribution. It has also 2 options of starting the byte-code reader engine — from command line or using the library classes. Since it is being used as library in all analyses — first it requires Soot Options class being setup. Options and Scene classes use classic singleton mechanism in order to prevent multiple engines working on the same code domain. Options class acts as duplicate to parameterized options from command line and it expects all necessary values to be entered:

```
Options.v().set_soot_classpath(project.getClassPathsString())
```

This command naturally sets the classpath for the code and libraries that will be used while reading the complete byte code package. As soon as all options concerning the source code classes, entry class and excluded/standard classes are set up — some general Soot options should be configured [10]:

```
Options.v().set_full_resolver(true); // setting resolution for managing simple and complex types in library
Options.v().set_whole_program(true); // takes into account all references from source code
Options.v().set_allow_phantom_refs(true); // allows non-reachable references from the code
Options.v().set_keep_line_number(true); // keeps original line numbers in the source code
```

In recent Soot versions very useful feature for including or excluding standard Java library classes has been implemented. It allows to decrease significantly amount of time and memory while constructing several Intermediate Representations within the Soot. It is called “no-bodies-for-excluded” and it remains experimental for the time-being.
All other manipulations with byte code reader and all IR implementations are done using the Scene class:

```java
Scene.v().getClasses(); // returns all known classes - library and application
Scene.v().getApplicationClasses(); // returns all application classes
```

Since it is entirely used as an interface between byte code and actual analysis engine; it is becoming useful after particular transformations are applied. This makes an alternative to use standard Soot collection classes in order to extract all needed information instead of iterating over all program entities (class, method, statement) and gathering exactly the same information with less time effort at the end.

### 3.1.4 Spark
Spark maintains basic options to be configured in order to get proper results. It starts with special class called SparkOptions that expects key-value map of definitions that are set to be included:

```java
HashMap<String, String> opt = new HashMap<String, String>();
opt.put("enabled","true");
SparkOptions sopt = new SparkOptions(opt);
```

In its constructor SparkOptions class interprets all properties that are written in given HashMap into Soot Scene extension in order to have them fully available during analysis runtime.

There are several options for defining analysis properties such as field-sensitivity, propagation, runtime garbage collection and others. However, most of them are strictly connected to Pointer Assignment Graph that will be as result of performed Spark analysis. The final step before getting the result is to perform transformation — singleton SparkTransformer is used for this:

```java
SparkTransformer.v().transform("",opt);
```

As the result — all Soot framework entities could be used for accessing internal program structure and Spark extension for Soot is the main source for extracting Points-To information:
PointsToAnalysis p2a = Scene.v().getPointsToAnalysis();

Obviously, it is returned as a Pointer Assignment Graph structure that is explicitly defined in Soot distribution (PAG class). It has all needed methods for finding correct Points-To sets (PointsToSetInternal):

```java
PointsToSetInternal p2s = (PointsToSetInternal) p2a.reachingObjects(l, sootField);
```

Where `l` is a current variable (in Soot terminology it is a JimpleLocal object) and `sootField` is a current field that may point to resulting Points-To set of objects.

All required information which will be used in metrics later on — could be retrieved either by iterating over all variables (namely collection of JimpleLocals) and getting Points-to set of each variable-field pair or by implementing custom handler for tracking allocation nodes in the method body (JimpleBody) and processing statements for field load-store operations.

### 3.1.5 Paddle

Paddle has its own set of options and transformation classes and they are being used right after Soot is finished with byte-code model construction. Options are the same simple key-value map with necessary fields for BDD operations backend, definition of context and propagation mechanism:

```java
opt.put("bdd", "true"); // identifies that BDD will be used
opt.put("backend", "buddy"); // sets BuDDy as low-level library for BDDs
opt.put("context", "objsens"); // sets context-sensitivity to ObjSens.
```

After all options are specified, they are intended to be used in transformation class PaddleTransformer:

```java
PaddleTransformer pt = new PaddleTransformer();
PaddleOptions paddle_opt = new PaddleOptions(opt);
pt.setup(paddle_opt);
pt.solve(paddle_opt);
```
By calling solve() method of PaddleTransformer Paddle will process with Soot program model and make Points-to information using BDDs in the runtime. According to Paddle's Soot extension (soot.jimple.paddle.Results) it will also adjust internal Soot model for keeping BDDs as main structure for Points-to sets — getPointsToAnalysis method will return BDDPointsToAnalysis object which obviously conforms the general PointsToAnalysis interface:

```
BDDPointsToAnalysis p2a = (BDDPointsToAnalysis) Scene.v().getPointsToAnalysis();
```

Since Paddle custom classes implement the same interfaces from Soot this means that they can be used for extracting the same information from internal collections of entities — JimpleLocals and PaddleFields.

