Preserving Unique References in Java Lists

Daniel Wayne Smith

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science In
Computer Science

Gregory Kulczycki
Csaba Egyhazy
Bill Frakes

12/7/2010
Falls Church, VA

Keywords: unique references, alias avoidance, software engineering

Preserving Unique References in Java Lists

Daniel Smith

ABSTRACT

The Java collection framework introduces aliasing when objects are added to and accessed from collections. This thesis describes a list component implemented in Java that preserves unique references of objects in the list, thereby avoiding undesired aliasing. We compared the running time of our list with three other lists from Java collections (Java collection framework, Google, and Functional Java) in five different applications. We found that the performance of our list was usually slightly slower than the performance of the Java list, but often much faster than the Google and Functional Java lists. We also compared the reasoning complexity of our list with Java's list by creating tracing tables for a method from a towers-of-Hanoi application and comparing the number of tokens in the table using our list with the number of tokens in the table using the Java list. We found that the number of tokens in the tracing table using the Java list was much higher than the number of tokens in the table using our list. We argue that this result will occur in any table for applications that use mutable list objects.

Contents

1 Introduction 1
1.1 Contributions and Hypotheses 2
1.1.1 Hypothesis H1 (Integrity) 2
1.1.2 Hypothesis H2 (Efficiency) 2
1.1.3 Hypothesis H3 (Reasoning) 3
1.2 Thesis Outline 3
2 Motivation and Related Work 4
2.1 Models of Variables 4
2.2 Aliasing 5
2.3 Immutable Types 6
2.4 Unique References 7
2.5 Achieving Value Model in Java 9
2.6 Related Work 10
3 From Java to Unique References Lists 13
3.1 Design Principles and Assumptions 13
3.2 List Comparison 14
3.2.1 Basic Operations 15
3.2.2 Iterator Associated Operations 20
3.2.3 Collection Associated Operations 20
3.2.4 Other Operations 22
4 Performance Comparison 24
4.1 Frequently Modified List Applications 24
4.1.1 Quicksort 24
4.1.2 Towers of Hanoi 28
4.2 Stable List Applications 31
4.2.1 Data Read 31
4.2.2 Depth First Search 34
4.2.3 Topological Sort 36
4.2.4 Performance Results 38
5 List Specification Comparison 41
5.1.1 Mathematical Sequences 41
5.1.2 Java List Specification 41
5.1.3 Unique References List Specification 44
5.2 Tracing Table Comparison 45
5.2.1 partition Method 46
5.2.2 solveHanoi Method 51

5.2.3 Reasoning Results 51
6 Conclusion 53
6.1 Open Questions 54
7 Appendix 55
7.1 Java solveHanoi 55
7.2 Unique References solveHanoi 57
7.3 Towers of Hanoi Code Listing 60
7.3.1 Java Class 60
7.3.2 Unique References Class 61
7.3.3 Google Collections Class 62
7.4 Data Read Code Listing 63
7.4.1 Java Class 63
7.4.2 Unique References Class 64
7.5 Quicksort Code Listing 64
7.5.1 Java Class 65
7.5.2 Unique References Class 66
7.6 DFS Code Listing 67
7.6.1 Java Class 67
7.6.2 Unique References Class 68
7.7 Topological Sort Listing 69
7.7.1 Java Class 70
7.7.2 Unique References Class 71
8 References 73

List of Figures

Figure 1 - Models of Variables 5
Figure 2 - Aliasing 5
Figure 3 - Immutable Objects 7
Figure 4 - Assignment in UR 8
Figure 5 - Parameter Passing in UR 8
Figure 6 - Unique References 9
Figure 7 – Pre-state (before) and post-state (after) for Java method call s.add(x) 16
Figure 8 - Pre-state (before) and post-state (after) for UR method call s.add(x) 16
Figure 9 – Pre-state (before) and post-state (after) for Java method call result = s.get(2) 17
Figure 10 – Pre-state (before) and post-state (after) for Java method call result = s.set(1, x) 19
Figure 11 – Pre-state (before) and post-state (after) for UR method call result = s.set(1, x) 20
Figure 12 – Pre-state (before) and post-state (after) for Java method call s.addAll(t) 21
Figure 13 – Pre-state (before) and post-state (after) for UR method call s.addAll(t) 21
Figure 14 - Quicksort performance 25
Figure 15 - Log Scale Quicksort Performance 26
Figure 16 - Quicksort performance 26
Figure 17 - Log Scale Quicksort Performance 27
Figure 18 – Quicksort Java vs. UR boxplot 27
Figure 19 - Towers of Hanoi Performance 29
Figure 20 - Towers of Hanoi Performance 29
Figure 21 - Towers of Hanoi Java vs. UR boxplot 30
Figure 22 - Data Read Graph 31
Figure 23 - Data Read Graph sans Functional Java 32
Figure 24 - Data Read Small Elements Boxplot 32
Figure 25 - Data Read Large Elements Boxplot 33
Figure 26 - DFS Performance 34
Figure 27 - DFS 1000 Elements boxplot 35
Figure 28 - Topological Sort performance 37
Figure 29 - Topological Sort 1000 elements boxplot 37
Figure 30 – Partition tracing metrics comparison 50
Figure 31 - solveHanoi metrics comparison 51

1 Introduction

Java is among the most commonly used programming languages today. Developers that use Java enjoy the ease of programming very complex applications in a small timeframe. The portability of Java is also a major draw to the language. Java programs can be run on almost any computer system. The Java Runtime Environment (JRE) has been ported to numerous operating systems making Java a major player in disparate systems and as a communication mechanism between them.

Java is used in college classrooms, commercial applications as well as in government applications. There are free Java libraries to do many common tasks. The Java graphical components can be altered at run time to make applications look like those applications that exist natively on the target system. Java is a very versatile and useful language that will, seemingly, be around for years to come.

Java is a reference based language. References are very common in more modern languages. Language creators enjoy the performance benefits that go along with references. Performance, of course, is often a reason to choose or not choose a language for a particular application.

References are not without consequences, though. Tony Hoare commented on their pitfalls over thirty years ago: "References are like jumps, leading wildly from one part of a data structure to another. Their introduction into high-level languages is a step backward from which we may never recover."[23] These jumps Hoare mentions are not unlike goto statements. Goto statements have all but been banished from computer programs. References, however, remain in the common lexicon of developers.

The reason why the goto statement fell out of favor was the difficulty to determine the outcome of a program by looking at it. It was difficult to discern what would happen given a set of circumstances. References share this same trait. It's difficult or impossible to tell what side effects will happen in a program by calling a mutator method on an object since it's impossible to tell how many other objects contain a reference to it.

The popularity of Java and the problems references introduce motivate the research problem addressed in this thesis: How can we allay the references problems in Java applications and make them easier to reason about. In answering this question, we focus on a frequently used data structure in Java: List<T>.

Aliasing in Java is introduced most obviously with assignment, when the reference held by one variable is directly copied to another. But aliasing can also be introduced when objects are passed into a method or values are returned from a method. Both of these cases occur routinely in the Java collection framework. The call *s*.*add*(*x*) introduces aliasing between the element *x* and the collection *s*, because after the call, *s* indirectly

references the same object that is referenced by *x*. Aliasing is also introduced by the call *x* = *s.get*(*i*) to access the *i*-th element in the collection. We focus on the List component in this thesis because it is part of the Java collection framework that exhibits this aliasing behavior, and similar solutions to alias-avoidance can be applied to other collections.

1.1 Contributions and Hypotheses

The primary criticism of references is their ability to create aliases and exacerbate the difficulty of reasoning about programs [5]. To mitigate this negative aspect, we introduce a modified List component whose operations do not introduce aliasing (unlike the traditional Java List).

In order to gauge the usefulness of our list component, several applications will be written comparing programs using our List to programs using Lists from other component libraries available for Java. These programs will be compared based on runtime. In addition, we will also compare a tracing table using our list component to a similar tracing table using a Java list component.

Another key contribution of this thesis is the use of metrics on tracing tables. Metrics have been used for decades on programs but have not seen much use in analyzing reasoning information about programs. The Halstead complexity measures are used to estimate complexity of programs [25]. They base their results on the following four calculations: number of distinct operators, number of distinct operands, total number of operators and the total number of operands. It is the contention that the more moving parts there are, the more the developer needs to keep track of and the more likely they are to have a bug. In our estimation of complexity of tracing tables, we simplified the metrics to lines of code and token count.

1.1.1 Hypothesis H1 (Integrity)

The operations of our list component do not introduce aliasing when its elements are mutable objects. We demonstrate this in Chapter 3 by describing each method in our list and explaining how it preserves the property of unique references for the objects that make up the elements in the list. An object has a unique reference if it only has one reference to it at any given point in the program. Since aliasing occurs when an object is accessible - directly or indirectly - through two or more variables, an object with only one reference cannot be aliased. We only need to demonstrate this for mutable objects (objects whose state can change), because immutable objects can be viewed with a value model of variables regardless of aliasing.

1.1.2 Hypothesis H2 (Efficiency)

The efficiency of an application made with our list is relatively close to that of applications made with a Java list. We test this hypothesis empirically in Chapter 4 by comparing our list with Java's list in five different applications and measuring performance times. In our tests, we include two other lists, both from 3rd party Java collections. One is an immutable list from Google collection classes and another is a list from Functional Java. At the end of this section, we discuss how these results might generalize depending on how lists are used in a given application.

1.1.3 Hypothesis H3 (Reasoning)

The reasoning complexity of an application made with our list is significantly less than that of applications made with a Java list. In Chapter 4, we give formal specifications for both our list component and Java's list. We select a method from the Towers-of-Hanoi application used in Chapter 3 and create tracing tables for the version using our list and the version using the Java list. We compare the reasoning complexity of these tables by counting the number of lines and tokens in each table. We argue that a good estimate to the number of extra tokens in any Java tracing table can be derived from information in a comparable tracing table using our list.

1.2 Thesis Outline

Chapter 2 provides an in-depth look at the problems created by references and the way in which our Unique References approach avoids them. Chapter 3 provides a comprehensive explanation of Java's List interface and describes what must be changed in order to preserve Unique References for list elements. Chapter 4 compares the results of performance tests between our list and others. Chapter 5 gives a full formal specification of both our list and the Java list, presents two tracing tables for a central method in the Towers-of-Hanoi program (one based on our list and one based on Java's list), and compares the complexity of the tracing tables. Chapter 6 contains possible avenues for future research as well as our conclusion.

2 Motivation and Related Work

All programmers reason about their code whether they do so formally or informally. Aliasing in mutable objects complicates that reasoning. By avoiding aliases, the behavior of programs can be reasoned about more easily and writing specifications becomes less challenging. Component writers can be assured that their data will not be corrupted without their knowledge. Complex frameworks for object change notification will not be required. Programming logic becomes simplified and more straightforward [5][12][15].

The reasoning problem with aliasing can be discussed in terms of the conceptual model of how variables relate to their objects. Reasoning is simplified when a type has a value model of variables. For example, even though immutable objects may be aliased, reasoning is not complicated because conceptually their variables can be viewed as values rather than references. Objects with unique references also have variables that can be viewed as values rather than references.

In this section, we discuss the differences between a value model of variables and a reference model of variables. We illustrate how immutable objects can maintain a value model even when aliasing is present. We discuss what unique references are and give some general guidelines for the approach to unique references that our list component will adopt. And we show how uniquely referenced objects also support a value model of variables. The final part of this section dicusses related work.

2.1 Models of Variables

A programming language can have a value model of variables or a reference model of variables or a mixture of both. In a language that uses a value model of variables, variables directly represent their object. In a language that uses a reference model of variables, a variable represents a reference to its object. Both of these models have their merits as well as their drawbacks. Variables that use a value model tend to be easier to reason about. Many object-oriented languages employ a reference model of variables for objects so that certain operations can be implemented efficiently without forcing programmers to deal with the challenges of manual memory management that come with explicit pointers.

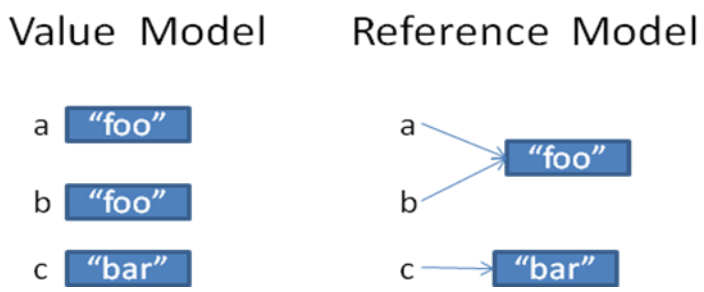


Figure 1 - Models of Variables

Figure 1 illustrates the difference between the value based and referenced based model of variables. On the left side, variables a, b and c represent the objects themselves. To assign variable b to the object c represents (e.g. b = c), the string "bar" is typically copied. On the right side, variables a, b and c represent references to their objects. Variable b can be assigned to c's object (e.g. b = c) with a simple reference copy.

2.2 Aliasing

In a reference model, the ability for multiple variables to reference (directly or indirectly) the same object is known as aliasing. Aliasing was designed with performance in mind. It has the ability to speed up many common operations. Aliasing is used with parameter passing and assignments and it's used extensively in collections. Adding and accessing elements from a Java List, for example, both create aliases.

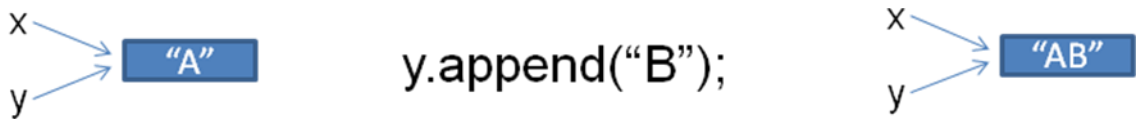


Figure 2 - Aliasing

Figure 2 illustrates what happens when two variables refer to the same StringBuffer object in Java. The left side is illustrating the state before the operation: both x and y refer to a StringBuffer that contains the string "A". The method append is called using variable passing in the string "B". The right side shows the state after the method call. After the call, both variables point to the same StringBuffer object with contents "AB". This type of behavior can lead to unintended consequences if programmers do not carefully monitor which variables point to the same object.

By itself, aliasing is not a bad thing. However, when used incorrectly aliasing can become very confusing and lead to hard to find bugs. Objects may have their member variables modified in another section of code putting them in an inconsistent state. Programmers must be cognizant of this to ensure their code can handle situations when their objects are modified via references in other areas.

Aliasing makes reasoning about object behavior more difficult. Aliasing complicates the reasoning process because "the execution of a method may change the behavior of a

seemingly uninvolved object, and this may happen without the affected object being accessed."[5] The ability to use aliased objects requires much more work to prove which variables are involved in interactions. Great lengths must be taken to ensure that only the correct objects are modified.

2.3 Immutable Types

Immutable objects are objects whose values cannot be modified. An immutable object will have the exact same state when it is garbage collected as it did when it was originally created. The canonical example of an immutable type in Java is the String class. The String class doesn't contain any methods which alter the state of the object. Any String altering method in the String class returns a new String (e.g. toLowerCase).

Joshua Bloch, author of Effective Java, recommends using immutable objects extensively. He states "classes should be immutable unless there's a very good reason to make them mutable" and that "it can be difficult or impossible to use mutable classes reliably."[6] Bloch also mentions how the simplicity of immutable objects allows a precise description of object behavior to be created making them easier to reason about.

Bloch gives the following five rules required to make a class immutable:

1. Don't provide any methods that modify the object's state
2. Ensure that the object can't be extended
3. Make all fields final
4. Make all fields private
5. Ensure exclusive access to any mutable components

By following these five rules, objects will remain unchanged after being created. They will have one state during their entire lifetime. Immutable objects also have other beneficial properties such as being thread safe. Defensive copies of immutable objects are not required. Since they cannot change state, there is no reason to be concerned of passing references to them around. Immutable objects make great building blocks when creating new objects. The only negative quality given is the necessity for having a new object for each distinct value. Requiring new objects for each step can be a performance problem in scenarios when multistep operations generate new objects at each step, and deep copying is required.

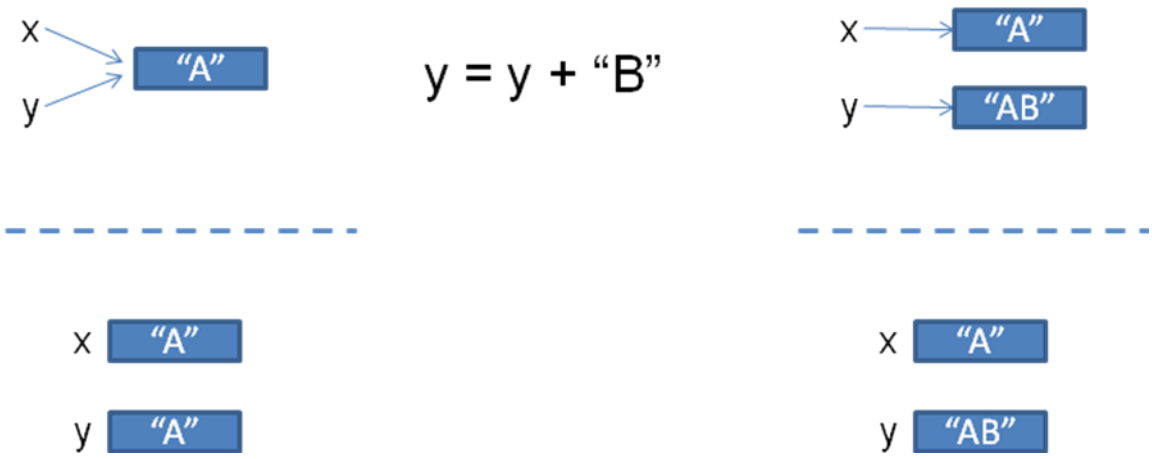


Figure 3 - Immutable Objects

Figure 3 illustrates what happens when updating an immutable object (String) in Java. The top of the figure shows an implementation view of the variables using a reference model. The bottom of the figure shows a conceptual view of the variables using a value model. On the left side, variables x and y refer to String "A". The operation 'y = y+ "B"' is executed. The right side shows the state after the operation completes. After the operation, variables x and y refer to different objects. Variable x was not impacted by the operation performed on y.

This example illustrates how immutable objects can be used in a language with a reference model of variables for objects without the same negative impact on reasoning as mutable objects. Several variables can point to the same value of an immutable object and be assured that the object will not be modified. This makes aliasing a non-issue for immutable objects. Everybody can have a reference to the same immutable object if necessary and reasoning about behavior can be accomplished with a value model of variables.

2.4 Unique References

Unique References (UR) is an alias avoidance technique that can be applied to Java eliminating aliasing. With this approach, object only have one reference to them at any given point in the program, guaranteeing that no aliasing can occur, and allowing them to be viewed using a value model of variables. Currently, maintaining unique references to an object requires a discipline on the part of a programmer. But with annotations (available in Java as of 1.5), a static checker can be created to ensure that objects of a specific type maintain only a single reference to them. The following two rules describe the approach to unique references that we used for our list component.

Main Rules for Unique References:

- Rule Number 1: If an object of a UR type is assigned to a different variable, the original variable becomes uninitialized

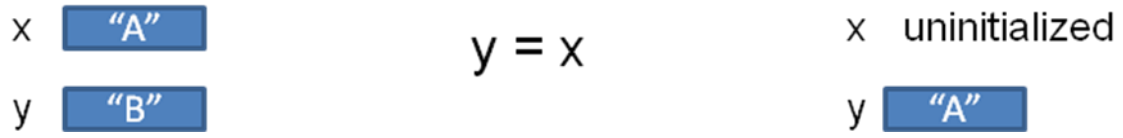


Figure 4 - Assignment in UR

Figure 4 illustrates what happens when one UR (unique reference) variable is assigned to another. On the left side, variable x represents a StringBuffer, "A", and y represents a StringBuffer, "B". The operation "y = x" is executed. The right side shows the state after the operation: y now represents the object "A" and x is uninitialized. This prevents aliasing on assignment. The variable x cannot be used again until it has been assigned. In Java, enforcing this rule would mean that the programmer must be disciplined and not use the variable x again until it is assigned a value. However, if a type StringBuffer were known to the compiler to be unique (through an annotation, for example), a static checker could ensure that the variable x was not used again before it was reinitialized by treating it exactly the same way the current Java compiler treats an uninitialized variable.

