A Balanced Verification Effort for the Java Language

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Diego Sebastian Zaccai, M.S., B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Bruce W. Weide, Advisor

Neelam Soundarajan

Paul A. G. Sivilotti
Abstract

Current tools used to verify software in languages with reference semantics, such as Java, are hampered by the possibility of aliases. Existing approaches to addressing this long-standing verification problem try not to sacrifice modularity because modular reasoning is what makes verification tractable. To achieve this, these approaches treat the value of a reference variable as a memory address in the heap.

A serious problem with this decision is that it severely limits the usefulness of generic collections because they must be specified as collections of references, and components of this kind are fundamentally flawed in design and implementation. The limitations become clear when attempting to verify clients of generic collections.

The first step in rectifying the situation is to redefine the “value” of a reference variable in terms of the abstract value of the object it references. A careful analysis of what the “value” of a reference variable might mean leads inevitably to this conclusion, which is consistent with the denotation of a variable in languages with value semantics, such as RESOLVE. Verification in languages with value semantics is straightforward compared to verification in languages with reference semantics precisely because of the lack of concern with heap properties. However, there is still a nagging problem: aliasing can occur in legal programs in languages with reference semantics, such as Java, and it must be handled when it does occur. The crux of
the issue is not in-your-face assignment statements that copy references but rather aliasing arising within (hidden) method bodies. The reason is that the effects of calls to these methods in client code must be summarized in their specifications in order to preserve modularity.

So, the second step is the introduction of a discipline restricting what a client can do with a reference that is aliased within a method. The discipline advertises the creation of such aliases in method specifications and prevents a client from engaging in behavior that would break the abstractions of the objects being referenced, as this would also prevent modular verification. These restrictions allow code to be specified in terms of the abstract values of objects instead of treating the values of references as memory addresses in the heap. Even though the discipline prevents some programming idioms, it remains flexible enough to allow for most common programs, including the use of iterators, without the need for workarounds.

A tool can verify that a program satisfies the provisions of this discipline. Further, it can generate verification conditions that rely only on abstract object values to demonstrate a program’s correctness. These verification conditions can be discharged by the theorem-proving tools currently used to verify RESOLVE programs.
For myself, but also
my parents, Cristina and Jorge,
my wife, Paloma,
and my son, Lucas.
Acknowledgments

This work was only possible due to the guidance and continuous support of my advisor, Dr. Bruce W. Weide. His patience proved to be stronger than what anyone could have requested, and I benefited greatly from it, since without his guidance this dissertation would only be an incomprehensible rambling about mathematical ideas and programming languages. His faith in me still encourages me to push myself farther, and I am thankful for it.

I would also like to thank the members of my committee, starting with Dr. Paul A. G. Sivilotti, who created the teaching materials that were later expanded by Dr. Bruce W. Weide and Paolo Bucci, which prompted me to formalize this work. My interactions with Paul, through classes and informal meetings, have been a highlight of my graduate studies. I would also like to thank Dr. Neelam Soundarajan for his help and insight, his seminars were the best introduction one could have to much of the topics related to this work, I would have never been able to grasp some of these concepts without his help.

I also would like to thank the members of the RSRG group, who had to endure numerous talks in which I tried to distill the ideas presented here into a coherent form. In particular, Dr. Wayne D. Heym, Dr. William F. Ogden, Dustin H. Hoffman, and Dr. Murali Sitaraman, with whom I exchanged plenty of ideas during the course of those talks. They provided a helpful skeptic view to the topics presented here, and
forced me to better formalize and explain my ideas. I also would like to acknowledge The Ohio State University, and in particular, the Department of Computer Science and Engineering. They have opened their doors for me far more times than I could have possibly expected, and their help has been invaluable.

It is difficult to express the enormous gratitude I have towards my parents, who always valued education and learning. Without their constant support and encouragement. I would have never been able to finish my undergraduate studies, much less continue onto graduate school. They have spent more time helping me navigate the pressures of living in an adult world in a foreign place than any son could ever hope for. I could never thank them enough for it. And my sister, whose contagious happiness helped me get through the hardest moments of self-doubt.

My wife Paloma has been the foundation from which I derived strength throughout my graduate studies. She has been both a cheerleader and a therapist, helping me in more ways than I count. She even provided me with what is commonly know as the ultimate push towards graduation: a son. And I cannot be happier, nor more grateful for that. Thank you Paloma, this would have not been possible without you.

---

Diego S. Zaccai

Columbus, Ohio

April 21, 2016
Vita

2009 ........................................ B.S. Computer Science and Engineering, The Ohio State University

2010-present ............................. Graduate Teaching Associate / Graduate Research Assistant, The Ohio State University

2013 ........................................ M.S. Computer and Science Engineering

Publications

Research Publications


**Fields of Study**

Major Field: Computer Science and Engineering

Studies in Formal Methods: Prof. Bruce W. Weide
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Source Listings</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Verifying Code Written in Real-World Languages</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Languages Designed for Verification</td>
<td>7</td>
</tr>
<tr>
<td>1.3 A Middle Ground</td>
<td>8</td>
</tr>
<tr>
<td>2. Software Verification</td>
<td>10</td>
</tr>
<tr>
<td>2.1 RESOLVE</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 The Client View</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 Verifying Client Code</td>
<td>18</td>
</tr>
<tr>
<td>2.1.3 The Implementer View</td>
<td>29</td>
</tr>
<tr>
<td>2.1.4 Verifying Implementer Code</td>
<td>37</td>
</tr>
</tbody>
</table>
### List of Source Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Implementation of a Set object represented by an array in VCC.</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Sample contract: UnboundedNaturalBaseFacility.</td>
<td>14</td>
</tr>
<tr>
<td>2.2 UnboundedDecrement enhancement for UNBF.</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Recursive implementation of the UnboundedDecrement enhancement for UnboundedNaturalBaseFacility.</td>
<td>20</td>
</tr>
<tr>
<td>2.4 The contract for StackTemplate.</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Selected parts of the ListTemplate contract.</td>
<td>32</td>
</tr>
<tr>
<td>2.6 Sample realization: ListRealization for StackTemplate.</td>
<td>33</td>
</tr>
<tr>
<td>3.1 A sample contract for List in KeY.</td>
<td>57</td>
</tr>
<tr>
<td>3.2 Sample Implementation: ArrayList in KeY.</td>
<td>60</td>
</tr>
<tr>
<td>3.3 The contract for java.lang.Object in KeY.</td>
<td>67</td>
</tr>
<tr>
<td>3.4 Parts of the Implementation of MySet in KeY.</td>
<td>69</td>
</tr>
<tr>
<td>4.1 The StackKernel interface in RESOLVE Java.</td>
<td>77</td>
</tr>
<tr>
<td>4.2 The enhanced Stack interface in RESOLVE Java.</td>
<td>83</td>
</tr>
<tr>
<td>4.3 An aliased object in client code obtained from two different methods.</td>
<td>89</td>
</tr>
<tr>
<td>4.4 Some of the methods in the NaturalNumber interface.</td>
<td>91</td>
</tr>
<tr>
<td>4.5 A call to foo with an encumbered repeated argument.</td>
<td>93</td>
</tr>
<tr>
<td>4.6 The QueueKernel interface.</td>
<td>96</td>
</tr>
<tr>
<td>4.7 The java.lang.Iterable interface.</td>
<td>99</td>
</tr>
<tr>
<td>4.8 The java.util.Iterator interface.</td>
<td>99</td>
</tr>
<tr>
<td>4.9 The Standard interface.</td>
<td>101</td>
</tr>
<tr>
<td>4.10 The newInstance method from Queue3.</td>
<td>102</td>
</tr>
<tr>
<td>4.11 The transferFrom method from Queue3.</td>
<td>103</td>
</tr>
<tr>
<td>4.12 The Queue interface with secondary methods.</td>
<td>105</td>
</tr>
<tr>
<td>4.13 A partial listing of the QueueSecondary class.</td>
<td>107</td>
</tr>
<tr>
<td>4.14 A partial listing of the bridge interface for java.lang.Object.</td>
<td>111</td>
</tr>
<tr>
<td>4.15 An implementation of the equals method for Queue.</td>
<td>113</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 A tracing table for the Decrement method of UnboundedDecrement enhancement for UnboundedNaturalBase devoid of any facts about the variable values in each state.</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Obligations.</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Facts from math subtypes and definitions.</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Facts from pre- and post-conditions.</td>
<td>25</td>
</tr>
<tr>
<td>2.5 Facts from pre-, post-conditions, and frame facts.</td>
<td>27</td>
</tr>
<tr>
<td>2.6 A partial tracing table for the Pop operation of the ListRealization of StackTemplate.</td>
<td>38</td>
</tr>
<tr>
<td>2.7 The full tracing table for the Pop operation of the ListRealization of StackTemplate.</td>
<td>39</td>
</tr>
<tr>
<td>2.8 A generic template for a verification condition.</td>
<td>40</td>
</tr>
<tr>
<td>2.9 A VC generated from the ensures clause of the Pop operation for the ListRealization of StackTemplate.</td>
<td>41</td>
</tr>
<tr>
<td>4.1 A sample tracing table for a call to s.length()</td>
<td>85</td>
</tr>
<tr>
<td>5.1 An incomplete tracing table for an equality expression for primitive types.</td>
<td>130</td>
</tr>
<tr>
<td>5.2 An incomplete tracing table for an equality expression for reference types.</td>
<td>130</td>
</tr>
</tbody>
</table>
5.18 A template tracing table for the execution of path through a while statement’s body in a linear tracing table. 

5.19 A complete tracing table for execution path A of Figure 5.8 representing flow chart for Listing 5.1. 

5.20 A complete tracing table for execution path A of Figure 5.20. 

5.21 A complete tracing table for execution path that goes through the loop in Figure 5.20. 

5.22 A complete tracing table for execution path ending in return false in Figure 5.21. 

5.23 A complete tracing table for execution path repeating the loop in Figure 5.21. 

5.24 A complete tracing table for execution path ending in return true in Figure 5.21. 

5.25 A complete tracing table for the enqueue method for Queue3. 

5.26 A complete tracing table for the dequeue method for Queue3.
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The effect of the constraint in UNBOUNDED_NATURAL_MODEL (N) in the abstract state space of integer (Z).</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>The effect of the convention in the representation’s state space of our List, and its correspondence to the abstract state space of Stack.</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Pictorial representation of the abstract value of s in a language with value semantics.</td>
<td>44</td>
</tr>
<tr>
<td>3.2</td>
<td>Pictorial representation of the concrete value of s in a language with value semantics.</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Pictorial representation of the reference value, abstract object-value and concrete object-value of s.</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>Pictorial representation of the concrete object-value of s.</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Pictorial representation of the concrete object-value of s in terms of its reference values.</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>Pictorial representation of the formula (x \mapsto 3, x).</td>
<td>51</td>
</tr>
<tr>
<td>3.7</td>
<td>Pictorial representation of the only possible situation described by the formula (x \mapsto 3, y \ast y \mapsto 3, x).</td>
<td>53</td>
</tr>
<tr>
<td>3.8</td>
<td>Pictorial representation of one of the possible situations described by the formula (x \mapsto 3, y \land y \mapsto 3, x).</td>
<td>53</td>
</tr>
<tr>
<td>3.9</td>
<td>Pictorial representation of the footprint of an ArrayList.</td>
<td>61</td>
</tr>
</tbody>
</table>
3.10 A demonstration of limitations of the contains method for KeY's ArrayList. .......................................................... 64

4.1 A representation of the temporary alias between an argument on the client side and a formal parameter on the implementer side. ........ 73

4.2 A representation of the alias created between an argument to a call and the internal representation of another object. .................. 74

4.3 The class structure of the Queue component family. .................. 95

5.1 A diagram of the proposed tool-chain. ................................. 120

5.2 A complete flow chart (including subexpressions) for the expression in Listing 5.1. .......................................................... 124

5.3 A representation of the AST for an expression with an operator. .... 128

5.4 A picture of three variables with the abstract object value of 42. .... 132

5.5 A representation of the AST for a method call. ........................ 134

5.6 A flow chart for the evaluation of the expression expr1 && expr2. .. 141

5.7 A flow chart for the evaluation of the expression expr1 || expr2. .. 142

5.8 A condensed control-flow diagram for the expression in Listing 5.1. 150

5.9 A representation of the AST for a variable declaration. ............... 160

5.10 A representation of the AST for an if-else statement. ............... 163

5.11 A flow chart for an if-else statement. ............................... 164

5.12 A flow chart for an if statement. ...................................... 164

5.13 A representation of the AST for a while statement. ................. 167

5.14 A control-flow diagram for a while statement in our discipline. ... 169

5.15 A template of for statements. ........................................... 173
5.16 Translation of the template in Figure 5.15 into a while loop. . . . . . . 174
5.17 A template of for-each loop statements. . . . . . . . . . . . . . . . . . 175
5.18 Translation of the template in Figure 5.17 into a while loop. . . . . . 176
5.19 A Component Coupling diagram with seven classes verified by the tools.179
5.20 A complete flow chart (including subexpressions) for the layered im-
    plementation of transferFrom. . . . . . . . . . . . . . . . . . . . . . . . . 183
5.21 A complete flow chart (including subexpressions) for the layered im-
    plementation of contains. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188
Chapter 1: Introduction

Software development has always been a complex endeavor. As computers become faster and cheaper, we involve them in more processes and expect more of them. However, the computers themselves still follow the same basic principles described by Alan Turing [55]. That is, computers today are much faster, but still no “smarter” than a Turing machine\(^1\). This means that the responsibility to provide all the new functionality falls squarely on the shoulders of software and the software engineers who build it.

It may come as no surprise then, that we are writing ever larger and more complex software. However, while our reliance on software has greatly increased, the way in which we develop it has not improved as much. Testing remains the primary form of quality control and it is a very dubious one. Dijkstra [16] has best expressed the limitations of using testing as our only tool for quality control:

Program testing can be used to show the presence of bugs, but never to show their absence.

Given this, some 21% of software maintenance cost is spent correcting errors (possibly found by users) in existing code [9]. We routinely interact with this fact in the form of crashes, vulnerabilities, and constant patches of our computers’ software.

\(^1\)This is demanded by the Church-Turing thesis, is a tenant of Computer Science.
In some cases, the shortfalls of this approach have dire consequences. Recently, a Spanish Airbus A400M crashed after configuration files were mistakenly removed from its computers, causing the engines to stop after take-off and the plane to crash at a cost of 17 lives [32, 54]. It is this type of mistake that reminds us that testing is not enough and reinforces the need for better quality assurance for software. For many, the ultimate solution to this problem might be formal verification. Formal verification has long been on the agenda of computer science [33, 21, 24]: that is, to have a compiler that not only checks the syntax of a program against fixed language rules, but also its semantics against human-provided behavioral specifications. C. A. R. Hoare has officially called this a grand challenge for computer science [25].

For all the promises of formal verification, the fact remains that it is difficult and thus expensive. In order to verify a program, we need to provide a description of its intended behavior. We do this by providing contracts for each method; we describe them in detail in Chapter 2, but for now let’s focus on the language of choice for writing such contracts: mathematics. This is hardly the language of choice for most humans, but the ambiguities inherent in any natural language\(^2\)—and the inability of computers to deal with them—make mathematics the only sensible choice. Mathematical statements allow us to describe software with an incredible amount of precision as well as providing us with the tools of reasoning that mathematicians/logicians have developed over the years.

The choice of math as our language also gives us one of the biggest challenges in formal verification: mathematics uses language that most programmers do not

\(^2\)Look no further than this text to find many.
know well\textsuperscript{3}. In fact, even where the concepts are understood, the language to express them is not standardized. Much of the difficulty this poses is due to the fact that in order to prove that software meets a contract, we first need the contract to precisely express what the software does\textsuperscript{4}. The difficulty in writing contracts has led some groups to verify partial properties of software instead of total correctness [2, 10]. While those efforts are important, from both a technical and practical perspective, this work focuses on \textit{full functional} and \textit{total correctness} verification. That is, this work focuses on verification of code where the behavior of the program is completely described in mathematical contracts and we can prove that the code will terminate for all possible inputs\textsuperscript{5}. We also call this \textit{full verification}. 

When it comes to full verification, work can be classified under one of two different approaches. The first attempts to fully verify code in “real-world” languages—such as C++, Java, or Ada—with all of their complexities. The second defines a research language that can lead to cleaner verification. Both come with their own complementary advantages and disadvantages.

1.1 Verifying Code Written in Real-World Languages

For those attempting to verify code in languages currently used in industry, the main advantage is familiarity: there already is a large base of software and many developers familiar with such code. For example, it is not hard to find programmers familiar with Java and C/C++. The hope behind verifying the software they are

\textsuperscript{3}This is particularly true about discrete mathematics, under which most digital computer programming falls.

\textsuperscript{4}And also what it does \textbf{not} do to the program.

\textsuperscript{5}This is achieved by providing progress metrics for loops and recursive methods.
already writing/using is that the fruit of that effort will be widely used (thus maximizing the return). Examples of people following this path include those trying to verify parts of the Linux Kernel [43]. While efforts like this are important, this type of verification remains too labor intensive\(^6\) to expect it to be widely used in new projects. We envision a world in which all “released” software has been verified, and manually doing this is impractical.

Multiple projects attempt to verify programs in real-world languages in a more automated way. For example, VCC [13] verifies low-level C code. This tool, developed by Microsoft, is capable of verifying some concurrent programs and has even been used to verify parts of the Windows kernel [14]. For all of its accomplishments, the tool’s focus on low-level software makes it less suitable for higher-level applications. Consider that programs written in C++, a superset of C, trend to have two to three times as many errors per line and take six times as long to debug than Java programs [44]. Since C makes information hiding and programming with objects even harder than C++, it is not too hard to imagine why programmers working on higher-level applications shy away from C. The same reasoning applies to VCC: the documentation required to specify the behavior of higher-level programs without objects, with a weak type system, and with unrestricted pointers is just too complex to be practical. A quick look at Listing 1.1, an implementation of a set of int\(^7\) in VCC [12], with pointers and quantifiers in the contracts, as well as the need to explicitly declare witnesses to the truth of universally quantified statements for the

\(^6\)This example was the result of a student trying obtain a master’s thesis, but the difficulty was so high and the number of hours invested in the effort so long that it became a doctoral project instead.