As the result for each pair of variable and field — Paddle will return unique Points-to set bounded to certain context:

```
p2s = (BDDPointsToSet) p2a.reachingObjects(l, sootField);
```

By iterating over Nodes inside each Points-to set the list of ContextAllocNodes could be maintained. Originally they should be linked to the context where they are allocated.

### 3.1.6 P2SSA
Points-To SSA analysis has quick adaptation on the technical level. Despite of heap objects that are used to store exact def-use operations over objects — Points-to SSA concentrates on context as container for all read-write and def-use operations. Since it already produces all required information in output files — the task was to re-use that information in more convenient way by implementing parser, extractor and file writer. Parser uses classic XML binding since Points-To SSA produces output files in that format. Extractor processes internal data structures that store all the necessary information for metrics and passes these collections to file writer. Internal domain model consists of the following classes:

- **AbstractObject** — represents single abstract object with information in regards to class that created this object, source code line number where allocation appears and type of the object (namely class name of this object);
- **AbstractObjectSet** — represents the collection of AbstractObjects with unique identifier within single program scope;
- **Context** — represents source method (this context belongs to), collection of AbstractObjects, collection of target methods and unique identifier within single program scope;
- **CustomHandler** — main SAX Default Handler implementation for SAX XML parser library; it uses code-defined parsing phases for keeping the parsing process in correct order according to hierarchy of internal collections of elements (AbstractObjectSets, HeapObjects and Call Graph edges);
• HeapObject — represents points-to sets information in regards to field association with instance object and value objects. Instance object is presented as link to AbstractObjectSet with single element while value objects are presented as AbstractObjectSet with multiple elements. Heap Object also contains the source line number where allocation occurs;

• XMLReader — main XML processing code with CustomHandler usage and internal CollectionContainer class which keeps internal collections in correct order (by using phases).

File writer takes into account the order in which object fields and other data is presented and outputs data in text format.

3.2 Functional structure

Analyses comparison relies on tool that runs all required procedures (Soot analysis, Points-To analysis) for certain projects. The general structure of the tool is presented in Figure 3.2:

![Figure 3.2: General structure of comparison tool](image)

Main class has a collection of Java projects to be analysed, counters for memory and time and dependencies for SceneHelper — main class for extracting information that can be taken from Soot Scene model without iterating over all hierarchical elements in the program binary code. Undoubtedly main class has an alternative for Scene approach — to use RunSparkP2AExtractor or RunP2APaddleExtractor for manual binary code parsing and collecting all elements.

SceneHelper class has public methods for accessing the metrics information from Scene and outputting it in the common format. Among with Scene object model — SceneHelper uses SparkVisitor and PaddleVisitor for iterating over resulting Points-To sets and collecting allocation nodes information.

As stated under Spark and Paddle descriptions — firstly SceneHelper uses corresponding PointsToSet derivatives from SimpleLocal variable representations and
SootField derivatives. Final collections of heap objects are being written to the file in correct format.

Since Points-To SSA propagated information comes within proper XML file — the processing architecture will look as Figure 3.3:

Figure 3.3: Structure of P2SSA processing part

Parsing starts with XMLReader class that implements standard SAX parse() method with corresponding operations in CustomHandler — startElement(), endElement() and characters().

The core functional elements presented on Figure 3.3 are entity classes that hold all analysis information (HeapObject, Context, AbstractObject and AbstractObjectSet) and processing elements that presented analysis data in common to Spark and Paddle format (CollectionContainer, CustomerHandler and XMLReader). Since the ordinary analysis data from Points-To SSA are large – entity elements (Context for instance) use lazy-loading model where each attribute is being loaded from XML element by request only.
4. Experiments

This chapter describes the experiments that have been taken in order to form complete vision of the state of the art in Points-To Analysis research.

4.1 Metrics

In order to compare certain software entities (in our case program analysis techniques) there should be metrics defined. In other words, some kind of concrete measurement system should be introduced. Metric is a property measurement. It is important for software analysis properties to have custom well-defined properties of the results (for example, number of objects processed) and not only the process properties (for example, time and memory).