- Rule Number 2: If an object of a UR type is transferred into another object (such as a collection) the original variable becomes uninitialized

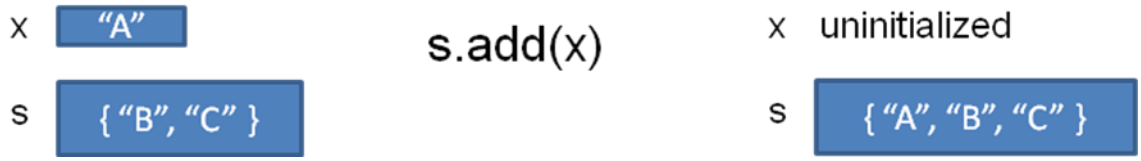


Figure 5 - Parameter Passing in UR

Figure 5 illustrates what happens in UR when an object is transferred into a method. On the left, x represents a StringBuffer, "A", and s represents a Set with the (StringBuffer) contents "B" and "C". The operation "s.add(x)" is then executed. The right side shows the state after the operation: s contains "A", "B" and "C" and the variable x is uninitialized. This prevents the variable x from being aliased to one of the elements of Set s. Note that programmers do not always want objects that are passed as parameters to be transferred to the calling object. For example, after the call to println(s), programmers would not want s to be uninitialized. Therefore an annotation is needed for those parameters that programmers want to be transferred rather than simply passed. We use the @transfer annotation to denote this in the list component.

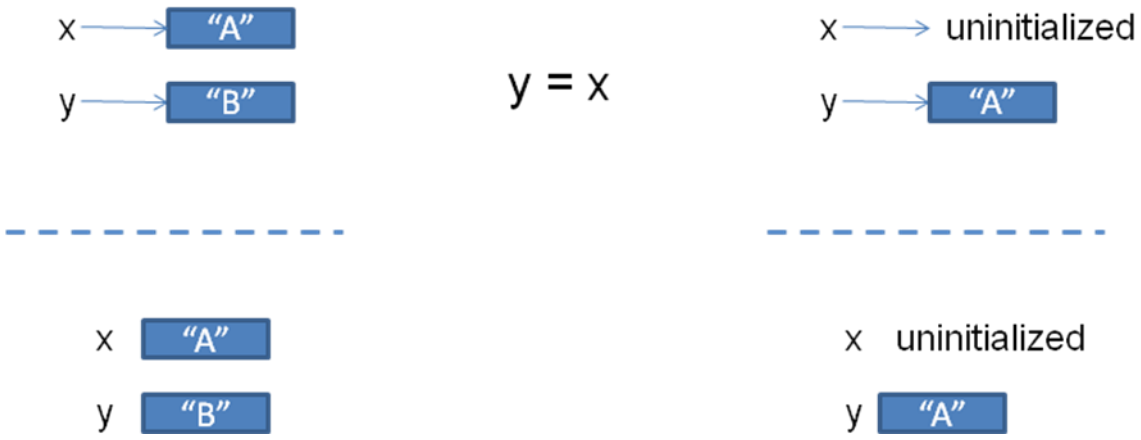


Figure 6 - Unique References

Figure 6 illustrates how one can view uniquely referenced objects with a value model of variables. The top part of the figure shows the variables as Java would represent them – using a reference model. The bottom part of the figure shows how the variables can be represented conceptually with a value model. The left hand shows the state of the objects before the assignment: x refers to "A" and y refers to "B". The operation "x = y" is performed. The right side shows the state after the operation: y refers to "A" and x is uninitialized. This shows how even though Java might use a reference model for all objects, those types that have unique references can still be reasoned about as if they used a value model.

The benefits of Unique References are that it allows easier reasoning of programs. If Unique References were implemented fully in a language it would simplify garbage collection. There will only ever be one reference to an object. Once that reference is gone, the memory can be reclaimed. Unique Reference can be used with mutable objects. The Unique References rules could be statically enforced with the Java compiler. It doesn't support this now but in theory, it could.

Using Unique References also has a few drawbacks. Since objects cannot be aliased, using containers becomes tricky. To access an element in a List, the recommended procedure would be to first swap in a null reference, inspect the element and swap the element back in. This would impair performance due to having to traverse the list twice.

2.5 Achieving Value Model in Java

To achieve a value model in Java, one of two things or a combination of the two must be done: use immutable types exclusively or use Unique References. Both of these prevent aliasing in Java.

Aside from assignment and parameter passing, aliasing can also occur when objects are returned from a method. To prevent this kind of aliasing for UR types, a static checker could be used to ensure the return statement follows the same rules as assignment statements.

2.6 Related Work

One of the works we leveraged in our research was a paper by Naftaly Minsky on how to modify Eiffel so that it supported a form of unique references [1]. Minsky provided the means for transforming the Eiffel language by adding u-variables. The u-variables he spoke of were unshareable variables. In other words, multiple references to them could not exist.

Minsky also elaborates on transfer semantics. Minsky provides a list of rules for when variables should be transferred and when they should not. As he states, there are times when variable consumption is expected and those when it is not. At times, having a variable consumed by an operation would yield unshareable variables practically useless.

Minsky's rules:

1. No transfer of regular variables into u-variables is allowed.
2. An assignments statement v := u, where u is a u-variable, is carried out as follows: first, the value of u is copied into v, then u is nullified; i.e., the value void (the null pointer of Eiffel) is stored in it.
3. The value of a formal u-parameter v declared as non-consumable cannot be changed. This entails the following constraints on the treatment of such parameters:
    a. No assignment into v is allowed.
    b. V cannot be assigned to any variable.
    c. V cannot be used as an actual argument in a procedure call if the corresponding formal parameter is declared as consumable.
4. Variables of a given class C can be declared as u-variable only if all the methods defined for this class treat their implicitly defined local variable Current as non-consumable formal parameter, satisfying the constraints of Rule 3.
5. The copying of a complete object must not be allowed to copy any u-attributes of it. Such attributes must be either moved, according to Rule 2, or not transferred at all by the copy routine.
6. A u-attribute of a class cannot be exported.
7. The result of a once function cannot be declared as unshareable.
8. Let there be a method recycle that can be applied to any u-variable u which is not an inconsumable argument of a procedure. Method recycle does nothing when u is void, and operates as follows otherwise:
    a. It applies recycle(recursively) to all u-attributes of u;
    b. It deallocates the object addressed by u, and nullifies variable u itself.
9. The recycle method introduced in Rule 8 is applied automatically as follows:
    a. Before a procedure exits all u-objects address by its local variables are recycled
    b. When an object is collected, during garbage collections, all its unshareable components are recycled.
    c. Before an assignment u := v is carried out, u is recycled.

Whereas Minsky's paper focused on changing Eiffel to support unique references, this thesis focuses on using a UR approach to create a Java list component. Our Unique

References approach implements Rule 1 in a slightly different manner. In Unique References, only certain kinds of classes can be stored and iterated through.

Our Unique References approach uses the transfer annotation for passing parameters, similar to Minsky's rule 3. A static checker for our Unique References approach would ensure that variables passed in as transfer parameters are not used prior to being assigned. Rule 4 and 5 were not addressed by our Unique References approach. Rules 6 and 7 do not apply to Java.

Minsky's work went a step further than the Unique References libraries and suggested changing the garbage collection and object reclamation strategy for Eiffel. Rules 8 and 9 had no impact on the Unique References library implementation. While these rules are good ones, they went a step beyond the scope of this paper.

Weide et al. [2] discussed the different possibilities for the assignment operator. They determined that in programming the swapping pattern is the best possible data movement operator and that assignment is the worst possible data movement operator.

When given a statement x = y, there is one possible value for x after the operation: the value of y. After the operation there are five possible values of y:

1. Y has no value, i.e., it is undefined
2. X has a unique, statically specified value of its type
3. X has some value of its type, but this is arbitrarily chosen from among a statically specified set of two or more values;
4. X has its own old value; or
5. X has y's old value

Weide et al. then compared these possibilities: efficiency, ease of management, ease of specification and reasoning and maximization of client knowledge. The swapping operator far exceeded all the other possibilities.

The swapping operator is not free from possible negative side effects. A situation exists in x = y where x is typed as a base class, and the object it references is of a subclass that is different than that of y in which swapping would be in error.

The transfer operator in our Unique References approach was used to enforce the rules of the swapping operator for parameters. A Unique References static checker would inspect methods and classes to verify that swapping occurs according to the specific pattern.

In [13], Sitaraman et al. describe why reasoning about software components is important. They describe the mathematical models that would be needed to verify the behavior of software components.

In order to prove program behavior and reason about components, the authors express the preconditions and postconditions of methods using requires and ensures clauses respectively. The requires clause defines the state of variables prior to the execution of a

method and the ensures clause defines the state of variables after the execution of a method. They define a language with enough rigor to suitably define the values of variables before and after a method has taken place. With the requires and ensures clause, what goes on inside of a method should be easily determined and how to call it should be easy to discern.

The use of tracing tables and symbolic reasoning tables were shown to define the ensures and requires clauses of the methods. A tracing table is sometimes used by people reviewing the code to determine the behavior. The tracing table has two columns: State and Facts. The State column states what operation or line in the source code is being evaluated. The facts column states the values of the variables at that point in the execution. By using the tracing table, a methods preconditions and postconditions can be verified. Tracing tables can also be used to help create ensures and requires clauses.

A symbolic reasoning table is more generalized form of a tracing table. Whereas in the tracing table, variables are specific values, the symbolic reasoning table uses names of objects. The symbolic reasoning table has four columns: State, Path Conditions, Facts and Obligations. The State and Facts columns are the same as the tracing table. The Path Conditions column defines what conditions were true to execute that statement. The Obligations column defines preconditions of the executed statements. The tracing table can prove the preconditions and postconditions for one set of input while the symbolic reasoning table can prove them for all inputs.

In section 5, we compare a tracing table written for a method that uses a Java list component with a similar tracing table written for a method that uses our UR list component. We find that the UR-based table uses less token than the Java-based table. We would expect to find similar result if we used symbolic reasoning table, but we have left this for future work.

3 From Java to Unique References Lists

In this section, we show how we constructed a List component that preserves unique references in its elements. Using the Java List interface as a starting point, we analyze each method for its effect on aliasing. If the method does not introduce aliasing, it preserves unique references, and we do not modify. If the method does introduce aliasing, it does not preserve unique references, and we discuss how to modify the method so that it does preserve unique references.

In addition to preserving unique references, we have chosen a few other minor software engineering guidelines for constructing our list. All of the principles and assumptions used are discussed in the next section.

3.1 Design Principles and Assumptions

Below are the principles we used to create the Unique References (UR) List:

- Unique Reference: Allow at most one reference to any object by avoiding the introduction of aliasing to mutable objects. By eliminating aliases, programs can be reasoned about more easily. When an alias to an object is created, that object can be modified in other parts of the code. This might lead to invalidating the state of objects that contain references to it. This confuses program logic and leads to hard to find bugs. The avoidance of aliasing is the primary objective of the Unique References list. The other three design principles are ones that simplify various aspects of the collection classes, but are not strictly necessary to avoid aliasing.

- Functions vs. Procedures: Hogg has observed that the division of methods into functions and procedures makes "object interaction more predictable" [9]. Functions are methods that do not alter the state of the object but do return a value, and procedures are methods that modify the state of the object but do not return values. An example of a function is the size method in the List interface; the insert method is an example of a procedure. The methods in the Unique References list follow this design principle.

- Copyable vs. Immutable Objects: Immutable objects are preferred. Immutable objects are thread safe and references to them can be passed anywhere without fear of negative consequences. To avoid aliasing, copyable objects must be deep copied. This will incur a performance penalty for non-trivial objects. In non-trivial objects, all of the mutable aggregate members must be deep copied; not just have their references copied. This process consumes clock cycles to allocate memory and copy memory. In principle, copyable objects could be used wherever we require immutable objects, but we are using immutable objects here for simplicity. Java has a clone method but this performs a shallow copy only, resulting in aliases for aggregate data in objects. Some deep copy libraries that exist such as http://robust-

it.co.uk/clone/, do not allow for custom implementations of copy methods. For maximum flexibility, we would allow copyable (deep-copy only) objects, and let the programmer decide whether they were willing to incur the performance penalty. In this scenario, the Unique References libraries could check not only whether the type is immutable but whether it is copyable and act appropriately for those conditions. This is still a possibility but is not currently implemented.

- Not implementing Collection: The Unique References List does not extend a base Collection interface. While this is something that could be done, we prefer to simplify our explanation of the UR List by avoiding it. For example, the add method of the Java Collection interface requires a boolean return value for Sets but not for lists. In the Java framework, the Map does not implement the Collection interface. Therefore, in the Unique References libraries, the Collection interface would only be implemented by two classes: List and Set. It was thought that a discussion of these types of issues would be a distraction to the reader and take focus away from the problem of aliasing.

3.2 List Comparison

public interface List<E> {	
//Java	//Unique References
//Basic list operations	
boolean add(Object o);	void add(@transfer Entry e);
void add(int index, E o);	void add(int index, @transfer E e);
void clear();	void clear();
E get(int index);	E get(int n); // E immutable
boolean isEmpty();	boolean isEmpty();
E remove(int index);	E remove(int index);
int size();	int size();
boolean contains(Object o);	
boolean equals(Object o);	
int hashCode();	
int indexOf(Object o);	
int lastIndexOf(Object o);	
boolean remove(Object o);	
E set(int index, E o);	E set(int index, @transfer E o);
List<E> subList(int fromIndex, int toIndex);	
//Iterator associated operations	
Iterator<E> iterator();	Iterator<E> iterator(); // E immutable

ListIterator<E> listIterator(); ListIterator<E> listIterator(int index);	
//Collection associated operations	

boolean addAll(Collection<? extends E> c);
boolean addAll(int index, Collection<? extends E> c);
boolean containsAll(Collection<?> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
	void addAll(List<E> c);

//Other operations	
Object[] toArray(); <T> T[] toArray(T[] a);	
}	

This table gives an overview of all the methods in the Java List interface as well as in the UR List interface.

3.2.1 Basic Operations

boolean add(Object o)

This method adds an element of type E to the end of the list. In accordance with the contract of the general Collection class, this method returns a boolean value, which is true if the collection changes as a result of the call. In the case of the list, if the method completes (i.e., no exceptions are thrown) this value is always true. It exists to account for the behavior of collections such as sets, where the add method can complete successfully without changing the set. For example, if you add 4 to the set {3, 4, 5}, the add method will return false because 4 is already in the set. The corresponding Unique References method does not return a value. This would violate the principle of keeping function and procedures separate. Since Unique References collection classes do not extend a base collection class, there is no need for the UR List add method to return a value.

A more serious problem with this method is that it creates an alias, violating our primary principle of alias-avoidance for the Unique References list. In Figure 7, the state of the program is depicted both before and after a call to the method s.add(x). A circle represents an objectand the angled brackets represent a list.Before the method call, x is the only reference to its object. After the call, x still references the object, but there is also a reference to the object from inside the collection.

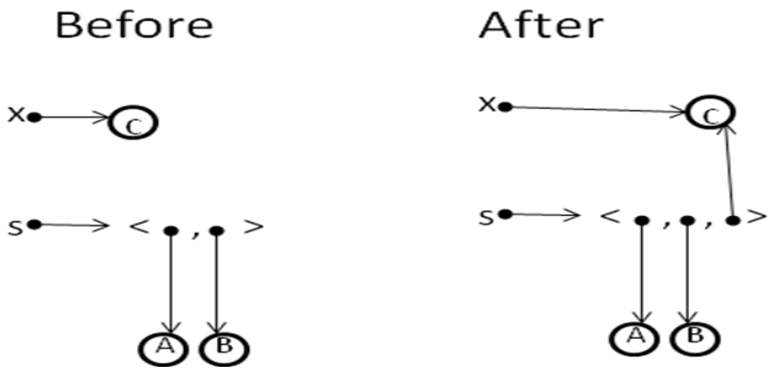


Figure 7 – Pre-state (before) and post-state (after) for Java method call s.add(x)

To avoid this problem, the Unique References add method annotates the passed parameter with a @transfer annotation. The @transfer annotation is unique to Unique References and is used specifically to help avoid aliasing by ensuring that mutable objects have only one reference to them. It indicates that after the execution of the method, the variable corresponding to the passed parameter is uninitialized, so the programmer should no longer use it. A programmer can adhere to this rule by following a discipline, or a static checker can be created to enforce it.

With the changes mentioned above, the signature of the Unique References method becomes **void** add(@transfer E o).

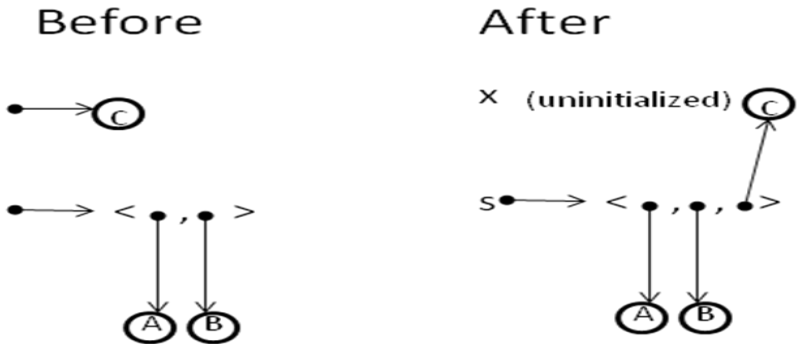


Figure 8 - Pre-state (before) and post-state (after) for UR method call s.add(x)

void add(**int** index, E o)
This method inserts an element of type E into the list at a given position. This shifts all existing elements from that position to the end by one to the right. This version of add has no return, since it is assumed to always succeed if there is no exception. If this method is called with an index that is out of range, an exception will be thrown. Like the previous add method, this method also creates an alias to the passed parameter. Therefore, in the UR List, the passed parameter is also marked with the @transfer annotation.

The signature of the Unique References method is **void** add(**int** index, @transfer E o).

void clear()

This method removes everything from the list. This method does not cause aliasing and was not changed in any way for the Unique References List.

E get(**int** index)
This method returns a handle (reference) to the element of type E at a given index in this list. Since a reference is being returned, this method causes aliasing. Figure 8 shows the state of the system before and after the statement 'result = s.get(i)' is executed. A circle represents an object, and the angled brackets represent a list. After the method executes, the variable result and the collection both contain references to the same object. Therefore, to avoid aliasing in mutable types, the Unique References version of this method requires E to be immutable.

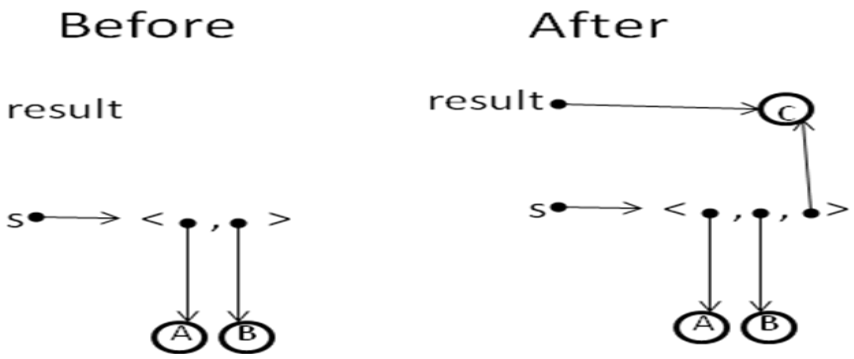


Figure 9 – Pre-state (before) and post-state (after) for Java method call result = s.get(2)

boolean isEmpty()
This method tests whether this list is empty. It is a pure function and it does not cause aliasing, so this method is unchanged in the Unique References List.