\(^7\)While it might look generic, it is not, and there are good reasons for this that will be explained in Chapter 3.
typedef int Val;

typedef struct Set {
    // abstract value of the set
    _(ghost bool mem[Val])
    // concrete representation
    Val data[SIZE];
    size_t len;
    __(invariant len <= SIZE)
    __(invariant \forall Val v; mem[v] <=>
        \exists size_t j; j < len && data[j] == v)
    // explicit witness
    _(ghost size_t idx[Val])
    __(invariant \forall size_t i;
        i < len ==> mem[data[i]])
    // witness for each abstract member
    __(invariant \forall Val v; mem[v] ==> 
        idx[v] < len && data[idx[v]] == v)
} Set;

void setNew(Set *s)
    _(requires \mutable(s))
    _(writes \extent(s))
    _(ensures \wrapped(s))
    _(ensures \forall Val v; !s->mem[v])
{
    s->len = 0;
    _(ghost s->mem = \lambda Val v; \false)
    _(wrap s)
}

_(pure) BOOL setMem(Set *s, Val v)
    _(requires \wrapped(s))
    _(reads s)
    _(ensures \result == s->mem[v])
{
    for (size_t i = 0; i < s->len; i++)
        _(invariant \forall size_t j; j < i
            ==> s->data[j] != v)
    { 
        if (s->data[i] == v) return TRUE;
    }
    return FALSE;
}

Listing 1.1: Implementation of a Set object represented by an array in VCC. (Cont.)
Listing 1.1: (Cont.)

```c
BOOL setAdd(Set *s, Val v)
(_(maintains \wrapped(s))
(_(writes s)
(_(ensures \forall Val x; s->mem[x] ==
  \old(s->mem[x]) || (\result && x == v))
{
  if (s->len == SIZE) return FALSE;
  _(unwrapping s) {
    s->data[s->len] = v;
    _(ghost s->mem[v] = \true)
    // update the witness
    _(ghost s->idx[v] = s->len)
    s->len++;
  }
  return TRUE;
}
```

code to be verified, illustrates some of the difficulties of using low-level languages for higher-level programs.

Our work is focused on higher level programming in Java. When it comes to verifying programs in Java, four projects stand out: JStar [40], Verifast [29], Jahob [37], and KeY [6]. All of these tools attempt to automatically verify properly documented Java programs, but they do so in different ways. The possibility\(^8\) of aliasing in Java requires much of the verification effort to focus on what is happening in the heap. Since the heap is a shared resource across multiple methods, difficulties controlling aliases necessitate a contract language that can define how the heap is affected by every operation. This usually results in the use of very complex specifications using either Separation Logic [46] in Verifast and JStar, or Dynamic Frames [30] in KeY. Jahob does not use either of these logics, opting instead to require programmers to

\(^8\)Some might even say this is inevitable given Java’s design.
specify in high-order logic how every pointer is affected in a method [57]. We discuss these efforts in more detail in Chapter 3.

1.2 Languages Designed for Verification

There is another possible path for those whose focus is squarely on verifying new software: create a language that forbids constructs that lead to unnecessary complexity. Both Dafny [38] and RESOLVE [23] fall in this camp, but they still have very different approaches.

Dafny is a language built from the ground up with the purpose of verifying pointer-based data structures. In order to do this, it introduces some level of sharing and aliasing, but it does so in a much more restricted way than in C/C++ or Java. Most objects inside of a method are assumed to occupy disjoint parts of the heap, making the verification effort much simpler. In cases where object sharing is required by a programming pattern, the advantages of Dafny over other efforts to verify code in a language like Java become less clear.

RESOLVE avoids all aliasing in the most extreme way: by preventing programs from sharing references altogether. As a result, RESOLVE follows value-semantics. This means every variable has a value that remains independent of what happens to any other variable. This creates clear semantics that make specification and verification easier [34] and modular. In fact, the semantics of RESOLVE are so simple that it is our language of choice later when we introduce the basics of verification. This simplicity does not restrict RESOLVE’s power; quite the contrary, as it supports programming over arbitrary mathematical types. It is worth noting that this simplicity

\[^9\text{The amount of documentation required when this is not the case might make it seem like this is a requirement, but it is not.}\]
with power does not come for free: since the language does not allow for aliasing, and inheritance is limited to specification inheritance [48, 39, 18, 11, 28], and there is no subtyping, most of the design patterns currently used in object-oriented design—such as those in Gamma [20]—are clumsy in RESOLVE. But they are possible: RESOLVE allows one to create modules with reference-like behavior, but there is no syntactic sugar that might encourage this [36, 50]. The apparent limitations of RESOLVE are a result of careful feature selection: any feature that can lead to misuse, or allows for uses that complicate verification, is not included in the language.

1.3 A Middle Ground

The work presented here represents a middle path, in which one can verify programs written in a significant proper subset of a real-world language (Java) while maintaining the reasoning and verification simplicity of RESOLVE. That is, the specification and verification effort does not demand specialized logics to deal with the heap, only basic knowledge of discrete mathematics\textsuperscript{10}. This middle approach restricts what a programmer can do with a Java reference. The methodology uses the standard RESOLVE contracts adapted to Java, but with a new notation that allows procedures to advertise when an object keeps an alias to another object that could be accessible by the caller after the method's execution is over. This advertisement of aliases allows us to ensure that there is a single reference to an object through which its value can be modified.

There are three main parts to the thesis of this work:

\textsuperscript{10}The hope is that this will make the specification and verification effort much more palatable to a regular programmer.
1. It is possible to establish a strong correlation between a main feature of real-world programming languages (reference semantics) and many of the shortcomings of the verification techniques described in Chapter 3.

2. There is a discipline that can be followed in Java that prevents programmers from writing programs that are gratuitously complex to specify and verify, yet allows most of what Java programmers normally do. Further, following this discipline allows for formal verification of Java programs without the previously discussed shortcomings.

3. A fully automated tool can check that a program conforms to said discipline, and can generate relatively simple verification conditions that can be solved by current RESOLVE tools.

Chapter 2 introduces basic concepts of software verification as well as related work. Chapter 3 demonstrates some of the complexities and shortfalls of attempting to verify a program without abstracting away from aliasing mechanisms. Chapter 4 introduces the discipline that allows us to verify Java programs with existing RESOLVE tools. Chapter 5 describes a tool that checks whether a component follows the rules of the previously introduced discipline and, if so, generates verification conditions that can be discharged to prove the code correct.
Chapter 2: Software Verification

In this chapter we explore current efforts in software verification. We introduce the basic concepts of contracts and modeling in RESOLVE, a language with value semantics and strict abstraction and information hiding. These features allow us to introduce the basic features and mechanisms of software verification without having to focus on the inherent complexities of sharing heap storage. Later, Chapter 3 will introduce the basic concepts of separation logic and dynamic frames. These two reasoning tools, combined with the ideas introduced in this chapter, are the ones most commonly used to reason about the shared heap across different methods in most of the systems that attempt to verify programs in real-world languages like Java. Understanding the concepts behind the verification of RESOLVE programs is important as they are used as a basis for the ideas behind software verification throughout this entire work.

2.1 RESOLVE

RESOLVE is a modular object-based programming language designed for full automatic verification. Like any regular object-oriented language, it has user-defined types—from which we can create variables—and operations related to those types that allow for the manipulation of variable values. It differs from regular object-oriented
languages in that operations (methods in regular OO parlance) are not attached to a variable\textsuperscript{11} (an object in OO parlance), and there is no subtyping of programming types and no polymorphism. The RESOLVE language enforces a strict separation between a type’s data representation and its model (the way we reason about it). This strict separation of concerns allows us to reason about components from two different points of view: the client’s and the implementer’s.

The client of a software component is the person\textsuperscript{12} making calls to operations in a contract, while the implementer is the one providing the code that implements a contract. As a general rule, a client is also an implementer of an operation\textsuperscript{13}, but unlike the implementer of a new data representation, the client reasons about variables involved in each operation’s specification in terms of their abstract values, not concerning herself about the way those values are represented. The operations in an enhancement can be implemented this way, providing added functionality to a realization without dealing with its internal complexities. This type of implementation is the focus of the upcoming explanation of how to verify client code.

Ideally, a contract specification comes first (provided by someone in the additional role of specifier) and the implementer is responsible for its correct implementation. We emphasize that is not the client who writes specifications to be used in client code; the contracts for components are provided along with possibly multiple implementations for each. A specifier creates an abstraction layer between the implementer’s data representation for a type—the variable’s concrete value—and the value of the variable

\textsuperscript{11}That is, there is no distinguished parameter. Operation calls in RESOLVE do not follow the syntax of \texttt{object.method()}.  

\textsuperscript{12}We say “person” but client and implementer should be considered roles that software developers play. For instance, a person might be both a client and an implementer at the same time.  

\textsuperscript{13}Otherwise why make a call to another operation at all?
as seen by a client—its *abstract value*. By the same token, using abstraction, the implementer is responsible for hiding the representation details from clients. This natural distinction allows us to explain the intricacies of RESOLVE in two different parts: The Client View and The Implementer View.

2.1.1 The Client View

As previously stated, the way we reason about a variable is to separate its abstract value from its representation. The description of what a variable is and what it can do—called an interface in many OO-languages—is presented in a *contract*. The *contract* provides both the mathematical model of a type—i.e., the way to think about the type’s values—as well as a basic set of operations that allow a client to manipulate it.

In Listing 2.1 we can find the *contract* for `UnboundedNaturalBaseFacility`. We use it to demonstrate some key concepts about interfaces in RESOLVE. The `UnboundedNaturalBaseFacility` defines, as the name implies, an unbounded natural number. By naming convention, parameterized (generic) interfaces are called templates and non-parameterized interfaces are called facilities.

In cases where there is a template parameter, as in the generic stack in Listing 2.4, the contract names its generic parameters inside parentheses right after the template’s name. Much like the meaning of a variable depends on its context, so too does the meaning of the type parameter. When we see a variable name inside a contract, such as `n` in line 19 of Listing 2.1, we are expected to reason about it in terms of its mathematical model. The same is true for a type parameter, e.g., `Item`: though it is a programming type, any appearance of a variable of that type in the contract
refers to its mathematical model. For any appearance of the type in an assertion such as a requires or ensures clause or in any other mathematical context such as a definition, the type name stands for its mathematical model type. This is a subtle yet important distinction that earlier versions of RESOLVE stated explicitly by the use of the notation math[Item]. Over time, it became apparent that this distinction was clear from context for those working in RESOLVE and thus the math[] notation was dropped.

Every contract that defines a type in RESOLVE provides a mathematical model for values of that type. Line 12 of Listing 2.1 provides both a name for a type defined by this facility, as well as the mathematical model. The keyword type in a contract is used to introduce a new programming type. It is always followed by the type's name, UnboundedNatural in this example. The mathematical model is introduced by the keywords is modeled by. In this case, every variable of the type UnboundedNatural can be considered to have a value as described in UNBOUNDED_NATURAL_MODEL.

The requirement to provide a mathematical model is what allows RESOLVE programmers to work over arbitrary mathematical objects such as integers, strings, sets, etc. One can constrain the state space of the model by providing a constraint. We can see this in the definition of the UNBOUNDED_NATURAL_MODEL in lines 4-6 of the contract. The math subtype keyword tells us that the resulting mathematical type is a subset of the possible values of its underlying type. In this case, UNBOUNDED_NATURAL_MODEL is a math subtype of integer (Z). The line exemplar n provides a name for an arbitrary variable of the type in the constraint n >= 0. Thus, a variable modeled by UNBOUNDED_NATURAL_MODEL can be thought of as an
contract UnboundedNaturalBaseFacility
  uses NaturalFacility

  math subtype UNBOUNDED_NATURAL_MODEL is integer
    exemplar n
    constraint n >= 0

  definition RADIX : integer
    satisfies restriction 2 <= RADIX and RADIX * RADIX < MAX

  type UnboundedNatural is modeled by UNBOUNDED_NATURAL_MODEL
    exemplar n
    initialization ensures n = 0

  procedure DivideByRadix(updates n: UnboundedNatural,
                           replaces k: Natural)
    ensures
    #n = n * RADIX + k and
    0 <= k and k < RADIX

  procedure MultiplyByRadix(updates n: UnboundedNatural,
                           clears k: Natural)
    requires
    0 <= k and k < RADIX
    ensures
    n = #n * RADIX + #k

  function IsZero (restores n: UnboundedNatural) : control
    ensures
    IsZero = (n = 0)

  function Radix (restores n: UnboundedNatural) : Natural
    ensures
    Radix = RADIX

end UnboundedNaturalBaseFacility

Listing 2.1: Sample contract: UnboundedNaturalBaseFacility.
integer that has a non-negative value. Figure 2.1 illustrates how a constraint relates the mathematical state space of a math type and a math subtype.

The lines 13 and 14 in the type definition provide key knowledge about the type. Unlike languages with reference semantics such as C and Java, RESOLVE automatically initializes every variable declared in a program to an initial value of its type. In the same way as in the math subtype definition, the exemplar keyword provides a name for any arbitrary UnboundedNatural variable, so that we can describe its initial value. In this case, every variable of type UnboundedNatural has an initial value of zero.

RESOLVE also allows programmers to define mathematical constants and functions. For example, we can see the introduction of the RADIX constant in lines 8-10. This definition tells us that the constant RADIX is an integer between 2 and the square root of the constant MAX defined in NaturalFacility (the contract for which can be found in Listing A.4).
After all the types declared in the contract are defined, we can proceed to define the operations to interact with them. All operations in RESOLVE—as in every other language for verification—have preconditions and postconditions that state what is true before and after a call. The precondition—or requires clause—states what needs to be true before a call is made, while the postcondition—or ensures clause—states what will be true after the operation returns (provided the precondition was met when the call was made).

The responsibilities and guarantees associated with the precondition and postcondition depend on whether you are the client of an operation or its implementer. A client needs to prove that the requires clause of an operation is satisfied before the operation is called, and can assume that if the precondition was met then the ensures clause is true after the operation returns. For the implementer of an operation the dual is true: she can assume the precondition to be true at the beginning the operation’s body, but must prove the postcondition is satisfied at the point of return.

There are two kinds of operations in RESOLVE: functions and procedures. Both are declared with a header that defines their name and formal parameters. The procedure keyword tells us that an operation does not produce a value as a result of its execution, but it may change the values of some of its arguments. The function keyword tells us that the operation acts like a mathematical function: the operation call produces a value, but the arguments are not modified.

Both the function and procedure keywords are followed by the operation’s name and a list of formal parameters enclosed in parentheses. Let’s use the procedure MultiplyByRadix as an example. Each parameter has a three-part declaration: the parameter mode, followed by its name, a colon, and the parameter’s programming
type. For example, the `MultiplyByRadix` operation has two such formal parameters: `n` and `k`. The `n` parameter is an `UnboundedNatural`—the type defined in this contract—while `k` is a `Natural`—the type imported from the `NaturalFacility` contract used in line 2.

A function cannot have a `requires` clause, which means programs function operations are total just like mathematical functions. The list of formal parameters is followed by the return type of the function. For example, `IsZero` returns a `control` value. This is the only type that can be used as an expression in a control-flow statement. That is, only functions returning a `control` value (or expressions created from them using `and`, `or`, and `not` connectives) can be used in the boolean condition that determines which path to take in an `if-then`, `if-then-else`, or `loop` statement. The math model of `control` is, obviously, the mathematical type `boolean`, and its the only built-in type in RESOLVE\(^{14}\).

Unlike a function, a procedure can impose prerequisites on its caller. These are expressed inside the optional `requires` clause, e.g., `0 <= k` and `k < RADIX` in the case of `MultiplyByRadix`. In these clauses, the names of formal parameters refer to the incoming argument values. For example, `k` here stands for the value of an integer—restricted to be between 0 and MAX—before the operation’s execution starts.

Every method has an `ensures` clause. It states what is true of the formal parameter’s corresponding arguments after the operation’s execution is over. Inside the `ensures` clause, the name of a formal parameter stands for its outgoing value, while the parameter prefixed with a `#` sign represents the incoming value (right before the

\(^{14}\)It is not even a fully-fledged type because you may not declare a variable of type `control`.\)
call starts). The **requires** clause never has # signs since all the parameters there can only denote incoming values.

There are four parameter modes in RESOLVE: **restores**, **updates**, **replaces**, and **clears**. They arise from a simple taxonomy of all possible uses of a parameter. A restores-mode parameter is one that has the same value before and after the operation’s execution\(^\text{15}\). It is equivalent to stating that \(x = \#x\) in the ensures clause for a parameter named \(x\). The updates parameter mode tells us that the value of the corresponding argument might be changed and that its incoming value is relevant to the method. One would expect both the incoming and outgoing values of an updates-mode parameter to be part of the ensures clause. The replaces parameter mode is used to state that the incoming value of the parameter is irrelevant to the operation’s behavior. That means that the parameter is only used to produce a value. Finally, clears tells us that the operation uses the incoming value of the parameter, but leaves it in its initial state—as expressed in the initialization clause for the type—upon return.

### 2.1.2 Verifying Client Code

When proving that an operation does what is intended we have two main concerns: that the postcondition of the operation we are implementing, including the parameter modes, is satisfied at the end of the operation body; and that every method called has its precondition satisfied at the point of the call. We need to show those things to be true for every possible execution path. There are many different ways to develop a proof. For example, one can construct a single putative theorem about what the

\(^\text{15}\)This does not necessarily mean that the value remains constant throughout the execution of the operation's body.
contract UnboundedDecrement
  enhances UnboundedNaturalBaseFacility

  procedure Decrement(updates n: UnboundedNatural)
    requires
      n > 0
    ensures
      n = #n - 1
  end UnboundedDecrement

Listing 2.2: UnboundedDecrement enhancement for UnboundedNaturalBaseFacility.

code does by following execution though all possible paths from the beginning to the end of execution, or from the termination up towards the code’s beginning. The reasoning methodology introduced in [24], known as Hoare Logic, is perhaps the most famous system of formal reasoning about programs. It constructs the theorems from the end of a code segment by a “backward sweep” towards its entry point. It has the advantage of familiarity to almost all graduate students in CS. However, it presents a reasoning challenge since most people do not think about programs operating that way.

This work uses a verification method that reasons about code in the same direction (i.e., forward) as code executes. While this is not the standard way of constructing a correctness proof, it is sound and relatively complete (like Hoare Logic [15]) as demonstrated in Heym’s doctoral dissertation [22]. The approach works by constructing a tracing table with four columns where the programming statements are interleaved with mathematical facts about the values of variables in the program at each point. Below, we construct such a tracing table for the UnboundedDecrement enhancement for UnboundedNaturalBaseFacility. This enhancement, shown in Listing 2.2, takes
realization Recursive

implements UnboundedDecrement for UnboundedNaturalBaseFacility

procedure Decrement (updates n: UnboundedNatural)
  decreases n
  variable z, d: Natural
  DivideByRadix(n, d)
  if AreEqual(d, z) then
    d := Radix(n)
    Decrement(n)
  end if
  Decrement(d)
  MultiplyByRadix(n, d)
end Decrement

end Recursive

Listing 2.3: Recursive implementation of the UnboundedDecrement enhancement, presented in Listing 2.2, for UnboundedNaturalBaseFacility, presented in Listing 2.1.

a non-zero NaturalNumber and decrements its value by one. Listing 2.3 is a proposed recursive implementation.