There are several metrics that reflect the program code complexity and value – object heap, call graph and reachable methods. Object heap holds all objects created in the memory through code. Call graph visualizes call sequence of the methods in more convenient way. Call graph edge consists of method caller at the start node and method callee at the end node. Reachable methods are all methods in the source code that can potentially be reached from program start point.

Traditionally, software analysis metrics have been defined as properties of data that is being processed. In pointer analysis case they are points-to sets (relation to objects in memory) and hierarchical elements (relation to call graph and reachable methods). By applying aggregated metrics on software entities comparison could be more efficient.

Therefore, the final list of metrics will be the following:

- Analysis time in seconds:
  We compute the time of running test exactly after Soot finishes byte code parsing;
- Used memory in megabytes:
  We track memory used for particular Java process in terms of difference between total memory allocated for process and free memory left;
- The comparison for abstract object heap file (number of heap objects):
  We track all field accesses (read/write) with structure called abstract heap and collect instance object (that initially assigned to the field of particular object) together with value objects (that later assigned to this field) into abstract object;
- The comparison for reachable methods (number of methods):
  We rely on Soot Scene infrastructure (transformed by analysis) while listing all methods that can be reached from program startup point;
- The comparison for call graph edges (number of edges):
  Again, we extract call graph edges (parent method and child method) from transformed Soot Scene infrastructure.

4.2 Experimental setup

In this section we dive into the details about configurations for each type of analysis implementation with regards to maximum memory consumption and BDD engines.
4.2.1 Spark configuration
The experiments have been started within Spark. Spark uses Soot singleton containers for byte code access and transformations. Since it is context-insensitive analysis its options are left by default and garbage collection (“force-ge” option) is turned off in order to measure used memory correctly. Spark is configured as following (Table 4.1):

<table>
<thead>
<tr>
<th>Name of the option</th>
<th>Value for the option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program arguments</td>
<td>spark</td>
</tr>
<tr>
<td>VM arguments</td>
<td>-Xmx2048M</td>
</tr>
</tbody>
</table>

Table 4.1: Spark run configuration

4.2.2 Paddle configuration
Since Paddle has extended set of features, all of them should be configured properly for treating it in terms of context-sensitivity. Paddle configuration is shown on the Table 4.2.

<table>
<thead>
<tr>
<th>Option source code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt.put(&quot;bdd&quot;, &quot;true&quot;);</td>
<td>Using BDD is set to true</td>
</tr>
<tr>
<td>opt.put(&quot;backend&quot;, &quot;buddy&quot;);</td>
<td>BuDDY is selected as target low-level BDD engine</td>
</tr>
<tr>
<td>opt.put(&quot;context&quot;, &quot;objsens&quot;);</td>
<td>Paddle options for context-sensitivity are mandatory - &quot;context&quot; as &quot;objsens&quot; and &quot;k&quot; as &quot;1&quot; (1-object sensitivity)</td>
</tr>
<tr>
<td>opt.put(&quot;k&quot;, &quot;1&quot;);</td>
<td></td>
</tr>
<tr>
<td>opt.put(&quot;conf&quot;, &quot;ofcg&quot;);</td>
<td>Call graph construction will be performed on the fly</td>
</tr>
</tbody>
</table>

Table 4.2: Paddle analysis configuration

Since the implementations of all low-level libraries that do BDD computations for Paddle require Java Native interface – additional run configuration options should be set as command line arguments for Java virtual machine – “-Djava.library.path” should point to directory where selected BDD library could be found. BDD library implementation is a dynamic library for target operation system – “dll” in Microsoft Windows case or “dylib” in Apple Mac OS X case.

Finally the Paddle run configuration will be as following (Table 4.3):

<table>
<thead>
<tr>
<th>Name of the option</th>
<th>Value for the option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program arguments</td>
<td>paddle</td>
</tr>
<tr>
<td>VM arguments</td>
<td>-Xmx2048M</td>
</tr>
<tr>
<td>VM arguments</td>
<td>-Djava.library.path=&quot;/Users/eugene/Dropbox/lnu/_master/p2a/lib/jedd-linux/&quot;</td>
</tr>
</tbody>
</table>

Table 4.3: Paddle run configuration
4.2.3 Extracting the information

Key differences between points-to analysis results in Spark and Paddle (described before) play major role in running methods for extraction. This is lead by differences in context-sensitivity aspect when Spark relies on PAG (pointer assignment graph) while Paddle uses BDDPointsToAnalysis. Both structures conform common PointsToAnalysis interface but objects that they hold – are different (AllocNode class for Spark and ContextAllocNode class for Paddle). However, methods for fetching the local variables (to iterate over them later) in both analyses are the same. After successful application of Spark and Paddle visitors (helper classes that enable common code to iterate over local points-to sets for AllocNodes and ContextAllocNodes respectively) points-to sets data are directly written into common XML file. All types of analysis were evaluated using Java standard library classes the benchmark programs code is linked to.