E remove(**int** index)
This method removes the element at a given position in this list. Even though this method returns a value, it also updates the current list, so it is considered a procedure (rather than a function) in Unique References. Since the object returned is removed from the list, this method does not create aliasing, so it is unchanged in the Unique References List.

int size()
This method gets the number of elements in this list. This method is a pure function and does not cause aliasing, so this method is unchanged in the Unique References List.

boolean contains(Object o)
This method tests whether this list contains a given object as one of its elements. To determine the result, the method uses the equals method on the content elements. If, for any element e in the container, e.equals(o), the method returns true, otherwise it returns false. By default in Java, the equals method checks the equality of references. References are unique in Unique References environment, so the result – assuming reference equality is checked – will always be false. For immutable objects (such as strings), the equals method is typically overridden to check for value equality. Therefore, a reasonable design decision would be to make this method available only for immutable objects, in which

case the signature would be **boolean** contains(E o). Currently, the method is unimplemented in our Unique References List.

boolean equals(Object o)
This method tests whether this list is equal to another object. This method is defined in the base Object class in Java. A List is defined to be equal to an object if that object is also a List, and the two lists have the same objects in the same order or if the two Lists are in fact references to the same List. This method suffers from the same problem as the contains method. Following the Unique References rules with Lists of mutable objects – assuming reference equality is checked – would always return false for different Lists. By using Lists of immutable objects with the Unique References rules, two different Lists could be equal. By not overriding this method, the default behavior of Java occurs. Namely, this will only return true if the parameter o is the same object as the List. This method could be implemented in a manner such that the Lists could be compared if they contained immutable objects. Currently this method is unimplemented in the Unique References List.

int hashCode()
This method returns a hash code value for an object. This value should be unique based on the value of an object. It is defined in the base class Object. Developers generally implement this method when their classes can be used as keys in hash tables. Since the List interface in Unique References defines a mutable object, these classes would be poor choices for keys since they are likely to change after being used in a hash table. This method is unimplemented in the Unique References List.

int indexOf(Object o)
This method obtains the first index at which a given object is to be found in this list. To determine if the parameter o equals an object contained in the list, the equals method is used. If, for any element in the list equals returns true, the method returns the index, otherwise it returns -1. Following the Unique References rules with Lists of mutable objects – assuming reference equality is checked – would always return -1. For immutable objects (such as strings), the equals method is typically overridden to check for value equality. Therefore, a reasonable design decision would be to make this method available only for immutable objects, in which case the signature would be **int** indexOf(E o). Currently, the method is unimplemented in the Unique References List.

int lastIndexOf(Object o)
This method obtains the last index at which a given object is to be found in this list. This method works in the same manner as the indexOf method except it starts the search for objects at the end of the list and searches toward the beginning. In a similar fashion to the indexOf method, if this method were to be used with immutable objects, the signature would be int lastIndexOf(E o). Currently, this method is unimplemented in the Unique References List.

boolean remove(Object o)

This method removes the first occurrence of an object from this list. To determine if the parameter o equals an object contained in the list, the equals method is used. If, for any element in the list equals returns true the object is removed from the list and true is returned. Otherwise, the method returns false. Following the Unique References rules with Lists of mutable objects – assuming reference equality is checked – would always return false. For immutable objects (such as strings), the equals method is typically overridden to check for value equality. Therefore, a reasonable design decision would be to make this method available only for immutable objects, in which case the signature would be boolean remove(E o). Currently, the method is unimplemented in the Unique References List.

E set(**int** index, E element)
This method replaces an element of type E in this List with another object of type E. Since Java cannot handle in-out parameters, this method returns the value that was replaced. If the given index is out of bounds, an exception is thrown. As in the add method, the main problem with this method is that it creates an alias, violating our primary principle of alias-avoidance for the Unique References List. In Figure 9, the state of the program is depicted both before and after a call to the method s.set(i, x). A circle represents an object, and the angled brackets represent a list. Before the method call, x is the only reference to its object. After the call, x still references the object, but there is also a reference to the object from inside the collection.

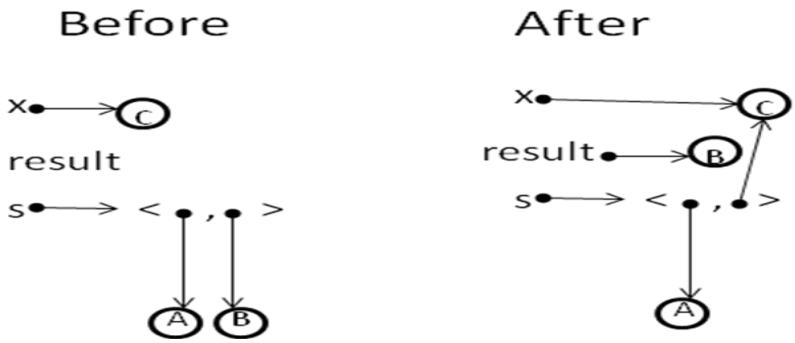


Figure 10 – Pre-state (before) and post-state (after) for Java method call result = s.set(1, x)

To avoid this problem, the Unique References add method annotates the o parameter with a @transfer annotation. The @transfer annotation is unique to Unique References and is used specifically to help avoid aliasing by ensuring that mutable objects have only one reference to them. It indicates that after the execution of the method, the variable corresponding to the o parameter is uninitialized, so the programmer should no longer use it.

The signature of the Unique References method, given changes above, is E set(**int** index, @transfer E o).

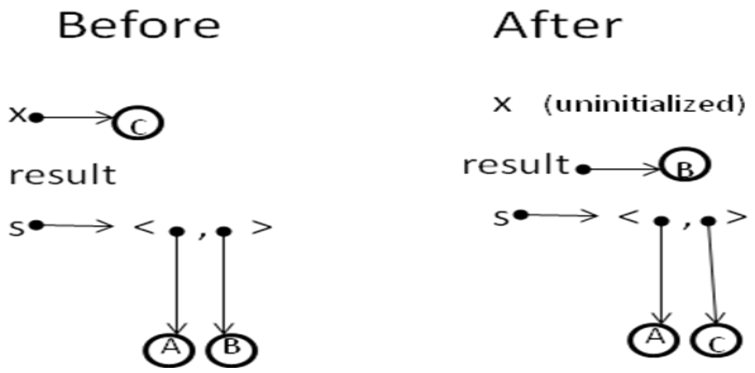


Figure 11 – Pre-state (before) and post-state (after) for UR method call result = s.set(1, x)

List subList(**int** fromIndex, **int** toIndex)
This method returns a List composed of a subsection of this List from fromIndex (inclusive) to toIndex (exclusive). In Java, this method creates a set of aliases to objects since all of the objects in the returned List also have references in this List. In the Unique References List, we require type E to be immutable to avoid aliasing of mutable objects.

3.2.2 Iterator Associated Operations

Iterator iterator()
This method obtains an Iterator over this list, whose sequence is the List order. To iterate through the elements in the List, the next() method is called which returns the next object of type E in the sequence. Iterators pose a potential aliasing problem since the objects inside the List are visible through the Iterators. To avoid the objects in the List from being aliased inappropriately, the type E must be immutable.

ListIterator listIterator()
This method obtains a ListIterator over this list, starting at the beginning. This method was not used in the Unique References List. Only the Iterator type was used.

ListIterator listIterator(int index)
This method obtains a ListIterator over this list, starting at a given position. This method was not used in the Unique References List for the same reason as the version that takes no parameters.

3.2.3 Collection Associated Operations

boolean addAll(**int** index, Collection c)
This method inserts the contents of a collection into the list at a given position. The List shifts all elements from that position to the end to the right by the number of elements inserted. In accordance with the contract of the general Collection class, this method returns a boolean value, which is true if the collection changes as a result of the call. In the case of the list, if the method completes (i.e., no exceptions are thrown) this value is always true. It exists to account for the behavior of collections such as sets, where the add method can complete successfully without changing the set. For example, if you add all of {4} to the set {3, 4, 5}, the add method will return false, because 4 is already in the set. The corresponding Unique References method does not return a value. This would violate the principle of keeping functions and procedures separate. Also, since the Unique

References List do not extend a base collection class, there is no need for the List add method to return a value.

A more serious problem with this method is that it creates aliases, violating our primary principle of alias-avoidance for the Unique References classes. In Figure 10, the state of the programming is depicted both before and after a call to the method s.addAll(t). A circle represents an object, and the angled brackets represent the list. Before the method call, t contains the only reference to the object that is contained within it. After the call, t still contains the reference to the object, but there is also a reference to the object from inside the List, s.

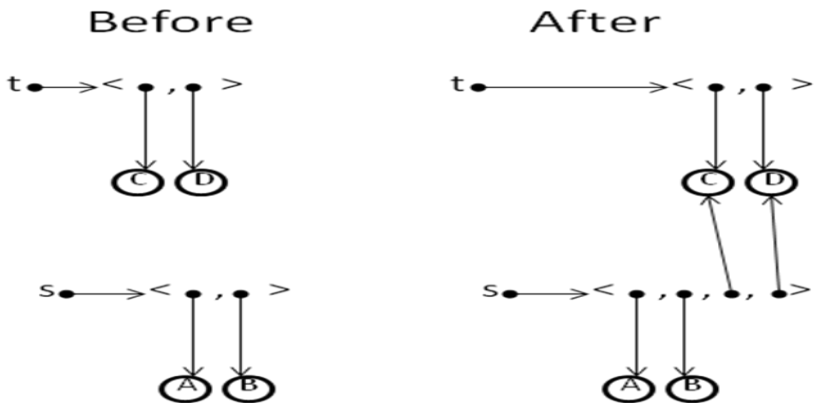


Figure 12 – Pre-state (before) and post-state (after) for Java method call s.addAll(t)

To avoid this problem, the Unique References addAll method clears the Collection c before returning control to the caller. This prevents any aliasing. Another reasonable design decision would be to prevent this method from being used with mutable elements.

With the changes mentioned above, the signature of the Unique References method is **void** addAll(**int** index, Collection c).

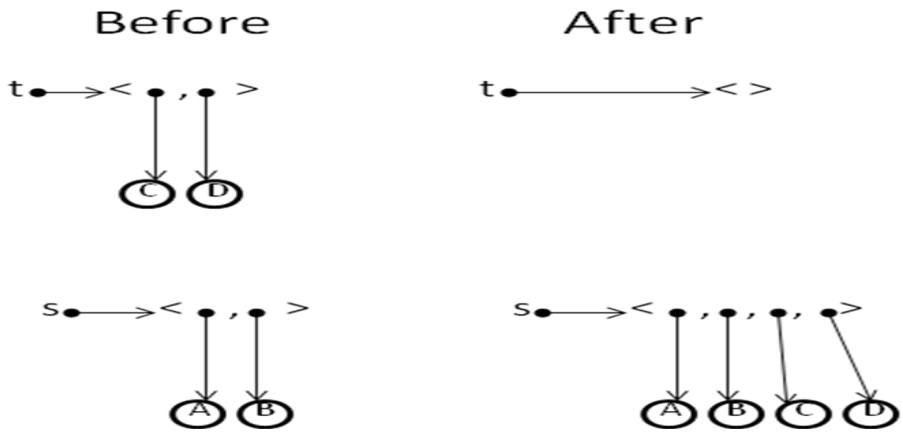


Figure 13 – Pre-state (before) and post-state (after) for UR method call s.addAll(t)

boolean addAll(Collection c)

This method adds the contents of a collection to the end of the list. In Java, this method creates aliases just as the version with the index does. The same changes and design considerations are used with this method.

With these changes, the signature of the Unique References method is
void addAll(Collection c).

boolean containsAll(Collection c)
This method tests whether this List contains every element in the Collection passed in. If every Object in the Collection was contained somewhere within the list, true is returned. Otherwise, it returns false. The obvious way to implement this method would be to iterate over each Object in the Collection and call contains(Object o). Since the Unique References List did not implement the single object version, it also did not implement the Collection version. Currently, this method is unimplemented in the Unique References List.

boolean removeAll(Collection c)
This method removes all elements of the Collection from this List. The method returns true if the List was changed. That is to say that if at least one item was removed, it returns true. Otherwise, it returns false. The obvious way to implement this method would be to iterate over each item in the Collection and call remove(Object o). Since the Unique References List did not implement the single object version, it also did not implement the Collection version. Currently, this method is unimplemented in the Unique References List.

boolean retainAll(Collection c)
This method removes all elements of this list that are not contained in a given collection. The return value will be true if the Collection has changed. That is to say that if the method removes any items in the List it will return true. Otherwise, it will return false. To determine if the List contains a given Object, the method uses the equals method of the Object. If, for any element of the Collection, c, the method returns true, the Object is retrained in the List. By default in Java, the equals method checks the equality of references. In Unique References, if the rules allow us to avoid aliasing by enforcing mutable objects to have unique reference, the result – assuming reference equality is checked – will always be true and the List will be cleared. For immutable objects (such as strings), the equals method is typically overridden to check for value equality. Therefore, a reasonable design decision would be to make this method available only for immutable objects. Currently, the method is unimplemented in the Unique References List.

3.2.4 Other Operations

Object[] toArray()
This method copies the current contents of this List into an array. This method creates an alias for every object contained in the List. This problem could be avoided by ensuring that E is an immutable type. Another possible solution would be to clear the items out of this List before return the array. Currently, this method is unimplemented in the Unique References List.

Object[] toArray(Object[] a)
This method copies the current contents of this List into the array passed in as a parameter. This method works in the same manner as the parameterless version works. Currently, this method is unimplemented in the Unique References List.

4 Performance Comparison

We compared the Unique References List to the Java List, an Immutable List from the Google Collections library and a List from the Functional Java library. We compared their performance as well as reasoning information. This section describes the performance comparison. Chapter 5 describes the reasoning comparison.

To compare the Unique References containers to those in Java, programs were written which accomplish common tasks and exercise common methods in the List interface. The following programs were created: Towers of Hanoi, Quicksort, DataRead, Depth First Search, and Topological Sort.

An interesting thing we learned doing our tests is that the compilation of byte code to machine code is not negligible. There is a clear difference in performance when this is occurring. To eliminate this, we always completed an extra series of tests at the beginning of our test suite and simply disregarded them. This ensured that all the test runs were done with byte code that had already been compiled to machine code.

The tests were run using Java 6 on a Windows XP machine with a Dual Core Intel Processor running at 2.93 GHz with 3 GB of ram. To measure the time System.nanotime() was used which "returns the current value of the most precise available system timer, in nanoseconds"[24].
Frequently Modified List Applications

4.1 Frequently Modified List Applications

We came up with two categories of applications to compare the performance. The first category of applications is frequently modified list applications. In these applications several inserts and removes are done on the lists throughout the application.

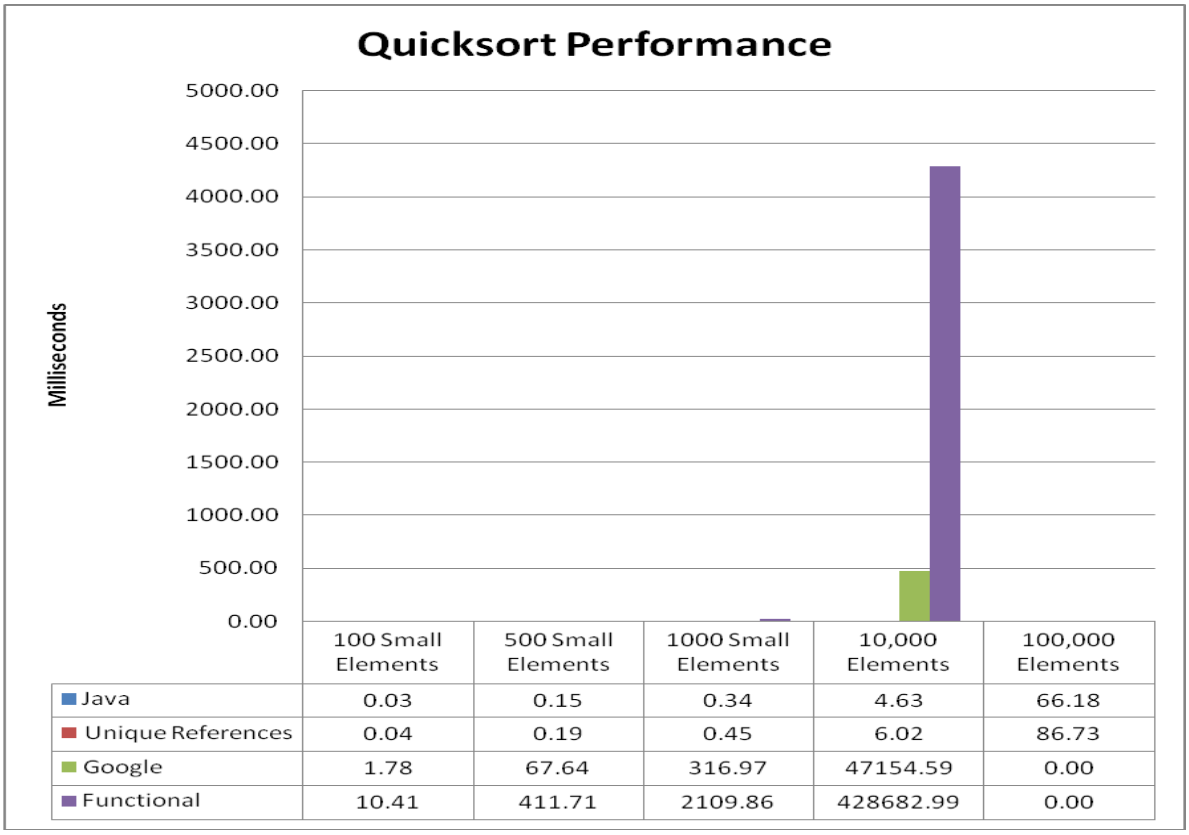
4.1.1 Quicksort

The Quicksort program used a list to sort a collection of objects. The Quicksort algorithm used was from Cormen[7]. The Wikipedia definition of Quicksort:

> *Quicksort is a sorting algorithm developed by C. A. R. Hoare that, on average, makes O(nlogn) (big O notation) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though if implemented correctly this behavior is rare. Typically, quicksort is significantly faster in practice than other O(nlogn) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data, it is possible to make design choices that minimize the probability of requiring quadratic time. Additionally, quicksort tends to make excellent usage of the memory hierarchy, taking perfect advantage of virtual memory and available caches. Coupled with the fact that quicksort is an in-place sort and uses no temporary memory, it is very well suited to modern computer architectures.*

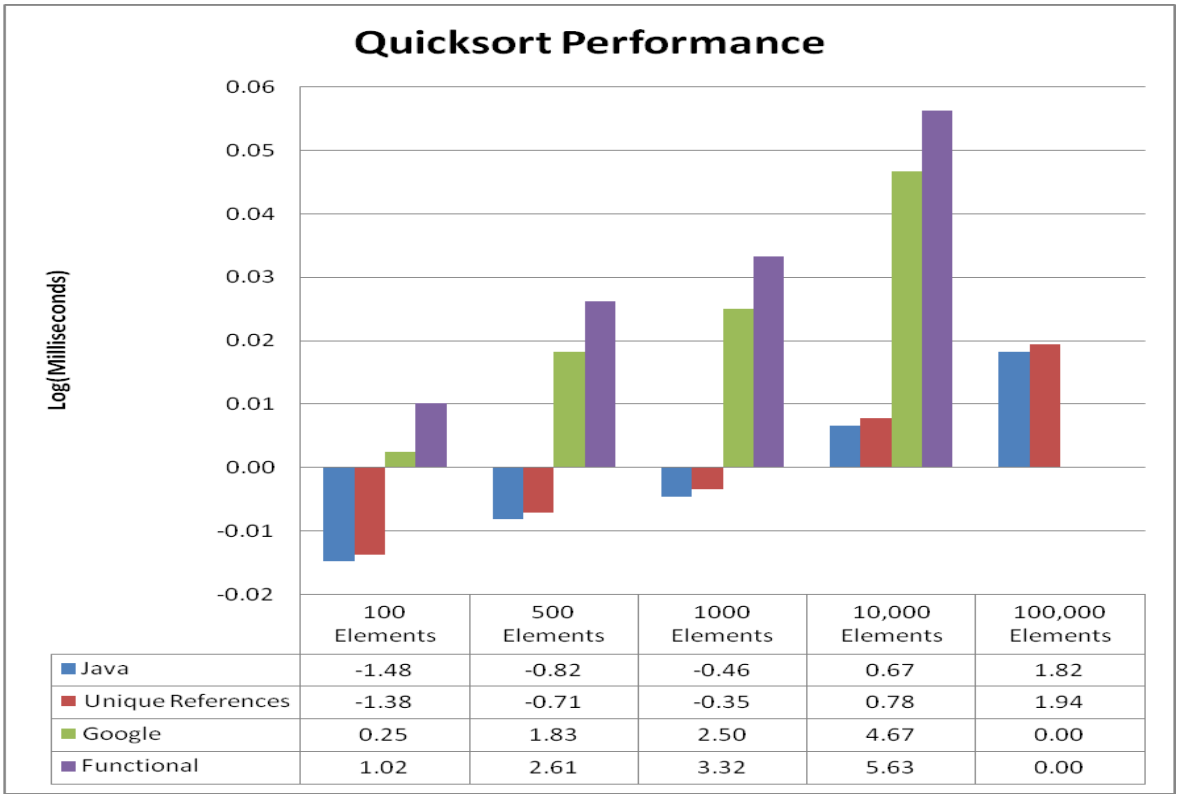
The collections sorted were 100, 500, and 1000 elements in length. Our program used lists to store the objects being sorted. The objects being sorted were Element objects we created. The Element objects contained integer data that was used to determine the sorting order.