We verify this code by creating a list of all the facts we know before and after every statement in the code. We call the mathematical values of all variables at a point between statements a state of a program. We number the states, starting from zero, in strict lexical order through the code—i.e., regardless of the program’s dynamic execution paths. Then we use a variable’s name subscripted with a state index to denote the (mathematical model) value of that variable in that state. Table 2.1 presents the body of the operation Decrement in an empty tracing table. Note that there is a state index before and after every statement of the method. This includes control-flow statements such as the state indexes 2 and 6 before and after the if-statement. Also note that all states—with the exception of the first and last one—are shared between two adjacent statements.
procedure Decrement (updates n: UnboundedNatural) decreases n
0     variable z, d: Natural
1     DivideByRadix (n, d)
2     if AreEqual (d, z) then
3       d := Radix (n)
4       Decrement (n)
5     end if
6     Decrement (d)
7     MultiplyByRadix (n, d)
8     end Decrement

Table 2.1: A tracing table for the Decrement method of UnboundedDecrement enhancement for UnboundedNaturalBase devoid of any facts about the variable values in each state.

In order to verify the code, we need to populate the Facts column with assertions we know to be true about the variables' values at each state, and the Obligations column with assertions we need to prove in order for the code to be correct.

The Obligations column is the most intuitive one to fill out. As stated before, every time we make a call we have to make sure that the method’s precondition is met. Thus, the requires clause of the method will appear in the state right before the method call (with the formal parameters replaced by their corresponding arguments). Table 2.2 fills the previous tracing table with all the obligations of the code. The obligations 4.1, 6.1, and 7.1 are a result of the requires clauses of the operations called immediately thereafter.

\footnote{This is shorthand notation for Obligation 1 at state index 4.}
Table 2.2: Obligations.

In Table 2.2, there are some obligations that are not part of any requires clause: Obligations 0.1 and 4.2. This is because, in order to prove total correctness, we must show that a piece of code will always terminate for all possible inputs. Termination is proven with the aid of a decreases metric provided by the implementation.

A decreases metric must be a non-negative integer-valued\textsuperscript{17} function. It also needs to be smaller in value whenever a recursive call is made—or by the end of the loop body in a while-do statement. In this case, the metric is simply the value of the formal parameter \( n \). As a result, we want to prove that the metric is non-negative before the operation starts (Obligation 0.1). We also need to prove progress in the method’s body before each recursive call, and thus we are required to prove that the value of \( n \) is smaller in state index 4 than at state index 0 (Obligation 4.2).

\textsuperscript{17}Technically, it is an ordinal-valued function, but non-negative integers usually suffice and are easier to understand for this explanation.
There is no assertion in the Obligations column before the call to \texttt{Decrement} in state 6. RESOLVE, like many modern languages, supports overloading. While both \(n\) and \(d\) are non-negative numbers, they have different programming types. The call after state 6 is not a recursive call, but rather a call to the \texttt{Decrement} method for a \texttt{Natural}, the one bounded by \(\text{MAX}\), as defined in \texttt{NaturalFacility}.

Finally, the goal of verifying a method is to show that it satisfies its postcondition. Obligation 8.1 is the \texttt{ensures} clause for \texttt{Decrement}. Proving all the Obligations of this operation body allows us to conclude that the code is correct. However, we currently have no Facts in the tracing table that would allow us to do that.

Perhaps some of the most intuitive (and repetitive) Facts in this table come from the restrictions of the mathematical models of the variables in scope. When dealing with existing theorem provers, it is far better to reuse mathematical theories than to create a new one. This is one reason the mathematical model of the variables \(n\), \(z\), and \(d\) is \texttt{integer} \((\mathbb{Z})^{18}\). We preserve their type restrictions by making them part of the Facts of our tracing table whenever they are in scope. In Table 2.3 we can see the Facts obtained this way. For example, we can see that in every state \(i\) where \(n\) is in scope, there is a Fact stating that \(n_i \geq 0\).

In Table 2.3, the Facts 0.2-0.4 are not directly related to any variable. They are the result of \texttt{definitions} made in the contracts of \texttt{UnboundedNaturalBaseFacility} and \texttt{NaturalFacility}. \texttt{RADIX} is defined in \texttt{UnboundedNaturalBaseFacility} as an integer satisfying the restriction \(2 \leq \text{RADIX} \text{ and } \text{RADIX} \times \text{RADIX} < \text{MAX}\), and we can directly see this in Facts 0.3 and 0.4. It is worth noting that, although mathematically no different from the value of a variable in a particular state index, the value of a \texttt{natural sub} \(\text{type in the specification.}\)

\footnote{The underlying mathematical type of which their model is a \texttt{math subtype} in the specification.}
Index | Path Cond. | Facts | Obligations |
--- | --- | --- | --- |
pro **cedure** Decrement *(updates n; UnboundedNatural)*
| | decreases n |
| 0 | 1 n₀ ≥ 0 |
| | 2 0 < MAX |
| | 3 2 ≤ RADIX |
| | 4 RADIX × RADIX < MAX |
| variable z, d: Natural |
| 1 | 0 ≤ d₁ ∧ d₁ ≤ MAX |
| | 2 0 ≤ z₁ ∧ z₁ ≤ MAX |
| | 3 n₁ ≥ 0 |
| DivideByRadix *(n, d)* |
| 1 | 0 ≤ d₂ ∧ d₂ ≤ MAX |
| | 2 0 ≤ z₂ ∧ z₂ ≤ MAX |
| | 3 n₂ ≥ 0 |
| if AreEqual *(d, z)* then |
| 1 | 0 ≤ d₃ ∧ d₃ ≤ MAX |
| | 2 0 ≤ z₃ ∧ z₃ ≤ MAX |
| | 3 n₃ ≥ 0 |
| d := Radix *(n)* |
| 1 | 0 ≤ z₄ ∧ z₄ ≤ MAX |
| | 2 d₄ ≤ MAX ∧ n₄ ≥ 0 |
| | 3 0 ≤ d₄ |
| Decrement *(n)* |
| 1 | 0 ≤ z₅ ∧ z₅ ≤ MAX |
| | 2 0 ≤ d₅ ∧ d₅ ≤ MAX |
| | 3 n₅ ≥ 0 |
| end if |
| 1 | 0 ≤ d₆ ∧ d₆ ≤ MAX |
| | 2 0 ≤ z₆ ∧ z₆ ≤ MAX |
| | 3 n₆ ≥ 0 |
| Decrement *(d)* |
| 1 | 0 ≤ d₇ ∧ d₇ ≤ MAX |
| | 2 0 ≤ z₇ ∧ z₇ ≤ MAX |
| | 3 n₇ ≥ 0 |
| MultiplyByRadix *(n, d)* |
| 1 | 0 ≤ d₈ ∧ d₈ ≤ MAX |
| | 2 0 ≤ z₈ ∧ z₈ ≤ MAX |
| | 3 n₈ ≥ 0 |
| end Decrement |

Table 2.3: Facts from math subtypes and definitions.
Table 2.4: Facts from pre- and post-conditions.

mathematical constant does not change during the method’s execution. Thus the facts about a constant need to be expressed only at the beginning of the code and without any subscript.

As was the case with the Obligations before, the most obvious sources of Facts are also pre- and post-conditions. Tracing table 2.4 has the precondition of the method being implemented as Fact 0.5. It also contains the Facts obtained from the called operations’ postconditions. These Facts are obtained by replacing the names of the formal parameters in the \texttt{ensures} clause of the method by their corresponding arguments and state index\textsuperscript{19}.

\textsuperscript{19}We skip the facts obtained by type constraints to save space, though we maintain the numbering so that their absence is noted.
Let’s use the statement `MultiplyByRadix (n, d)` after state index 7 to further demonstrate how an operation’s contract affects the Facts and Obligations. Recall from Listing 2.1 that the operation’s precondition is \( 0 \leq k \text{ and } k < \text{RADIX} \), where \( k \) is the second formal parameter. We can see that the value of the argument \( d \)—the one in \( k \)’s place—right before the call is denoted by \( d_7 \). This substitution yields Obligation 7.1. Fact 8.4 is obtained by a similar replacement, except this time variables preceded by a \# sign refer to state index 7, and non-prefixed variables refer to state index 8.

Fact 8.5 is the result of variable \( d \)’s `clears` parameter mode. It tells us that, after the call, \( d \) it will have its initial value—defined as 0 in `NaturalFacility`. A similar description of the initial value of a `Natural` variable can be seen in Facts 1.5 and 1.6. As previously stated, all variables are automatically initialized when they are declared in RESOLVE.

While the number of Facts in the table is already significant, there are more. Two key pieces of information are missing: what happens to a variable that is not mentioned in a statement, and how does the if-statement affect the value of the variables? We call the facts produced by both of these concerns `frame facts`, and Table 2.5 is the result of adding them as well as path conditions to Table 2.4.

Due to RESOLVE’s value semantics, frame facts surrounding an operation call are surprisingly simple: if an operation does not mention a variable in scope, then that variable’s value remains unchanged. That is how we obtain Facts such as 7.5 and 7.6: the call to `Decrement` does not mention either \( n \) or \( z \), thus their values must remain the same.

In the case of control-flow statements, frame facts need to express how the values of all variables in scope are affected along each execution path. The result of this can
procedure Decrement (updates n: UnboundedNatural)
  decreases n
  variable z, d: Natural
  1 z_1 = 0
  2 d_1 = 0

DivideByRadix (n, d)
  3 n_1 = n_2 \times \text{RADIX} + d_2
  4 0 \leq d_2
  5 d_2 < \text{RADIX}
  6 z_2 = z_1

if AreEqual (d, z) then
  3 d_2 = z_2
  4 n_3 = n_2
  5 z_3 = z_2
  6 d_3 = d_2

  d := \text{Radix} (n)
  4 d_4 = \text{RADIX}
  5 z_4 = z_3
  6 n_4 = n_3

Decrement (n)
  4 n_5 = n_4 - 1
  5 d_2 = z_2
  6 z_5 = z_4

end if
  4 d_2 = z_2 \Rightarrow d_6 = d_5
  5 d_2 = z_2 \Rightarrow n_6 = n_5
  6 d_2 = z_2 \Rightarrow z_6 = z_5
  7 d_2 \neq z_2 \Rightarrow d_6 = d_2
  8 d_2 \neq z_2 \Rightarrow n_6 = n_2
  9 d_2 \neq z_2 \Rightarrow z_6 = z_2

Decrement (d)
  4 d_7 = d_6 - 1
  5 n_7 = n_6
  6 z_7 = z_6

MultiplyByRadix (n, d)
  4 n_8 = n_7 \times \text{RADIX} + d_7
  5 d_8 = 0
  6 z_8 = z_7

end Decrement

Table 2.5: Facts from pre-, post-conditions, and frame facts.
be seen in states 3-6. We can see that states 3-5 have a path condition in them. It arises from the *ensures* clause of the `AreEqual` function. In a state that has a path condition, such as state index 3, the path condition needs to hold for a variable to have a value in that state index. In practice, this means that when trying to prove an obligation inside a state with a path condition, the path condition can be assumed to be a fact.

State 6 uses implications to unify the frame facts after the if-statement, at the confluence of multiple possible execution paths. Facts 6.4-6.6 tell us what is true about the variables’ values if the then-block of the statement is executed: the variables have the values they had at state 5. Similarly, Facts 6.7-6.9 use an implication with the negation of the conditional to express what will happen if the then-block is skipped: the variables have the values they had at state 2. Note that evaluating the if-condition cannot change the values of any variable, which is crucial for this to work with state 2.

We have discussed the origins of all facts and obligations in the tracing table for the `Decrement` procedure. The final step is to create and prove a theorem for each of the obligations in the tracing table along each execution path that reaches that state. Each theorem states that its obligation is implied by the conjunction of all the facts up to that point. Each one of those theorems is called a Verification Condition, and their generation is explained in more detail in Subsection 2.1.4: Verifying Implementer Code.
2.1.3 The Implementer View

The previous section demonstrated what a client needs to know about a component to use it by describing the UnboundedIncrement enhancement for UnboundedNatural BaseFacility. The client was the one writing that operation body. This section uses the ListRealization for StackTemplate to demonstrate how we can map the representation of a variable’s value (in Java terms, instance variables) to its abstract mathematical model value as seen by a client. It is worth noting that, due to RE-SOLVE’s approach that we program over arbitrary mathematical types, the constructs described here map the math model values of the representation variables to the math model value they represent. These math models may, of course, involve completely different math types.

The contract for StackTemplate can be found in Listing 2.4. It is a Stack—Last-In First-Out line—modeled by a string of Item (the parameterized type). Stack maintains the LIFO order by adding and removing elements from the same “end” of the string. There are three operation of interest:

- **Push** puts an element on the left of the string of Item.
- **Pop** removes the left-most element of the string of Item; it requires the string to not be empty before the call.
- **IsEmpty** allows a client to ask whether if the string of Item is empty.

The ensures clause of Pop is particularly interesting, since it is a relational clause. In it, we need to “solve” for the outgoing values of the parameters s and x by finding a concatenation that results in the same value as the incoming value of the stack (#s). The clause happens to have a unique solution, but this fact helps to illustrate the
contract StackTemplate (type Item)
  uses UnboundedIntegerFacility

  math subtype STACK_MODEL is string of Item

  type Stack is modeled by STACK_MODEL
    exemplar s
      initialization ensures s = empty_string

  procedure Push(updates s: Stack,
    clears x: Item)
    ensures
      s = <#x> * #s

  procedure Pop(updates s: Stack,
    replaces x: Item)
    requires
      s /= empty_string
    ensures
      #s = <x> * s

  function IsEmpty (restores s: Stack) : control
    ensures
      IsEmpty = (s = empty_string)

  function Length (restores s: Stack) : Integer
    ensures
      Length = |s|

end StackTemplate

Listing 2.4: The contract for StackTemplate.
difference between = in a contract (mathematical equality) and its meaning in most programming languages (assignment). In RESOLVE, assignment is denoted by the := symbol.

Since the representation of Stack that we use as an example in this section is based on the ListTemplate component, we briefly describe how that component is viewed by a client. The mathematical model for a List is a tuple of two string of Item named left and right. The operation Advance allows a client to move the front object from right to the end of left. In this way, the boundary between the two strings acts like an implicit “cursor.” The operations Insert and Remove can respectively add or remove an element to the front right. The methods Reset and AdvanceToEnd move all the elements to right and left, respectively. This makes them the equivalent of moving an imaginary cursor to the front or end of the List. Finally, the functions LeftLength and RightLength return the lengths of their respective strings. The math model for ListTemplate and the contract for Remove can be seen in Listing 2.5, and the full contract can be found in Listing A.7 in Appendix A.

The implementer of StackTemplate has two related but separate concerns. First, he needs to decide what variables he’ll use for representing a Stack as a List, and find a mapping from the representation to the math model of a Stack. Second, he needs to write code that manipulates the representation in a way that the resulting configuration represents the values each operation contract describes. We focus on the first part of the puzzle in this section and explain how each operation gets verified in Subsection 2.1.4.

In Listing 2.6, we can see the code that builds a Stack from a List. We focus on lines 9-13 of the realization in this subsection. The line type representation for
Stack is (itemlist: List) defines the representation for our Stack. The representation is a record/tuple that holds all variables used in the data representation. In this case, there is only one such variable: a List named itemlist. For uniformity, even a one-variable representation is declared as a record/tuple.

The line exemplar $s$ provides a name to refer to the representation record when writing the convention and correspondence clauses. Much like in the case of the type definition in UnboundedNaturalNumber, since there is no restriction attached to the exemplar, the properties described about $s$ universally apply to every Stack variable with this realization.

The correspondence—also known as an abstraction function—of a realization is the mapping from the values of the representation variables (as seen by the implementer)
realization ListRealization

   implements StackTemplate

   uses ListTemplate
   uses IsPositive for UnboundedIntegerFacility

   facility ListFacility is ListTemplate(Item)

   type representation for Stack is (itemlist: List)
     exemplar s
       convention s.itemlist.left = empty_string
       correspondence function s.itemlist.right
     end Stack

   procedure Push (updates s: Stack,
       clears x: Item)
     Insert(s.itemlist, x)
   end Push

   procedure Pop (updates s: Stack,
       replaces x: Item)
     Remove(s.itemlist, x)
   end Pop

   function Length (restores s: Stack) : Integer
     Length := RightLength(s.itemlist)
   end Length

   function IsEmpty (restores s: Stack) : control
     variable len: Integer
     len := RightLength(s.itemlist)
     IsEmpty := not IsPositive(len)
   end IsEmpty

end ListRealization

Listing 2.6: Sample realization: ListRealization for StackTemplate.
to the abstract value (as seen by the client). That is, it provides a way to interpret the representation into the abstract value. The correspondence can be either a function or a relation, but for this work we limit the focus to abstraction functions (for a study of the consequences of this restriction see [47]). In this particular example, the correspondence function is simple enough: \texttt{s.itemlist.right}. That is, the Stack’s abstract value is the same as the right string of \texttt{itemlist} in its representation.

An astute reader will notice that the correspondence function does not mention the \texttt{s.itemlist.left} string. This would apparently allow an implementer to do anything to it without affecting the perceived value of the Stack. However, such extraneous manipulations would likely result in inefficient use of memory and add a new possible source of errors. The implementer does not intend this, of course. The convention—or representation invariant—allows the implementer to rule out manipulations that would make no sense or artificially complicate the code. Moreover, the convention characterizes the domain of the correspondence function.

The convention is an optional clause for type representations (the default is simply \texttt{true}) that needs to be true right after a representation is constructed for a new variable as well as after every operation’s body. In a language with strong encapsulation like RESOLVE, where the representation of a variable’s value cannot be manipulated by outside operations, this means that the convention can be assumed to hold at the beginning of every operation (except for initialization)\footnote{There is one more rule needed to ensure this: you may not refer to a Stack variable as a client anywhere inside the realization where Stack is represented. This could be stated more clearly by saying that a public operation of a realization cannot make a call to another public method in the realization. Similarly, a private operation in the realization may not take a Stack as a parameter. These rules of RESOLVE are not enforced in most other languages, e.g., commercial object-oriented languages such as Java.}. This inductive pattern of assuming the convention at the beginning of a method’s body and proving
it at the end resembles that of a loop invariant. Significantly for provers, there is no explicit inductive proof required; that is handled at a “meta-level” in constructing the proof rules of RESOLVE.

For ListRealization, the convention \( s.itemlist.left = \text{empty_string} \) simply states that the left string in itemlist is empty at the end of every operation body. This restricts the possible values for the representation that would be valid for the List but nonsensical to the Stack.