Two of three (Spark and Paddle) analysis tests (for the complete set of benchmark programs) have been performed on the same computer – Apple Macbook Pro, 4GB RAM, Intel Core i5 2,3 GHz under Mac OS X 10.6 Lion. The Java version is 1.6.0_29, 64bit.

4.3 Comparing Analysis

A few series of experiments for the target analysis frameworks have been performed on the following benchmark programs (Table 4.4). In this table the following data is listed: name and version of the benchmark program, link to its homepage and functional description. These programs have been selected as basis for all experiments after series of tests for general analysis compliance level (projects such as ABC, Jython, JavaC have been filtered from general list due to number of problems when certain analysis implementation fails to continue analyzing the project binary files).

<table>
<thead>
<tr>
<th>Name, version</th>
<th>Homepage</th>
<th>Functional description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javacc, 5</td>
<td><a href="http://javacc.java.net/">http://javacc.java.net/</a></td>
<td>The Java Compiler is a parser generator originally by Sun Microsystems.</td>
</tr>
<tr>
<td>lucene, 3.5</td>
<td><a href="http://lucene.apache.org/">http://lucene.apache.org/</a></td>
<td>Provides a Java-based indexing and search implementation, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities.</td>
</tr>
<tr>
<td>pmd, 4.3</td>
<td><a href="http://pmd.sourceforge.net/">http://pmd.sourceforge.net/</a></td>
<td>PMD scans Java source code and looks for potential problems.</td>
</tr>
<tr>
<td>sablecc, 3</td>
<td><a href="http://sablecc.org/">http://sablecc.org/</a></td>
<td>SableCC is a parser generator that generates fully featured object-oriented frameworks for building compilers, interpreters and other text parsers.</td>
</tr>
</tbody>
</table>

Table 4.4: Benchmark programs
Spark and Paddle results for these benchmark programs have been extracted through actual execution with environment configurations described before while analysis results for P2SSA have been adopted from already predefined XML-based files. P2SSA results were rewritten to the target format in order to take place in analysis comparison.

Keeping in mind that metrics, presented in Metrics section – Abstract Object Heap, Call Graph and Reachable Methods – have been adopted for analysis techniques used in this thesis we can conclude that precision for metrics is sufficient enough to make a comparison out of it.

The following Table 4.5 and Figure 4.1 represents analysis execution time with regards to described benchmark programs (X axis represents benchmark program and Y axis represents time in seconds):

<table>
<thead>
<tr>
<th>Program</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>javacc</td>
<td>59</td>
<td>1841</td>
<td>32</td>
</tr>
<tr>
<td>lucene</td>
<td>23</td>
<td>4218</td>
<td>168</td>
</tr>
<tr>
<td>pmd</td>
<td>83</td>
<td>3202</td>
<td>79</td>
</tr>
<tr>
<td>sablecc</td>
<td>80</td>
<td>3443</td>
<td>128</td>
</tr>
</tbody>
</table>

Table 4.5: Result for analysis time in seconds

These results prove that analysis time logically depends on complexity of the program in terms of lines of code and class depth. However, each case is unique and context-sensitivity may reflect that complexity too.

Table 4.6 and Figure 4.2 present memory used for performing analysis and transformation in each particular case (X axis represents benchmark program and Y axis represents memory used in megabytes).
Memory results reflect not only the complexity of the benchmark program but also the efficiency of garbage collection and memory management in certain analysis case. Spark turns out to be poor implemented for these program examples.

A few minor things have been discovered while adapting Spark and Paddle for common format representing abstract object heap in XML file. One of those is related to Soot byte code line numbers. Since Spark, Paddle and P2SSA are using Soot as byte code reader – the interface for them have to be common. However, Spark and Paddle do not reflect value objects (the objects that have been assigned to the field of certain object) by line numbers sometimes. In these cases the line numbers are specified with “-1”. The example of output XML file is presented in Appendix A.

