Analysing Aliasing in Java Applications

Björn Berggren
Abstract

Analysing Aliasing in Java Applications

Björn Berggren

Aliasing refers to the possibility of having multiple references to the same memory location and is a cornerstone concept in the imperative programming paradigm. As applications grow large, it is hard for programmers to keep track of all places in the code where they employ aliasing, possibly resulting in hard-to-predict runtime errors. I present a tool, model and query language to perform dynamic aliasing analysis on Java programs. The tool uses the model to simulate the full execution of a program, including how and when objects and references between them are created and removed. The query language consists of constructs that can be nested to form logical expressions, operating on the model to check whether a certain object remains in a certain condition throughout the execution. The language is designed to monitor lifetimes of specific objects and collect information about the execution as a whole. I performed an experiment using the tool, examining traces from programs for profiling, searching and parsing, showing that applications have different aliasing behaviour. All programs mostly use objects based on built-in Java classes, indicating that programmers could benefit from gaining the ability to control and limit how such objects can be aliased. The experiment shows that the tool works and can be useful in this research field. In future work, the tool should be optimised and tested in more extensive experiments.
5.2.1 Hypothesis 1 ................................................. 28
5.2.2 Hypothesis 2 ................................................. 28
5.2.3 Hypothesis 3 ................................................. 29
5.2.4 General Data ............................................... 29
5.2.5 Summary .................................................... 30
5.3 Threats to Validity ............................................. 30

6 Conclusion and Future Work 31

References 32
1 Introduction

In computing, aliasing refers to situations when there exist multiple references to the same memory location [1]. While this is useful in certain situations, this also cause problems. Programmers are only human and are prone to errors and mistakes. Especially in big projects it can be easy to forget which parts of the code that are interrelated; changing the value of a variable in one section will also affect variables in other sections if they are referring to the same memory location. This is particularly pertinent in parallel and concurrent programming where there is a very high demand on the programmer to keep track of when changes may occur and what consequences they might have.

Understanding aliasing could be instrumental for new imperative programming languages. It could offer a much simpler solution to problems such as data races, than the existing ones using locks and atomic operations to achieve synchronized code. In the context of object-oriented programming languages, such as Java (which is the target language for this thesis work), aliasing is an essential and omnipresent concept, as simply passing an object as a method argument or reassigning it to a different variable creates a new alias [2].

There exists a lot of previous work within the field of aliasing analysis. Hogg, Lea, Willis, deChampeaux and Holt identify four different categories that seem to represent the main research approaches [1]: detection, advertisement, prevention and control. Research on aliasing advertisement, prevention and control often involves the development of language extensions, like annotation frameworks [3], or other auxiliary language constructs to help programmers work with aliasing. Aliasing detection, on the other hand, is about analysing and diagnosing existing programs to find information on where, when and how aliasing appears.

1.1 Purpose

The purpose of this project is to provide a tool that allows for researchers to explore custom trace data of Java program executions using a versatile and powerful query language, to gather information about aliasing statistics over the execution as a whole or even examine the lives of particular objects. The purpose is not to provide a complete and comprehensive application of fixed functionality, but rather a prototype of a flexible and extendable tool that can be adjusted for particular goals.

Additionally, the project involves showing that the developed tool is actually fit for use in the field of analysing aliasing. The objective is not, however, to provide results that are necessarily useful in the research context, but rather to serve as a proof-of-concept that show that the tool actually is valuable and useful.
1.2 Hypotheses

To prove the tool’s usefulness, I use an experimental approach. Using the tool, I examine different categories of Java programs, trying to establish whether a number of relevant hypotheses hold. If the tool shows results that seem accurate or at least different between the categorised programs, I can establish that the tool has the potential to be useful.

For the informal hypotheses to be used by the tool in the experiment, they have to be converted into query constructs part of the tool’s query language (see Section 3.5); hence, the experiment results are naturally affected by the translation accuracy (see Section 5.3).

The hypotheses are presented and motivated below.

**Hypothesis 1.** There is a considerable difference in aliasing behaviour between different Java applications.

**Hypothesis 2.** There is a considerable difference in the frequency of using custom versus built-in objects between different Java applications.

**Hypothesis 3.** Some Java applications rarely or never use more than four incoming references at the same time to any custom object during the execution.

Hypothesis 1 is a simple hypothesis theorising that the results actually will show different results in terms of how objects are aliased throughout the execution. This could determine whether some kinds of programs are more interesting than others to investigate and, for instance, implement optimisation methods for.

Hypothesis 2 aims to find out if some kinds of programs tend to mostly use built-in classes or if they actually contain a large amount of user-defined types. Knowledge about this could be pertinent in the relevance of developing some kind of Java annotation framework or language extension to help programmers deal with aliasing problems, much like Almeida proposed balloon types [4], allowing programmers to enforce better encapsulation on their custom classes.

Hypothesis 3 addresses potential garbage collection optimisation. Reference counting is a common garbage collection technique, which is implemented by having live objects keep track of how many incoming references they have respectively [5]. This is typically achieved by adding a "reference counter" field to the object "header" of each object [5]. By being able to assure that the reference count never will exceed a certain number, the size of that field can be limited and fixed, which could allow for potential compilation or garbage collection optimisations. In this case, being able to assure that the reference count to any object never exceeds four would mean that the size of the reference counter field could, theoretically, be limited to two bits (in practice, the lowest assignable size is most probably one byte). The decision of looking for objects with no more than four incoming references in this hypothesis is mostly arbitrary and more of a proof-of
concept that the tool can be used for this kind of goals. Furthermore, the decision of only examining custom objects is mostly based on the purpose of showing the tool’s ability of applying larger, composite queries.

1.3 Limitations

Rather than presenting a complete large-scale application of wide functionality and adaptability (which is out of the scope of this thesis), the purpose of this project is to provide a prototype of a tool with the potential and capacity of becoming truly useful within the field of trace-based analysis of aliasing. The tool is more of a proof-of-concept and is implemented in a modular way, providing interfaces for both the model for the object graph (see Section 2.2) and the query language, making it possible to adjust and extend the tool for specific purposes. The tool analyses program execution traces, but unlike Spencer [6] (see Section 1.4.5), which provides a large but determined dataset of traces, my tool expects the users to input the traces themselves.

The project involves an experimental part, where the tool is used in an attempt to find relations between aliasing behaviour and categories of different Java programs. As an extensive experiment would require the gathering of a thorough and relevant assortment of Java programs to be representative for the different categories of Java applications (which is out of the scope of this thesis), this experiment serves more as a proof-of-concept that the tool actually can detect different behaviour in aliasing in Java applications, and therefore is operational and handy, rather than an experiment that yields results from which definite conclusions can be drawn.

1.4 Related Work

In previous work, multiple methods of aliasing analysis have been presented. Among them are snapshot analysis, static analysis and trace-based analysis. All of them are useful in different ways and have their advantages and disadvantages.

1.4.1 Snapshot Analysis

When performing snapshot analysis, the researchers collect "snapshots" of the program state (such as the heap) at different times during an execution of it, and further investigate how objects are aliased by performing offline examinations on the data, letting the collection of snapshots represent the full execution of the program. By looking at each allocated memory location and counting the number of references to it, this method provides good overall measurements of aliasing. However, the results of the analysis might be inaccurate depending on the timing and frequency of the examined snapshots.