Figure 2.2 shows how the convention restricts the space of values for itemlist as well as how its value relates to the Stack’s value through the correspondence. In the bottom set named Representation Value we can observe possible values for a List, and in the top set labeled Abstract Value we can observe how the values that satisfy the convention map to the abstract value of a Stack. We can also see how List values like \((\langle 2 \rangle, \langle \rangle)\) reside outside of the convention.

Missing here is the code to initialize the List for the Stack. We can leave that implicit since the construction of the representation record would create a List with the initial value of two empty strings, \((\langle \rangle, \langle \rangle)\). Not only does this initial value satisfy the convention, it is also mapped by the correspondence function to an empty Stack, \(\langle \rangle\), which is the initial abstract value for stack. This mapping can also be observed in Figure 2.2. The VCs generated for this are not shown but are important for the proof of correctness.
Figure 2.2: The effect of the convention in the representation’s state space of our List, and its correspondence to the abstract state space of Stack.
2.1.4 Verifying Implementer Code

Proving the correctness of the data representation for a type in a realization follows the same rules used to verify client code. That is, we need to create a tracing table that contains the following:

- For each called operation, its precondition is an Obligation in the state right before the call, and its postcondition is a Fact in the state right after.
- The implemented method’s precondition is a Fact at the beginning of the code, and its postcondition is an Obligation at the end.
- Restrictions on the mathematical models of all variables in scope—arising from math subtyping—are treated as Facts.
- All variables not mentioned in a statement are assumed to remain unchanged (frame fact).
- Control-flow statements create path conditions as well as “merging” frames.

The only difference is in the treatment of types whose type representation is defined in the realization. For those types, we need to connect the abstract value of a variable to the values of its representation’s variables. As stated before, this is done through the correspondence function. We also need to ensure that the representation invariant—the convention—is maintained.

As discussed before, in order to avoid unsoundness, the body of a method that assumes the representation invariant cannot refer to the variable’s abstract value inside of the procedure’s body. This rule allows us to worry about the correspondence function as well as the convention at only two points: the beginning of execution
Procedure `Pop (updates s: Stack, x: Item)`

```
procedure Pop (updates s: Stack, x: Item)

0

1  \(s_0 \neq \lambda\)

Remove (s.itemlist, x)

1  \(s_{itemlist_1.left} = s_{itemlist_0.left}\)

2  \(s_{itemlist_0.right} = (x_1) \circ s_{itemlist_1.right}\)

end Pop
```

Table 2.6: A partial tracing table for the `Pop` operation of the `ListRealization` of `StackTemplate`.

and its end. Since RESOLVE does not allow for break statements, the beginning and end of execution are always the first and last states in our tracing table respectively.

Similarly, since we only reason about the abstract value of a variable of the new type we are representing before and after execution of an operation, it follows that any calls made in an operation body deal only with the variables in the representation record of a parameter of that type. For example, in Listing 2.6, all operation bodies refer only to the `itemlist` member of the formal parameter `s`. That is, every mention of `s` references `s.itemlist`, the variable inside `s`'s representation.

In this subsection, we use the code for `Pop` as an example to demonstrate what additional Facts and Obligations are needed when verifying an operation body that has access to the representation of a variable. Table 2.6 is a tracing table with all the Facts obtained in the methodology described in Verifying Client Code.

While the body is only one line long, it is still complex enough to illustrate how the correspondence and convention affect our tracing tables. In Table 2.6, we can identify the `requires` (Fact 0.1) and `ensures` (Obligation 1.1) clauses of the implemented operation, `Pop`. Conversely, the `requires` clause of `Remove`, with the appropriate instantiation, can be seen in Obligation 0.1, and its `ensures` clause can

\(^{21}\)Note that in both cases they only refer to the abstract value of the `Stack`.

38
<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>procedure Pop ((\text{updates } s: \text{Stack}, x: \text{Item}))</td>
<td>1 (s_0 \neq \lambda)</td>
<td>1 (s_{\text{itemlist}_0}.right \neq \lambda)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (s_0 = s_{\text{itemlist}_0}.right)</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (s_{\text{itemlist}_0}.left = \lambda)</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>Remove ((s_{\text{itemlist}}, x))</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1 (s_{\text{itemlist}<em>1}.left = s</em>{\text{itemlist}_0}.left)</td>
<td>1 (s_0 = (x_1) \circ s_1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (s_{\text{itemlist}<em>0}.right = (x_1) \circ s</em>{\text{itemlist}_1}.right)</td>
<td>2 (s_{\text{itemlist}_1}.left = \lambda)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 (s_1 = s_{\text{itemlist}_1}.right)</td>
<td>()</td>
</tr>
</tbody>
</table>

Table 2.7: A complete tracing table for the Pop operation of the ListRealization of StackTemplate.

be seen split into Facts 1.1 and 1.2. There is no need for frame rules since all the variables in scope are used in the single operation called. Similarly, since there are no mathematical subtypes or control-flow statements, this would be the complete tracing table from a client’s perspective.

However, it is easy to see the disconnect between the Facts and Obligations from the operation’s contract and those from the code. While both refer to \(x\), the Item, in the same way, the facts obtained from Pop’s contract only talk about the value of \(s\), a Stack, while the facts in the code refer only to the values of \(s_{\text{itemlist}}\), the List representing it. It is here that we can use the knowledge from the correspondence function to obtain the Tracing Table 2.7.

In Table 2.7, Facts 0.2 and 1.3 are obtained from the correspondence function. With them, we can finally connect in our tracing table the value of the Stack, \(s\), to the value of the variable representing it, \(s_{\text{itemlist}}\). It is worth noting that the correspondence is treated as an unverified claim about the variable’s representation, and as such, it is always a Fact. An erroneous correspondence function would lead to
Prove:

\[
\begin{align*}
\text{Obligation}_i \\
\text{Given:} \\
0: & \quad \text{PathCondition}_0 \rightarrow \text{Fact}_0 \\
1: & \quad \land \text{PathCondition}_1 \rightarrow \text{Fact}_1 \\
\ldots \\
i-1: & \quad \land \text{PathCondition}_{i-1} \rightarrow \text{Fact}_{i-1} \\
i: & \quad \land \text{PathCondition}_i \rightarrow \text{Fact}_i
\end{align*}
\]

Table 2.8: A generic template for a verification condition.

unprovable VCs, but there is no reason to attempt to prove that the correspondence function is “correct”.

Fact 0.3 and Obligation 1.2 are a result of the convention. As previously stated, we assume the convention to hold before execution begins (state 0), but we must prove it holds after the operation returns. It should not be surprising that the convention is not relevant to the method call or the variable’s abstract value (it is only there to prevent an unwise use of the s.itemlist.left string).

The final step in verifying the code is to generate a Verification Condition (VC) for each one of the Obligations in the tracing table. What we need to prove is that for each Obligation, the conjunction of the Facts for each possible execution path to that point imply the Obligation. The Facts about a particular state \( i \) are only relevant if the Path Condition at that state holds. Thus, the facts in each VC are also an implication, of the form \( \text{PathCondition}_i \rightarrow \text{Facts}_i \). The VC is an implication stating that if the conjunction of implications of Path Conditions and Facts, from the starting state of execution up to the state index of the Obligation, then the Obligation.

We can see a simplified VC in Table 2.9. This is the VC for Obligation 1.1. The antecedent, presented under the Given heading, is the conjunction of Facts 0.1-0.3
Prove:
\[ s_0 = \langle x_1 \rangle \circ s_1 \]

Given:

0: \[ s_0 \neq \lambda \]
1: \[ \wedge s_0 = \text{s.itemlist}_0.\text{right} \]
2: \[ \wedge \text{s.itemlist}_0.\text{left} = \lambda \]
3: \[ \wedge \text{s.itemlist}_1.\text{left} = \text{s.itemlist}_0.\text{left} \]
4: \[ \wedge \text{s.itemlist}_0.\text{right} = \langle x_1 \rangle \circ \text{s.itemlist}_1.\text{right} \]
5: \[ \wedge s_1 = \text{s.itemlist}_1.\text{right} \]

Table 2.9: A VC generated from the \textbf{ensures} clause of the \texttt{Pop} operation for the \texttt{ListRealization} of \texttt{StackTemplate}.

and 1.1-1.3. Since there is no Path Condition in either state, the \textit{true} antecedent in front of each fact is simplified away. The consequent, presented under the Prove heading, is Obligation 1.1. A simple substitution of the Givens 1 and 5 in Given 4 yields the desired proof.

We can declare the code to be correct (up to mathematics incompleteness) if and only if all the VCs generated from a tracing table are proven to be true.
Chapter 3: Dealing With the Heap in Modern Languages

In the previous chapter we explained the extremely simple frame rule of RESOLVE: if a variable is not mentioned in a statement, then its value is not changed by it. Unfortunately, this fails in a language such as Java in the face of aliasing. When two references point to the same object, using one to affect the value of the object will change the value we observe via the other reference, too. Not only that, but modifying the references that are part of another object’s representation can also lead to a client breaking the object’s representation invariant, or convention. An example is when a HashSet has its key externally modified, leaving the element in an incorrect bucket.

This means that we need a more complex way to reason about the effects that a call to a method can have on the values of variables. This reasoning is further complicated by the fact that the heap is implicitly shared across all objects, making any allocated object potentially accessible by any method\(^\text{22}\).

The presence of references creates two distinct problems when trying to frame the effects of a method call on variables. The first one is a problem of modularity: we need to realize which variables are modified by a method call, and which ones are

\(^{22}\text{This is not strictly true for languages such as Java that do not allow for pointer arithmetic, but the possible presence of aliases remains a severe problem.}\)
not. The second problem is that we need to realize how the modification of an object contained in the representation of another object can affect the overall program state.

Even though they share a source—aliasing—these two problems trend to create opposing incentives. That is, the methodologies that help reason in a modular way about program behavior do so by also restricting what we can say about the objects these variables point to. Even worse, as we’ll explain later, trying to solve the second problem without limiting access to aliased objects breaks any modular reasoning and makes the verification process intractable.

When dealing with the heap there are three basic approaches in the literature: to explicitly address how every reference is affected in every method, to use separation logic to explain what happens to parts of the heap not mentioned, or to use dynamic frames to describe which parts of the heap can be modified by a method. All three of these methodologies focus on the problem of modularity, with their own shortcomings.

The first option is the one taken by Jahob [37]. It has the distinct advantage of being able to explain data structures without the need to introduce concepts that are extraneous to the program. The problem with this methodology is that it breaks modularity. We might be able to describe what happens to every reference inside a linked list, but it is much harder to do that for a Map from Sets to Lists. Even worse, when programming to an interface as one should, we might not know what representations of maps, lists, and sets we are dealing with, making the task of tracking the internal representations of objects infeasible.

23The Java type Map<Set<E>,List<T>> is our example.

24How are the references arranged inside the representation of an object of the type Map<Set<E>, List<T>>? How does modifying an object that might be stored inside them change the value of the map? How do we know whether the object is referenced inside of it?
3.1 The Meaning of a Variable’s Value

Before going into further detail, we need to make sure we stay consistent with our terminology. In particular, we need to make clear the differences between abstract and concrete values, and how the meanings of those terms change in languages with reference semantics.

3.1.1 Value Semantics

As described in Chapter 2, in a language with value semantics such as RESOLVE, a variable can be seen from either a client or an implementer’s side. The abstract value of a variable is the one defined by the type’s contract; it abstracts away the unnecessary complexities of the data representation and is simply the way a client should think about the value of the variable. The concrete value, or data representation, of a variable is the complete structure used to represent the abstract value of the variable.

Figure 3.1 shows the abstract value of a stack-typed variable $s$. By convention, an abstract value will be presented inside a green cloud. As you can see, the value inside the cloud is an abstraction from how a stack might be implemented since it hides its
Figure 3.2: Pictorial representation of the concrete value of $s$ in a language with value semantics.

representation details while maintaining the information that a client is interested in. That is, the value of the stack $s$ is viewed as the string of strings ("Abc", "Def").

Figure 3.2 is a pictorial representation of a possible concrete value of $s$. Nothing prevents the representation from involving pointers—as long as the pointer-based structures do not "escape" the component, as discussed later. In this particular case, the representation uses an array to store the information inside $s$. It is worth noting that, while the data representation of $s$ includes all the variables inside of it, it does not depend on or expose the internal representations of the strings it holds, nor the representation of the integer used to store its size\(^{25}\). That is, even though the data representation exposes a layer "below" the abstract value of a variable, it does not break the abstraction of the variables used in the data representation by exposing "deeper" layers. In this way, the concrete value of a variable is not the complete memory layout of every component in the data representation laid bare, but only

\(^{25}\)In this case, the size is the length of the abstract value of $s$. This is a naming discrepancy that can easily be avoided, but it will preserved in this presentation to remain congruent with the naming conventions in Java.
a look into “the top layer” of the representation, the complete unraveling of which would generally resemble an onion.

Note that Figure 3.2 presents the value of the size integer inside of a cloud. This is because the number shown is the abstract value, not the data representation\textsuperscript{26}. This level of abstraction is present with every data-type; however, primitive data-types such as integers and characters are simple enough that making an explicit note about the fact that the value presented is an abstraction is too pedantic. For brevity and simplicity, diagrams will omit the clouds when the abstraction is obvious, as it is in the case of integers.

### 3.1.2 Reference Semantics

The difference between a variable’s abstract and concrete values becomes more complex when we consider a language with reference semantics. Most popular languages, such as Java, feature reference semantics and they differ from languages with value semantics due to their pervasive use of references. A reference acts as an extra layer of indirection whenever someone tries to access a “value”. As such, there are three possible meanings for the “value” of a reference variable in such a language: its reference value, its abstract object-value, or its concrete object-value.

Figure 3.3 shows how the three possible values of a variable are depicted in our diagrams. The reference value is shown by the triangle and arrow on top of the variable’s name (s). This is an abstraction from the actual value of a reference, which is really a memory address\textsuperscript{27}. Drawing references this way is a sound practice as long

\textsuperscript{26}Perhaps a 32-bit two’s-complement binary number.

\textsuperscript{27}That is, a word-sized unsigned integer whose value matches the memory location of the first cell of the object the reference points to. For example, 0x00AB3125 in a 32-bit architecture.
as two references are drawn with arrows pointing to the same place if and only if they are aliased. The main advantage of drawing references this way is that it abstracts away the exact memory locations from our diagrams. In languages that do not allow for pointer arithmetic, such as Java, the abstraction afforded by drawing references this way is so basic that there is no need to differentiate between the abstract and concrete values of a reference: the exact memory location is always irrelevant to a user of the language, whether client or implementer of a component.

References are introduced in these languages as a way to access objects. Those objects, in turn, are constructed in ways that require us to distinguish between values as seen by clients vs. implementers (just as in a language with value semantics). We depict these two values in the same way we did for languages with value semantics: an abstract object-value is shown in a green cloud, while a concrete object-value is shown in a dotted blue oval. Further, since there is no distinction between abstract and concrete reference values, whenever the discussion focuses on a reference variable, any mention of an abstract value means an abstract object-value. Similarly, mentions of concrete object-value are simplified to either concrete value or data representation.

It is also worth noting that not all variables need to be accessed through a reference in a language with reference semantics. There are variable types that, mostly due to
their predictable size in memory, can be accessed directly. These are primitive types, and they behave just as a variable would in a language with value semantics. This is why fields such as \texttt{s.size} and the length of the \texttt{s.array} array simply have abstract values displayed. The previous discussion about those being actual abstract values remains true; the cloud is omitted for simplicity.

Note that in Figure 3.3 the arrow from \texttt{s} to its concrete value points to something inside of the oval. This because the reference does not point to the overall structure of an instance, but rather to a particular variable inside of it. Figure 3.4 expands the representation of \texttt{s} to what might be its data representation. The representation of the stack using an array to store its components is similar to, indeed essentially equivalent to, the one presented in Figure 3.2. However, the need to use references to access an object makes the pictures appear considerably different. And the reasoning about the two is different, which is why we want different pictures to refer to when doing that reasoning. Still, there is a clear parallel between the description of a
concrete value in a language with value semantics and the concrete object-value in a language with reference semantics.

Defining the abstract object-value of a variable is a little more complicated. In practice, we want the abstract object-value of the variable $s$ to match the one we gave it in value semantics—i.e., the string $\langle \text{"Abc"}, \text{"Def"} \rangle$. However, the fact that the stack is holding references to strings and not the strings themselves allows for the possibility of defining the abstract value of $s$ in term of the references it holds instead of the abstract object-values of the objects they point to.

In order to be able to include the values of the strings in our stack’s abstract value, we must consider the abstract values of the strings referenced by entries of $s.array$ as part of the concrete object-value of $s$. They need to be included because considering those abstract values to be outside the data representation should prevent us from using them in any abstraction function. Including abstract values inside of a concrete representation is not a trivial matter, since it requires objects to have abstract values in the first place and the programming logic to deal with mathematical types inside a program.

Some programming logics avoid this problem by treating the abstract value of $s$ as a stack of references. Figure 3.5 presents a diagram of what would be considered to be part of the representation of $s$ if we considered it to hold references instead of strings. This is the preferred methodology of the logics described in this chapter. However, even though reasoning about objects in this way has many advantages when it comes to modularity, it does not match the way programmers use abstraction when dealing with complex systems. It also leads to shortcomings that will be discussed in Section 3.3. Due to the use of this unfortunate simplification, the works discussed in
this chapter do not deal well with abstract and object values. Thus, unless otherwise noted, any mention of a variable’s value refers only to its reference value or the value of a primitive type. Similarly, wherever we intend to differentiate an object’s abstract vs. concrete value, we make this distinction explicit.

3.2 Describing the Heap

As noted before, the heap is an ever-present shared store across all objects and methods in most modern languages. Whenever possible, this discussion merges the notations from [30, 46] with the notation of RESOLVE to avoid, as much as possible, re-introducing the same concepts with different names.

The set of all possible states of the heap is denoted by $\Sigma$. We consider a state of the heap $\sigma \subset \Sigma$ to be a finite map between locations, $Loc$, and values. The set of memory locations $Loc$ is assumed to be infinite, but any state $\sigma$ is finite. The
values that a particular cell in \( \sigma \) can map to could be literals—i.e., primitive types in Java—or references to other parts of memory—i.e., the addresses of other cells. In other words, a state is a finite subset of the memory addresses \( Loc \) and the values they hold at a specific time in a program’s execution.

For any state \( \sigma \) of the heap, we define its domain to be the set of used locations, that is \( Used = Dom(\sigma) \). Conversely, the set of unused locations contains all other locations, defined as \( Unused = Loc \setminus Used \).