<table>
<thead>
<tr>
<th>Program</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>javacc</td>
<td>712</td>
<td>273</td>
<td>260</td>
</tr>
<tr>
<td>lucene</td>
<td>802</td>
<td>278</td>
<td>341</td>
</tr>
<tr>
<td>pmd</td>
<td>792</td>
<td>330</td>
<td>310</td>
</tr>
<tr>
<td>sablecc</td>
<td>801</td>
<td>278</td>
<td>268</td>
</tr>
</tbody>
</table>

Table 4.6: Result for used memory in megabytes

![Figure 4.2: Result for used memory in megabytes](image)

Table 4.7: Results for abstract object heap size
Table 4.7 and Figure 4.3 compare the size of abstract object heap for benchmark programs (X axis represents benchmark program and Y axis represents number of objects in the heap).

As seen from the abstract object heap P2SSA provided with less precise results on SableCC project. After more detailed review for files we can conclude that P2SSA treats internal fields for classes as separate heap objects even if they participate only in static context. Another trick was discovered concerning the overridden class constructors that use “this” term for accessing the current object. P2SSA computes abstract heap objects for them (specifying field name as “ClassName$1.this$0”) even when they may not be used in current scope.

The results for reachable methods are presented in table and diagram view on Table 4.8 and Figure 4.4 (X axis represents benchmark program and Y axis represents number of reachable methods) respectively. Call graph edges are presented on Table 4.9 and Figure 4.5 (X axis represents benchmark program and Y axis represents number of call edges).

P2SSA has less precise results for PMD project due to the same problem discovered for abstract object heap for SableCC – excess reachable methods and call graph edges are defined in excess overridden constructors.

<table>
<thead>
<tr>
<th>Program</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>javacc</td>
<td>1072</td>
<td>951</td>
<td>883</td>
</tr>
<tr>
<td>lucene</td>
<td>2229</td>
<td>1140</td>
<td>1109</td>
</tr>
<tr>
<td>pmd</td>
<td>2708</td>
<td>2488</td>
<td>2490</td>
</tr>
<tr>
<td>sablecc</td>
<td>2036</td>
<td>1952</td>
<td>1672</td>
</tr>
</tbody>
</table>

Table 4.8: Results for number of reachable methods
Figure 4.4: Results for number of reachable methods

<table>
<thead>
<tr>
<th>Program</th>
<th>Spark</th>
<th>Paddle</th>
<th>P2SSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>javacc</td>
<td>5812</td>
<td>2788</td>
<td>2120</td>
</tr>
<tr>
<td>lucene</td>
<td>5062</td>
<td>2414</td>
<td>1981</td>
</tr>
<tr>
<td>pmd</td>
<td>5617</td>
<td>5513</td>
<td>6747</td>
</tr>
<tr>
<td>sablecc</td>
<td>9542</td>
<td>9044</td>
<td>8074</td>
</tr>
</tbody>
</table>

Table 4.9: Results for number of call graph edges

Figure 4.5: Results for number of call graph edges

Minor differences in Spark and Paddle (described before) have been studied and described. Take a look at the following call graph edges of Paddle output for open-source project Jython:
The right arrow distinguishes the caller and the callee respectively in all cases of graph edges. The problem in this particular case is in Paddle’s way of meaning for constructor index of super class/interface by “$” sign. After reviewing the source code for these edges – the conclusion is following – Paddle calls the constructor by its index in super class (PyObject in this case) instead of calling the constructor of concrete class when reflection is used for creating new object.

The other case of difference in analyses types is that Spark does not track abstract class overridden methods in nested classes. The example is following (with outlook in source code):

```
<org.python.core.AbstractArray: void
<init>(java.lang.Class, int) ->
<org.python.core.PyArray$ArrayDelegate: void
setArray(java.lang.Object)
```
Some of the Spark experiments have shown the issues in Spark memory management with resulting OutOfMemory Java exception.

Another strange behavior of Paddle call graph edges proved that Paddle wraps <clinit> for the Class itself and continues with graph edge even if the reference to required Class is not found.

Since the analysis results for P2SSA have been extracted on one machine and have been reprocessed on the other, it is hard to overview the differences between Paddle and P2SSA so far.