Potanin, Noble and Biddle developed a tool, called Fox [2], to analyse object ownership in Java programs using a snapshot analysis method. Along with the tool, they present
the Fox Query Language (FQL), a domain-specific query language that allows users to filter and find information about objects and their properties.

1.4.2 Static Analysis

Static analysis in this context typically refers to the technique of analysing program source code and inferring aliasing behaviour of the program, by exploring all possible code paths. A correct implementation of this method is able to guarantee certain properties of a program and can therefore sometimes be more useful than other dynamic approaches. However, there are limitations of static analysis as well. For example, it is hard to infer how a program written in a language of a more dynamic nature is going to behave during execution, which restricts what a static analysis can actually confirm.

Woo, Gaudiot and Wendelborn introduce a static analysis algorithm for analysing aliasing in Java applications, in their 2004 article "Alias Analysis in Java with Reference-Set Representation for High-Performance Computing" [7]. They extend an existing algorithm analysing aliasing in C++ applications and apply it to Java programs.

1.4.3 Trace-Based Analysis

When using trace-based analysis, researchers trace various events during the execution of a program to be able to establish facts about the aliasing behaviour of the program. Much like snapshot analysis, this is a dynamic approach and has some of the same advantages and disadvantages, such as how extensive the trace logs are and the level of detail in them.

Brandauer and Wrigstad presented a web service, called Spencer [6], that performs trace-based heap analysis on a large dataset of Java programs in different categories of functionality. They also implemented a query language allowing users to build composite queries able to cover information about the heap as a whole or even particular properties about specific objects during the execution of a program. Spencer is closely related to the tool developed in this project (see Section 1.4.5 and 1.5.2).

1.4.4 Other Work

In 1991, Hogg addresses alias control by introducing the concept of islands in his article "Islands: Aliasing Protection In Object-Oriented Languages" [8]. An island is a group of objects that may keep aliases between themselves in any way, but are not allowed to be aliased from any object outside of the group, except by a certain bridge object, through which all external references to the objects within the island must go. Aliasing is controlled through a "destructive read" operation, meaning that as soon as an object is aliased, its previous alias is removed, retaining reference uniqueness. The concept of islands introduces a new, more isolated approach to alias control.

In their article "The Geneva Convention On The Treatment of Object Aliasing" [1] from
1992, Hogg et al. clarify the meaning and consequences of aliasing in object-oriented programming. They suggest ways of detecting, declaring, preventing and controlling aliasing.

In 1997, in his article "Balloon Types: Controlling Sharing of State in Data Types" [4], Almeida presents the concept of balloon types as a syntactical construct part of object-oriented programming languages. He argues that the definition of encapsulation often is too weak as it allows for external objects to have references into state of other objects that are supposed to have its state only reachable by public methods. Balloon types offer programmers the possibility of declaring their custom types as "truly" encapsulated; external objects cannot keep references into the state of any balloon object, much like the kind of encapsulation primitive types such as integers, floats, and strings provide.

1.4.5 Spencer

The work most closely related to my project, is that of Brandauer and Wrigstad and their development of the Spencer web service [6], presented in 2017. The web service allows users to specify flexible queries part of a domain-specific query language against a large set of Java program trace data. The query language is written in a very modular way, making it possible to combine multiple subqueries into advanced composite queries, that can be customised to find very specific bits of information about objects and references between them.

The article [6] presented by Brandauer and Wrigstad along with their implementation of the Spencer service, is one of the articles upon which Brandauer’s doctoral dissertation [9] is based. In his dissertation, published 2018, he discusses alias control through reference uniqueness (an object may only be referenced by one other object at the same time), encapsulation (restricted reachability of data contained in an object), immutability (object is not modifiable through the reference) and more. He explores previous work and results within the field of aliasing analysis, specifically regarding the mentioned concepts, in an attempt to find out how common they are in practice.

Originally, my work was supposed to be involved with that of Brandauer and Wrigstad, but eventually I ended up developing a tool that, although similar to Spencer in many ways, approaches aliasing analysis differently (see Section 1.5.2).

1.5 Thesis History

This project is about the analysis of data generated by the bachelor thesis work A Tool For Analysis of Aliases Inside the Java VM [10] by Erik Samuelsson. Erik developed a tool that extracts information about how a program works in Java, by examining the instructions passed through the Java Virtual Machine [11] (JVM) during the execution of the program.
1.5.1 The Java Virtual Machine

The JVM is responsible for the execution of a program written in the Java programming language. Much like many other programming languages, a Java program’s source code needs to be compiled into machine instructions before it can be executed. However, instead of machine instructions native to the machine on which it is compiled, Java programs are compiled into bytecode instructions [12], executable by the JVM. The compiled bytecode contains a sequence of instructions part of a large instruction set, where each instruction is represented by an operation identifier, or opcode, and a list of operands [13].

The JVM is then responsible for executing the correct machine instructions native to the current machine, corresponding to the bytecode [11]. This enables mobility and makes it easy to write cross-platform applications, at the cost of introducing a bit of complexity into the execution of a program, which may have an impact on the program performance.

The JVM allows for instrumentation tools, or agents, to be attached to it during its execution [14]. This makes it possible to collect information about a running program, such as the executed bytecode instructions, which is integral in the implementation of this project.

The tool that Erik developed is a JVM agent that collects information necessary for trace-based analysis of aliasing, presenting the logs in a structured output file easy to analyse. Detailed data such as when an object is created, referenced, accessed or removed is saved in a large output file.

1.5.2 Relation to Spencer

During my work on this thesis, Brandauer and Wrigstad were my supervisor and reviewer, respectively. The original intention was for my work to be part of their development of the Spencer service, but in the end I decided to develop a tool aiming for different purposes than that of Spencer. Although they both perform trace-based aliasing analysis in similar ways (simulating the object graph throughout a program execution and performing analysis on objects using a comparable query language), I consider the two following main concepts to differ between Spencer and my tool.

Trace Data The Spencer service provides a prespecified base of programs from which to analyse traces. My tool allows, or rather requires, researchers to provide their own trace data, which they can obtain using Erik’s tool by attaching his Java agent to the JVM during an execution of their desired program. However, as Brandauer and Wrigstad point out [6], the Spencer service is an open-source application under the MIT license, so users can download and customise their own version of it, making it possible to specify their own trace data. The difference is that my tool provides this input modularity as an out-of-the-box feature.
**Extensibility**  Spencer provides a fixed domain-specific query language to be used, and while it is flexible and allows for advanced query compositions, my tool provides a query language that is designed and supposed to be extended. Once again, although the code base for Spencer is provided publicly under the MIT license, allowing for extension and own specialisation as well, my tool is written with the purpose of further implementation in mind.

2 Method

Achieving the goals of the project involves understanding the output trace logs of Erik’s tool and developing a tool able to parse through them and structure the essential data in them in a way such that a researcher can request particular pieces of information from it. The ability for researchers to perform such requests involves the development of an intuitive domain-specific query language which, to retain the property of extensibility, needs to be implemented in a modular way. The structure model and query language are introduced in this section and their implementations are explained in detail under Section 3.