Ten runs were done for each setup. The results of these runs were averaged to give the following graph.



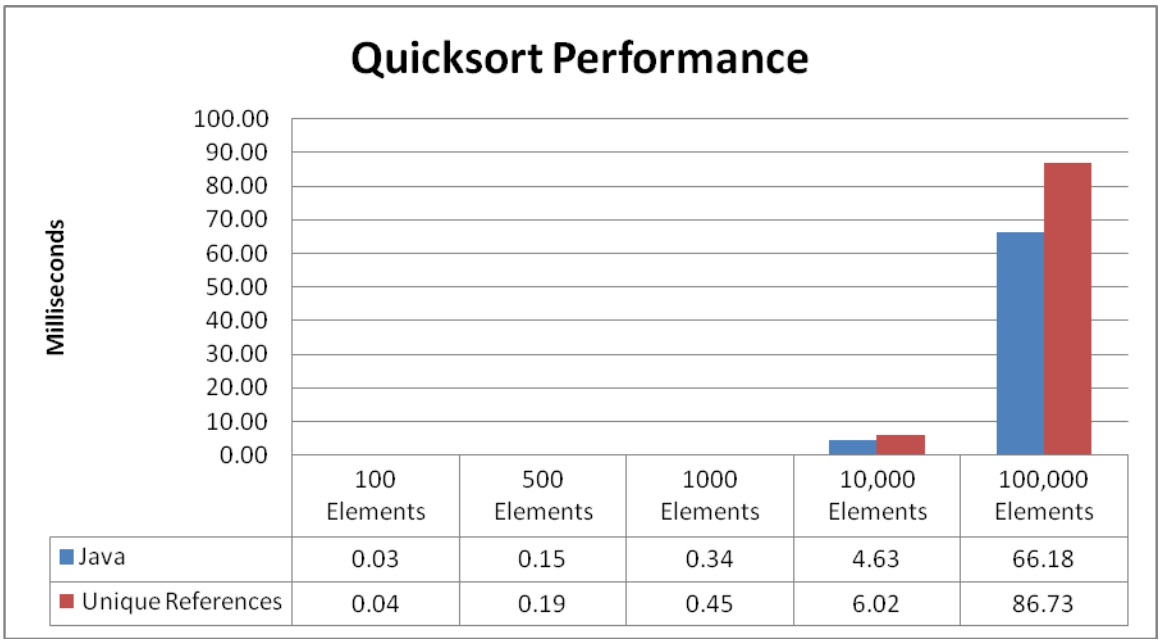
	100 Small Elements	500 Small Elements	1000 Small Elements	10,000 Elements	100,000 Elements
Java	0.03	0.15	0.34	4.63	66.18
Unique References	0.04	0.19	0.45	6.02	86.73
Google	1.78	67.64	316.97	47154.59	0.00
Functional	10.41	411.71	2109.86	428682.99	0.00

Figure 14 - Quicksort performance



	100 Elements	500 Elements	1000 Elements	10,000 Elements	100,000 Elements
Java	-1.48	-0.82	-0.46	0.67	1.82
Unique References	-1.38	-0.71	-0.35	0.78	1.94
Google	0.25	1.83	2.50	4.67	0.00
Functional	1.02	2.61	3.32	5.63	0.00

Figure 15 - Log Scale Quicksort Performance



	100 Elements	500 Elements	1000 Elements	10,000 Elements	100,000 Elements
Java	0.03	0.15	0.34	4.63	66.18
Unique References	0.04	0.19	0.45	6.02	86.73

Figure 16 - Quicksort performance

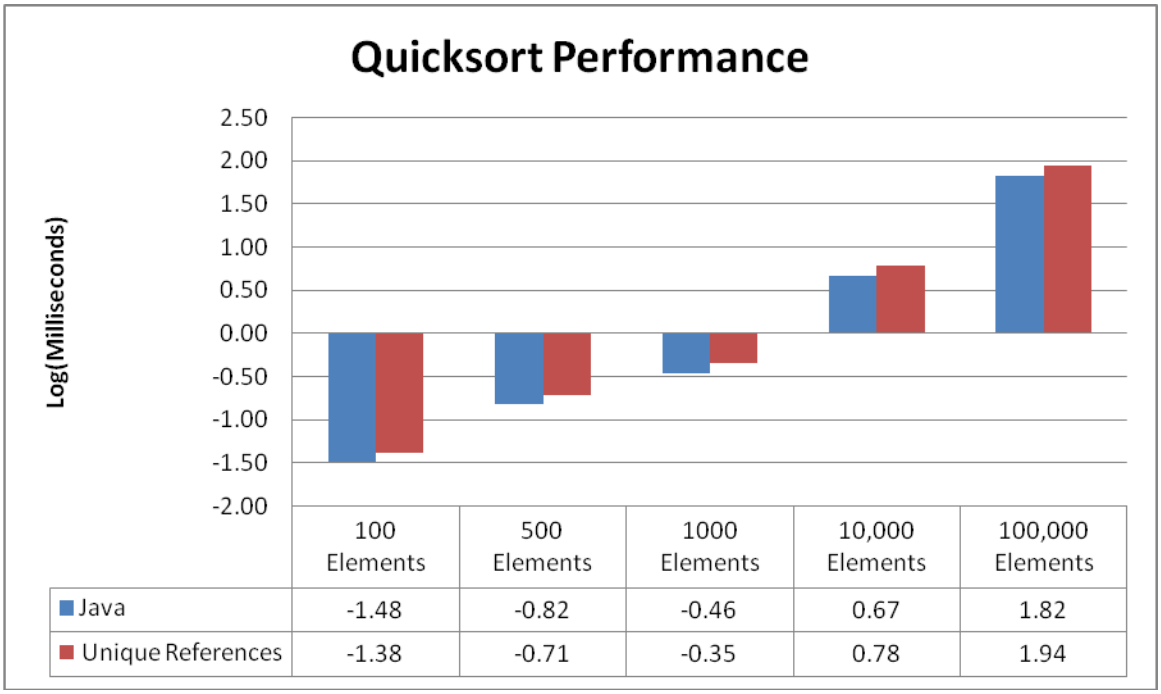


Figure 17 - Log Scale Quicksort Performance

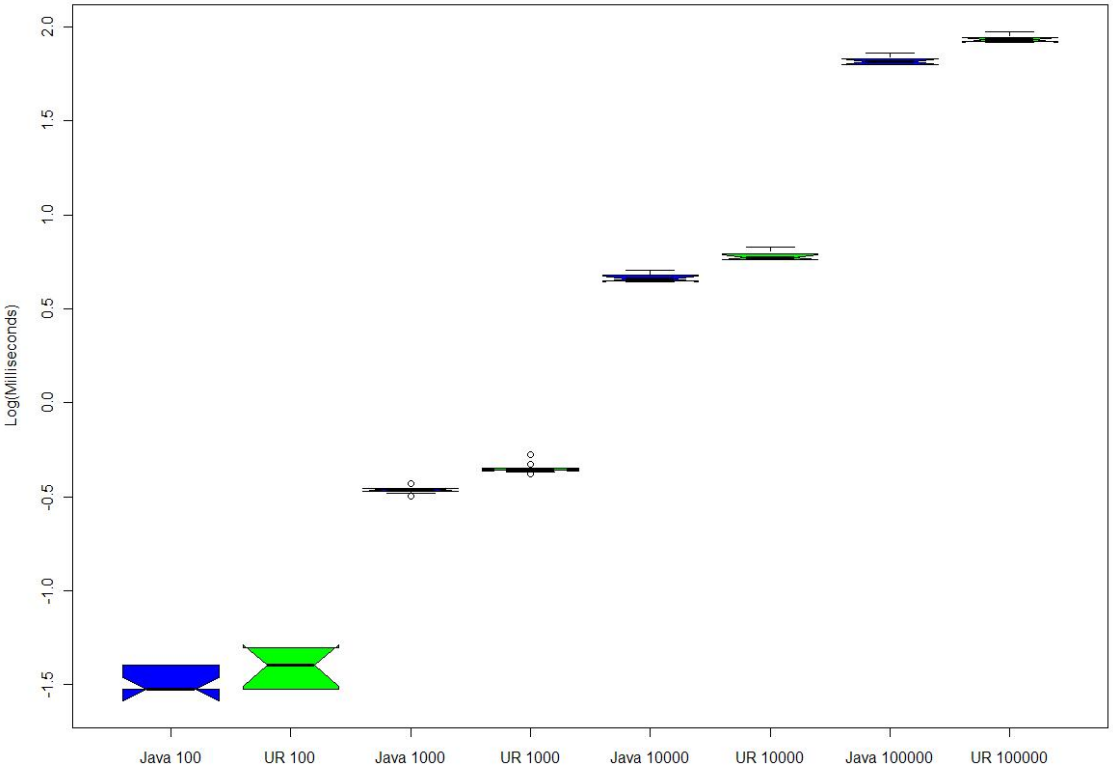


Figure 18 – Quicksort Java vs. UR boxplot

The unique references containers were approximately 30% slower while the Functional and Google lists performed far worse. The reason for the poor performance was the nature of the lists and the nature of the application. The Quicksort application does many inserts and removes from the list. The Google ImmutableList and the Functional Java list do not perform well in these scenarios. Each time an element is added or removed from one of these lists, a copy has to be made of every element in the list. The Java and UR lists do not share this characteristic.

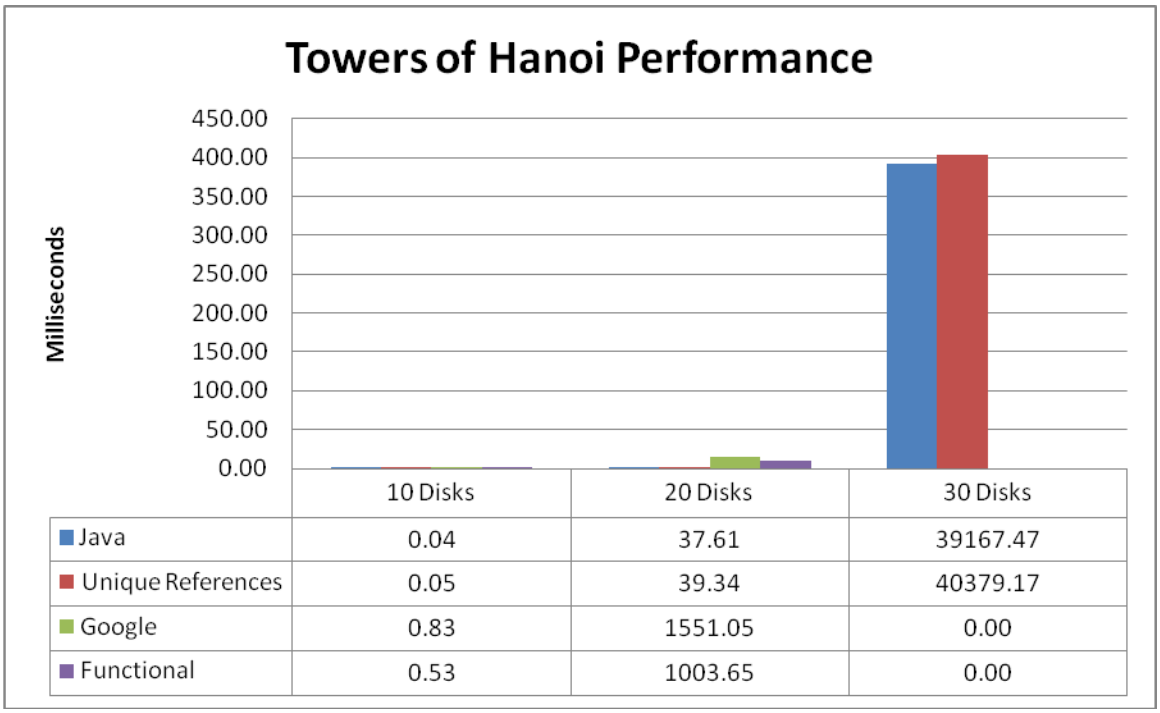
The distributions came out a little varied and there is no clear explanation for this either. The Google list might have been optimizing its data structure every time an item was retrieved from it, somehow caching the results. The outliers might have come from a garbage collection cycle being trigger or might have been impacted negatively by an operating system operation. Another reason for the outliers could have been the data. A completely new random list of objects was created then passed to each Quicksorter search object. It is possible that some of these cases were more expensive than others.

4.1.2 Towers of Hanoi

The towers of Hanoi program solved the classic towers of Hanoi problem. The Wikipedia definition of the towers of Hanoi problem:

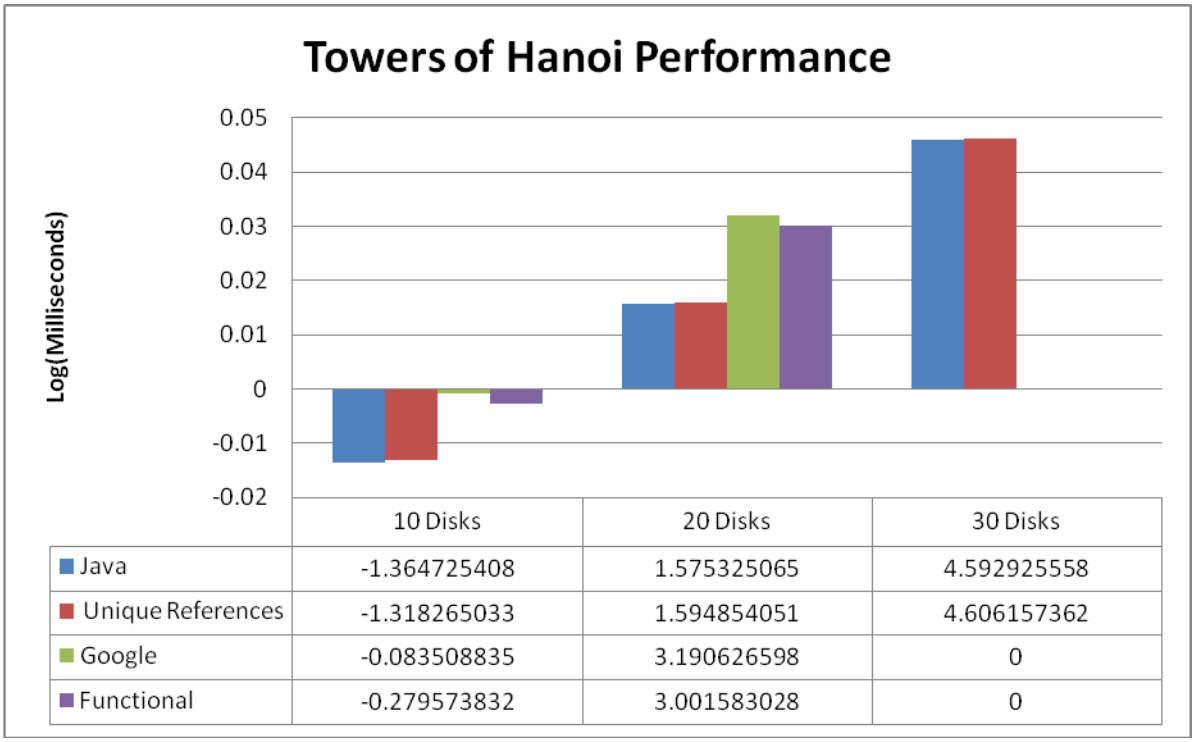
> *[Towers of Hanoi] consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.*
>
> *The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:*
>
> - *Only one disk may be moved at a time.*
> - *Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.*
> - *No disk may be placed on top of a smaller disk.*

Our program modeled the rods as Lists and the disks as Disk objects. This application does many inserts and removes from the Lists as it moves Disks from one rod to another. Ten runs were done for each setup. Google and Functional took prohibitively long to execute the 30 disk runs. The results of these runs were averaged to give the following graph.