### 3.2.1 Separation Logic

In order to describe how separation logic helps with framing, we first have to explain its operators. The points-to operator, \( \mapsto \), expresses what a variable (i.e., a cell in the run-time stack or the heap) points to (in the heap). For example, when we say \( x \mapsto 3, x \)" it means that variable \( x \) holds the address of the first of two consecutive cells in the heap, which hold the values of 3 and a reference to the cell itself, respectively. Figure 3.6 illustrates how the heap would have to look for the previous formula to hold. Notice that this formula also states that \( x \) has the address of the only allocated cell in the heap.
The separating conjunction, represented by the * operator, is what allows us to express that two cells are in different locations of the heap. Consider the example \( x \mapsto 3, y \ast y \mapsto 3, x \). As the name implies, the * operator is a conjunction, so for the previous formula to hold, both sides have to be true. That is, \( x \) needs to point to two cells, the first one containing the value 3 and the second one containing a reference to the same address as \( y \). A similar situation needs to be true for the address that \( y \) holds.

What is special about the separating conjunction is that it requires both sides to be satisfied by disjoint parts of the heap. That is, for the separating conjunction to hold, the formulas on each side have to be satisfied by different cells. For the previous assertion to hold for a run-time stack \( s \) and a heap \( \sigma \), written \( \langle x \mapsto 3, y \ast y \mapsto 3, x \rangle_{\text{assert}}(s, \sigma) \), we need to find heaps \( \sigma_0 \) and \( \sigma_1 \) such that they are disjoint and their union yields \( \sigma \). Moreover, the partitions of the heap need to satisfy the two formulas respectively, i.e., \( \langle x \mapsto 3, y \rangle_{\text{assert}}(s, \sigma_0) \) and \( \langle y \mapsto 3, x \rangle_{\text{assert}}(s, \sigma_1) \).

Notice that the requirement for the heap parts to be disjoint is what prevents the stack variables \( x \) and \( y \) from being aliases. That is, if the formula were \( \langle x \mapsto 3, y \land y \mapsto 3, x \rangle \) instead, both the heaps shown in Figure 3.7 and 3.8 would satisfy the formula. That is because in both cases, \( x \) and \( y \) point to a cell containing the value of 3 and a reference to the same address as the other variable.

It is worth noticing that the separating conjunction forces the cells to be separated, but it does not prevent aliasing from within the cells. In our example in Figure 3.7,

\(^{29}\)Here, the run-time stack stack is also a mapping from names to values. But unlike the heap, the stack only holds the local variables declared in a program, which are automatically allocated and deleted.
Figure 3.7: Pictorial representation of the only possible situation described by the formula $x \mapsto 3, y \ast y \mapsto 3, x$.

Figure 3.8: Pictorial representation of one of the possible situations described by the formula $x \mapsto 3, y \wedge y \mapsto 3, x$. The other situation is depicted in Figure 3.7.
the second values of the cells contain references to each other, and yet they are still disjoint. This will become important as we discuss limitations of separation logic.

Having an understanding of separating conjunctions allows us to explain the frame rule in separation logic. Much like RESOLVE’s frame rule, it states that every part of memory not mentioned by a statement remains the same. It ensures this by using a separating conjunction between the facts derived from a statement \((p\) and \(q)\) and the ones obtained from frame properties \((r)\).

\[
\frac{\{p\}c\{q\}}{\{p \ast r\}c\{q \ast r\}}
\]  

Equation 3.1 presents the frame rule for separation logic. In it, no variable—i.e., part of the stack—in \(r\) can be assigned by \(c\), but we also need to prove that none of the heap locations mentioned in \(p\) and \(q\) are part of \(r\). Notice that the requirement for \(r\) to be from a disjoint part of the heap is an onerous one, and, in cases where we cannot show such a strict separation, the proofs need to rely on explicit tracking of every reference in the program.

### 3.2.2 Dynamic Frames and JML*

If separation logic focuses on reasoning about what parts of the heap are not altered by a statement, then dynamic frames [30] is its converse, focusing on what are the parts of the heap that can be modified by a statement or method. It achieves this by declaring fields inside an object—both programmatic and abstract variables—to be members of a region and forcing methods to be framed by a region. By default, each variable is its own region.
We will discuss the basics of dynamic frames in the context of JML* [56], a modification of JML that extends the original accessible and assignable clauses in JML so they work on regions instead of variables. JML* is the specification language used by the KeY verifier [7, 56] to reason about Java programs

In order to better understand how to reason about regions and frames, let’s focus on the example in Listing 3.1. It is the interface—i.e., contract—of a List from [56]. This interface presents a subset of the one in the sample code in version 2.4.1 of the KeY verifying system, available at [4].

As in JML, contracts in JML* can have code in them and some specifications are executable. JML* introduces the type \texttt{locset} into JML. This primitive type—i.e., it does not extend \texttt{java.lang.Object}—stands for sets of memory locations and is used in the writes (assignable) and depends (accessible) clauses that will be explained later. These are analogous to the regions described by Kassios in his dissertation [31]. The use of user-defined regions instead of variables is of particular importance when the value of a variable depends on the composition of other variables: the regions allow for a more granular specification, while abstracting away the details of an internal representation.

Line 3 of the contract introduces footprint as the region that will contain the variables of a List instance. That is, footprint abstracts away the memory locations that could represent the List in all of its possible subclasses. It is also used in the modifies clauses—denoted by the assignable keyword—of methods that change the

\footnote{Some readers might frown upon the lack of type parameters in the implementations of collections. This is not due to bad programming practices, but to a limitation of the specification language. JML* works on Java version 1.4 which does not support generics.}
value of a list, as well as the depends clauses—denoted by the `accessible` keyword—of methods that depend on its value.

Unlike the work from Kassios [30], JML* does not introduce the framing operator, opting instead to rely on modifies and depends clauses. An equivalent effect to the framing operator can be achieved by using a clause of the form `accessible m: s`, where `m` is a model field and `s` is an expression of type `\locset`. Such a depends clause means that `s` frames `m`—i.e., `m` may depend at most on the locations in `s`—as long as the invariants for the objects involved hold in a particular state.

There are three main operators in Kassios work: the preservation operator $\Xi$, the modification operator $\Delta$, and the swinging pivots operator $\Lambda$. Each of these has an analogous operator in JML*. The `accessible` clause on a method replaces the preservation operator. Similarly, the modification operator ($\Delta$) is replaced by clauses of the form `assignable s` for a `\locset`-typed variable `s`.

An expression `fresh(s)`, for a `\locset`-typed variable `s`, is satisfied if and only if all of the locations in `s` do not belong to any object that was allocated in the heap before the call. The expression `new elems_fresh(s)` is an abbreviated form for the expression `fresh(set_minus(s, old(s)))`. It is satisfied if and only if all the locations contained in `s` after the call were either part of `s` before the call or fresh locations. This makes `new elems_fresh` equivalent to the swinging pivots operator ($\Lambda$).

All objects inherit the model field `\inv`, a special field defined in JML*'s version of `java.lang.Object`, and inherited by all of its subtypes. The invariant declarations, such as the one in line 7 of Listing 3.1, as well as those inherited from all super-classes, are joined together in a conjunction to form a `private represents` clause that will
public interface List {

//@ public model instance \locset footprint;
//@ public accessible \inv: footprint;
//@ public accessible footprint: footprint;

//@ public instance invariant 0 <= size();

/*@ public normal\_behaviour
 @ accessible footprint;
 @ ensures \result == size();
/*@
 public /*@ pure */ int size();

/*@ public normal\_behaviour
 @ requires 0 <= index && index < size();
 @ accessible footprint;
 @ ensures \result == get(index);
 @
 @ also public exceptional\_behaviour
 @ requires index < 0 || size() <= index;
 @ signals\_only IndexOutOfBoundsException;
/*@
 public /*@ pure */ Object get(int index);

/*@ public normal\_behaviour
 @ assignable footprint;
 @ ensures \result == (\exists int i;
 @ 0 <= i && i < size(); get(i) == o);
 @*/
 public /*@ pure */ boolean contains(Object o);

/*@ public normal\_behaviour
 @ assignable footprint;
 @ ensures size() == \old(size()) + 1
 @
 @ ensures (\forall int i; 0 <= i && i < size() - 1;
 @ get(i) == \old(get(i)));
 @
 @ ensures \new\_elems\_fresh(footprint);
 @*/
 public void add(Object o);

}

Listing 3.1: A sample contract for List in KeY.
become the invariant inside each dynamic class. By default, the `\inv` clause behaves the same way as the `convention` in RESOLVE. That is, all method contracts (except for constructors) contain an implicit `requires this.\inv` and implicit postconditions `ensures this.\inv` as well as `signals(Exception) this.\inv`.

The lack of a separation between abstract and concrete spaces in JML makes the invariant a combination of multiple clauses in RESOLVE. The invariant includes the information from both the `convention` and `correspondence`, as well as the `constraint` of a type's math model in a `contract` in RESOLVE. This is particularly exemplified by the statement in line 7. It states that the result of calling the List's `size()` method is a non-negative `int`. This would be a restriction on the List's math model in RESOLVE\(^{32}\), though no explicit statement would be necessary since a List would be modeled by a string whose length is always non-negative.

Ideally, we would like to say that the invariant of an object depends only on the variables contained in its data representation\(^{33}\). Line 4 of the contract does exactly that by framing the object invariant of a List by its `footprint`—i.e., the invariant can depend at most on the locations in `footprint`. A common practice in dynamic frames is for regions to frame themselves; line 5 does just that. Thus, the values, contracts and methods that depend exclusively on the `footprint`\(^{34}\) can only be evaluated from the variables included inside it, nothing more. Conversely, any change to a variable that is not part of the `footprint` cannot have any effect on the variables inside of it.

\(^{31}\)This allows a method to `throw` an unchecked exception whenever a call that does not satisfy the `this.\inv` precondition is made.

\(^{32}\)Provided by a `constraint` on a `math type` inside of a `contract`.

\(^{33}\)This is true for a List, but it might not be the case for objects that depend on external values.

\(^{34}\)As noted by the the `accessible` and `assignable` clauses.
This is one of the limitations of this approach when we consider what is not included in the footprint.

Listing 3.2 presents an implementation of the List contract presented in Listing 3.1. We use it to better understand what the footprint region represents for an implementation of a List. The example is a subset of the example in the KeY verification system that was also used in [56].

Lines 3 and 4 declare the only fields that an instance of this object will have: an array of objects (to keep references to the elements stored) and an integer (to keep track of the List’s size). Both of them have default initializations that allow the constructor in line 18 to be correct with an empty body.

Lines 8-12 extend the \texttt{\_inv} model field. They state that the reference array will never be \texttt{null}, and that the value of \texttt{size} will remain between zero and the array’s length\footnote{This particular field is not part of the footprint, yet it is readable because its value cannot be modified by the program. This is the same mechanism that allows the prover to ignore the \texttt{class} references inside every instance, since it is a read-only field that cannot be modified.}. The third invariant tells us that the values inside of a list cannot be \texttt{null}. The fourth invariant might look confusing at first. It is there to prevent implementers from pointing the array reference to an array of a subclass of \texttt{java.lang.Object}. That is, the assertion in line 12 prevents a developer assigning to array an array of \texttt{String[]} . Such assignment is not an error to the compiler, but can lead to a run-time type error when trying to add to the List an object of a type other than \texttt{String}. This is a well known problem with arrays: they are not type-safe due to the fact that arrays are covariant [8].

Line 6 of the class defines what the footprint region will be for this implementation of the List. It includes the reference to array (the array of objects), the
public class ArrayList implements List {

    private /*@ nullable @*/ Object[] array = new Object[5];
    private int size = 0;

    //@ private represents footprint = array, array[*], size;
    /*@ private invariant array != null;
    @ private invariant 0 <= size && size <= array.length;
    @ private invariant (\forall int i; 0 <= i && i < size;
    @ array[i] != null);
    @ private invariant typeof(array) == \type(Object[]);
    */

    /*@ public normal Behaviour
    @ ensures size() == 0 && fresh(footprint);
    */
    public /*@ pure @*/ ArrayList() { }

    public int size() { ... }

    public Object get(int index) { ... }

    public void add(Object o) { ... }

    public boolean contains(Object o) {
        /*@ loop invariant 0 <= i && i <= size &&
        @ (\forall int x; 0 <= x && x < i; array[x] != o);
        @ assignable \nothing;
        @ decreases size - i;
        */
        for(int i = 0; i < size; i++) {
            if(array[i] == o) {
                return true;
            }
        }

        return false;
    }
}

Listing 3.2: Sample Implementation: ArrayList in KeY.
size integer, and all the references inside the array, represented by the short-hand notation array[*]. Figure 3.9 presents a pictorial representation of the footprint region. Note that all the methods that access or assign to footprint in this class can only refer to the variables inside it.

3.3 Variables and Abstraction in Current Tools

As stated before, there are two main things that a verifying compiler needs to do when proving code that works on the heap. First, it needs to figure out which variables’ object values might be modified by a method call and which cannot be modified by it. Secondly, it needs to figure out how aliases within an object’s representation can be manipulated to affect its abstract object value (via possible modification outside of the object’s methods).
These two responsibilities tend to manifest orthogonal requirements. The reasoning methods that help make verification more modular tend to over-simplify the objects themselves. Similarly, reasoning about object values in terms of reference values of variables nested in their representations makes it impossible to abstract away the implementation details of an object from its client, leading to a failure of modularity.

If there is one common thread between separation logic and dynamic frames, it’s their exclusive reliance on variable reference values to describe the state of a program. This reliance on reference values is sensible at first blush: variables have a direct relationship with memory cells so it is easier to separate the regions of the heap that are modified by a statement from those that are not. Because of this focus on reference instead of object values, both logics appropriately address the problem of separating the heap (i.e., the first problem). However, the absence of object values presents a challenge when reasoning about programs in the way a programmer generally intends to. To be able to talk about a list as a string of objects using only the reference values that can be reached from a List variable requires us to do away with the internal objects’ abstraction—we’ll need to see inside their internal representations to understand their object values.

Modularity is a necessary property of a practical verification system [58]. Any attempt to verify a large system without the possibility of abstracting implementation details out of the proof would be intractable\textsuperscript{36}. There is, however, a very serious

\textsuperscript{36}Imagine trying to verify a Java program. If doing so demanded expanding all details of the JVM, as well as, the computer’s architecture because their implementation details “leaked” rather than being hidden by their usual abstractions, the amount of work needed to verify the simplest program would be tremendous.
downside to modeling memory only in terms of reference values and ignoring some object values. Essentially, no object values can be ignored.

### 3.3.1 Variables vs. Objects

In order to understand the limitations of these reasoning tools, let’s re-examine the `ArrayList` class in Listing 3.2. The `List` interface defines a list’s value in terms of reference values—i.e., the memory locations of the objects stored in the list—and not the abstract object values stored in the list. This allows the prover to consider the proof of a `List` implementation in a modular way: defining the abstract object value of the list as a list of references ensures that, once a reference is placed inside the list, its value will not be changed by an alias from outside the class. Before we go any further, let’s note that while the problem will be presented here in terms of JML* and dynamic frames, this problem is similar for separation logic. The example provided in Listing 1.1 has the same shortcomings as the ones in this `List`.

The limitations of using reference values as the values in the `List` become clear once you closely inspect the `contains` method. The function mimics the one you would expect to find in `java.util.List`. There is one major difference: the one verified by KeY only checks for reference equality, while the one in `java.util.List` uses the `equals` method to check for equality. That is, both methods appear to be searching for an object inside of the list, but in reality one of them is searching for an aliased reference while the other is searching for an object value.

The difference between these two is explained by using Figure 3.10. In it, we can see that the objects referenced by `n1` and `n2` have the same abstract values, and that
n2 is an alias to an object in list. As expected, both the KeY and Java implementations of List would evaluate the expression list.contains(n2) to true. However, unlike the java.util implementation of List, the implementation verified with KeY would evaluate list.contains(n1) to false. The way contains is implemented in the KeY version would be considered a beginner mistake in an introductory Java course, but here it seems to be a case of wishful naming. The method might be named like a familiar one that provides useful functionality, but its contract—and thus its implementation—do not express the typical programmer’s expected behavior for a method with that name. And the programmer is correct; the KeY version’s functionality in practice is virtually useless.

The root of this problem becomes clearer when we try fixing it. The solution to this problem is not as simple as replacing the mentions of the == operator by a call to the equals method. This is because the equals method would depend on regions outside of the method’s assignable region: the List’s footprint. Further, in the
case of a generic list, there is no way of describing what the method would depend on! There is no special abstract field that represents the data inside all possible extensions of `java.lang.Object`.

The footprint problem cannot be solved by extending the footprint to include all fields pointed to by the references in `array`—something like `array[*]`—since that only moves the problem one layer over. This approach might be able to work without generics—i.e., if you have a fixed depth to primitive types—but not if you have the possibility of storing an arbitrary nesting of reference types. To see why this would be a problem, consider a list of another container object. The previous solution would include the fields inside of the representation of each list, `array` and `size`, but not the closure of the objects that those lists point to. Thus, the use of `equals` would still violate the `accessible` clause for the method.

Even more problematic is the fact that doing this would make it very difficult to prove that regions are disjoint. Adding an element to a list would entangle the regions of that element to the regions on the list. This would make it impossible to separate the value of the list from the values of the aliases to objects that were already added to the list. That is, the prover would need to understand how modifying an object that was already aliased in the list would change the list's value. This would be impossible without knowledge about the way an object is stored inside of it.

### 3.3.2 The Need for Systematic Abstraction

The problem described above is compounded when we consider the meaning of the `equals` method defined in `java.lang.Object`. One would expect one of the most
commonly used (and overridden) methods in the Java language would have a contract exposing some form of commonality across all of its implementations.

However, that expectation would turn out to be wrong. Listing 3.3 presents the “documented” body of the Object class for KeY. It is worth noting the absence of either abstract fields (to denote the abstract value of an object), regions (to denote a footprint), or contracts for most methods.

The only requirement for any implementation of the equals method is that needs to be pure. While it is hard to argue with that requirement (it is a sensible one), it is hard to believe that there is no common meaning to the method. Later in this work, I’ll argue that the meaning is that the function should evaluate to true if the two arguments have math models\(^{37}\) defined in the same class (or interface) and the arguments have the same abstract object values.

But even the previous definition presents a problem: The default implementation of the method is incorrect. That is, every class that contains an instance field should be required to override the equals method. This is a tough argument when trying to prove “real-world” programs, since it requires a level of structure for classes that greatly exceeds what the average programmer expects to do. It is worth noting that, while this requirement might be considered too restrictive by some, it is part of the best practices recommended by Joshua Bloch [8]. Those who consider it too restrictive simply have not thought deeply enough about it, as Bloch makes clear.