2.1 Trace Logs

The output trace logs of Erik’s tool come in the form of a text file in which each line represents certain events during the execution of the monitored Java program. Each line consists of an integer followed by a number of parameters (integers or strings) separated by space characters, where the initial integer is the identifier of an event and the following parameters represent object identifiers or class names. Table 1 contains a summary of each event in the documentation [15].

2.1.1 Inconsistencies

Through different test executions of Erik’s tool, certain anomalies and inconsistencies was unexpectedly observed when comparing example programs and their output trace logs, which had to be taken into account when implementing the analyser.

Sometimes, object IDs appear as 0 as part of FLOAD, FSTORE, MCALL, MEXIT and VSTORE events. This mostly depends on the objects being null or of primitive types (such as `int`, `float` and `char`). These objects are simply ignored by the analyser, as null or immutable objects are of little interest when analysing aliasing. However, it seems FLOAD events in Erik’s tool reports objects loaded from static fields as having the ID 0, even if they are allocated objects assigned actual IDs. Since it is impossible to determine whether a field is static from the trace logs, the analyser simply cannot do anything but to ignore the event in this case. Fortunately, the FLOAD event is one of the less interesting events as it does not alter the object graph in any way. Hence, this does not pose much of a problem for the accuracy of the results.
<table>
<thead>
<tr>
<th>ID</th>
<th>Name/Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Allocation/ALLOC</td>
<td>The event triggered when a new object is allocated. The event ID is followed by parameters for the ID of the allocated object, the type of the allocated object and the ID of the object or class performing the allocation.</td>
</tr>
<tr>
<td>2</td>
<td>Field Load/FLOAD</td>
<td>The event triggered when an object is loaded/used from a field of another object or class. The event ID is followed by parameters for the field name, ID of the loaded object, ID of the object or class performing the load and ID of the object owning the field.</td>
</tr>
<tr>
<td>3</td>
<td>Field Store/FSTORE</td>
<td>The event triggered when an object is stored in a field of another object or class. The event ID is followed by parameters for the field name, ID of the stored object, ID of the object stored in the field before, ID of the object or class performing the store and ID of the object owning the field.</td>
</tr>
<tr>
<td>4</td>
<td>Method Call/MCALL</td>
<td>The event triggered when a method is called. The event ID is followed by parameters for the method name, ID of the object or class calling the method, ID of the object or class owning the method, and a range of zero or more identifiers representing the objects passed as arguments to the method.</td>
</tr>
<tr>
<td>5</td>
<td>Deallocation/DEALLOC</td>
<td>The event triggered when an object is garbage collected. The event ID is followed by a parameter representing the ID of the deallocated object.</td>
</tr>
<tr>
<td>6</td>
<td>Method Exit/MEXIT</td>
<td>The event triggered when a method is exited. The event ID is followed by parameters for the method name, ID of the returned object (0 if void), ID of the object or class originally calling the method, ID of the object or class owning the method and a range of zero or more identifiers representing the objects going out of scope after the method exit.</td>
</tr>
<tr>
<td>7</td>
<td>Variable Store/VSTORE</td>
<td>The event triggered when an object is assigned to a local variable. The event ID is followed by parameters for the ID of the stored object, ID of the object previously assigned to the variable and ID of the object or class performing the assignment.</td>
</tr>
</tbody>
</table>

Table 1: A summary of each possible event in the output of Erik’s tool.
String objects that are not explicitly allocated using the new keyword will have their object ID appear as 0 in most events, which is fine as strings are immutable and not that interesting when analysing aliasing, so they are simply ignored by the analyser. However, as part of MCALL events, they will actually have IDs assigned to them. This needs to be taken into account by the analyser, as one would expect an object to be allocated using the ALLOC event before using it.

Autoboxing refers to the automatic conversion between primitive types and their corresponding wrapper classes (e.g. int and Integer, float and Float, char and Character, etc.) [16]. This conversion occurs for example when an int value is passed as argument to a method expecting an Integer object. The conversion is done by an implicit call to the wrapper type’s valueOf method, which creates an instance of the wrapper class based on a primitive value [16]. For some wrapper classes, the valueOf method also implements caching such that if the provided primitive value has been converted before, a reference to the cached wrapper object will be returned instead of allocating a new object [17]. FSTORE and VSTORE events record when an autoboxed object is stored in a field or variable, but the ID of the stored object will be 0. This poses no problem for the same reason as explained above; wrapper objects of primitive types are immutable and of little interest to the aliasing analysis. However, if the autoboxed object is fetched from an autobox cache, the stored object’s ID will be an actual ID, corresponding to the wrapper object in the cache. As no ALLOC events are recorded for the creation of autoboxed objects, these objects will appear to be part of the object graph even though they actually are not, which has to be taken into account by the analyser.

It also seems that upon allocating a new object, the actual allocation event (ALLOC) is preceded by the call to the constructor method of the object’s class (MCALL/MEXIT). Hence, the object’s ID appears before it actually exists, which of course has to be taken into account by the analyser as well.

So far, based on the observations of when objects in the trace logs appear with ID 0, they are either primitive, null, autoboxed or loaded from a static field. In all of those cases, the objects are either immutable or part of an FLOAD event and hence not of particular interest to the aliasing analysis. Therefore, objects with ID 0 are simply ignored by the analyser.

Furthermore, based on the observations of when objects in the trace logs appear with an actual object ID when they actually should not, they are either string objects, autoboxed objects or objects in the process of being constructed. Strings and autoboxed objects are of little interest to the analysis since they both are immutable, but objects having their impending ALLOC event waiting should not be ignored. Because it is impossible to distinguish these kinds of objects from each other using the trace logs alone, the analyser simply adds previously unseen objects to the model. String and autoboxed objects will not affect the model or the analysis much and objects in the process of being constructed will be succeeded by a corresponding ALLOC event that
will define their type, which the analyser can add to the model at that point.

Finally, even though the inconsistencies explained above were resolved with decent methods, they complicated the implementation of the tool and may have an impact on its performance and ability to correctly present the aliasing behaviour of analysed programs.

2.2 Object Graph Simulation

To be able to follow the lifetimes of individual objects (from allocation to deallocation) during the execution of a program, and when and how they obtain references to other objects or get referenced by other objects, the tool in this project simulates the program execution regarding object behaviour by creating and maintaining an abstract object graph. The tool parses the trace data output from Erik’s tool and processes it sequentially. Each time an event is registered that regards object allocations, references or deallocations, the abstract object graph is modified accordingly. Figure 2.1 illustrates an object graph simulation example.

2.3 The Model

The model is the part of the tool that performs the object graph simulation. It is responsible for representing the state of the object graph at all times, corresponding to the trace logs. To achieve this, the model exposes functionality that lets the user create, read, update and delete objects and references between them. The model differentiates between "heap references" and "stack references". Heap references, $A \rightarrow_h B$, are established between object $A$ (outgoing) and $B$ (incoming) upon an FSTORE event, when object $A$ owns the field and object $B$ is the object assigned to it. Stack references, $A \rightarrow_s B$, are established between object $A$ (outgoing) and $B$ (incoming) upon a VSTORE event, when object $A$ owns the local variable and object $B$ is the object assigned to it. The fact the the model exposes functionality for treating specific objects' stack and heap references separately enhances the tool’s versatility.