	10 Disks	20 Disks	30 Disks
Java	0.04	37.61	39167.47
Unique References	0.05	39.34	40379.17
Google	0.83	1551.05	0.00
Functional	0.53	1003.65	0.00

Figure 19 - Towers of Hanoi Performance



	10 Disks	20 Disks	30 Disks
Java	-1.364725408	1.575325065	4.592925558
Unique References	-1.318265033	1.594854051	4.606157362
Google	-0.083508835	3.190626598	0
Functional	-0.279573832	3.001583028	0

Figure 20 - Towers of Hanoi Performance

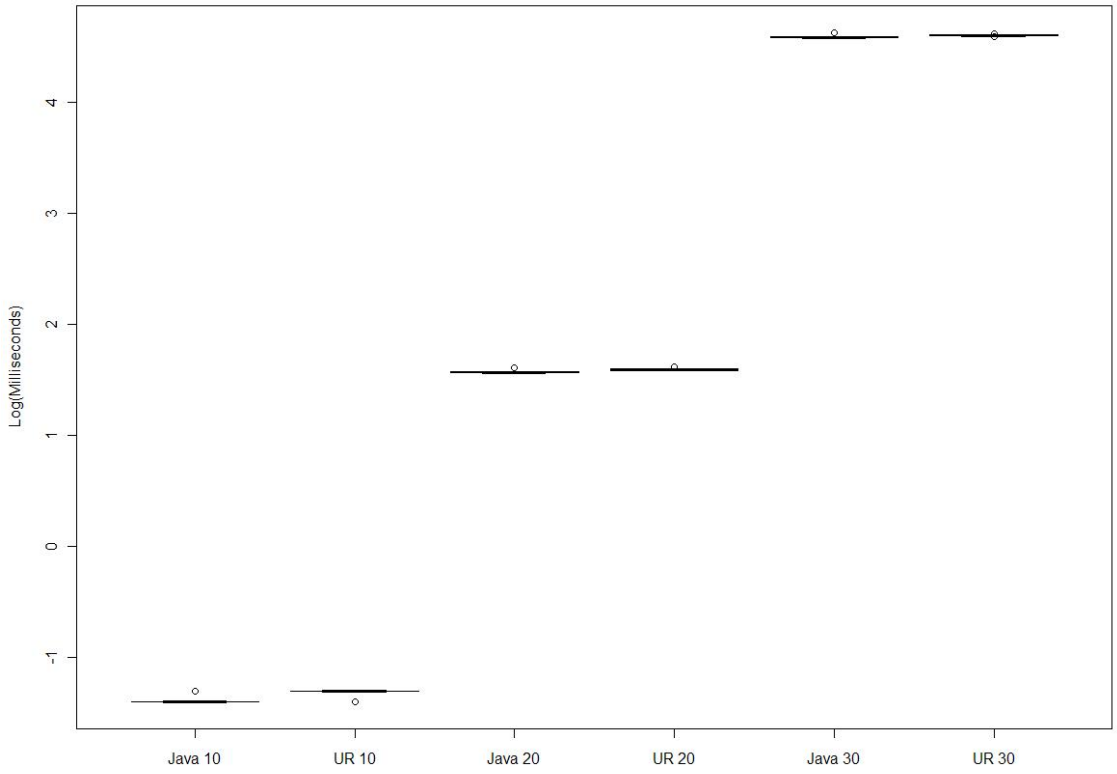


Figure 21 - Towers of Hanoi Java vs. UR boxplot

Our results showed that the Unique References list fared very well against the Java list. The time required was approximately 5% longer for the Unique References list in comparison to the Java list. The same cannot be said of the Functional list and the Google Collections immutable list. Their poor performance is due to the nature of their lists. To add or remove an element for a Google Immutable List or a Functional Java list, a new list has to be created and the references to each of the elements in the list have to be copied into the new list.

The reason why the UR List performed so well in Towers of Hanoi is the nature of the application. In this application, the Disks are removed from the List every time. There are no swaps in and out. It is simply removing an item from one list and inserting into another. Therefore the number of operations more closely resembles that of Java. This application is comparing the insert and remove operations of the Java and UR List.

In the Towers of Hanoi, the Functional List outperformed the Google List. This is in contrast with the Quicksort application that had the Google List outperforming the Functional List. While the reason for this is uncertain, it could potentially be related to the fact that in the Towers of Hanoi application, all of the inserts and removes from the List are done at the top whereas in Quicksort, elements are added and removed from the middle of the lists.

The distributions came out a little varied and there is no clear explanation for this either. The Google list might have been optimizing its data structure every time an item was retrieved from it, somehow caching the results. The outliers might have come from a garbage collection cycle being trigger or might have been impacted negatively by an operating system operation.

4.2 Stable List Applications

The second category of applications we chose to compare the performance one was stable list applications. In these applications, the lists are created at the beginning of the application and remain the same throughout the application.

4.2.1 Data Read

The Data Read application was used initially to test out various implementations of the Unique References list. In this application, a list of 50 lists of 50 elements was created. Then 10,000 random reads were done picking one of the 50 lists and one of 50 the elements in the list to inspect. The Data Read application elements allowed the size of their internal data to be set upon creation. Ten runs were completed for each of the different list types with small elements and then again with large elements. The small vs. large elements comparison was done in order to determine if any deep copying was being done on the elements.

Mutable objects were used as the elements in the List. By using mutable objects, the Unique References List was forced to swap in a null value to retrieve an element then later swap the element back in.

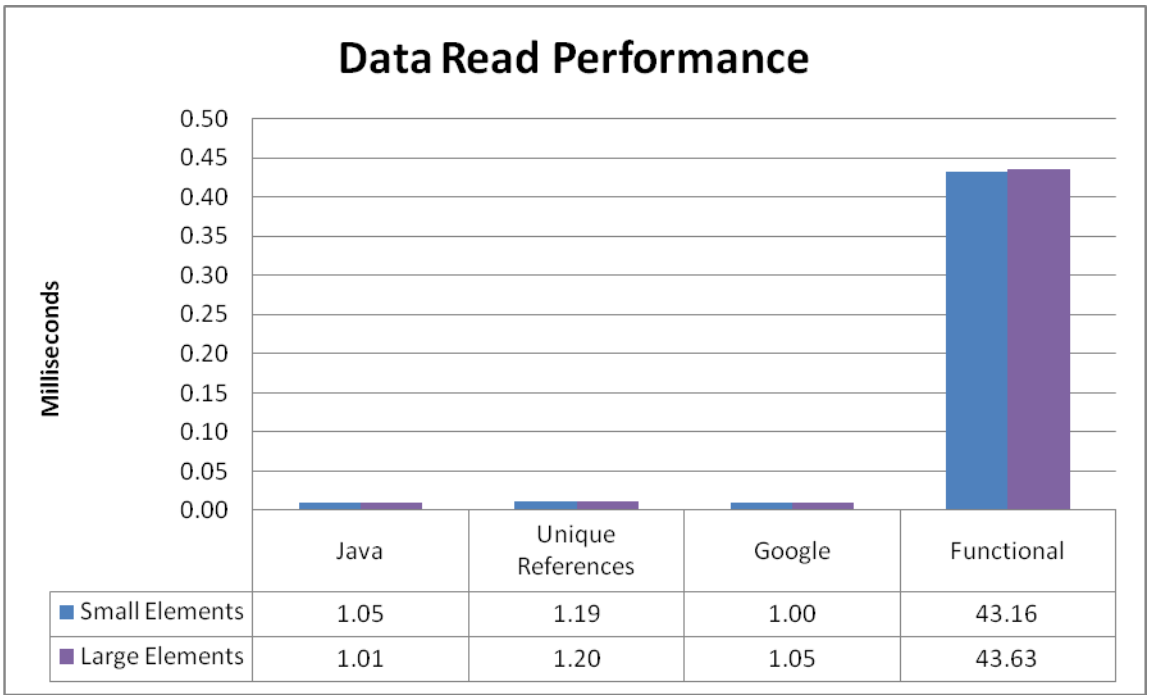


Figure 22 - Data Read Graph

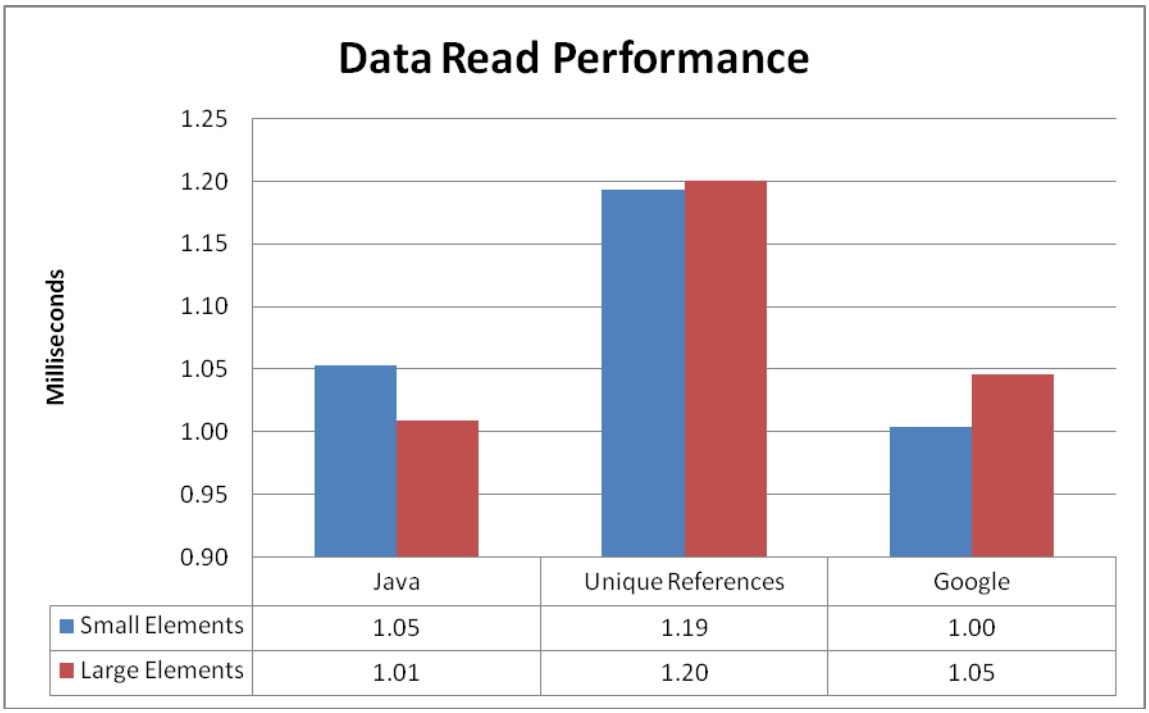


Figure 23 - Data Read Graph sans Functional Java

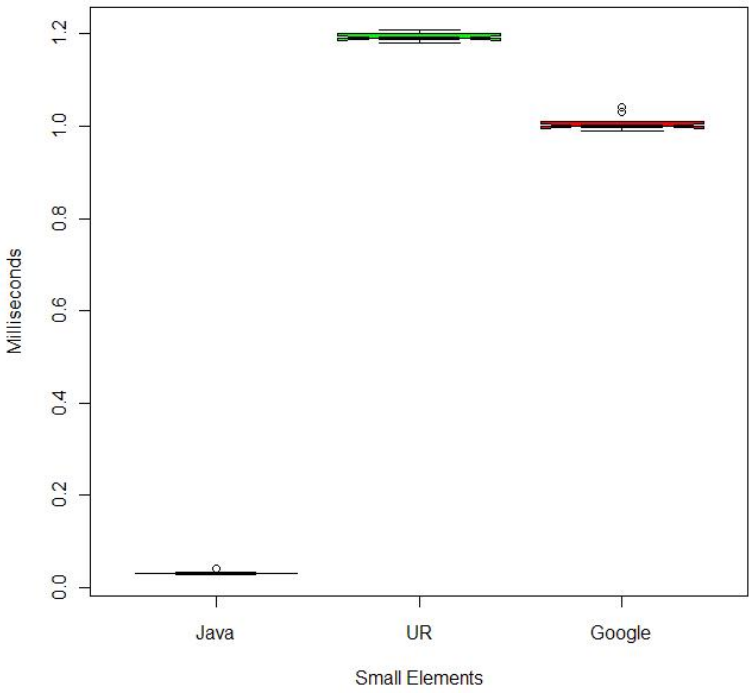


Figure 24 - Data Read Small Elements Boxplot

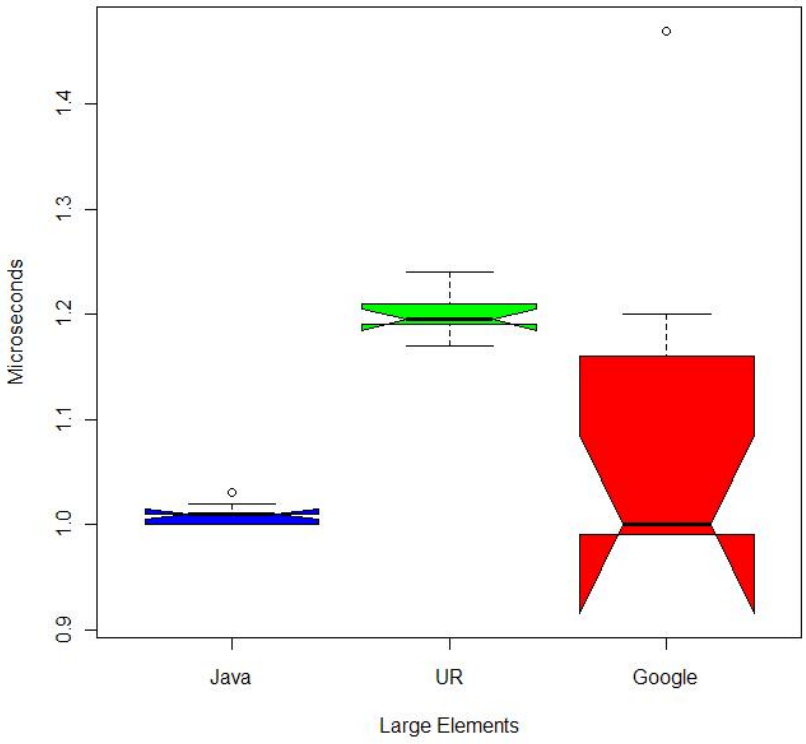


Figure 25 - Data Read Large Elements Boxplot

The results of Data Read showed that the runs with UR were within 2 microseconds for 10,000 element reads. This does equate to an 18% longer run time for the Unique References List. Considering the UR List had to do two operations compared to one in Java (get vs. swapping null in and out), this is very good. The Google Collections list performed very well. The Functional Java list was quite terrible in comparison to the other three. There is no indication why the Functional List is so slow. The presumption is that the library is simply slower due to the functional nature it's trying to display. These tests also revealed that there was no deep copying going on since the runs with large elements were actually faster than the runs with small elements. The slight edge in performance with large elements is believed to be attributed to the continued optimization of the code by the Java Virtual Machine.

It is unclear why the run time of Java went down slightly for Large Elements while the run time of Google went up slightly. It could have to do with the underlying data structure (e.g. Array or Linked List). It might also have to do with fragmented memory on the heap. The UR List uses the Java List as its underlying structure. This is why it also went up.

The distributions came out a little varied and there is no clear explanation for this either. The Google list might have been optimizing its data structure every time an item was

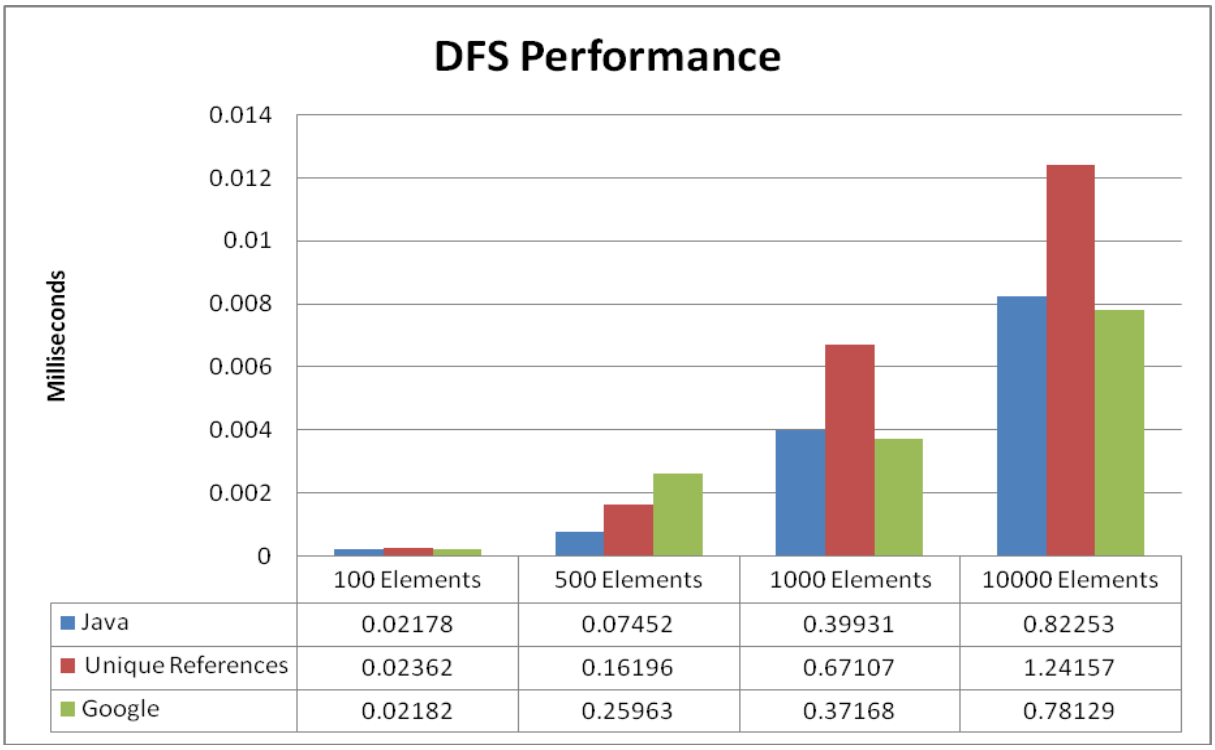
retrieved from it, somehow caching the results. The outliers might have come from a garbage collection cycle being trigger or might have been impacted negatively by an operating system operation.

4.2.2 Depth First Search

The Depth First Search application was created to perform a classic Depth First Search. The Wikipedia definition of Depth First Search:

> *Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.*

The algorithm used to perform the depth first search was taken from Cormen [7]. Our implementation of the DFS application created a graph of nodes stored in a List. The nodes were represented as mutable Node objects. The nodes contained a List of adjacent node indices. The indices represented the index in the List of nodes where their adjacent node existed. The application did a Depth First Search traversing each node in a depth first manner assigning discovery and finish times. The maximum number of nodes in the Lists for both the DFS and Topological sort runs were capped at 10,000. This is due to receiving StackOverflowExceptions at larger list lengths. Ten runs were completed for each Java, Unique References and Google lists. Their run times were averaged together to give the following graph.



	100 Elements	500 Elements	1000 Elements	10000 Elements
Java	0.02178	0.07452	0.39931	0.82253
Unique References	0.02362	0.16196	0.67107	1.24157
Google	0.02182	0.25963	0.37168	0.78129

Figure 26 - DFS Performance

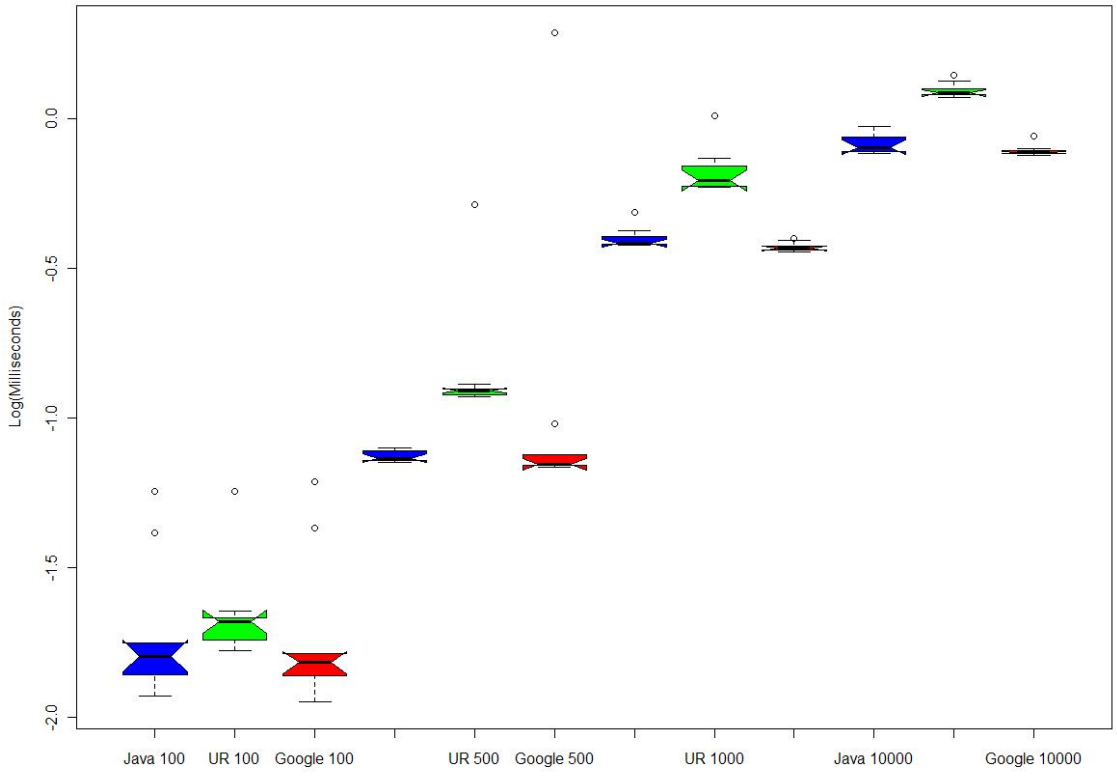


Figure 27 - DFS 1000 Elements boxplot

Our results showed that the Google Immutable List performed the best. The Unique References list performed the worst. However, the results were fairly in time. However, in the 10,000 element run, the UR List took approximately 50% longer even though 50% was only .4 ms. The reason for the Unique References reduced performance was the need to swap the Node objects in and out of the list to modify them. This reduction in performance should be a linear one. The trend on the bar graph echoes the linear progression.