The alternative utilized by KeY is to make class abstraction optional (it is missing in our previous List example). Leaving abstraction as an option makes it impossible to provide a contract for the equals method: There is no abstract object value to

\(^{37}\)The math models referenced here are the same as the ones in RESOLVE.
```java
package java.lang;

public class Object {
    /*@
    public normal_behavior
    assignable \nothing;
    @*/
    public /*@ pure @*/ Object() {}

    public /*@ pure @*/ boolean equals(java.lang.Object o);
    public int hashCode();

    public java.lang.String toString();

    protected void finalize() throws java.lang.Throwable {}
    protected java.lang.Object clone() throws java.lang.CloneNotSupportedException {}
    public final void notify();
    public final void notifyAll();
    public final void wait() throws java.lang.InterruptedException;
    public final void wait(long ms) throws java.lang.InterruptedException;
    public final void wait(long ms, int ns) throws java.lang.InterruptedException;
}
```

Listing 3.3: The contract for java.lang.Object in KeY.

talk about. Even more problematic is the fact that providing a contract that expresses what the default implementation does—check for reference equality—would make any correct implementation of the method unprovable. That is, all overridden implementations would inherit the clause ensures \result \nt null \result == this == o, preventing the method from evaluating to true when both arguments are not aliased but have the same abstract object value (like n1 and n2 in Figure 3.10).
One could easily argue that this is a Java problem and not a problem with the tools used to verify Java code. However, the inability of a tool to work around this problem dooms any attempt at creating a usable generic component that needs to identify objects inside of it—like a Set or a Map.

Listing 3.4 provides an example of how the lack of abstraction leads to entire components with wishful naming. We previously described how the contains method in a List did not perform as expected. The examples from the KeY tool include a Set class. As was the case with List, instead of creating a set of abstract object values, it creates a set of reference values.

Consider using this set to store keywords to be provided by a user from a file. Ideally we would want the set to contain unique representations of each keyword, and the add method appears to do this. However, since it can only talk about reference values, nothing prevents the set from holding two references that point to distinct String objects with the same abstract value. We could take this to the extreme and create a set of strings that only contains references to distinct objects with the same abstract value. Further, checking whether a keyword is in a set would also be likely to fail, since the check for membership would have the same shortcoming as the contains method illustrated in Figure 3.10. The problems in these representations illustrate why there is a need for the strict rules presented in the next section.

The impossibility of restricting aliasing in a language like Java adds another dimension when considering the limitations of this approach. In Java, clients of MySet can continue to hold references to objects after they are added to a set. This “representation leakage” can allow for a client to change the abstract object value of a set
```java
public class MySet {

    private List list;

    //@ private represents footprint = this.*, list.footprint;

    /*@ private invariant list.\inv;
     * @ private invariant \disjoint(list.footprint, this.*);
     * @ private invariant (forall int x, y; 0 <= x this.*);
     * @ && x < list.size() && 0 <= y this.*);
     * @ && y < list.size() && x != y; this.*);
     * @ list.get(x) != list.get(y));
     * @*/
    ...

    /*@ normalBehaviour
     * @ accessible footprint;
     * @ ensures \result == contains(o);
     * @*/
    public /*@ pure @*/ boolean contains(Object o) {
        return list.contains(o);
    }

    /*@ public normalBehaviour
     * @ assignable footprint;
     * @ ensures (forall Object x;
     * @     contains(x) == (\old contains(x)) || o == x));
     * @ ensures \new_elems_fresh(footprint);
     * @*/
    public void add(Object o) {
        if (!list.contains(o)) {
            list.add(o);
        }
    }

    ...
}

Listing 3.4: Parts of the Implementation of MySet in KeY.
```
without ever making a call involving the set itself. This pervasive aliasing, combined with the difficulties created by making abstraction optional, leads to verification of components that might look valuable at first glance. However, these components have severe shortcomings that would make them unusable in the real-world problems they purport to address.

In order to fix these problems we need to be able to describe the abstract object value of a set in terms of the abstract object values of the objects it holds—not just their reference values. This requires abstraction to be a first priority of the reasoning system. We also need to prevent clients from modifying internal objects in ways that can break an object’s internal representation. To do this, without forcing clients to expand the internal representations of all objects they use, requires us to prevent clients from modifying an object once it is part of the data representation of another object—otherwise we would not be able to reason about object values in a modular fashion. The next chapter shows how a system like this would work.

\footnote{To be fair, in the presented examples changes to these abstract object values would not be seen as a problem, since the set is only concerned about the values of the references it holds. But this is also precisely what is wrong with them.}
Chapter 4: Allowing Abstraction in the Presence of Aliases

As the previous chapter illustrated, it is very difficult to verify components in an object oriented language with reference semantics. The presence of aliases, both among the objects that references point to in application programs, as well as within the representations of objects, makes it very hard to modularly reason about a program’s state. This problem is exacerbated if we try to verify a component in a language such as Java, where providing an argument in a method call creates an alias; necessitating that any class that incorporates an object into its representation—as an instance variable—leaves its client with a direct reference to part of its representation. The problem is compounded when we realize that this alias into an object’s representation can potentially be used to change its internal state while bypassing its interface, leaving an object whose value can only be understood by exposing its internal representation. This can negate the advantages of having an abstract object value and force the client to maintain an object’s representation invariant.

The work in this chapter presents a methodology that has two different aims. First, we introduce a formalism based on abstract object values instead of references, allowing structures for program verification—i.e., contracts—in Java to mimic those of RESOLVE. Second, we constrain the problems created by aliasing though advertising
the possibility of aliasing in a method specification (contract) as well as restricting what can be done with an aliased reference.

The problem of having aliases to objects within the representation of another object is that using those references can break modularity. This is because modifying an object’s representation without going through its interface can break its invariants—either its convention or the model’s constraint in RESOLVE’s terminology. Even worse, keeping track of how the value of an object that had its representation changed outside of its interface—even when its invariants are maintained—requires the clients to keep track of the object’s internal representation instead of its abstract value. This type of expansion of an object’s internal representation runs counter to any form of information hiding and abstraction, and would make modular reasoning impossible. This is a problem that has been noted by proponents of separation logic when they propose to “blame the client” [19].

In this work, we focus on solving this problem by preventing unintended aliases from being used to modify objects. That means that regardless of how many references there are to an object, only one of them can have the authority to change the object’s abstract value. Keeping track of aliases is a relatively simple task when the aliasing arises from the direct use of the assignment operator =, since that leads by definition—with the exception of primitive types—to the creation of an alias between the variables involved in the assignment.

The problem becomes more subtle when we are dealing with aliases created inside a method body. Every method call in Java passes arguments into the method’s body by creating an alias of the arguments provided by the client, as illustrated in Figure 4.1. Here, we can observe how the reference value of each parameter is copied
to the corresponding formal parameter of the method, aliasing the two of them. These aliases can be considered to be mostly benign in single-threaded code, since a reference passed as an argument by the caller cannot be used until the method’s execution ends, eliminating the aliased formal parameter. That is, while an alias to an object exists both in the client and the method’s body, the client is prevented from continuing while the method holds the alias to the argument in a formal parameter. Similarly, the aliased reference in the formal parameter disappears when the method terminates before the client can resume execution.

This aliasing is not always as benign, however, since the method body can create an alias to one of its formal parameters. If this alias is to a local variable, the result would be the same, since the variable would also disappear at the end of the method’s

\[39\] The case where the argument is also in scope from within the method body must also be forbidden. This might not a common restriction in single-threaded code, but it becomes important in concurrent code [42], making it a best practice.

Figure 4.1: A representation of the temporary alias between an argument on the client side and a formal parameter on the implementer side.
execution. However, creating an alias in an instance variable\textsuperscript{40} would allow for the alias to outlive the method body’s execution. This type of aliasing, represented in Figure 4.2, is one of the two ways of aliasing that can lead to an object exposing parts of its internal representation to its client. The discipline we introduce in this chapter deals with this type of alias by preventing a client from using a reference to an object once it’s aliased inside a method this way.

The other way in which an internal part of the representation of an object can be leaked is by a \texttt{return} statement. Much like the arguments in a method call, an object gets “returned” to a caller by making a copy of a reference to it. This can create an alias between a part of an object’s representation and a variable in

\textsuperscript{40}Or a static field, but as we will later discuss, that will not be allowed in our discipline.
the client code. As with aliases created by passing arguments, this alias is benign when the returned object is not referenced by any object that would outlive the method body’s execution, since this would result in only a temporary alias; but it is problematic when the reference returned is part of the data representation of the distinguished parameter, or any of the formal parameters to the method.

However, we cannot be as restrictive as we were with parameter aliasing and simply forbid the use of those aliased return values, since the reason to call a method that returns an alias is to get such alias in the first place. That means that, unlike many of the aliases created by parameter passing, the alias returned by a method is the goal of the call and not a side-effect of the language design. This aliased variable can be considered a reference to an object borrowed from another variable. The borrowed object needs to be treated in a special way, preventing a client from changing its value. Similarly, the object needs to be considered ephemeral, and its value “lost” whenever any method is called on the object from which it was “borrowed”. This prevents a client from perceiving a spontaneous change to an aliased object due to the fact that it was modified within the representation of its owner.

4.1 Leveraging Java’s Documentation

The work presented in this chapter is not designed to create a new language for specifications nor a new way of writing them. Instead it aims to, whenever possible, leverage the currently existing tools for RESOLVE and Java. To do this, we use a style of contracts that mimics the one we would expect to see in RESOLVE, but be presented inside of Java’s documentation system: JavaDoc. For example, most of
the documentation keywords in RESOLVE contracts lead to equivalent JavaDoc tags inside a Java program, such as the `ensures` keyword and the `@ensures` tag.

Figure 4.1 presents the interface for `StackKernel`. This interface is used to illustrate how the documentation in Java can be leveraged to document code with contracts similar those in Chapter 2. JavaDocs are the multi-line comments that start with an extra star symbol—i.e., the comments between the `/**` and `*/` delimiters.

Our documentation does not preclude the regular use of JavaDoc. For example, in line 6 there is an optional paragraph in English that informally describes the interface. All standard JavaDoc tags retain their meanings, and their informal nature aimed to explain the contracts in English. Readers familiar with JavaDoc will notice that tags like `@param` and `@return` retain their expected format and content. Since the content of standard tags and explanatory sentences is orthogonal to the formal documentation of a component, they are omitted in future listings. They were only left in this example to demonstrate how they can still be used whenever a programmer desires to do so.

The annotations below the standard JavaDoc are the focus of this chapter. We begin with the parameter modes, and move them into the pre- and post-conditions of a method. We discuss the mathematical models of an object when we talk about how a class should be structured in our family of components.

### 4.2 Parameter Mode Annotations

RESOLVE's parameter modes, introduced in Chapter 2, provided a concise way of covering the taxonomy of possible uses of a parameter. They are also part of our

Their informal nature makes them unimportant to a verification effort, but valuable to a typical user for informal understanding.
package components.stack;

import components.standard.Standard;

/**
 * LIFO stack kernel component with primary methods.
 * 
 * @param <T> the {StackKernel} entries
 * @mathmodel type StackKernel is modeled by string of T
 * @initially default:
 * @ensures this = <>
 * @requires this /= <>
 * @update this
 * @updates this
 * @depletes x
 * @depletes this
 * @requires this /= <>
 * @ensures #this = <pop> * #this
 * @requires this /= <>
 * @depletes this
 * @depletes this
 * @ensures this = <#x> * #this
 * @initially default:
 * @ensures this = <>
 * @depletes this
 * @depletes this
 * @requires this /= <>
 */

public interface StackKernel<T> extends
    Standard<Stack<T>>, Iterable<T> {

    /**
     * Adds {x} to {this}'s top.
     * 
     * @param x
     * @depletes x
     * @depletes this
     * @requires this /= <>
     * @ensures this = <#x> * #this
     */
    void push(T x);

    /**
     * Removes {x} from {this}'s top.
     * 
     * @return the entry removed
     * @depletes this
     * @depletes this
     * @requires this /= <>
     * @ensures #this = <pop> * #this
     */
    T pop();

    /**
     * Reports {this}'s length.
     * 
     * @return {this}'s length.
     * @depletes this
     * @depletes this
     * @requires length = #this
     */
    int length();
}

Listing 4.1: The StackKernel interface in RESOLVE Java.
documentation for Java methods. However, the presence of references, and the aliases that can be created with them, requires the introduction of two new parameter modes.

Unlike RESOLVE, Java does not have a reserved parameter mode slot inside of a list of parameters. Thus, thus parameter modes are presented as tags in the JavaDoc before a method, followed by a list of formal parameter names. The parameter modes remain mutually exclusive despite the change in presentation, so a formal parameter to a method can be listed as having at most one parameter mode.

4.2.1 RESOLVE’s Parameter Modes in Java

RESOLVE’s four parameter modes are represented by similarly named tags in our documentation: @updates, @replaces, @restores, and @clears. Each one of them can appear as an optional tag in a method’s JavaDoc followed by a list of formal parameter names—which might include this in the case of instance methods. As in RESOLVE, all four of them are concerned with abstract object values and not reference values.

The @restores parameter mode is considered the default parameter mode. That is, unless otherwise noted, each parameter to a method is expected to have its abstract value after the call restored to its abstract value before the call. Since it is the default mode, it is never explicitly mentioned in a contract.

The other three parameter modes borrowed from RESOLVE behave in the expected way. A parameter listed after a @replaces tag has its outgoing abstract value updated, and the method’s results are independent of its incoming abstract value.

42Remember, that does not necessarily mean that the concrete object value of the parameter is unchanged, nor that its abstract object value cannot be changed inside of the method (so long as its changed back before the method body’s execution ends).
A parameter listed after a @clears tag has its outgoing abstract value set to an initial value for its type. Here, since Java does not automatically initialize variables, we need to specify what that value is. The default initial value of an object is the abstract value—or one of a set of values—produced by a call to the class’ constructor that takes no parameters. We go into more detail about this in Section 4.4.1. The @updates parameter mode remains the most general one, since it states that the outgoing abstract value of a formal parameter can be modified by a method in a way that depends on its incoming value.

It is worth noting that not all Java types can be used with all of these parameter modes. Both references and primitive types are passed to a method by copying. Since a primitive-typed formal parameter in a method is a separate copy from the variable used as an argument by the client, the caller would never be able to observe any changes made to a formal parameter inside the method’s body. As a result, the only possible parameter mode for a parameter of a primitive type is @restores.

A similar problem arises with immutable types. An object of an immutable type, once allocated, has a concrete object value that cannot be changed\(^{43}\) (and thus its abstract object value cannot be changed either). This also makes changing the value of a formal parameter of an immutable type impossible. Thus, it seems the only parameter mode they can have regarding their abstract object value is @restores. However, the references to objects of immutable types can also be affected by the @depletes parameter mode introduced later.

Due to Java’s parameter passing mechanisms, a client is always unaware of any assignment to a formal parameter within a method. Thus, assigning a value to a

\(^{43}\)This is an oversimplified definition. Depending on whose version of “immutable type” is adopted, the restrictions on the value of an immutable object might be even much stronger than that.
formal parameter is, at worst, a misguided attempt at providing a result to a client or, at best, a way to avoid declaring a reference of the same type as a formal parameter. One of those uses is wrong, while the other is a questionable practice. It should not be a surprise then, that it is considered a best practice to never assign to a formal parameter [1], and this best practice is enforced in our discipline.

4.2.2 Parameter Modes for Reference Values

As we stated before, RESOLVE’s four parameter modes of @updates, @replaces, @restores, and @clears present a taxonomy of how an argument of a call might be used in a language with value semantics. However, Java is not such a language and even though our contracts focus on the abstract values of variables, sometimes there is a need to advertise the changes to program state that references make possible.

There are two ways in which a method can create an alias that outlives its execution: by aliasing a reference that is part of a data representation to a formal parameter, or by returning an alias to an object that is part of a data representation. Two different parameter modes aim to alert a possible caller of these situations.

The Depletes Parameter Mode

The @depletes parameter mode is used to specify that a method might create and save (within a data representation) an alias to the argument. This possibility is depicted in Figure 4.2. This parameter mode does not describe any changes to the value of either the reference or the object, but it does denote that the reference is no longer readable by the client after the call. This implies, the client cannot “see” the value of the referenced object after the call. We call references in this state depleted.

This also applies to static fields, but aliasing them is not allowed in our discipline.
and they behave in a similar fashion to uninitialized variables in Java: the only way for a depleted variable to become usable again is to assign it a value. And, as long as a reference is depleted, it cannot be dereferenced. That is, it is not possible to use the dot operator to access the object it references\(^{45}\), nor can it be used inside any expression—which excludes both using it on the right-hand side of an assignment statement as well as an argument to a method call.

The \texttt{@depletes} parameter mode restricts the possible interpretations of a program’s state in Java, but it does not change the meaning of a Java program. This is true because having a depleted reference simply prevents a program from using a part of its state, but it does not change any of the values visible by other variables (both in the concrete and abstract sense). This restriction to the usability of a variable does not change the meaning of a correct program in our discipline with respect to a regular Java program. Executing a program that follows this restriction in a regular Java environment will yield the expected result. It is worth noting that the converse is not true, a correct Java program can be declared as illegal in this discipline.

The restrictions imposed by the \texttt{@depletes} parameter mode address what has long been a problem in Java representations. For example, consider the disclaimer in the `java.util.Set` documentation in the Java 1.7 API [3]:

> Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

\(^{45}\)That also prevents access to its fields and methods.
The \texttt{@depletes} notation tackles this problem by disallowing the mechanism that creates it: modifying the data representation of an object through aliases to data representation objects held in client code.

The Encumbers Relation

The encumbers relation is used to advertise that the result of a return statement exposes part of an object’s data representation. As such, it is not a parameter mode but a parameter effect. It is different from other parameter modes in that it affects the result of a function, and not its arguments. Since the function is exposing a part of the data representation of one of its parameters, it is usually the case that the returned value is encumbered by \texttt{this}. But that is not always the case, as even a static method is capable of returning an alias to the data representation of one of its formal parameters. To account for this, the \texttt{@encumbers} notation provides the identifier of the formal parameter to whose data representation the resulting value might be aliased.

Let’s study the contract for \texttt{top()} in Listing 4.2 to see how we document this potential source of aliases in our contracts. The \texttt{top()} method returns an alias to the top element of a \texttt{Stack}. This alias is documented by the line \texttt{@encumbers top via this}. As the documentation suggests, the value returned by \texttt{top} is encumbered by \texttt{this}.

Before describing how it can affect a program, let’s define the properties of the encumbers relation. The encumbers relation \((\rightarrow)\) is a binary relation on variables in the program state. We write \(a \rightarrow b\) as shorthand for “\(a\) encumbers \(b\)” or “\(b\) is encumbered by \(a\)”.