The model also makes it possible to attach queries (from the tool’s query language) to particular objects. The queries can then probe the object graph throughout the simulated execution of the program. When the simulation is finished, the query results can be gathered and analysed.

2.4 The Query Language

The query language is the part of the tool that enables the user to acquire information about particular objects during a program execution. It presents a base query, the Observe query, and a set of advanced queries that are used in a composite and recursive fashion, but that all finally rely on one or more Observe queries. A list of all available queries and their descriptions is found in Table 2.
<table>
<thead>
<tr>
<th>Trace Data</th>
<th>Object Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Trace Data" /></td>
<td><img src="image2.png" alt="Object Graph" /></td>
</tr>
</tbody>
</table>

Figure 2.1: An illustration of an object graph simulation according to the specified example of trace data excerpt.
<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observe</td>
<td>The base of the query language. Is associated with a model instance and a specific object part of the model. Checks whether the model and object are in a certain condition. Enters an accepting or non-accepting state depending on the result of that test.</td>
</tr>
<tr>
<td>Not</td>
<td>Is in an accepting state when its subquery is in a non-accepting state. Can never be frozen.</td>
</tr>
<tr>
<td>Ever</td>
<td>Enters an accepting state if its subquery enters an accepting state. Becomes frozen as soon as this happens. Remains in a non-accepting state until then.</td>
</tr>
<tr>
<td>Always</td>
<td>Enters a non-accepting state when its subquery enters a non-accepting state. Becomes frozen as soon as this happens. Remains in an accepting state until then.</td>
</tr>
<tr>
<td>Any</td>
<td>Is in an accepting state when any of its subqueries is in an accepting state. Becomes frozen if any of its subqueries becomes frozen in an accepting state, or if all of its subqueries become frozen in non-accepting states.</td>
</tr>
<tr>
<td>All</td>
<td>Is in an accepting state when all of its subqueries are in accepting states. Becomes frozen if all of its subqueries become frozen in accepting states, or if any of its subqueries becomes frozen in a non-accepting state.</td>
</tr>
<tr>
<td>Immediately</td>
<td>Is in an accepting state when its subquery is in an accepting state. Runs only one time. Becomes frozen as soon as it is run.</td>
</tr>
</tbody>
</table>

Table 2: A list of all queries in the query language, and their descriptions.

The Observe query acts as the foundation of the query language. It is associated with a model instance and a specific object part of the model. The Observe query is used to check whether or not the model and associated object are in a certain condition. When run, the Observe query subsequently enters an either accepting or non-accepting state.

Advanced queries are based on one or more subqueries, which eventually rely on one or more Observe queries. For example, a query checking whether an object always remains unaliased (having at most one incoming reference) can be constructed as `Always(Not(Observe("Object has > 1 incoming references")))` (in pseudo-code). The accepting state of an advanced query depends on the accepting state of its subquery/subqueries.

Some advanced queries can end up in a "frozen" state. If a query checks for a certain condition such that its accepting or non-accepting state can never change once entered, the query can be declared frozen. For example, the Always query remains in an
accepting state as long as its wrapped subquery remains in an accepting state. If the subquery ever enters a non-accepting state, the *Always* query will be stuck in a non-accepting state, since the inherent meaning of it does not allow for a single occasion in which the subquery may enter such a condition. As a result, the *Always* query can be declared frozen in this case.

The fact that a query object is associated with an object in the model, means that the query object follows the object’s lifetime throughout the program execution simulation. When the execution terminates, the result of the query is represented by the state (accepting/non-accepting) of the query object.

The selection of queries provided with this tool can be considered limited, with only six advanced queries; however, it lays a foundation large enough for the purposes of this project. The tool is supposed to be extended and adapted in each project employing it, and for the experimental part of this work, the provided queries are sufficient to express the specified hypotheses and achieve interesting results that can be analysed and discussed.

### 2.5 Data Gathering

There are a lot of places where one can find freely available Java applications, that could be relevant and interesting in an experiment such as this, for example at GitHub [18], where developers all over the world keep their open-source software in repositories. However, instead of having to scour the Internet for Java applications of varied categories but similar in size, one can acquire a Java benchmark suite containing just that: a number of different, reasonably sized Java applications intended for benchmarking the performance of the JVM on different systems. This project does not in itself involve any interest in performing JVM benchmarks, but the individual applications part of a benchmark suite can be extracted and executed by themselves while monitored by Erik’s tool, collecting the desired trace information. There are a lot of Java benchmark suites and other code corpora available for free use, published in the purpose of research.

In 2008, the Standard Performance Evaluation Corporation (SPEC) [19] released their first freely available Java Runtime Environment (JRE) benchmark suite [20]. Unlike their previous releases, which they split into client- and server-side versions, this suite focuses more on general purpose Java applications of core functionality, suitable for measuring basic performance in both client- and server-side systems.

Blackburn et al. published the DaCapo benchmark suite in 2006 [21]. In their article, they argue that the main benchmark providers at that time (most prominently SPEC) fail to correctly address certain parts of the JVM performance, such as compilation and garbage collection. They present the DaCapo benchmarks [22], a freely available benchmark suite containing several general purpose, open-source Java applications, in an attempt to improve upon existing suites.
In 2010, Tempero et al. presented the Qualitas Corpus [23], a comprehensive collection of 100 open-source Java applications, trying to reduce the cost of performing extensive empirical studies by providing a complete assemblage of varied programs, but with properties that make them easy to analyse and compare.

In 2019, Prokopec et al. published the Renaissance benchmark suite [24]. They argue that current popular Java benchmark suites fail to take into account applications employing concurrency and parallelism, and offer a suite that focuses on Java programs of such kind.

In the experimental part of this project, the DaCapo 9.12 benchmark suite, released in 2009, was used [22]. DaCapo’s collection of appropriately sized, general purpose Java applications was deemed fit for the purposes of this project, although other benchmark suites or code corpora would certainly have worked as well. The Qualitas Corpus consists of a vast amount of varied programs that surely would have been adequate in this experiment, but the programs are only provided in source code format, requiring the manual work of compiling each program first, before they can be run with Erik’s tool to achieve the trace data. The time assigned to this experiment was limited and since other options were available (e.g. benchmark suites), Qualitas and other source code corpora were ruled out as options. The Renaissance benchmark suite was considered lacking in variety as the majority of its applications are focused on concurrency. The SPEC suite is similar to DaCapo in many ways, but was disregarded since DaCapo was developed as an improvement over SPEC. There are undoubtedly plenty of other suites fit for this experiment, but in the end, familiarity also had an impact on the decision of settling upon DaCapo, as it has been used successfully in previous work by all of Wrigstad, Brandauer, Erik and me.

The DaCapo benchmark suite consists of the programs shown in Table 3.