The DFS results are what we anticipated after the Data Read results. The lists in the DFS application were stable and constant throughout the life of the application. In the DFS application, however, the contents of the list were modified. The modification of the elements was the differentiating factor between DFS and Data Read.

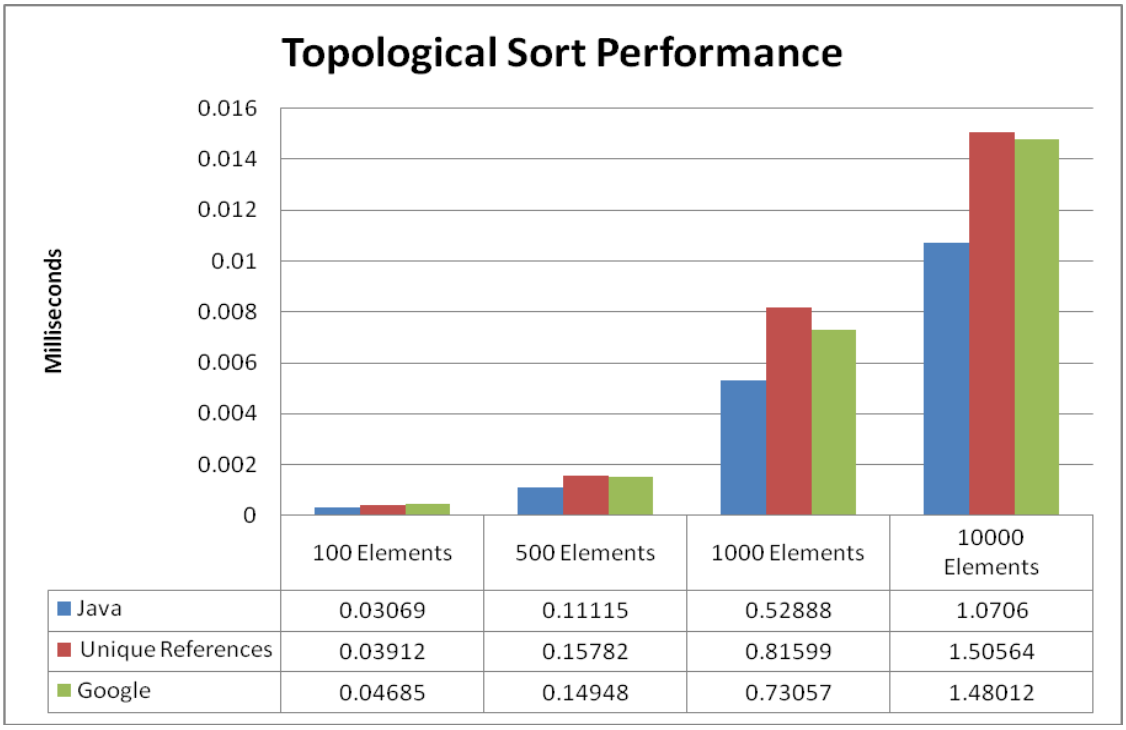
The distributions came out a little varied and there is no clear explanation for this either. The Google list might have been optimizing its data structure every time an item was retrieved from it, somehow caching the results. The outliers might have come from a garbage collection cycle being trigger or might have been impacted negatively by an operating system operation. Another reason for the outliers could have been the data. A completely new random graph was created then passed to each DFS search object. It is possible that some of these cases were more expensive than others.

4.2.3 Topological Sort

The Topological Sort was created to perform a classic Topological Sort. The Wikipedia definition of Topological Sort:

> *In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.*
>
> *More formally, define the reachability relation R over the nodes of the DAG such that xRy if and only if there is a directed path from x to y. Then, R is a partial order, and a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.*

The topological sort algorithm was taken from Cormen [7].Just like in the Depth First Search application, the Topological Sort application created a graph of nodes stored in a List. The nodes were represented as Node objects. The nodes contained a List of adjacent nodes indices. The indices represented the index in the List of nodes where their adjacent node existed. The application did a Depth First Search traversing each node in a depth first manner assigning discovery and finish times. As the nodes were finished in the topological sort, they were added to a list. Ten runs were completed for each Java, Unique References and Google lists.



	100 Elements	500 Elements	1000 Elements	10000 Elements
Java	0.03069	0.11115	0.52888	1.0706
Unique References	0.03912	0.15782	0.81599	1.50564
Google	0.04685	0.14948	0.73057	1.48012

Figure 28 - Topological Sort performance

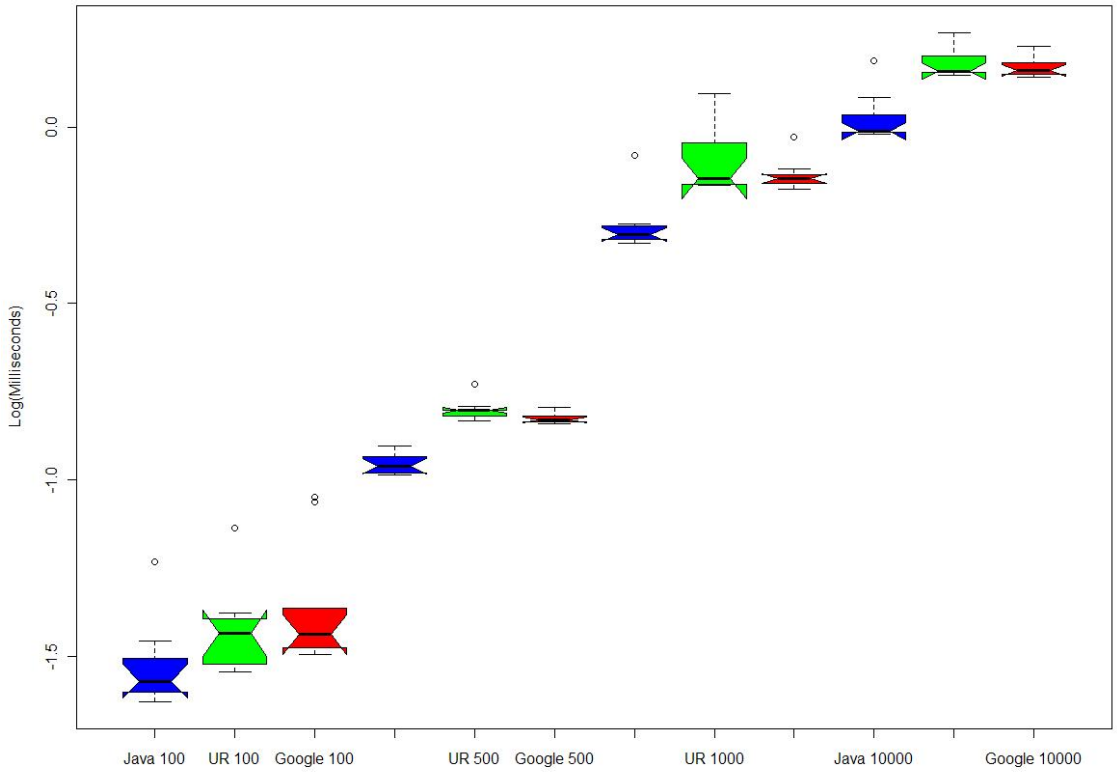


Figure 29 - Topological Sort 1000 elements boxplot

The results of the Topological Sort application were slightly different than that of the DFS application. In particular, the Google List is now the slowest. This difference stems from the need to the build a list to return. The topological sort algorithm is very similar to the DFS algorithm. The difference in topological sort and DFS is that in DFS after a node has been traveresed, its index is added to a list. The result is a list of node indicies ordered by finish time. Creating this Google List requires a build method call on the immutable list builder. This call will copy all of the values from the builder into a new immutable list which is the cause of the Google List's reduced performance. The relationship between Java and UR is identical in Topological Sort to DFS.

The distributions came out a little varied and there is no clear explanation for this either. The Google list might have been optimizing its data structure every time an item was retrieved from it, somehow caching the results. The outliers might have come from a garbage collection cycle being trigger or might have been impacted negatively by an operating system operation. Another reason for the outliers could have been the data. A completely new random graph was created then passed to each Topological Sorter object. It is possible that some of these cases were more expensive than others.

4.2.4 Performance Results

The performance test showed that the programs written using the Unique References List executed nearly as fast as those written using the Java List. The tests showed that the programs written using the Google Collections List executed nearly as fast as those written using the Java List and actually executed slightly faster in programs where the List was stable. In programs where the Lists were being modified frequently, the Google Collections List performed very poorly. The Functional Java list was slow in every scenario.

These differences should not be incredibly surprising. The differences between a Java List program and a UR program are very small. The UR List is essentially just a wrapper for the Java List. The performance degradation from each method call should be minimal. The crucial difference in using the Java List and using the UR List is how to retrieve elements. Instead of simply calling get(int index) and being returned a reference to your object, you must call set(int index, T element) passing in null. With UR you must also remember to return your object to the List after using it by calling set again. It's this second method call that really causes the discrepancy in the run times of the applications.

To give a sort of rough estimate on the overhead of two swaps, we'll investigate the Data Read application. This application contained a List of Lists of Elements, List<List<Element>>. Ten thousand reads were done randomly from this list. That's 20,000 gets from the Java List versus 40,000 sets of the UR List. The average run times were 1.05ms and 1.19 ms for Java and UR respectively. This shows there isn't a lot of overhead with the swapping paradigm itself. The Quicksort application took a mere 20 ms longer in Unique References for sorting a list of 100,000 elements. This happened to

be 31% but is still inconsequential in terms of impact. Typically in an application, the bottlenecks wouldn't be inserting an element into a list; they would be retrieving data from a database or computing a complex function. The effect sizes here should be of interest because they are so small.

Effects size of software performance is not something that is directly or widely discussed in the literature. However, researchers in computer science theory focus on how performance (complexity) is affected relative to input size. For example, does the algorithm run in constant time, linear time, polynomial time, or exponential time? In the case of the Java and unique references lists, the complexity of similar algorithms will be the same. For example, if an algorithm based on a Java list takes linear time, a similar algorithm based on a unique references list will also take linear time. The reason for this is that the practical differences between the lists are that where you would only need a handle to access or update an element in a Java list, you may potentially need to swap an element out of a UR list, access or update it, and then swap it back in. The swapping, like the referencing in Java lists, is a constant time operation, so it will not add to the computational complexity of an algorithm.

Experiment	Multiplier (UR/Java)
Quicksort (100 small elements)	1.33
Quicksort (500 small elements)	1.26
Quicksort (1000 small elements)	1.32
Quicksort (10,000 elements)	1.3
Quicksort (100,000 elements)	1.3
Towers of Hanoi 10 Disks	1.25
Towers of Hanoi 20 Disks	1.05
Towers of Hanoi 20 Disks	1.03
Data Read (small elements)	1.13
Data Read (large elements)	1.19
Depth First Search (100 small elements)	1.08
Depth First Search (500 small elements)	2.17
Depth First Search (1,000 small elements)	1.68
Depth First Search (10,000 small elements)	1.5
Topological Sort (100 small elements)	1.27
Topological Sort (500 small elements)	1.42
Topological Sort (1,000 small elements)	1.54
Topological Sort (10,000 small elements)	1.41

From a practitioner's point of view, the effect size is rooted in practical concerns. For example, the computational complexity of the quicksort algorithm is n^2 in the worst case,

whereas the complexity of heap sort is n log n. In practice, however, quicksort is often faster than heap sort because the worst case does not occur that often and the constant multiplier for quicksort is small. In almost all of our experiments the runtime of the UR list algorithm was no more than 1.5 times that of the corresponding Java algorithm (see below). We believe that this difference is small enough that the vast majority of software practitioners could safely ignore it provided that clear reasoning benefits existed from using the UR list. Those reasoning benefits are the topic of the next section.

5 List Specification Comparison

Specifications were created for the Java List as well as the Unique References List. The specifications were used to verify the behavior of the objects to determine the specific side effects of their methods. Using the specifications we could ensure that the Unique References rules were being broken by Java and obeyed by the Unique References List we created.

In heavyweight specifications we use traditional mathematical concepts (e.g. sequences, sets and functions) to model each type and describe the behavior of the class using assertions (e.g. invariants, preconditions and postconditions) written in terms of the mathematical model.

5.1.1 Mathematical Sequences

The List classes use a math sequence as their mathematical model. A math sequence can be thought of as an ordered sequence of objects with functions to insert or remove elements.

Suppose s = [a, b, c, d] and t = [e, f, g, h] are mathematical sequences. Then the following are true statements about mathematical them:

- |s| is the length of s (= 4 in this case)
- 0 is the first index of s
- |s| – 1 is the last index of s
- [ ] is the empty list
- [x] is a singleton list containing x
- s[1..3] = [b c d]
- s[1..0] = [ ] or $s_{1..0}$ = [ ]
- s[1] = b or s_0 = b
- s + t = [a b c d e f g h]

5.1.2 Java List Specification

Interface List<E>{

static model contents: Location ---> MathSeq<E>;
model Location **this**;

void add(int n, Object e)
updates contents;
requires 0 ≤ n ≤ |contents(**this**)|
ensures contents = #contents (+)
{ #**this** |--> #contents(#**this**)[0..(#n–1)] + [#e] +
#contents(#**this**)[n..(|#contents(#**this**)|–1)] }
ensures n = #n
ensures e = #e

void add(int n, Object e)
    updates contents;
    requires 0 ≤ n ≤ |***this**|
    ensures contents = #contents (+)
        { #**this maps_to** *#**this**[0..(#n–1)] + [#e] + *#**this**[n..(|*#**this**|–1)] }
    ensures n = #n
    ensures e = #e

boolean add(Object e)
    ensures this = #**this** + [#e]
    ensures e = #e

boolean addAll(Collection c)
    requires 0 ≤ n ≤ |**this**|
    ensures this = #**this**[0..(#n–1)] + #c + #**this**[n..(|#**this**|–1)]
    ensures c = #c

boolean addAll(int n, Collection c)
    requires 0 ≤ n ≤ |**this**|
    ensures this = #**this**[0..(#n–1)] + #c + #**this**[n..(|#**this**|–1)]
    ensures n = #n
    ensures c = #c

void clear()
    ensures this = []

boolean contains(Object o)
    ensures result = true if #**this**[i] = object, false otherwise

boolean containsAll(Collection c)
    ensures result = true if #**this**[i] = o for all o in targets, false otherwise

boolean equals(Object o)
    ensures result = (#**this** = #o)

Object get(int index)
    requires 0 ≤ index < |**this**|
    ensures result = #**this**[index]

int hashCode()
ensures result = */*int*/*

int indexOf(Object o)
    ensures result = i: #**this**[i] = object

boolean isEmpty()

ensures result = true if |#**this**| = 0, false otherwise

Iterator iterator()
requires /* *E is immutable* */
ensures this = #**this**
ensures result = Iterator

int lastIndexOf(Object o)
ensures result = i: #**this**[i] = object

ListIterator listIterator()
ensures result = /* ListIterator */

ListIterator listIterator(int index)
requires 0 ≤ index < |**this**|
ensures result = /* ListIterator */

Entry remove(int index)
updates contents
requires 0 ≤ index < |***this**|
ensures this = #**this**
ensures index = #index
ensures contents = #contents (+)
{#**this** |--> *#**this**[0..#index–1] + *#**this**[(#index+1)..|*#**this**|]}
ensures result = *#**this**[#index]

boolean remove(Entry e)
ensures this = #**this**
ensures e = #e
ensures contents = #contents (+)
{#**this maps_to** ALL_BUT_FIRST(*#**this**, #e)}
ensures result = #e **not_in** *#**this**

boolean removeAll(Collection c)
ensures this contains no elements in c

boolean retainAll(Collection c)
ensures this contains only elements in c

Object set(int index, Object element)
requires 0 ≤ index < |**this**|
ensures element = #element
ensures n = #n
ensures this = #**this**[0.n(#n-1)] + |#o| + #**this**[(n+1)..(|#**this**|–1)]
ensures result = #**this**[#n]

```
    int size()
        ensures this = #this
        ensures result = |#this|

    List subList(int fromIndex, int toIndex)
        requires fromIndex 0 ≤ fromIndex < |this|
        requires toIndex fromIndex ≤ toIndex < |this|
        ensures result = /* List<E> */

    Object[] toArray()
        ensures result = /* Array */

    Object[] toArray(Object[] a)
        ensures a contains #this
}
```

5.1.3 Unique References List Specification

```
interface List<E> {
        model MathSeq<E> this;

    void add(@transfer E e);
        ensures this = #this + [#e]
        ensures e = ??

    void add(int n, @transfer E e);
        requires 0 ≤ n ≤ |#this|
        ensures this = #this[0..(#n–1)] + [#e] + #this[n..(|#this|–1)]
        ensures n = #n
        ensures e = ??

    void clear();
        ensures this = []

    E get(int n);
        requires /* E is immutable */
        requires 0 ≤ n < |#this|
        ensures this = #this
        ensures n = #n
        ensures result = #this[#n]

    boolean isEmpty();
        ensures this = #this
        ensures result = |#this|

    E remove(int n);
        requires 0 ≤ n < |this|
        ensures this = #this[0.n(#n-1)] + #this[(n+1)..(|#this|–1)]
```

```
        ensures n = #n
        ensures result = #this[n]

    int size();
        ensures this = #this
        ensures result = |#this|

    E swap(int n, @transfer E o);
                requires 0 ≤ n < |this|
                ensures o = ??
                ensures n = #n
        ensures this = #this[0.n(#n-1)] + |#o| + #this[(n+1)..(|#this|–1)]
                ensures result = #this[#n]

    Iterator<E> iterator();   // E immutable
        requires /* E is immutable */
        ensures this = #this
        ensures result = Iterator

    void addAll(int n, List<E> s);
        requires 0 ≤ n ≤ |this|
        ensures this = #this[0..(#n–1)] + #s + #this[n..(|#this|–1)]
        ensures n = #n
        ensures s = [ ]
}
```

5.2 Tracing Table Comparison

To compare the understandability of the programs, tracing tables were created for a method from both the Towers of Hanoi and the Quicksort program. The metrics collected from the tracing tables were compared for number of steps, number of tokens, lines of code and tokens per step.

The number of steps was compared to determine if one of the lists required more effort to write the program with. The LOC equals the sum of the number of variables in each state. The token count compares all the bits of data that have to be analyzed.