It is transitive, that is, if \(a \rightarrow b\) and \(b \rightarrow c\), then \(a \rightarrow c\). It is also asymmetric, so if \(a \rightarrow b\) then \(\neg(b \rightarrow a)\). From the relation’s asymmetry we
```java
package components.stack;

public interface Stack<T> extends StackKernel<T> {

    /**
     * @encumbers top via this
     * @requires this /= <>
     * @ensures <top> = this[0, 1)
     */
    T top();

    /**
     * @depletes x
     * @updates this
     * @requires this /= <>
     * @ensures
     *     <replaceTop> = this[0, 1) and
     *     this = <#x> * #this[1, |#this|)
     */
    T replaceTop(T x);

    /**
     * @updates this
     * @ensures this = rev(#this)
     */
    void flip();
}
```

Listing 4.2: The enhanced Stack interface in RESOLVE Java.
can also see that $\mapsto$ is both irreflexive—i.e., $\neg(a \mapsto a)$—and antisymmetric—i.e.,

$$(a \mapsto b \land a \neq b) \Rightarrow \neg(b \mapsto a).$$

Since the encumbers relation is both transitive and asymmetric, it follows that there can be no cycles in the encumbers relation\(^{46}\).

When client code contains the statement $x = s.top()$, the method returns an alias to part of the data representation of $s$. The client has the knowledge that the resulting value in $x$ is “shared” with the variable $s$. This is the relationship that encumbers captures. The relation is directional because we consider the variable resulting from the method to be “borrowed”, and its possible uses to be limited by the encumbering variable $s$. The abstract value of an encumbered variable may not be altered, since doing so could potentially alter another variable’s data representation. This means that an encumbered variable can only be used in method calls with a \texttt{@restores} parameter mode. This convention prevents a client from modifying an object that might be part of another object’s representation. It also forbids the client from aliasing it to another data representation, since a method that creates such an alias to a parameter would have \texttt{@depletes} as its parameter mode.

Table 4.1 presents a tracing table similar to those in Chapter 2. These tables are further discussed in Chapter 5, but for now let’s focus our attention on the Depleted and Encumbered columns. The Encumbered column tracks the encumbers relation between variables. While the relation is transitive, not all possible related pairs are written, so if we have $a \mapsto b$ and $b \mapsto c$ at a particular state, the table does not redundantly present $a \mapsto c$. The Depleted column keeps track of variables that are depleted at a state index. An invariant of these tables is that the depleted variables cannot appear inside of the Path Conditions, Facts, or Obligations columns.

\(^{46}\)We explain how this property is maintained later in this Section.
Table 4.1: A sample tracing table for a call to `s.length()`. Notice that the previously encumbered `top` variable becomes depleted as a result of the use of `Stack` variable `s`.

In state index 1, we know \( s \to x \) as a result of the call to `top()`. This means that the value of \( x \) is dependent on the value of \( s \), and any changes to the data representation \( s \) could potentially result in changes to \( x \). The call to `length()` after state index 1 restores the abstract object value of \( s \), yet there is no guarantee that its data representation remains unchanged after it. Thus, by conservative rules, \( x \) becomes depleted at state index 2. That is, every time an encumbering variable is used as an argument to a call, all the variables encumbered by it become depleted. This rule balances the need of a client to access a part of the `Stack`'s data representation with the need to maintain the data representation's integrity.

This rule might sound too restrictive for a `Stack`. But, it would pose a burden on the implementer of a data structure to require the implementation to preserve the values of any object in it that can be potentially exposed through a return statement. For an example of why relaxing this reasoning rule could be complicated, consider a representation of a rational number where the numerator and denominator are `NaturalNumbers^47`. Also, assume that there is a method that allows the client to

^47The interfaces involved are not important for this example, but the fact that the data representation contains mutable types is.
obtain an encumbered reference to either part of the fraction. Checking whether two rational numbers are equal would restore the value of both, i.e., the quotient between the two NaturalNumbers would remain constant for both rational numbers. Yet in doing the check, an implementer could “simplify” the stored fraction to lowest terms, and possibly change the value of any variable encumbered by it. It is the possibility of changes to the concrete object value of a variable that can happen even when the abstract object value remains constant that forces us to be conservative in our approach: any use of an encumbering variable depletes all the variables encumbered by it.

For another example, consider the statement \texttt{s.push(s.top());}, where \texttt{s} is a Stack. Our tool would separate the nested expressions by assigning the result of \texttt{s.top()} to a temporary variable, say \texttt{@temp}, and using that temporary variable as the argument to \texttt{push}. Thus, the statement would be split into \texttt{@temp = s.top()} and \texttt{s.push(@temp);}. Our tools would note that \texttt{s \mapsto @temp} before the call to \texttt{push}, and deplete it before the call (since \texttt{s} is a parameter to the call). This in turn would make the one of the arguments to \texttt{push} a depleted variable, which is an error. This error would prevent some of the problems explained in Sections 3.3.1 and 3.3.2.

It is worth noting that the dynamic encumbers relation can only have the shape of a chain. That is, there is no branching or tree structure to the local encumbers relationship generated in our discipline. This is due to the fact that Java allows for the return of, at most, a single variable (creating at most a single encumbers arrow at a time). Further, the creation of said arrow requires using the variable which encumbers the resulting variable, depleting any other variable that was encumbered by it beforehand.
4.2.3 Implications of the New Annotations

The previous sections described what the @depletes and @encumbers relations are about. There are implications for the three subjects involved in software development: the client, the implementer, and the specifier. Similar efforts have already been done for the four standard parameter modes in RESOLVE, and the resulting rules can be found in [52, 26, 51].

From an implementer’s perspective, the presence of a @depletes parameter mode does not require that the parameter is depleted or aliased to the data representation within a method’s body. The converse is true: the absence of a @depletes parameter mode prevents the implementer from depleting or aliasing the formal parameter. Thus, any method that might need to add to the data representation of a formal parameter another formal parameter, or part of it, must mark the second one with a @depletes parameter mode.

For the client, it is irrelevant whether the implementation of a method with a @depletes parameter mode actually depleted the formal parameter. Given the possibility of an alias, the client has to assume that the variable was depleted. Thus, the variable that was marked as potentially depleted by a method remains unusable until it is assigned a new value. That is, the client needs to take the worst-case approach and assume that a parameter that is marked as @depletes is always depleted by the method. The specifications should account for this fact, too. So, as a rule, @ensures clauses should refrain from expressing anything about the outgoing value of a parameter that is listed as @depletes.

The presence of an @encumbers annotation in a contract is similarly not restrictive for an implementer. It gives the implementer the option to return an encumbered
variable, but it does not require her to do so\textsuperscript{48}. The converse is also true: if the returned variable is does not have an \texttt{@encumbers} modifier in the method’s contract, then the returned variable can never be encumbered by the implementer. Further, the variable can only be encumbered by the parameter that is singled out in the contract, and not by any other formal parameter. That is, the returned variable can only be a part of the data representation of the formal parameter named after the \texttt{via} keyword in the \texttt{@encumbers} clause.

For a client, the resulting variable—i.e., the one that the result of the function is assigned to—will be restricted both in its access and availability. The access restriction stems from the requirement that any use of the variable is done with the \texttt{@restores} parameter mode. This effectively prevents the encumbered variable from having its abstract object value modified (and thus, inadvertently changing the value of another object’s data representation). It also prevents aliasing within and across different data representations, since the methods that would create such aliases would mark the formal parameter as having a \texttt{@depletes} parameter mode and not \texttt{@restores}.

The availability restrictions on the encumbered variable are a result of its depletion whenever an encumbering variable is used. This makes the encumbered variables rather short lived, but prevents clients from holding multiple references to the data representation of the same object that were obtained through different methods. For an example of why this could be problematic, consider the code in Listing 4.3. After both statements finish execution, references $x$ and $y$ would seem to point to the same object. In our discipline, $s \mapsto x$ right before the call to $s.pop()$. Thus, the variable $x$ is depleted by the time it is aliased with $y$.

\textsuperscript{48}Though, it is worth noting that it would be rare to have documentation that allows for an encumbered variable to be returned when it is not necessary.
Listing 4.3: An aliased object in client code obtained from two different methods.

1 ...  
2 x = s.top();  
3 y = s.pop();  
4 ...  

But consider what would be the effect of not being conservative with our encumbered variables, and allowing them to “live” after their encumbering objects are accessed. After `pop`, the variable `y` would no longer be associated with `s`, but `x` would be! What’s worse, any changes to the object referenced by `y` would affect `x`, a seemingly unrelated variable. Our approach maintains the more restrictive, but safer, approach of depleting the encumbered variable `x`, preventing the client from observing this problem altogether.

### 4.3 Abstract Object Values, Not References

The pre- and post-conditions might look very similar in RESOLVE and RESOLVE Java. Yet for all the similarities, there is one major difference between the two. As discussed in Chapter 3, they have very different ways of treating a variable. The fact that Java has reference semantics raises the question of what it means when the ensures clause of `length` for a stack states that `length = |this|`.

It should be clear that Java treats any mention of `this` as a reference to an object. However, we are interested in abstract object values in contracts. In all `@ensures` and `@requires` clauses, any mention of a parameter denotes not its reference value, but rather its abstract object value.

It is worth noticing that, while this simplifies reasoning about contracts, it creates some serious limitations when dealing with a reference. For starters, a `null` reference
does not have an object associated with it. Even though the reference to this could never be null in a method call, the other references might be, leading to a contract that is unsatisfiable by design.

Our discipline’s solution to this problem is to forbid clients from using a null reference as an argument to any call. That is, any call that uses a null reference as an argument is, by convention, violating the pre-condition of the method. While this convention is not enforced as standard Java practice, it is not uncommon either: multiple classes inside the Java API require parameters to be non-null. For example, the class java.util.TreeSet prohibits its clients from adding null elements to it, and attempting to add a null reference results in the unchecked NullPointerException [3]. Our approach is to make this a convention for all methods.

Not all variables in Java are references. Variables of primitive types—byte, short, int, long, float, double, boolean, or char—do not have a reference associated with them. They instead behave the same way a variable would in a language with value semantics. Thus, variables of primitive types remain consistent with our contracts, since the contracts can only refer their abstract values.

4.3.1 Repeated Arguments

One of the problems with referring only to abstract values in a language with reference semantics is that, while not the focus of a program’s state, the reference values can provide important information about the arguments to a call. This becomes apparent when we examine the problem of repeated arguments. Consider the contract for multiply in Listing 4.4.
package components.naturalnumber;

public interface NaturalNumber extends Comparable<NaturalNumber>,
        NaturalNumberKernel {

    ...}
The `NaturalNumber` type is modeled in the same way as RESOLVE’s type specified by the same name: a mathematical integer whose value is greater than or equal to zero. The model is described in the `NaturalNumberKernel` interface that is extended by the one in Listing 4.4; the reasoning behind this division is explained in Section 4.4.

The `multiply` method updates the value of the distinguished parameter `this` to its incoming value times `n`. Consider the statement `m.multiply(n)`, and evaluate the `@ensures` clause `this = #this * n` when `m` and `n` are aliases with an abstract object value of 2. The conclusion is that the abstract object value of `m` should be 4 after the call. Similarly, the abstract object value of `n` should be restored to 2. That is, the object that both `m` and `n` reference should have the abstract object value of both 2 and 4 after the call. This makes the contract unsatisfiable.

One could argue about whether being able to express reference values in the contract would solve this problem, but the problem lies in the presence of a single object with multiple names: aliasing. There are many ways of deciding what a method should to do in the presence of repeated arguments. Some solutions to this problem are described in [35]. However, in our discipline we hold clients responsible for these types of calls. As such, clients are forbidden from using repeated arguments in a method call. We believe this to be the simplest approach.

The previous example of a repeated argument was rather obvious. However, the presence of aliases to part of the data representation of an object can lead to similar reasoning problems. Consider the call to `foo(s, m)` in Listing 4.5. In it, the behavior of the `foo` method is completely irrelevant to our example, but the variable state before it is not. The call to `s.iterator()` produces an encumbered variable, thus after line 10 `s ↦ it`. The call `it.next()` on line 11 also returns an encumbered
import components.stack.Stack;
import java.util.iterator;

public class Test{

  /**
   * @requires |s| > 0
   */
  public static void repeatedArguments(Stack<NaturalNumber> s){
    Iterator<NaturalNumber> it = s.iterator();
    NaturalNumber m = it.next();
    foo(s, m);
  }
}

Listing 4.5: A call to foo with an encumbered repeated argument.

variable, so it↦→ m after the call. Thus, right before the call to foo(s, m), we know that s↦→ it, and it↦→ m.

As we previously discussed when we introduced parameter modes, this call violates the rules for a depleted variable: the use of s in the parameter list of foo depletes both it and m before the call, making the second parameter to foo a depleted variable and violating our discipline. Even ignoring that, this call would have a repeated argument. Since s↦→ it and it↦→ m the transitivity of the encumbers relation implies that s↦→ m. Thus, there is a possibility that m is aliased to a part of the data representation of s. This makes m a different name for a part of s and, therefore, a repeated argument. Similar arguments can be made with array elements (when there is no proof that the indexes are not equal), as well as variable accesses through the dot operator (s.a). The key here is that encumbered variables, much like aliased ones, are treated as repeated arguments in the most conservative way: we might not know that m is an alias to a part of the data representation of s, but the possibility
that it might be is enough to mark this call as a violation of the repeated argument rule.

4.4 The Structure of a Type

Throughout this chapter we have introduced interfaces that have names ending in “Kernel” or that extended those. This is not an accident but rather a result of a deliberate focus on having contracts and components allow for maximum reusability while following a strict separation of concerns. The “primary” methods in Kernel interfaces are separated from the “secondary” methods in extensions to create a strict distinction between the methods necessary for a type to be usable and those that can be implemented by layering code on top of other methods for the type. We use the contracts for Queue in our examples but the ideas demonstrated here apply to any component designed under our discipline. It is worth noting that not all components can follow this strict structure, and when that is not possible methodologies like those introduced later in Section 4.5 help bridge the gap.

Figure 4.3 presents a sample dependency structure of a component in our discipline. In it, interfaces are shown by blue ovals, classes are shown by blue rectangles, and abstract classes are shown by blue rectangles with rounded corners. The arrows labeled with ‘e’ represent the extends relationship between either two classes or two interfaces. Arrows labeled with an ‘i’ represent the implements relationship that can only appear between a class and an interface.

The structure presented here is designed to provide maximum reusability of general contracts such as those provided in Standard. It also separates into a Kernel the
basic methods necessary to implement and use a type, from all of its other methods. We use the \texttt{Queue} family of components to explore this structure in detail.

4.4.1 The Kernel Interface

The kernel interface defines the mathematical model of a type, as well as its core methods—i.e., the minimum set of methods necessary to manipulate the values of this type\footnote{Sometimes there might be an extra method of two in a Kernel for parsimony of design, but we try to avoid it.}. Listing 4.6 shows the \texttt{QueueKernel} interface.

The documentation for the interface in lines 5-12 contains information about the type defined by the \texttt{QueueKernel} interface. The mathematical model of an object of type \texttt{QueueKernel} is defined in line 6. The type definition is introduced by the

\begin{figure}
\centering
\includegraphics[width=\textwidth]{class_structure.png}
\caption{The class structure of the \texttt{Queue} component family.}
\end{figure}
```java
package components.queue;

import components.standard.Standard;

/**
 * @mathmodel type QueueKernel is modeled by string of T
 * @initially default:
 * @ensures this = <>
 * @iterator ~this.seen * ~this.unseen = this
 */
public interface QueueKernel<T> extends Standard<Queue<T>>,
        Iterable<T> {

    /**
     * @depletes x
     * @updates this
     * @ensures this = #this + <#x>
     */
    void enqueue(T x);

    /**
     * @updates this
     * @requires this /= <>
     * @ensures #this = <dequeue> * this
     */
    T dequeue();

    /**
     * @ensures length = |this|
     */
    int length();
}

Listing 4.6: The QueueKernel interface.
```
@mathmodel tag, and much like in RESOLVE, the keyword type is followed by the name of the programming type we are going to model. When modeling a programmatic type, the name will always be the same as the interface or class the JavaDoc is documenting, but this will not be the case when introducing new mathematical types. The name of the type is followed by the is modeled by keywords and its model. The QueueKernel is a string of objects of the mathematical model of the $\mathcal{T}$ parameter.

Interfaces cannot define constructors in Java. However, we need to be able to introduce an initial value every type we model in our discipline. Further, this initial value needs to be the same as the abstract value resulting from calling the constructor with no parameters. This limitation leads to the introduction of the @initially label. This label introduces the constructors that an implementer of this interface must provide.

In the case of QueueKernel the clause only introduces the default constructor (lines 8-10). A call to a default constructor—i.e., the one without any formal parameters—will produce a new empty queue—modeled by $<>$, the empty string. In all constructors, the clauses and parameter modes appear without a leading @ sign to avoid conflicts with the javadoc tool, a program used to format the documentation into web-pages.

Refer to the NaturalNumberKernel interface in Listing B.2 to see how multiple constructors are introduced. In it, multiple constructors are introduced after the @initially tag. Each of them follows the same structure, with the default keyword replaced by a list of formal parameters to the constructor—such as int $i$—followed by a colon and the contract for the constructor.
In lines 13 and 14 we can see where the two extends arrows emanating from the QueueKernel in Figure 4.3 come from. Extending Standard is a convention for all mutable types created in our discipline. The methods in this interface are discussed in Section 4.4.2.

Extending Iterable

Extending the java.lang.Iterable interface adds the iterator method to the list of methods provided by the QueueKernel interface. The reason that this is part of a Kernel interface is that, for most data structures, it is impossible to efficiently create an iterator in Java without accessing an object’s internal representation. Much like the contracts in java.lang.Object, the iterator() method lacks the necessary knowledge about the type implementing it. Thus a one-size-fits-all specification cannot fully express the value of the object it produces, which calls for new specification machinery.

Listings 4.7 and 4.8 contain the contracts for the Iterable and Iterator interfaces. In our discipline, the java.util.Iterator is modeled by two strings of objects: seen and unseen. Their names come from the fact that a client has accessed (i.e., “seen”) some entries in the collection by using the next() method, and has not yet accessed the others. The use of strings records the temporal order of their access. The iterator() method returns an object of type java.lang.Iterator that is encumbered by this—the object implementing the method\textsuperscript{50}.