Most of the applications in the suite differ a lot from the others in terms of functional purpose, making all of them suitable choices for this project’s experiment in that sense, as they all can represent distinct categories. However, the trace logs they generate should preferably be of similar size, to be able to present a fair comparison when analysing the aliasing behaviour throughout their executions. Therefore, Erik’s tool was run on all of them and the sizes of their respective outputs were compared. Three benchmark programs generated trace logs of similar size: pmd, luindex and fop. Since the interest of this thesis not necessarily lies in researching any particular categories of Java applications, but merely show results as a proof-of-concept for the implemented tool, the categories of programs can be selected arbitrarily. The categories and representative programs can be seen in Table 4.

The category names are determined by the description of the program used in each benchmark. The pmd benchmark uses the PMD program, which is described as a source code analyser that "finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth" [26]. The luindex benchmark uses
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>Simulates a number of programs run on a grid of AVR microcontrollers.</td>
</tr>
<tr>
<td>batik</td>
<td>Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.</td>
</tr>
<tr>
<td>eclipse</td>
<td>Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.</td>
</tr>
<tr>
<td>fop</td>
<td>Takes an XSL-FO file, parses it and formats it, generating a PDF file.</td>
</tr>
<tr>
<td>h2</td>
<td>Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark.</td>
</tr>
<tr>
<td>jython</td>
<td>Interprets the pybench Python benchmark.</td>
</tr>
<tr>
<td>luindex</td>
<td>Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.</td>
</tr>
<tr>
<td>lusearch</td>
<td>Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.</td>
</tr>
<tr>
<td>pmd</td>
<td>Analyzes a set of Java classes for a range of source code problems.</td>
</tr>
<tr>
<td>sunflow</td>
<td>Renders a set of images using ray tracing.</td>
</tr>
<tr>
<td>tomcat</td>
<td>Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.</td>
</tr>
<tr>
<td>tradebeans</td>
<td>Runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database.</td>
</tr>
<tr>
<td>tradesoap</td>
<td>Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in memory h2 as the underlying database.</td>
</tr>
<tr>
<td>xalan</td>
<td>Transforms XML documents into HTML.</td>
</tr>
</tbody>
</table>

Table 3: A summary of the Java benchmarks in the DaCapo 9.12 benchmark suite [25].

<table>
<thead>
<tr>
<th>Category</th>
<th>Representative program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling</td>
<td>PMD [26]</td>
</tr>
<tr>
<td>Searching</td>
<td>Apache Lucene [27]</td>
</tr>
<tr>
<td>Parsing</td>
<td>Apache FOP [28]</td>
</tr>
</tbody>
</table>

Table 4: The selected categories of Java programs for which aliasing behaviour should be examined, and the programs representing them, upon which the tool implemented in this project will actually be used.
the Apache Lucene program, which is described as "a Java library providing powerful indexing and search features, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities" [27]. The fop benchmark uses the Apache FOP program, which is described as "a print formatter driven by XSL formatting objects (XSL-FO) and an output independent formatter" and "a Java application that reads a formatting object (FO) tree and renders the resulting pages to a specified output" [28].

Of course, a wider range of representative programs, gone through a more meticulous selection process, would most likely yield better results. However, since that is not the main interest in this experiment, one program per category was deemed to suffice.

3 Tool Implementation

The tool was implemented using the Python programming language and consists of two main components, the model interface and the query language, along with the analyser main program that uses a provided model implementation and set of queries to perform the actual Java program execution simulation and data collection.

The user of the tool is expected to implement the model interface, using as foundation a data structure suitable for the purposes of their experiment. To run the analyser, they are then expected to provide an instance of the implemented model, to which a list of queries and data collectors are attached, along with a file containing the trace logs to analyse.

The input trace logs file should be generated from Erik’s tool by attaching his agent to the JVM during an execution of the program to analyse. The list of queries should be created using the query constructs part of the tool’s query language. The data collectors should be a list of functions operating on the model as a whole, collecting general information about the execution. The analyser then parses the trace logs and incrementally modifies the model to simulate the objects’ behaviours, and at the same time collects information about the execution as well as the objects and their interactions using the queries and data collectors attached to the model.

The tool's components will be explained in detail in the rest of this section.

3.1 Model Interface

The model is provided as an interface, in the form of an abstract Python class. To use the analyser tool, this interface must first be implemented. Table 5 presents and explains the methods exposed by the model interface. The analyser parses through the trace logs and modifies the model according to each event, using the exposed methods. The model is, as explained in Section 2.3, responsible for representing the simulated object graph at all times during the execution.

When creating an instance of a model, the user is expected to provide two parameters.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialises the model by storing the provided query factories and data collectors, initialising the data collectors’ output files, setting up the data structure and initialising an empty query results dictionary.</td>
</tr>
<tr>
<td>add_obj</td>
<td>Adds an object of given ID and type to the model and generates queries for it, using the query factories.</td>
</tr>
<tr>
<td>set_obj_type</td>
<td>Sets the type of an object in the model.</td>
</tr>
<tr>
<td>get_obj_type</td>
<td>Gets the type of an object in the model.</td>
</tr>
<tr>
<td>add_stack_ref</td>
<td>Adds a stack reference between two objects.</td>
</tr>
<tr>
<td>add_heap_ref</td>
<td>Adds a heap reference between two objects.</td>
</tr>
<tr>
<td>remove_obj</td>
<td>Removes an object from the model after saving its associated queries to the results dictionary.</td>
</tr>
<tr>
<td>remove_stack_ref</td>
<td>Removes a stack reference between two objects.</td>
</tr>
<tr>
<td>remove_heap_ref</td>
<td>Removes a heap reference between two objects.</td>
</tr>
<tr>
<td>has_obj</td>
<td>Checks whether an object exists in the model.</td>
</tr>
<tr>
<td>has_stack_ref</td>
<td>Checks whether there exists at least one stack reference between two objects.</td>
</tr>
<tr>
<td>has_heap_ref</td>
<td>Checks whether there exists at least one heap reference between two objects.</td>
</tr>
<tr>
<td>get_obj_ids</td>
<td>Returns a list of IDs corresponding to the currently existing objects in the model.</td>
</tr>
<tr>
<td>get_obj_queries</td>
<td>Returns a reference to the list of query instances associated with a given object.</td>
</tr>
<tr>
<td>reset_obj_queries</td>
<td>Resets the state of all queries associated with a given object.</td>
</tr>
<tr>
<td>collect_data</td>
<td>Runs all data collectors.</td>
</tr>
<tr>
<td>get_results</td>
<td>Returns a copy of the model’s results dictionary.</td>
</tr>
</tbody>
</table>

Table 5: A list of all methods exposed by the model interface, and their descriptions.

The first parameter is a list of query "factories", that is, a list of functions that when run returns an instance of a specific query. Each factory function should always return a new instance of the same query. When a new object is added to the model, the model generates a list of query instances by running each query factory, and attaches those query instances to the object.

The second parameter is a list of "data collectors", that is, a list of functions that when
run can collect arbitrary information about the model using its exposed methods. Each data collector is associated with a collection function operating on the model, and a filename pointing to the file to which the collection function’s results should be printed throughout the execution.