Metrics example:

State	Facts
6	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
	ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]}
	list = @43
	left= 0
	right= 4
	i = -1
	j = 1

if (list.get(j).compareTo(pivot) <= 0) {

In this example, there is one step. There are seven lines of code.

For a line of code, "ArrayList.Contents = {@43 -> [@85, @92]}", the table below lists the tokens.

ArrayList
.
Contents
=
{
@43
->
[
@85
,
@92
]
}

5.2.1 partition Method

The partition method is from the Quicksort application. The purpose of this method is to rearrange a sub-array in place.

Java partition Method Tracing table

State	Facts
1	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]} list = @43 left= 0 right= 4
	private static int partition(ArrayList<Element> list, int left, int right){
2	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]} list = @43 left= 0 right= 4
	Element pivot = list.get(right);
3	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]} list = @43

left= 0
right= 4

int i = left-1;

4
Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]}
list = @43
left= 0
right= 4
i = -1
j = 0

for (int j = left; j <= right-1; j++) {

5
Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]}
list = @43
left= 0
right= 4
i = -1
j = 1

for (int j = left; j <= right-1; j++) {

6
Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]}
list = @43
left= 0
right= 4
i = -1
j = 1

if (list.get(j).compareTo(pivot) <= 0) {

7
Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
ArrayList.Contents = {@43 -> [@85, @71, @64, @28, @62]}
list = @43
left= 0
right= 4
i = -1
j = 1

swap(list, ++i, j);

8
Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5}
ArrayList.Contents = {@43 -> [@71, @85, @64, @28, @62]}
list = @43
left= 0
right= 4
i = 0
j = 2

for (int j = left; j <= right-1; j++) {

9	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@71, @85, @64, @28, @62]} list = @43 left= 0 right= 4 i = 0 j = 3
	for (int j = left; j <= right-1; j++) {
10	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@71, @85, @64, @28, @62]} list = @43 left= 0 right= 4 i = 0
	swap(list, ++i, right);
11	Element.Contents = {@64 -> 30, @85 -> 20, @62 -> 10, @28 -> 15, @71 -> 5} ArrayList.Contents = {@43 -> [@71, @62, @64, @28, @85]} list = @43 left= 0 right= 4 i = 1
	return i;

Steps: 11, Tokens: 636, Lines Of Code: 69

Unique References partition Method Tracing Table

State	Facts
1	list = [20, 5, 30, 15, 10] left = 0 right = 4
	private static int partition(JArrayList<Element> list, int left, int right){
2	list = [20, 5, 30, 15, 10] left = 0 right = 4
	Element pivot = list.swap(right, null);
3	list = [20, 5, 30, 15, NULL] left = 0 right = 4
	int i = left-1;
4	list = [20, 5, 30, 15, NULL] left = 0 right = 4 i = -1

	for (int j = left; j <= right-1; j++) {
5	list = [20, 5, 30, 15, NULL] left = 0 right = 4 i = -1 j = 0
	if (compareTo(list, j, pivot, right) <= 0) {
6	list = [20, 5, 30, 15, NULL] left = 0 right = 4 i = -1
	for (int j = left; j <= right-1; j++) {
7	list = [20, 5, 30, 15, NULL] left = 0 right = 4 i = -1 j = 1
	if (compareTo(list, j, pivot, right) <= 0) {
8	list = [20, 5, 30, 15, NULL] left = 0 right = 4 i = -1 j = 1
	swap(list, ++i, j);
9	list = [5, 20, 30, 15, NULL] left = 0 right = 4 i = 0
	for (int j = left; j <= right-1; j++) {
10	list = [5, 20, 30, 15, NULL] left = 0 right = 4 i = 0 j = 2
	if (compareTo(list, j, pivot, right) <= 0) {
11	list = [5, 20, 30, 15, NULL] left = 0 right = 4 i = 0
	for (int j = left; j <= right-1; j++) {
12	list = [5, 20, 30, 15, NULL] left = 0 right = 4

	i = 0 j = 3
	if (compareTo(list, j, pivot, right) <= 0) {
13	list = [5, 20, 30, 15, NULL] left = 0 right = 4
	else{
14	list = [5, 20, 30, 15, NULL] left = 0 right = 4 i = 1
	Element k = list.swap(i, pivot);
15	list = [5, 10, 30, 15, NULL] left = 0 right = 4 i = 1
	list.swap(right, k);
16	list = [5, 10, 30, 15, 20] left = 0 right = 4 i = 1
	return i;

Steps: 16, Tokens: 355, Lines Of Code: 65

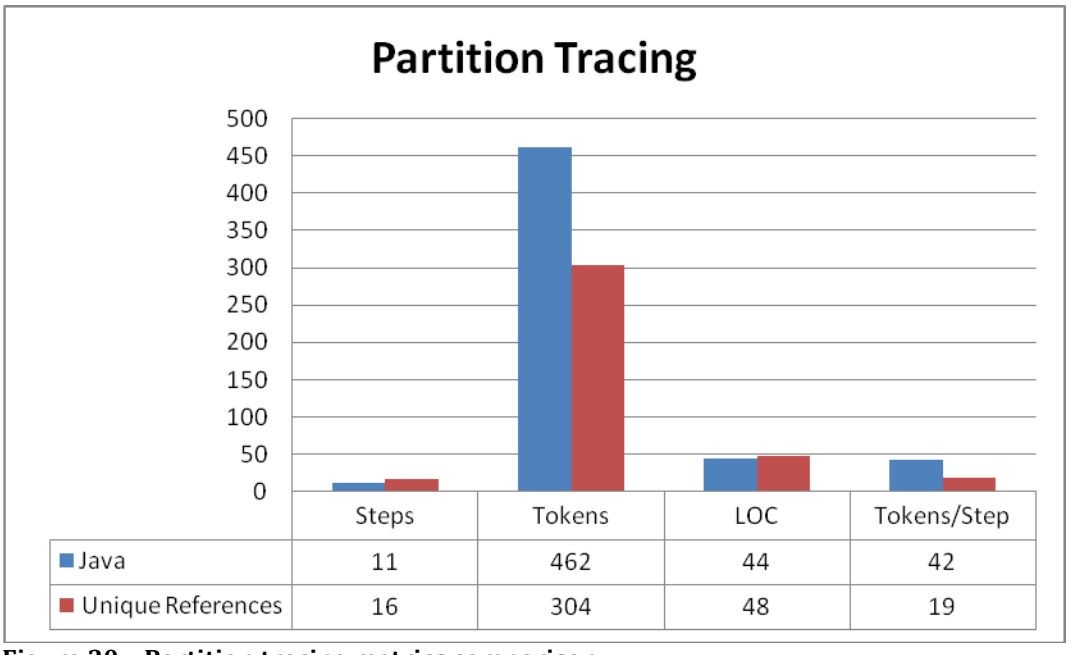
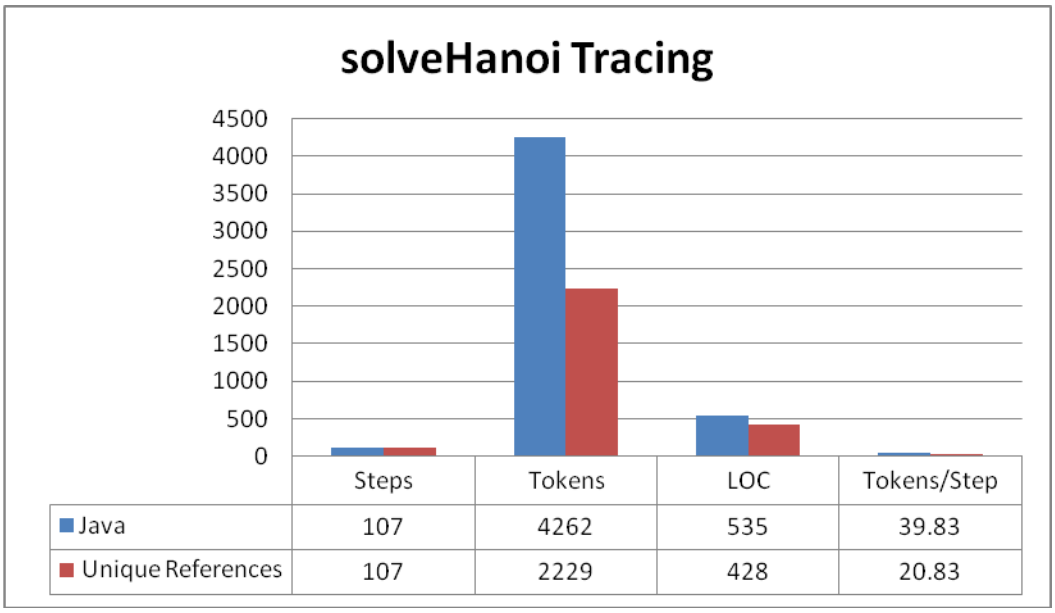


Figure 30 – Partition tracing metrics comparison

Paragraphs here describing what happened.

5.2.2 solveHanoi Method

Java: In appendix
Unique References: In appendix



	Steps	Tokens	LOC	Tokens/Step
Java	107	4262	535	39.83
Unique References	107	2229	428	20.83

Figure 31 - solveHanoi metrics comparison

The Unique References containers were designed to facilitate better reasoning about programs. This becomes apparent in these tracing table numbers as there are just over half as many tokens in the Unique References tracing table as there are in the Java tracing table. This preponderance of tokens in Java comes from the necessity of having a global variable in the tracing table containing references to all the objects.

5.2.3 Reasoning Results

The tracing tables showed us that Unique References had significantly fewer tokens than Java. Fewer tokens and fewer lines of code in the tracing table should translate into fewer things for the developer to remember when writing code using the List classes. It stands to reason that if code is easier to understand and write it should have fewer bugs.

The reason why Java has so many more tokens and lines of code is the aliasing that Java allows. To reason about the aliasing you have to include maps from items to their memory locations so that they might be used in multiple places. This adds an additional variable and a few extra tokens per object.

5.2.3.1 UR Token Formula

The following formula is a rough estimate of the number of tokens you can expect in a step of a tracing table for a UR List application:

Tokens_in_step = variables * 3 + objects_in_lists * 2

I'll now describe each part of this formula.
variables *3: variables is a value that is equal to the number of variables in scope. Each variable will receive a line in the tracing table like "left = 0". This is three tokens.

objects_in_lists * 2: objects_in_lists is a value that is equal to the number of objects in the lists in scope. The general form for a List in the tracing table is [list variable] = [value1, value2, … valuen]. This part of the formula is adding up all the tokens for the value and the commas.

This will give a rough estimate of the tokens in a single step for a UR tracing table.

5.2.3.2 Java Token Formula

The following formula is a rough estimate of the number of tokens you can expect in a step of a tracing table for a Java application:

Tokens_in_step = variables * 3 + objects_in_lists * 2 + mutable_objects * 6 + mutable_types * 6

I'll now describe each part of this formula.
variables * 3 and objects_in_lists *2 are the same as they were in UR.

mutable_types * 6: mutable_types is a value that is equal to the number of mutable types in scope. Each mutable type will have an entry in the tracing table like "[type name].Contents = {@address1->value1, @address2->value2, …, @addressn->valuen}". This part of the formula is adding up the six tokens that form this line excluding the values in the map: [type name].Contents = {}.

mutable_objects * 6: mutable_objects is a value that is equal to the number of mutable objects in scope. This part of the formula is adding up all the tokens for the "@address->value," part of the mapping.

6 Conclusion

The problems that references and aliasing create are not new. Parallels have been drawn between references and goto statements. Goto statements are now highly regarded as mostly inappropriate and it's rare to see them in new code. At times, however, goto statements can make things simpler and more efficient just as references can. The goal with Unique References is not to ban the use of references altogether but, instead, to use references in a way that facilitates reasoning about programs as if they were value models.

Unique References seems to function well as a bridge between the reference model language implementation and the value model semantics. The applications written using UR were very close in performance to those written using standard Java libraries while the reasoning information was significantly simpler.

To make UR applicable to real world applications, more verification support is necessary. A static verifier could be implemented to ensure that Java applications conformed to the rules of UR. The optimal solution would be compiler support as well as IDE support. Syntax highlighting of UR errors in Eclipse would make following UR rules trivial.

The only drawback found while using UR was remembering to put objects back into the List. Forgetting to swap elements caused a couple problems early in the development of the test applications. Aside from that, using the UR List was identical to the Java List.

Work is still needed to integrate UR into larger and more complex applications. It is believed that the results from the small applications introduced in this paper would be mirrored there. It would also be of interest to create additional container types such as Hashtable, Set and map.

For the results of our hypotheses:

- **Hypothesis 1**
  We determined that we removed all of the aliasing behavior for mutable objects. This was determined during creation of the Unique References interface. Each method of the Java List interface was examined for possible alias creating behavior and modified to prevent aliasing.

- **Hypothesis 2**
  It was determined that the swapping null out to access objects took between 30% and 50% longer in Unique References lists. This was determined by evaluating each of the performance results. The application which did no swapping, Towers of Hanoi, the performance of Unique References was almost identical to Java. In the other applications which all required swapping for Unique References, the performance was negatively impacted.

- **Hypothesis 3**

The tracing tables for the Unique References List contained far fewer tokens and fewer lines of code. The token count was 40% to 50% higher in the tracing tables for the Java List. The lines of code metric were around 20% higher in Java. This was determined by creating tracing tables for the solveHanoi method of the Towers of Hanoi application and the partition method of the Quicksort application and calculating the metrics based on those. A formula for estimating the metrics was also created after inspecting the tracing tables to get the general pattern.

6.1 Open Questions

There are still many open questions to be answered for Unique References. Of particular interest is how the results would fare on different platforms and given larger applications. Due to time restrains a single platform was used to test the given applications. It would be beneficial to prove the particular ratios on a slower machine and a faster machine as well as machine with more memory and one with less memory.

The applications used in this Thesis were rather small encompassing a single jar file. It would be interesting to use Unique References in an industrial sized application. Upon embarking upon such a task, it might also be necessary to create a static verifier to prove that the Unique References rules were being upheld.

Developer reviews would also be a crucial step in the adoption of Unique References. Would developers be able to learn and obey the Unique References rules and would they be like Unique References? Surveys could be taken to gauge these questions.

7 Appendix

7.1 Java solveHanoi

Java solveHanoi Method Tracing Table

State	Facts
1	disks = 4 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []} fromPole = @17 toPole = @93 withPole = @88
	solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){
2	disks = 4 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []} fromPole = @17 toPole = @93 withPole = @88
	if (disks >= 1) {
3	disks = 4 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []} fromPole = @17 toPole = @93 withPole = @88
	solveHanoi(disks-1, fromPole, withPole, toPole);
4	disks = 3 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []} fromPole = @17 toPole = @88 withPole = @93
	solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){
5	disks = 3 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []} fromPole = @17 toPole = @88 withPole = @93
	if (disks >= 1) {
6	disks = 3

Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []}
fromPole = @17
toPole = @88
withPole = @93

solveHanoi(disks-1, fromPole, withPole, toPole);

7 disks = 2
Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []}
fromPole = @17
toPole = @93
withPole = @88

solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){

8 disks = 2
Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []}
fromPole = @17
toPole = @93
withPole = @88

if (disks >= 1) {

9 disks = 2
Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []}
fromPole = @17
toPole = @93
withPole = @88

solveHanoi(disks-1, fromPole, withPole, toPole);

10 disks = 1
Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [@17, @88, @93, @12], @93 -> [], @88 -> []}
fromPole = @17
toPole = @88
withPole = @93

solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){

…

102 disks = 0
Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3}
List.Contents = {@17 -> [], @93 -> [@17, @88, @93], @88 -> [@12]}
fromPole = @88
toPole = @17
withPole = @93

	solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){
103	disks = 0 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [], @93 -> [@17, @88, @93], @88 -> [@12]} fromPole = @88 toPole = @17 withPole = @93
	if (disks >= 1) {
104	disks = 1 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [], @93 -> [@17, @88, @93], @88 -> [@12]} fromPole = @88 toPole = @93 withPole = @17
	toPole.add(fromPole.remove(fromPole.size() - 1));
105	disks = 1 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [], @93 -> [@17, @88, @93, @12], @88 -> []} fromPole = @88 toPole = @93 withPole = @17
	solveHanoi(disks-1, withPole, toPole, fromPole);
106	disks = 0 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [], @93 -> [@17, @88, @93, @12], @88 -> []} fromPole = @17 toPole = @93 withPole = @88
	solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole, ArrayList<Disk> withPole){
107	disks = 0 Disk.Contents = {@17 -> 4, @93 -> 2, @12 -> 1, @88 -> 3} List.Contents = {@17 -> [], @93 -> [@17, @88, @93, @12], @88 -> []} fromPole = @17 toPole = @93 withPole = @88
	if (disks >= 1) {

Steps: 107, Tokens: 6295, Lines Of Code: 642

7.2 Unique References solveHanoi

UR solveHanoi Method Tracing table

State	Facts

1	disks = 4 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
2	disks = 4 fromPole = [4,3,2,1] toPole = [] withPole = []
	if (disks >= 1) {
3	disks = 4 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(disks-1, fromPole, withPole, toPole);
4	disks = 3 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
5	disks = 3 fromPole = [4,3,2,1] toPole = [] withPole = []
	if (disks >= 1) {
6	disks = 3 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(disks-1, fromPole, withPole, toPole);
7	disks = 2 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
8	disks = 2 fromPole = [4,3,2,1] toPole = [] withPole = []
	if (disks >= 1) {
9	disks = 2 fromPole = [4,3,2,1] toPole = []

	withPole = []
	solveHanoi(disks-1, fromPole, withPole, toPole);
10	disks = 1 fromPole = [4,3,2,1] toPole = [] withPole = []
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
	…
102	disks = 0 fromPole = [1] toPole = [] withPole = [4,3,2]
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
103	disks = 0 fromPole = [1] toPole = [] withPole = [4,3,2]
	if (disks >= 1) {
104	disks = 1 fromPole = [1] toPole = [4,3,2] withPole = []
	toPole.add(fromPole.remove(fromPole.size() - 1));
105	disks = 1 fromPole = [] toPole = [4,3,2,1] withPole = []
	solveHanoi(disks-1, withPole, toPole, fromPole);
106	disks = 0 fromPole = [] toPole = [4,3,2,1] withPole = []
	solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole, JArrayList<Disk> withPole){
107	disks = 0 fromPole = [] toPole = [4,3,2,1] withPole = []
	if (disks >= 1) {

Steps: 107, Tokens: 2229, Lines Of Code: 428

7.3 Towers of Hanoi Code Listing

This appendix provides an overview of Towers of Hanoi application.