4.4 Summary
In this chapter the results obtained from static analysis technique implementations of Spark, Paddle and P2SSA have been compared. The comparison is based on defined metrics. These metrics are modified in order to have common format for files. Since it is hard to make them 100% common, some exceptions in regards to differences are specified. During the adaptation process of Spark and Paddle a few problems have been discovered and finally presented in this chapter. This makes a complete overview of context-insensitive and context-sensitive approaches for precise and efficient program analyses. Merging the metric values and presenting them onto the single line concludes that P2SSA has higher precision against Paddle while Paddle is more precise than Spark as was expected.
5. Conclusions and Future Work

The general purpose of this master thesis was to compare different static program analysis approaches from two research groups with regards to context-sensitivity. As the results we received an extended comparison. This comparison provided us with detailed overview of state of the art in static program analysis and the role of context-sensitivity in it. It concludes that Soot framework is definitely a standard for Java byte code manipulations and transformations while concrete approaches leave algorithms and data management under their own infrastructure. The comparison results represent a significant achievement in analysis precision with the help of context-sensitivity used in Paddle and P2SSA. Section 2 dives into the theory background for technologies that have been used in this master thesis. Section 3 represents implementation details of the practical part with regards to Soot, Spark, Paddle and P2SSA application. Section 4 describes some bottlenecks in using Spark and Paddle that have been discovered while implementing the comparison.

Since research topic of static program analysis definitely moves forward nowadays the future of this work should be related to recent upgrades in existing analysis approaches. Some of the specified bottlenecks shall be fixed in the nearest future and then they will have direct impact on analysis results that might change the efficiency and precision. In the future there is also a possibility to adapt and include other analysis techniques (Doop [16], for example) into common metric approaches.
References


http://en.wikipedia.org/wiki/Program_analysis (20111102)


http://www.cse.lehigh.edu/~gtan/projects/multilingual.html (20111102)

http://en.wikipedia.org/wiki/Java_Native_Interface (20111102)


Appendices

Appendix A: The example of abstract object heap file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<heapobj field='org.javacc.parser.Action.action_tokens'>
  <instance class='org.javacc.parser.Action' line='735'/>
  <value class='java.util.ArrayList' line='44'/>
</heapobj>

<heapobj field='org.javacc.parser.Action.action_tokens'>
  <instance class='org.javacc.parser.Action' line='150'/>
  <value class='java.util.ArrayList' line='44'/>
</heapobj>

<heapobj field='org.javacc.parser.Action.action_tokens'>
  <instance class='org.javacc.parser.Action' line='756'/>
  <value class='java.util.ArrayList' line='44'/>
</heapobj>

<heapobj field='org.javacc.parser.Action.action_tokens'>
  <instance class='org.javacc.parser.Action' line='556'/>
  <value class='java.util.ArrayList' line='44'/>
</heapobj>

<heapobj field='org.javacc.parser.BNFProduction.declaration_tokens'>
  <instance class='org.javacc.parser.BNFProduction' line='347'/>
  <value class='java.util.ArrayList' line='42'/>
</heapobj>

<heapobj field='org.javacc.parser.Choice.choices'>
  <instance class='org.javacc.parser.Choice' line='617'/>
  <value class='java.util.ArrayList' line='46'/>
</heapobj>

<heapobj field='org.javacc.parser.Choice.choices'>
  <instance class='org.javacc.parser.Choice' line='725'/>
  <value class='java.util.ArrayList' line='46'/>
</heapobj>

<heapobj field='org.javacc.parser.Choice.choices'>
  <instance class='org.javacc.parser.Choice' line='752'/>
  <value class='java.util.ArrayList' line='46'/>
</heapobj>

<heapobj field='org.javacc.parser.Container.member'>
  <instance class='org.javacc.parser.Container' line='630'/>
  <value class='org.javacc.parser.RChoice' line='1033'/>
</heapobj>

<heapobj field='org.javacc.parser.Container.member'>
  <instance class='org.javacc.parser.SingleCharacter' line='1246'/>
</heapobj>

<heapobj field='org.javacc.parser.Container.member'>
  <instance class='org.javacc.parser.SingleCharacter' line='1182'/>
</heapobj>

<heapobj field='org.javacc.parser.Container.member'>
  <instance class='org.javacc.parser.SingleCharacter' line='1182'/>
</heapobj>
```

30
<value class='org.javacc.parser.CharacterRange' line='1239'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.RZeroOrOne' line='1123'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.RRepetitionRange' line='1150'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.RJustName' line='1097'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.RZeroOrMore' line='1119'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.ROneOrMore' line='1115'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1049'/>
<value class='org.javacc.parser.RStringLiteral' line='1091'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='630'/>
<value class='org.javacc.parser.TryBlock' line='303'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='630'/>
<value class='org.javacc.parser.REndOfFile' line='997'/>
</heapobj>
<heapobj field='org.javacc.parser.Container.member'>
<instance class='org.javacc.parser.Container' line='1630'/>
<value class='org.javacc.parser.RJustName' line='990'/>
</heapobj>