For each line in the trace logs, the analyser main program performs four actions:

1. Parses the line into an event.
2. Performs the appropriate modifications to the model.
3. Applies the queries of each object in the model.
4. Runs the model’s data collectors.

When an object is removed from the model, its query results should be saved in a results dictionary, mapping object IDs to query results.

As the number of objects in the model can get very large during long executions of big applications, the operation of running the data collectors and all objects’ queries for each event can take a long time, especially if there is a large number of query factories and data collectors. However, in the common case, the queries and data collectors do not need to be run for each event, as many events will not have an impact on the model (e.g. FLOAD). Therefore, when running the analyser, the user is given the option of providing parameters ("query rate" and "collect rate") for the frequencies of how often the analyser should apply the queries and data collectors attached to the model.

The implementation of the interface should involve some kind of scalable data structure, able to keep track of the existing objects and their interactions, as it should simulate the object graph throughout the entire program.

Along with this tool comes an example implementation of the model interface, to be able to perform the experiment in this project. The example implementation uses a directed graph data structure [29], part of the networkx Python module [30]. Using this data structure, the model’s methods are implemented by adding and removing nodes and edges in the graph, corresponding to the events of objects getting allocated/deallocated and referenced/dereferenced.

### 3.2 Query Language

As presented in Section 2.4, the query language consists of a range of queries: the basic Observe query and a number of advanced queries that can be constructed in a composite fashion, but all finally relying on one or more Observe queries. The query language is implemented such that all queries implement a query interface, which exposed methods are explained in Table 6.

When an instance of the Observe query is created, it is given a name and a conditional function operating on its associated model and object. The conditional function is a
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialises the query object. In the case of an Observe query, it is given a name and a conditional test function. Otherwise, in the case of advanced queries, they are given one or more instances of other query objects, around which this new query will wrap to form a composite query.</td>
</tr>
<tr>
<td>apply</td>
<td>Applies the query on the model and object to which it is associated.</td>
</tr>
<tr>
<td>isAccepting</td>
<td>Checks whether the query is currently in an accepting state.</td>
</tr>
<tr>
<td>isFrozen</td>
<td>Checks whether the query is currently in a frozen state.</td>
</tr>
<tr>
<td>clone</td>
<td>Returns a new identical instance of the query.</td>
</tr>
<tr>
<td>toString</td>
<td>Returns the string representation of the query. In the case of the Observe query, this is the name given to it during its initialisation.</td>
</tr>
</tbody>
</table>

Table 6: A list of all methods exposed by the query interface, and their descriptions.

boolean function and determines whether or not the query is in an accepting state. The test is performed when the query’s apply method is called and the query enters an accepting state only if the test function returns true.

When an advanced query object’s apply method is run, it subsequently runs the apply method of its subquery/subqueries, which continues recursively until the final Observe query/queries are applied. The accepting state of the advanced query object is then determined by its isAccepting method, that in some way depends on the accepting state of its subquery/subqueries, according the type of the advanced query object.

Furthermore, queries ending up in frozen states can be optimised to not invoke the apply chain each time they are run. In a query’s apply method, before calling the apply method of its subquery/subqueries, it can check whether it is frozen and if so simply remain in its current state, by immediately exiting the method. The concept of the frozen state improves the performance of the analysis tool drastically.

Finally, the fact the queries are based on an interface retains the query language’s modularity and extensibility and makes it easy for researchers to augment and adjust the language for their own experimental purposes.

For reference, Listing 1 and 2 present the Python code for the Observe and Always queries respectively.
class Observe(query.Query):
    def __init__(self, name, tst):
        self.name = name
        self.tst = tst
        self.state = None

    def apply(self):
        self.state = self.tst()

    def isAccepting(self):
        return self.state

    def isFrozen(self):
        return False

    def clone(self):
        c = Observe(self.name, self.tst)
        c.state = self.state
        return c

    def toString(self):
        return self.name

Listing 1: The Python source code for the Observe query.

class Always(query.Query):
    def __init__(self, q):
        self.q = q
        self.failure = False

    def apply(self):
        if self.failure: return
        self.q.apply()
        if not self.q.isAccepting(): self.failure = True

    def isAccepting(self):
        return not self.failure

    def isFrozen(self):
        return self.failure

    def clone(self):
        c = Always(self.q.clone())
        c.failure = self.failure
        return c

    def toString(self):
        return "Always({0})".format(self.q.toString())

Listing 2: The Python source code for the Always query.
3.3 Data Collectors

The data collectors attached to the model collect information about aliasing behaviour during the program execution as a whole. The data collector functions lack the flexibility the query language offers in following the lifetimes of particular objects, but they are useful to collect information about matters such as, for instance, the average number of incoming references to any object during the entire execution, which would be hard to formulate using queries. The data collectors are attached to the model and executed by the analyser when parsing the events. The frequency of how often they are executed are, like the queries, determined by a parameter passed to the analyser.

A data collector consists of three components:

- A collection function operating on the model, returning statistics in some output format.
- A function returning a string representation of the output format of the collection function.
- A filename which the model will use to create a file, to which the model will print the string representation of the results of the collection function.

3.4 Output

The tool outputs results for both the applied queries and the data collectors.

3.4.1 Query Results

After a successful execution, the analyser gathers and returns the final results of the queries of each object. Apart from the query results, the analyser returns the execution time and execution parameters, such as query rate and collect rate.

Normally, the result of a query is determined by its state (accepting or non-accepting) at the point during the execution where it was deallocated. However, as the deallocation event of some objects are not recorded in the trace logs (usually because the JVM garbage collector did not have time to deallocate them before the termination of the program), the analyser additionally has to collect the query results of each object still remaining in the model. The results are in this case represented by the state (accepting or non-accepting) in which the objects currently are.

3.4.2 Data Collector Results

The data collector results are printed successively to their respective output files during the execution of the analyser. Each line in an output file represents the data collector’s results gathered by the analyser at a point during the execution, hence the size of the output files strictly depends on the data collection frequency of the analyser, given by the collect rate parameter passed to it.
As the user determines the output of the data collection functions, the containing format of the output files mostly depends on themselves. However, the analyser does print a number in the beginning of each line, representing the progress percentage of the execution at that point. This is useful when for example plotting the results.

### 3.5 Converting Hypotheses to Queries

To perform the experiment, the hypotheses needed to be converted into queries, and potentially data collectors. The hypotheses’ corresponding queries are shown in Table 7.

<table>
<thead>
<tr>
<th>Hyp.</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Always(Observe(&quot;Object is unaliased&quot;))</td>
</tr>
<tr>
<td>2</td>
<td>Immediately(Not(Observe(&quot;Object is built-in&quot;)))</td>
</tr>
<tr>
<td>3</td>
<td>All([Immediately(Not(Observe(&quot;Object is built-in&quot;))), Always(Observe(&quot;Object has &lt;= 4 incoming references&quot;))])</td>
</tr>
</tbody>
</table>

Table 7: The hypotheses’ corresponding queries.

The query for Hypothesis 1 will find all objects that always uphold the condition of being unaliased, meaning always having at most one incoming reference. The ratio of unaliased objects can be compared among the different categories of programs.