File name
Disk.java
FunctionalController.java
CoogleController.java
JavaController.java (6.3.1)
Main.java
URController.java (6.3.2)

7.3.1 Java Class

```
package dsmith;

import java.util.ArrayList;

public class JavaController {
	private int disks;
	private ArrayList<Disk> start;
	private ArrayList<Disk> middle;
	private ArrayList<Disk> end;

	private Hashtable<Integer, ArrayList<Disk>> ListContents = new Hashtable<Integer,
ArrayList<Disk>>();

	public JavaController(int disks, int elementSize){
		this.disks = disks;
		start = new ArrayList<Disk>();

		middle = new ArrayList<Disk>();

		end = new ArrayList<Disk>();

		int currentOrdinal = disks;
		while(currentOrdinal > 0){
			start.add(new Disk(currentOrdinal--, elementSize));
		}
	}

	public int getDisks(){
		return disks;
	}

	public void solveHanoi(){
		solveHanoi(disks, start, end, middle);
	}

	private void solveHanoi(int disks, ArrayList<Disk> fromPole, ArrayList<Disk> toPole,
ArrayList<Disk> withPole){

		if (disks >= 1) {
```

```
                solveHanoi(disks-1, fromPole, withPole, toPole);

                toPole.add(fromPole.remove(fromPole.size() - 1));

                solveHanoi(disks-1, withPole, toPole, fromPole);
            }
        }
}
```

7.3.2 Unique References Class

```
package dsmith;
import tuna.util.JArrayList;


public class URController {
        private int disks;
        private JArrayList<Disk> start;
        private JArrayList<Disk> middle;
        private JArrayList<Disk> end;

        public URController (int disks, int elementSize){
            this.disks = disks;
            start = new JArrayList<Disk>();
            middle = new JArrayList<Disk>();
            end = new JArrayList<Disk>();

            int currentOrdinal = disks;
            while(currentOrdinal > 0){
                start.add(new Disk(currentOrdinal--, elementSize));
            }
        }

        public int getDisks(){
            return disks;
        }

        public void solveHanoi(){
            solveHanoi(disks, start, end, middle);
        }

        private void solveHanoi(int disks, JArrayList<Disk> fromPole, JArrayList<Disk> toPole,
JArrayList<Disk> withPole){
            if (disks >= 1) {
                solveHanoi(disks-1, fromPole, withPole, toPole);

                toPole.add(fromPole.remove(fromPole.size() - 1));

                solveHanoi(disks-1, withPole, toPole, fromPole);
            }
        }
}
```

7.3.3 Google Collections Class

```
package dsmith;

import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableList.Builder;

public class GoogleController {
	private int disks;
	private ImmutableList<Disk> start;
	private ImmutableList<Disk> middle;
	private ImmutableList<Disk> end;

	public GoogleController(int disks, int elementSize){
		this.disks = disks;
		end = ImmutableList.of();
		middle = ImmutableList.of();

		int currentOrdinal = disks;

		Builder<Disk> b = ImmutableList.builder();
		while(currentOrdinal > 0){
			b.add(new Disk(currentOrdinal--, elementSize));
		}
		start = b.build();
	}

	public int getDisks(){
		return disks;
	}

	public boolean isDone(){
		return false;
	}

	public void solveHanoi(){
		solveHanoi(disks, start, end, middle);
	}

	private void solveHanoi(int disks, ImmutableList<Disk> fromPole, ImmutableList<Disk> toPole,
ImmutableList<Disk> withPole){
		if (disks >= 1) {
			solveHanoi(disks-1, fromPole, withPole, toPole);
			int size = fromPole.size();
			Disk toAdd = null;
			if(size >= 1){
				toAdd = fromPole.get(size - 1);
				fromPole = removeDisk(fromPole);
				toPole = addDisk(toPole, toAdd);
			}
			solveHanoi(disks-1, withPole, toPole, fromPole);
		}
	}

	private ImmutableList<Disk> addDisk(ImmutableList<Disk> list, Disk toAdd){
			Builder<Disk> b = ImmutableList.builder();
			b.addAll(list);
```

```
                b.add(toAdd);
                return b.build();
        }

        private ImmutableList<Disk> removeDisk(ImmutableList<Disk> list){
                int size = list.size();
                if(size == 0){
                        return list;
                }

                Builder<Disk> b = ImmutableList.builder();
                for(int i = 0; i < size - 1; i++)
                {
                        b.add(list.get(i));
                }

                return b.build();
        }
}
```

7.4 Data Read Code Listing

This appendix provides an overview of the Data Read application.

File name
DataReadDriver.java
ElementList.java
FunctionaList.java
GoogleList.java
JavaArrayList.java
URJArrayList.java

7.4.1 Java Class

```
package dsmith.DataRead;

import dsmith.Element;
import java.util.ArrayList;

public class JavaArrayList implements ElementList{

        private ArrayList<ArrayList<Element>> elements;

        @Override
        public void initialize(Element[][] elements) {
                this.elements = new ArrayList<ArrayList<Element>>();
                for(int i = 0; i < elements.length; i++)
                {
                        ArrayList<Element> current = new ArrayList<Element>();
                        for(int j = 0; j < elements[i].length;j++)
                        {
                                current.add(elements[i][j]);
```

```
                        }
                        this.elements.add(current);
                }
        }

        @Override
        public void readElement(int index1, int index2) {
                ArrayList<Element> list = elements.get(index1);
                Element e = list.get(index2);
        }
}
```

7.4.2 Unique References Class

```
package dsmith.DataRead;

import tuna.util.JArrayList;
import dsmith.Element;

public class URJArrayList implements ElementList {

        private JArrayList<JArrayList<Element>> elements;

        @Override
        public void initialize(Element[][] elements) {
                this.elements = new JArrayList<JArrayList<Element>>();
                for(int i = 0; i < elements.length; i++)
                {
                        JArrayList<Element> current = new JArrayList<Element>();
                        for(int j = 0; j < elements[i].length;j++)
                        {
                                current.add(elements[i][j]);
                        }
                        this.elements.add(current);
                }
        }

        @Override
        public void readElement(int index1, int index2) {
                JArrayList<Element> list = elements.swap(index1, null);
                Element e = list.swap(index2, null);
                list.swap(index2, e);
                elements.swap(index1, list);
        }
}
```

7.5 Quicksort Code Listing

This appendix provides an overview of the Quicksort application.

File name
FunctionaJavaQuickSorter.java
GoogleCollectionsQuickSorter.java
JavaQuickSorter.java

QuickSortDriver.java
QuickSorter.java
URQuickSorter.java

7.5.1 Java Class

```
package dsmith.QuickSort;

import dsmith.Element;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.Hashtable;

public class JavaQuickSorter implements QuickSorter{
	private ArrayList<Element> elementList;

	public void quickSort()
	{
		quickSort(elementList, 0, elementList.size() - 1);
	}

	public void initialize(Element[] elements)
	{
		elementList = new ArrayList<Element>();
		for(int i = 0; i < elements.length; i++)
		{
			elementList.add(elements[i]);
		}
	}

	private static int partition(ArrayList<Element> list, int left, int right){
		Element pivot = list.get(right);
		int i = left-1;
		for (int j = left; j <= right-1; j++) {
			if (list.get(j).compareTo(pivot) <= 0) {
				swap(list, ++i, j);
			}
		}

		swap(list, ++i, right);
		return i;
	}

	private static void swap(ArrayList<Element> list, int one, int two){
		Element temp = list.get(one);
		list.set(one, list.get(two));
		list.set(two, temp);
	}

	private static void quickSort(ArrayList<Element> list, int left, int right) {
		if (left < right) {
			int q = partition(list, left, right);
			quickSort(list, left, q-1);
			quickSort(list, q+1, right);
```

```
                }
        }
}
```

7.5.2 Unique References Class

```
package dsmith.QuickSort;

import tuna.util.JArrayList;
import dsmith.Element;

public class URQuickSorter implements QuickSorter{
        private JArrayList<Element> elementList;

        public void quickSort(){
                quickSort(elementList, 0, elementList.size() - 1);
        }

        public void initialize(Element[] elements){
                elementList = new JArrayList<Element>();
                for(int i = 0; i < elements.length; i++)
                {
                        elementList.add(elements[i]);
                }
        }

        private static int partition(JArrayList<Element> list, int left, int right){
                Element pivot = list.swap(right, null);
                int i = left-1;

                for (int j = left; j <= right-1; j++) {
                        if (compareTo(list, j, pivot, right) <= 0) {
                                swap(list, ++i, j);
                        }
                }

                if(++i == right){
                        list.swap(right, pivot);
                }
                else{
                        Element k = list.swap(i, pivot);
                        list.swap(right, k);
                }

                return i;
        }

        private static int compareTo(JArrayList<Element> list, int j, Element e, int pivotIndex) {
                if(j == pivotIndex) return 0;
                Element temp = list.swap(j, null);
                int value = temp.compareTo(e);
                list.swap(j, temp);
                return value;
        }
```

```
	private static void swap(JArrayList<Element> list, int one, int two){
		if(one == two)return;
		Element temp = list.swap(one, null);
		temp = list.swap(two, temp);
		list.swap(one, temp);
	}

	private static JArrayList<Element> quickSort(JArrayList<Element> list, int left, int right) {
		if (left < right) {
			int q = partition(list, left, right);
			quickSort(list, left, q-1);
			quickSort(list, q+1, right);
		}
		return list;
	}
}
```

7.6 DFS Code Listing

This appendix provides an overview of the Quicksort application.

File name
DFSDriver.java
DFSResultNodeList.java
GoogleDFSearcher.java
IDFSearcher.java
JavaDFSearcher.java
Node.java
URDFSearcher.java

7.6.1 Java Class

```
package dsmith.DFS;

import java.util.ArrayList;

public class JavaDFSearcher implements IDFSearcher{

	private ArrayList<Node> nodes;
	private int time;

	@Override
	public DFSResultNodeList search() {
		time = 0;

		int size = nodes.size();
		for(int i = 0; i < size; i++){
			Node node = nodes.get(i);
			node.setColor(Node.White);
			node.setPredecessor(Node.InvalidPredecessor);
		}
		for(int i = 0; i < size; i++){
```

```
                Node node = nodes.get(i);
                if(node.getColor() == Node.White){
                        DFSVisit(node, Node.InvalidPredecessor);
                }
        }

        return new DFSResultNodeList(nodes);
    }

    private void DFSVisit(Node node, int parent){
        node.setColor(Node.Gray);
        node.setPredecessor(parent);
        node.setDiscovered(++time);

        int[] adjList = node.GetAdjacencyList();
        for(int i = 0; i < adjList.length; i++){
            int childIndex = adjList[i];
            Node child = nodes.get(childIndex);
            if(child.getColor() == Node.White){
                DFSVisit(child, node.getIndex());
            }
        }
        node.setFinished(++time);
        node.setColor(Node.Black);
    }

    @Override
    public void createList(Node[] nodes) {
        this.nodes = new ArrayList<Node>();
        for(int i = 0; i < nodes.length; i++){
            this.nodes.add(nodes[i]);
        }
    }
}
```

7.6.2 Unique References Class

```
package dsmith.DFS;

import tuna.util.JArrayList;

public class URDFSearcher implements IDFSearcher {
    private JArrayList<Node> nodes;
    private int time;

    @Override
    public DFSResultNodeList search() {
        time = 0;

        int size = nodes.size();
        for(int i = 0; i < size; i++){
            Node node = nodes.swap(i, null);
            node.setColor(Node.White);
            node.setPredecessor(Node.InvalidPredecessor);
            nodes.swap(i, node);
```

```
                }
                for(int i = 0; i < size; i++){
                        Node node = nodes.swap(i, null);
                        if(node.getColor() == Node.White){
                                DFSVisit(node, Node.InvalidPredecessor);
                        }
                        nodes.swap(i, node);
                }

                return new DFSResultNodeList(nodes);
        }

        private void DFSVisit(Node node, int parent){
                node.setColor(Node.Gray);
                node.setPredecessor(parent);
                node.setDiscovered(++time);

                int[] adjList = node.GetAdjacencyList();
                for(int i = 0; i < adjList.length; i++){
                        int childIndex = adjList[i];
                        Node child = nodes.swap(childIndex, null);
                        if(child == null)continue;
                        if(child.getColor() == Node.White){
                                DFSVisit(child, node.getIndex());
                        }
                        nodes.swap(childIndex, child);
                }
                node.setFinished(++time);
                node.setColor(Node.Black);
        }

        @Override
        public void createList(Node[] nodes) {
                this.nodes = new JArrayList<Node>();
                for(int i = 0; i < nodes.length; i++){
                        this.nodes.add(nodes[i].copy());
                }

        }
}
```

7.7 Topological Sort Listing

This appendix provides an overview of the Toplogical Sort application.

File name
GoogleToplogicalSort.java
ITopologicalSorter.java
JavaTopologicalSorter.java
TopologicalSorterDriver.java
TopologicalSortResultNodeList.java
TunaTopologicalSort.java

7.7.1 Java Class

```
package dsmith.TopologicalSort;

import java.util.ArrayList;
import dsmith.DFS.Node;

public class JavaTopologicalSorter implements IToplogicalSorter{

	private ArrayList<Node> nodes;
	private int time;
	ArrayList<Integer> results;

	@Override
	public TopologicalSortResultNodeList search() {
		time = 0;
		results = new ArrayList<Integer>();

		int size = nodes.size();
		for(int i = 0; i < size; i++){
			Node node = nodes.get(i);
			node.setColor(Node.White);
			node.setPredecessor(Node.InvalidPredecessor);
		}
		for(int i = 0; i < size; i++){
			Node node = nodes.get(i);
			if(node.getColor() == Node.White){
				DFSVisit(node, Node.InvalidPredecessor);
			}
		}

		return new TopologicalSortResultNodeList(results);
	}

	private void DFSVisit(Node node, int parent){
		node.setColor(Node.Gray);
		node.setPredecessor(parent);
		node.setDiscovered(++time);

		int[] adjList = node.GetAdjacencyList();
		for(int i = 0; i < adjList.length; i++){
			int childIndex = adjList[i];
			Node child = nodes.get(childIndex);
			if(child.getColor() == Node.White){
				DFSVisit(child, node.getIndex());
			}
		}
		node.setFinished(++time);
		results.add(new Integer(node.getIndex()));
		node.setColor(Node.Black);
	}

	@Override
	public void createList(Node[] nodes) {
		this.nodes = new ArrayList<Node>();
		for(int i = 0; i < nodes.length; i++){
			this.nodes.add(nodes[i]);
```

```
		}
	}
}
```

7.7.2 Unique References Class

```
package dsmith.TopologicalSort;

import tuna.util.JArrayList;
import dsmith.DFS.Node;

public class URTopologicalSort implements IToplogicalSorter {
	private JArrayList<Node> nodes;
	private int time;
	JArrayList<Integer> results;

	@Override
	public TopologicalSortResultNodeList search() {
		time = 0;
		results = new JArrayList<Integer>();

		int size = nodes.size();
		for(int i = 0; i < size; i++){
			Node node = nodes.swap(i, null);
			node.setColor(Node.White);
			node.setPredecessor(Node.InvalidPredecessor);
			nodes.swap(i, node);
		}
		for(int i = 0; i < size; i++){
			Node node = nodes.swap(i, null);
			if(node.getColor() == Node.White){
				DFSVisit(node, Node.InvalidPredecessor);
			}
			nodes.swap(i, node);
		}

		return new TopologicalSortResultNodeList(results);
	}

	private void DFSVisit(Node node, int parent){
		node.setColor(Node.Gray);
		node.setPredecessor(parent);
		node.setDiscovered(++time);

		int[] adjList = node.GetAdjacencyList();
		for(int i = 0; i < adjList.length; i++){
			int childIndex = adjList[i];
			Node child = nodes.swap(childIndex, null);
			if(child == null) continue;
			if(child.getColor() == Node.White){
				DFSVisit(child, node.getIndex());
			}
			nodes.swap(childIndex, child);
		}
		node.setFinished(++time);
		results.add(node.getIndex());
		node.setColor(Node.Black);
```

```
	}

	@Override
	public void createList(Node[] nodes) {
		this.nodes = new JArrayList<Node>();
		for(int i = 0; i < nodes.length; i++){
			this.nodes.add(nodes[i].copy());
		}

	}
}
```

8 References

1. Naftaly H. Minsky, Towards Alias-Free Pointers, *Proceedings of the 10th European Conference on Object-Oriented Programming*, p.189-209, July 08-12, 1996.

2. B.W. Weide, S.M. Pike, and W.D. Heym, Why Swapping*?*, *Proceedings RESOLVE 2002 Workshop*, 2002, http://people.cs.vt.edu/edwards/RESOLVE2002/proceedings/Weide2.html.

3. M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. Hollingsworth, Reasoning about Software-Component Behavior, *Proceedings ICSR-6, Springer Verlag*, 2000, LNCS vol. 1844, pp. 266-283.

4. SCOTT, M. L. 2000. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, San Francisco, CA.

5. John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. *The Geneva Convention on the treatment of object aliasing*. OOPS Messenger, 3(2), April 1992.

6. J. Bloch. Effective Java. The Java Series. Addison-Wesley, 2001.

7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, 2001

8. G. Kulczycki, J. Vasudeo, Simplifying reasoning about objects with Tako, In *Fifth International Workshop on Specification and Verification of Component-Based Systems*, p. 57-64, New York, NY, USA, 2006. ACM.

9. John Hogg. Islands: aliasing protection in object-oriented languages. In Proceedings of OOPSLA ’91, pages 271–285. ACM Press, 1991.

10. James Noble, Jan Vitek, Doug Lea, and Paulo Sergio Almeida. Aliasing in object oriented systems. In *ECOOP ’99 Workshops, volume 1743 of Lecture Notes in Computer Science*, pages 136–163. Springer-Verlag, 1999.

11. Scott M. Pike, Wayne D. Heym, Bruce Adcock, Derek Bronish, Jason Kirschenbaum, and Bruce W. Weide. 2009. Traditional assignment considered harmful. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (OOPSLA '09). ACM, New York, NY, USA, 909-916.

12. James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming* (ECCOP '98), Eric Jul (Ed.). Springer-Verlag, London, UK, 158-185.

13. Murali Sitaraman, Steven Atkinson, Gregory Kulczycki, Bruce W. Weide, Timothy J. Long, Paolo Bucci, Wayne D. Heym, Scott M. Pike, and Joseph E. Hollingsworth. 2000. Reasoning about Software-Component Behavior. In *Proceedings of the 6th International Conerence on Software Reuse: Advances in Software Reusability* (ICSR-6), William B. Frakes (Ed.). Springer-Verlag, London, UK, 266-283.

14. Wayne David Heym. 1996. *Computer Program Verification: Improvements for Human Reasoning*. Ph.D. Dissertation. Ohio State University, Columbus, OH, USA. UMI Order No. GAX96-12193.

15. Gregory Kulczycki, Murali Sitaraman, Bruce W. Weide, and Atanas Rountev. 2005. A specification-based approach to reasoning about pointers. *SIGSOFT Softw. Eng. Notes* 31, 2, Article 7 (September 2005).

16. David G. Clarke, John M. Potter, and James Noble. 1998. Ownership types for flexible alias protection. *SIGPLAN Not.* 33, 10 (October 1998), 48-64.

17. Douglas E. Harms and Bruce W. Weide. 1991. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Trans. Softw. Eng.* 17, 5 (May 1991), 424-435.

18. Richard B. Kieburtz. 1976. Programming without pointer variables. *SIGMOD Rec.* 8, 2 (March 1976), 95-107.

19. Joan E. Krone. 1988. *The Role of Verification in Software Reusability*. Ph.D. Dissertation. The Ohio State University. Advisor(s) William F. Ogden. AAI8824547.

20. Peter Müller and Arnd Poetzsch-Heffter. 2000. Modular specification and verification techniques for object-oriented software components. In *Foundations of component-based systems*, Gary T. Leavens and Murali Sitaraman (Eds.). Cambridge University Press, New York, NY, USA 137-159.

21. Alex Borgida, John Mylopoulos, and Raymond Reiter. 1993. "…And nothing else changes": the frame problem in procedure specifications. In *Proceedings of the 15th international conference on Software Engineering* (ICSE '93). IEEE Computer Society Press, Los Alamitos, CA, USA, 303-314.

22. Scott M. Pike, Bruce W. Weide, and Joseph E. Hollingsworth. 2000. Checkmate: cornering C++ dynamic memory errors with checked pointers. *SIGCSE Bull.* 32, 1 (March 2000), 352-356.

23. Hoare, C. A. R. 1973. Hints on programming language design. In Proceedings ACM Principles of Programming Languages 1973. ACM Press.

24. http://download.oracle.com/javase/1.5.0/docs/api/java/lang/System.html

25. http://en.wikipedia.org/wiki/Halstead_complexity_measures