The query for Hypothesis 2 will find all objects that are of custom classes. The ratio of queries staying in accepting versus non-accepting state can be compared among the different categories of programs.

The query for Hypothesis 3 will find all objects that both are of custom classes and always have at most four incoming references. The ratio of queries staying in accepting versus non-accepting state can be compared among the different categories of programs.

Obviously, the `Observe` queries receive more parameters than just a name, but the more technical information is abstracted from this article, as the experiment primarily serves to show how informal phrasings can be turned into formal queries.

Additionally, two data collectors were created. One for collecting the minimum, maximum and average number of incoming references to any given object, and one for collecting the number of built-in and custom objects respectively, during all times of the execution.

### 4 Results

First, Erik’s tool was run on the DaCapo benchmark program of each selected category to receive the trace logs on which to run the analyser implemented in this project. The
file sizes of the output trace logs were 200 MiB, 600 MiB and 900 MiB for the pmd, luindex and fop programs respectively. The analyser was then run on each of the trace logs, using the queries and data collectors described in Section 3.5 along with both query and collect rate parameters set to 1000 (meaning that the analyser executes the queries and data collectors at every 1000th event). Presented in the below sections are the query and data collector results for each benchmark program.

The experiments were run on a Windows 10 computer with a Intel Core i7 4790K CPU [31] and 16 GiB of RAM [32]. During the execution of both Erik’s tool and the tool of this project, the memory limit was never exceeded, but the processor ran at 100% the entire time.

### 4.1 Query Results

Query results for each specific object were omitted here, as that kind of output is too large to fit in this article. Only the general statistics are presented.

<table>
<thead>
<tr>
<th></th>
<th>pmd</th>
<th>luindex</th>
<th>fop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace logs file size (MiB)</td>
<td>200</td>
<td>600</td>
<td>900</td>
</tr>
<tr>
<td>Query rate</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Collect rate</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Execution time (hh:mm:ss)</td>
<td>01:24:51</td>
<td>08:02:49</td>
<td>31:43:43</td>
</tr>
<tr>
<td>Objects created</td>
<td>105006</td>
<td>127449</td>
<td>574859</td>
</tr>
<tr>
<td>Objects satisfying Q1 (%)</td>
<td>79.19</td>
<td>90.58</td>
<td>77.58</td>
</tr>
<tr>
<td>Objects satisfying Q2 (%)</td>
<td>13.82</td>
<td>5.45</td>
<td>23.48</td>
</tr>
<tr>
<td>Objects satisfying Q3 (%)</td>
<td>11.99</td>
<td>0.50</td>
<td>21.20</td>
</tr>
</tbody>
</table>

Table 8: Specifications, parameters and results for each of the benchmark programs. Q1, Q2 and Q3 respectively refer to the queries that were converted from the project’s hypotheses, presented in Table 7.

### 4.2 Data Collector Results

Presented in the following sections are the output of the data collectors, plotted into images.
4.2.1 pmd

Figure 4.1: The minimum, maximum and average number of incoming references to any given object during the execution of the pmd DaCapo benchmark program (the Min curve coincides with the x-axis, remaining at 0 for the whole execution). For the y-axis, a base-10 logarithmic scale is used.

Figure 4.2: The number of built-in and custom objects during the execution of the pmd DaCapo benchmark program.
4.2.2 luindex

Figure 4.3: The minimum, maximum and average number of incoming references to any given object during the execution of the luindex DaCapo benchmark program (the Min curve coincides with the x-axis, remaining at 0 for the whole execution). For the y-axis, a base-10 logarithmic scale is used.

Figure 4.4: The number of built-in and custom objects during the execution of the luindex DaCapo benchmark program.
Figure 4.5: The minimum, maximum and average number of incoming references to any given object during the execution of the fop DaCapo benchmark program (the Min curve coincides with the x-axis, remaining at 0 for the whole execution). For the y-axis, a base-10 logarithmic scale is used.

Figure 4.6: The number of built-in and custom objects during the execution of the fop DaCapo benchmark program.
5 Analysis

In this section, the results are analysed both in terms of the general performance of the tool and if it is possible to see any relationships between aliasing behaviour and the selected categories of Java programs.

5.1 Execution Evaluation

The benchmark programs’ trace logs’ file sizes, execution times and number of objects created during the execution (see Table 8) are summarised in Table 9.

<table>
<thead>
<tr>
<th>Benchmark program</th>
<th>Logs size (MiB)</th>
<th>Execution time (hh:mm:ss)</th>
<th>Objects created</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmd</td>
<td>200</td>
<td>01:24:51</td>
<td>105006</td>
</tr>
<tr>
<td>luindex</td>
<td>600</td>
<td>08:02:49</td>
<td>127449</td>
</tr>
<tr>
<td>fop</td>
<td>900</td>
<td>31:43:43</td>
<td>574859</td>
</tr>
</tbody>
</table>

Table 9: For each benchmark program, the file sizes of the trace logs acquired from Erik’s tool, along with the execution times when run through the analyser and the number of objects created during the execution.

Based on the execution times of each program, the tool seems to scale quite poorly. The file sizes of the benchmark program’s trace logs increase in a quite linear fashion, but the analyser’s execution times for each of them increase much faster, even though they use the same query and collect rate parameters, and the same queries and data collectors. This might depend on the number of objects created increasing drastically between fop and the other two programs, but pmd and luindex are still quite similar in size and number of objects created. Nevertheless, the tool provided as part of this thesis is, as mentioned, merely supposed to be a prototype of a modular tool that should be extended in the future, and there is plenty of room for optimisation.

The reason for the poor performance might come from the fact that the graph model implemented as part of the experiment possibly relies on unoptimised data structures and methods for constructing and maintaining the object graph. Another possibility is that the computer on which the experiments were run does not have the necessary hardware specifications needed for performing the analysis of each respective program.

Better hardware specifications on the computer on which the programs were run, might have helped getting better performance, but it cannot make up for potential unoptimisations. Furthermore, the tool is currently implemented using a sequential algorithm. Introducing concurrency and parallelism can probably improve the tool’s performance considerably.
5.2 Results Discussion

In this section, all of the results from the queries and data collectors are discussed, in terms of whether the hypotheses held or not and if the information could be useful. The discussions rely on the premise of letting the benchmark programs pmd, luindex and fop be representative for the categories of "profiling", "searching" and "parsing" applications respectively, which of course probably not is enough to support the subsequent arguments. However, as mentioned before, this experiment is rather about showing that the tool has the potential of being useful in the conduct of experiments of this kind, and so the below analysis is more of an exhibition of how the tool is capable of providing results from which valuable conclusions can be drawn.

To reiterate, the hypotheses were:

Hypothesis 1. There is a considerable difference in aliasing behaviour between different kinds of Java applications.

Hypothesis 2. There is a considerable difference in the frequency of using custom versus built-in objects between different kinds of Java applications.

Hypothesis 3. Some kinds of Java applications rarely or never use more than four incoming references at the same time to any custom object during the execution.

Judging from the results, it seems the hypotheses held to a certain degree.

5.2.1 Hypothesis 1

The first hypothesis was formalised into a query looking for all objects being deallocated in a state such that they had never been aliased during their lifetime (see Table 7). Comparing the query results of the different benchmark programs pmd, luindex and fop, they all seem to exhibit an overall quite low level of aliasing (79.19 %, 90.58 % and 77.58 % unaliased objects respectively). It seems systems employing profiling and parsing functionality exercise aliasing quite sparingly, while search applications use almost no aliasing. This can be because e.g. parsing applications might apply some recursion when parsing nested structures, rendering objects related to each other. Searching algorithms probably do not involve much object interaction, but rather a lot of compare operations between a given parameter and existing data.

5.2.2 Hypothesis 2

The second hypothesis was formalised into a query looking for objects instantiated from custom classes. It seems applications in all of the categories use quite a low level of custom objects (13.82 %, 5.45 % and 23.48 % respectively for profiling, searching and parsing), all of them below 25 %. However, systems performing parsing operations seem to use a considerably larger amount of custom objects than those of profiling
functionality. Parsing applications might require a larger amount of custom classes depending on the comprehensiveness of the potential grammar able to describe the structure of the target subjects of the parser.

5.2.3 Hypothesis 3

The third hypothesis was formalised into a query looking for objects satisfying the two conditions of being both based on a custom class and having at most four incoming references. Overall, the amount of objects fulfilling those requirements was quite small for all categories, only 11.99 %, 0.50 % and 21.20 % for profiling, searching and parsing respectively. The most considerable difference is between searching and parsing applications, which are differing by about 20 %. Since one of the conditions involved that of Hypothesis 2 (objects being based on custom classes), and those results being quite low, the results of this hypothesis is expected to be quite low as well.

5.2.4 General Data

The results presented by the data collectors show how programs in the different categories behave in terms of number of built-in, custom and unaliased objects throughout the execution. It seems the average number of incoming references to any given object during the execution of all programs is close to 0 (Figure 4.1, 4.3 and 4.5), implying a generally low level of aliasing for all categories.

Interestingly, however, while searching and parsing applications have a continuously growing maximum number of incoming references to objects throughout their executions, profiling applications seem to exhibit a large number of maximum incoming references at the beginning on their execution, although quickly falling to and mostly remaining at a relatively low level. For searching and parsing applications, this might be because they generally work in a lazy fashion, only loading information when needed, hence the degree of aliasing increasing as the execution progresses (Figure 4.3 and 4.5). Profiling applications, on the other hand, might work in a more greedy fashion, where the entire source code target for the profiler is loaded into memory immediately and the data structures are set up completely at the very beginning of the program. The quick subsequent decay in the curve (Figure 4.1) might suggest that the profiler releases a lot of unused objects or references as they have served their use early in the program.

Profiling, searching and parsing applications all seem to use a similar number of built-in and custom objects during the execution (Figure 4.2, 4.4 and 4.6). The number of custom objects is generally very low compared to the number of built-in objects used, indicating that Java’s built-in classes hold a lot of responsibility in controlling the aliasing behaviour of Java applications in general. This suggests that it might be a good idea to provide some kind of annotation framework or language extension for controlling aliasing as an out-of-the-box feature in the default Java installation.
5.2.5 Summary

The results show some diversity in aliasing behaviour of different categories of Java programs. Some results exposed information that might be worth attention. Especially luindex seems interesting, with 90.58% unaliased objects (Table 8); letting that program alone be representative for the entire category of programs employing some kind of searching algorithms, it seems programs of that kind could benefit from some kind of Java annotation framework or language extension, for example letting programmers declare objects as unique [2] (allowing at most one incoming reference). Perhaps such an annotation framework or language extension could even implicitly declare objects as unique by default, requiring the programmer to explicitly declare objects as non-unique and able to be aliased when needed. This way, programmers are less prone to accidentally introduce unnecessary aliasing, that otherwise potentially more easily could cause errors as the application grows larger.

However, looking at Figure 4.4 displaying the number built-in and custom objects of the luindex program, it seems the number of built-in objects is quite small during the entire execution. This might invalidate the argument of searching applications being suitable subjects for the kind of language augmentations previously discussed.

Regarding Hypothesis 3, the results of its corresponding query is not satisfactory enough for potential garbage collection optimisation, at least when it concerns custom objects. The experiment could be repeated, removing the query’s restraint on only looking for custom objects. If the majority of all objects were to satisfy that query, it could be possible to use that information in the purpose of optimising garbage collectors implementing reference counting [5].

5.3 Threats to Validity

Issues potentially leading to inaccurate results could be possible bugs in both Erik’s tool and the analysis tool developed here, as both tools are developed as part of bachelor theses and therefore limited by the time the theses allow for implementation. Certain anomalies and inconsistencies (presented in Section 2.1.1) were found in Erik’s tool, resulting in the need of a few workarounds during the development of the tool in this project, which can have had a negative impact on both the performance of the tool and accuracy of the results.

Another possible source of error is that the queries formulated in Section 3.5 do not adequately represent the hypotheses established for the experiment. Further experimentation can be made, either trying to convert the hypotheses using other query constructs, or modifying the hypotheses slightly to make them easier to transform into queries.
6 Conclusion and Future Work

In this thesis, I have presented a tool that, along with Erik’s tool [10], is able to perform a trace-based analysis of the execution of a Java program for aliasing behaviour. The tool provides a versatile and extensible query language and a model interface, which when implemented will represent and simulate the object graph of an executed program. Trace logs representing the execution are acquired using Erik’s tool. Queries are attached to the model, collecting desired information about specific objects and their incoming and outgoing references as the analyser uses the program’s trace logs and the model to perform a simulation of the execution. The tool is developed with an approach focused on modularity and extensibility; the model and query language are provided as interfaces, such that the tool can be extended and adapted for specific purposes.

I also carried out an experiment, to show the tool’s functionality and usability. In the experiment, I used the tool to analyse Java program traces from three different program categories: profiling, searching and parsing. The results showed different aliasing behaviour among the three; for instance, parsing applications has a higher percentage of custom objects used than applications of the other two categories, and search applications seem to use almost no aliasing. Overall, for applications of all categories, it seems the number of custom objects is very low compared to the number of built-in objects. Consequently, the most beneficial approach for controlling aliasing would naturally be to provide some kind of annotation framework or language extension that lets programmers control how objects of any class can be referenced, as part of the default Java installation.

However, the accuracy of the experiment results is questionable. Inconsistencies in Erik’s tool and potential bugs in mine could have had a negative impact, but most importantly, the programs analysed were too few and can hardly be considered representative for each category. Having said that, the purpose of the experiment was just to exhibit the tool’s potential and possible value in future research within the field of aliasing analysis. Whereas the results might be of little research value, they show that the tool is able to distinguish aliasing behaviour between different kinds of Java applications and hence has the potential to be of value in further projects.

In future work, the tool can be used in real experiments with the original purpose of actually finding something presentable, that is, results valuable in the field of research. The validity of the tool’s results needs to be confirmed by testing it and it allows for plenty of optimisation. Analysing aliasing can be very influential for the future of object-oriented languages and hopefully the tool contributed in this thesis can assist in future studies within that field.
References