The \texttt{@ensures} clause for the iterator() method states that the seen string of the iterator is empty right after the call. The contract purposely leaves undefined the initial value of the unseen string. The missing information about the initial value

\textsuperscript{50}And thus the one we’ll be iterating over.
package java.lang;
import java.util.Iterator;

public interface Iterable<T> {
    /**
     * @encumbers iterator via this
     * @ensures iterator.seen = empty_string
     */
    public Iterator<T> iterator();
}

Listing 4.7: The java.lang.Iterable interface.

package java.util;

/**
 * @mathmodel type Iterator is modeled by (seen: string of E,
 * unseen: string of E)
 */
public interface Iterator<E> {
    /**
     * @ensures hasNext = (|this.unseen| > 0)
     */
    boolean hasNext();

    /**
     * @encumbers next via this
     * @updates this
     * @requires |this.unseen| > 0
     * @ensures this.seen = #this.seen * <next> and
     * <next> * this.unseen = #this.unseen
     */
    E next();

    ...}

Listing 4.8: The java.util.Iterator interface.
of an iterator needs to be provided by the type implementing it. For a Queue, this information is provided in line 11 of QueueKernel. The expression after the @iterator tag is an invariant that relates the value of a Queue, this, to the value of its iterator, denoted by ~this. Here, it states that the Iterator presents the elements of the Queue in order from first to last. The invariant is added to the @ensures clause of the iterator method for a QueueKernel object. That is, the complete @ensures clause for the iterator method of a Queue is the combination of the @ensures clause in the Iterable interface and the @iterator invariant in QueueKernel. Thus, the initial value for the iterator returned by the iterator() method on a queue is defined by the clause (iterator.seen = empty_string) and (iterator.seen * iterator.unseen = this).

In the Iterator interface, we can see that the next() method moves an element from the front of the unseen string to the back of seen, and the resulting variable is encumbered by the iterator. The optional method remove() is not included in the Iterator interface specification since, by convention, the method is not implemented in any of our components.

4.4.2 The Standard Interface

The Standard interface in the top-right corner of Figure 4.3 provides a minimum set of operations for all mutable types in our discipline. These three methods provide functionality that allows for the efficient movement of data among objects without the need for aliasing their representations. Note that Standard is a parameterized interface, and is intended to be parameterized by the interface or class extending it.
The `clear` method resets the abstract value of the distinguished parameter to its initial value. The `newInstance` method, by convention, returns a new object with the same dynamic type as the receiver. The language we use in our contracts follows a strict separation between programming types and mathematical types (i.e., contracts only refer to the mathematical model of a variable). A side-effect of this is that our specification language does not support any mention of the “dynamic type” of a variable, and thus cannot express this convention. Instead, it is enforced by using a standardized, generated body of the method, and ensuring that every class that implements the `Standard` interface has a `public default` constructor that is called with reflection. This ensures that the resulting object has an appropriate initial value, since the default constructor is the one that defines the initial value of a type.
@SuppressWarnings("unchecked")
@override
public final Queue<T> newInstance() {
    try {
        return this.getClass().newInstance();
    } catch (ReflectiveOperationException e) {
        throw new AssertionError(
            "Cannot construct object of type 
            + this.getClass());
    }
}

Listing 4.10: The newInstance method from Queue3.

Listing 4.10 presents the newInstance method from the Queue3 class. The implementation of newInstance always looks like this in our discipline\textsuperscript{51}. Thus, while we do not have a way of expressing anything about the dynamic type resulting from the method, we can enforce our convention by forcing the code to always follow this template. That is, the body of a method implementing newInstance always consists of exactly this try-catch statement.

The transferFrom method moves the value of source to this. The method, much like newInstance, is implemented in a systematic way. This is because, when both this and source have the same dynamic type, the transfer can be done in constant time. Listing 4.11 presents a sample body for the transferFrom method taken from the Queue3 class.

The implementation extending QueueSecondary provides an implementation of transferFrom that checks whether other has the same dynamic type as this (in line 6). If both arguments have the same dynamic type, the method proceeds with a constant-time transfer. This is achieved by aliasing all of the fields from source

\textsuperscript{51}Except for the return type, which differs accordingly.
```java
@Override
public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    if (source instanceof Queue3<?>) {
        Queue3<T> localSource = (Queue3<T>) source;
        this.entries = localSource.entries;
        localSource.createNewRep();
    } else {
        super.transferFrom(source);
    }
}
```

Listing 4.11: The `transferFrom` method from `Queue3`.

to `this` (line 8) and then allocating a fresh representation for `source` (by calling `createNewRep` in line 9). The `createNewRep` method is a `private` method that does exactly the same as the default constructor.

A call to an instance method of a variable after we referenced its representation would normally violate our discipline. We are not allowed to refer to the abstract object value of a variable in a call when we have access to its data representation. Yet that is exactly what we are doing in the call to `localSource.createNewRep()`. However, the `createNewRep` method is an exception to this rule: it assumes that all the fields in the data representation of the receiver are uninitialized or depleted before the call. Thus, in our discipline, this call cannot lead into the problems described in Chapter 2: assuming the entire representation to be uninitialized precludes the method from assuming any `@convention` before the method begins.

However, the implementer cannot assume `source` has the same dynamic type as `this`. For example, `this` might refer to a `Queue3<Integer>` object while `source` refers to a `queue2<Integer>` object. When the dynamic types do not match, we need to use
a layered implementation of this method (line 11). In order to avoid repeating the code for the layered implementation in every class implementing a kernel, the layered implementation is provided in the “secondary” abstract class as explained in detail in the next section. The implementation for it can be seen in Listing 4.13.

4.4.3 The Secondary Methods

The Queue interface in Figure 4.3 contains the secondary methods for a queue. The secondary methods are those that are not necessary to manipulate variables of a type, yet provide useful functionality. In RESOLVE, those methods would be split into their own extensions that allow for methods to be added to a type without changing its name. Creating something akin to that in Java is not possible, because a type is equal to a class in Java.

The solution is to provide all of the extended behavior for a type in a secondary interface. This interface is named by the type’s name, since it is the one that all clients are supposed to use. Thus, while the Kernel interface defines a type, the secondary interface that extends its functionality defines the type that should be used in variable declarations by both clients and implementers\(^{52}\). Listing 4.12 presents the secondary methods for a Queue.

It should not be hard to see how append and flip can be implemented by code that calls the Kernel methods (i.e., by layering code on the Kernel methods). Similarly, the front method can easily be implemented using the Iterator for a Queue. That is, while the functionality provided by these methods might be important to a client, their implementations can be written independently of the Queue’s internal representation.

\(^{52}\)Following standard best practices, our discipline stipulates that variables be declared using interface types.
package components.queue;

public interface Queue<T> extends QueueKernel<T> {

    /**
     * @encumbers front via this
     * @requires this /= <>
     * @ensures <front> = this[0, 1)
     */
    T front();

    /**
     * @depletes x
     * @updates this
     * @encumbers replaceFront via this
     * @requires this /= <>
     * @ensures <replaceFront> = #this[0, 1) and
     *          this = <#x> * #this[1, |#this|)
     */
    T replaceFront(T x);

    /**
     * @updates this
     * @clears q
     * @ensures this = #this * #q
     */
    void append(Queue<T> q);

    /**
     * @updates this
     * @ensures this = rev(#this)
     */
    void flip();

    /**
     * @ensures contains = x is in elements(this)
     */
    boolean contains(T x);
}

The ability to implement these methods in a layered way, as well as the need for them to be present in all implementations of Queue, lead to the introduction of the QueueSecondary abstract class. The secondary class provides the layered implementations of the methods introduced in the Queue interface. This provides for maximum code reuse and greatly reduces the onus on an implementer of a Queue: they only need to provide bodies for the methods introduced in the Kernel interface.

Listing 4.13 presents the layered implementation bodies for the methods introduced in the secondary interface of our Queue. In Java, all classes implicitly extend the java.lang.Object class. As we stated in the previous chapter, this class contains the common methods equals and hashCode. The implementations of these methods, whenever a Kernel interface is well designed, should also be possible in a layered fashion. Thus, the abstract class needs to provide bodies for them. In the previous chapter we argued that the default implementations of these methods are incorrect, so failing to override them would result in trouble. The details of these methods’ implementations are later explained in Section 4.5.1.

The Layered transferFrom Body

As stated before, the transferFrom method creates a potential performance problem. When the two parameters have the same dynamic type, the transfer can be performed very fast by simply rearranging the references of the representations of this and source. However, this kind of implementation fails whenever the objects have different dynamic types. The solution is to have multiple implementations of this method.
package components.queue;

import java.util.Iterator;

public abstract class QueueSecondary<T> implements Queue<T> {

    // Common methods (from Object) ---------------------------
    @Override
    public final boolean equals(Object obj) { ... }

    @Override
    public int hashCode() { ... }

    @Override
    public String toString() { ... }

    // Modular transferFrom implementation -------------------
    @Override
    public void transferFrom(Queue<T> source) {
        this.clear();
        /**
         * @updates this, source
         * @maintains this * source = #this * #source
         * @decreases |source|
         */
        while (source.length() > 0) {
            this.enqueue(source.dequeue());
        }
    }

    // Other non-kernel methods ------------------------------
    @Override
    public T front() {
        assert this.length() > 0 : "Violation of: this /= <>";
        return this.iterator().next();
    }

    @Override
    public T replaceFront(T x) {
        assert this.length() > 0 : "Violation of: this /= <>";
        Queue<T> q = this.newInstance();
        T front = this.dequeue();
        q.enqueue(x);
        q.append(this);
        this.transferFrom(q);
        return front;
    }

Listing 4.13: A partial listing of the QueueSecondary class. (Cont.)
Listing 4.13: (Cont.)

```java
@Override
public void append(Queue<T> q) {
    assert q != null : "Violation of: q is not null";
    assert q != this : "Violation of: q is not this";
    /**
     * @updates this, q
     * @maintains this * q = #this * #q
     * @decreases |q|
     */
    while (q.length() > 0) {
        T x = q.dequeue();
        this.enqueue(x);
    }
}

@Override
public void flip() {
    /**
     * @decreases |this|
     */
    if (this.length() > 1) {
        T x = this.dequeue();
        this.flip();
        this.enqueue(x);
    }
}

@Override
public boolean contains(T x) {
    /**
     * @updates ~this
     * @maintains ~this.seen * ~this.unseen =
     * ~this.seen * ~this.unseen
     * and x is not in elements(~this.seen)
     * @decreases |-this.unseen|
     */
    for (T y : this) {
        if (x.equals(y)) {
            return true;
        }
    }
    return false;
}
```
In our discipline, a complete implementation of a Queue\textsuperscript{53} is required to implement all of the methods introduced by the Kernel interface (including the methods in the interfaces it extends). That includes the methods in the Standard interface. Yet, as stated before, all of the implementations for transferFrom would share the same code whenever the dynamic types of this and source do not match: the lack of knowledge about the source’s representation requires the implementer to transfer the values using only the Kernel methods. Moreover, by the same convention explained in Chapter 2, these methods cannot be called inside of a class that is implementing the abstraction.

In order to avoid repeating code in every implementation of a component, the secondary class provides a layered implementation of transferFrom that specific implementations—such as Queue3 in Figure B.1—can call whenever the dynamic types of the arguments do not match. This provides both a high level of code reuse, and a best-case performance for the method. It is also one of the few performance concessions in our discipline, since in most cases when there are trade-offs between complexity and performance our decisions heavily skew in favor of lowering complexity. It turns out that transferFrom is so important—it is the data movement operator of choice in our discipline—that an efficient implementation when the dynamic types are the same is vital in practice. This choice is based in previous work in [45] that has shown that a “move” operator should always be preferred to aliasing.

\textsuperscript{53}I.e., a class that extends QueueSecondary and is not an \textit{abstract class}. 
4.5 Leveraging Java’s components

Not all of the components used in a Java project can be implemented following the best practices described in Section 4.4. Some of the components that are part of the Java language are just impossible to ignore. For example, it is hard to imagine a Java program that does not use the `String` class, or primitive types. The solution to this problem is to create “bridge” interfaces. These interfaces connect components created outside of our discipline with those we intend to verify.

Verification of a program that interacts with other programs is, unsurprisingly, contingent on the correct documentation and implementation of the other programs, too. This is a complicating factor when using components that are not verified, since the level of confidence on the whole process is greatly diminished. However, in order to verify anything in Java, one needs to trust a lot of components that, as of now, are not verified.

It would be wrong to allow the lack of preexisting verified implementations in the language to prevent verifying implementations of applications that are written in it. The proposed solution is to create interfaces for those unverified components in the hope that they can be verified in the future, and with the knowledge that, despite their lack of verification, most of these components come with a high degree of confidence in their correctness. The primary concern is then whether their actual behavior can be correctly specified.

4.5.1 The `java.lang.Object` interface

The `java.lang.Object` class is the most inescapable `class` in any Java program. Any implementation of a `class`—i.e., a reference type—implicitly extends it. This is
Further complicated by the fact that the default implementations provided for some of the methods in it are simply wrong once abstract object values are important. As we discussed in Chapter 3, the methods introduced in the `Object` class presents a problem for both reasoning and verification of a program. This is because many of methods introduced by `Object` have a meaning that depends the mathematical model of the type—something that the implementation of `Object` does not have access to. To solve this problem, these methods need to be overridden by any class that attempts to implement an interface in our discipline. Specifically, this means that our abstract classes with layered implementations of secondary methods also need to provide bodies for both the `equals` and `hashCode` methods.

Listing 4.14 presents the bridge interface for the Object class. Notice that while in Java Object is a class, we do not verify the method bodies in it. Thus, the bridge is declared as an interface meant to provide a contract for Object, but not a body. This interface is not actually compiled, but conceptually replaces Object in that each interface in our discipline is viewed as extending it.

Implementing the equals Method

As we stated before, the equals method provides a challenge, since its meaning is defined without any knowledge about the type’s math model. Since our discipline does not allow for behavioral subtyping, only one of the interfaces implemented by a class can define a math model. The @ensures clause states that the equals method returns true iff the abstract value of this is equal to the abstract value of other. However, due to the type-signature of the method, the type of other may not match the type of this. Similarly, since this method is not defined inside of our discipline, we cannot assume all calls to it follow the convention of not using null references or repeated arguments.

The correct way of implementing the equals method in Java is not a secret; we follow the methodology presented in [8]. A sample of an implementation of the method for Queue following best practices is presented in Listing 4.15. We use this method to describe the expected structure of any implementation of equals.

As we can see in lines 3-8, the first two checks are related to the reference values of the formal parameters other and this. Any implementation must check whether the other argument is null (in which case equals must return false) or an alias to the distinguished parameter (in which case it must return true).
```java
@override
public final boolean equals(Object other) {
    if (other == this) {
        return true;
    }
    if (other == null) {
        return false;
    }
    if (!(other instanceof Queue<?>)) {
        return false;
    }
    Queue<?> q = (Queue<?>) other;
    if (this.length() != q.length()) {
        return false;
    }
    Iterator<T> it1 = this.iterator();
    Iterator<?> it2 = q.iterator();
    while (it1.hasNext()) {
        T x1 = it1.next();
        Object x2 = it2.next();
        if (!x1.equals(x2)) {
            return false;
        }
    }
    return true;
}
```

Listing 4.15: An implementation of the equals method for Queue.
Next, the implementation needs to check that the two objects share a math model. It’s not enough for this and other to just share a mathematical model type; they also need to share the place where the type was declared\(^\text{54}\). This is why lines 9-11 will return false if the parameters do not implement the same interface. Satisfying this check allows the implementer to use the methods defined in the Queue interface (after type-casting other, as in line 12). Once the cast is made, the rest of the method’s body needs to be verified in the same way as any other method, but with the new casted variable \(q\) in place of other. The process for this is described in Chapter 5.

An astute reader might have noticed that the equals method requires both formal parameters to have the same math model in order to be true. Yet, the method in Listing 4.14 checks for an instance of a Queue instead of a QueueKernel (where the math model is defined). This is not a problem since, by convention, no class will only implement a Kernel interface without implementing the secondary methods, too. The use of a common abstract class that implements all of the secondary methods in a reusable manner greatly reduces the burden imposed by this restriction.

### 4.6 Conclusion

The contracts and rules presented in this chapter have two different aims: to force every variable to have an abstract value, and to prevent references from accessing parts of an object’s data representation in a way that side-steps its interface. This allows most of our contracts to focus on abstract object values and not on references and dynamic types. Whenever the contracts are not expressive enough to capture\(^\text{54}\)It is not equivalent to have a String variable with the value of “Hi” and a Stack<Character> variable with the value of \(\langle H', 'i'\rangle\), even though both of them have the exact same mathematical models and abstract values.
a problem, as in the case of the `equals` method, we propose a standardized code template for implementations that bridge the gap and provide the necessary guarantees by prescription, without the need for case-by-case verification of properties we otherwise can’t verify.

The first problem is solved by requiring every class to implement an interface that defines a `@mathmodel`. In the case of a component that is not part of our discipline, the bridge interface provides both the math model and all of the methods for the type. In the case of a component created in our discipline, the kernel interface provides the math model. Thus, every programmatic type in our discipline has an abstract value associated with it.

The problem of aliases to parts of an object’s data representation is solved by using the two reference parameter modes—`@depletes` and `@encumbers`—to advertise that a method can create an alias to one of its parameters.

When looking at code in the most concrete way, the only way that a variable can be aliased to another is through the assignment operator (`=`). However, when we try to abstract away the behavior of a method from its body, two new possible sources of aliasing emerge: an alias created by a method between parts of data representations of its arguments\(^{55}\), and an alias created when a reference is returned by a method\(^{56}\).

The aliases created by assignment can be locally traced and a VC generator can record how those affect program state. The simplest way of doing this is to assume that every assignment of a reference depletes the reference on the right-hand side.

\(^{55}\)This is the type of alias that a call `q.enqueue(x)` generates between `q` and `x`.

\(^{56}\)This is the type of alias that a call `x = q.front()` generates between `q` and `x`. 
Keeping only one reference that is able to access and modify an object is conservative but effective, and as a practical matter barely impacts client code written with components designed using our discipline.

Dealing with aliases created by methods is just as simple. The @depletes parameter mode ensures that one of aliased arguments to a call will not be accessed after the call, preventing the client from directly accessing the representation of an object, bypassing its interface methods. The @encumbers rules prevent a client from nefariously using an alias, even if that alias was created on purpose by the implementer. The next chapter describes how we can generate verification conditions that rely only on a variable’s abstract value for a program that follows the rules and structure introduced in this chapter.
Chapter 5: Verifying a Resolve Java Program

In the previous chapter we introduced techniques that allow us to specify code in terms of abstract object values. We did this in a language that has reference semantics by limiting the way that aliased objects can be used. The limitation was presented in terms of two new parameter modes, with the goal of preventing any use of an aliased reference in a way that might alter the observed abstract object value of another variable.

In this chapter we introduce methods and tools that can create verification conditions similar to those in Chapter 2. The similarity between those is another goal, since it allows us to re-use the mathematical provers currently used by the RESOLVE tools, such as Split Decision [5] and Z3 [53]. The main reason for the similarity between the Java and RESOLVE VCs is that both refer only to the abstract values of the variables involved in a program (abstract variable values for primitive types and abstract object values for reference types).

The use of Java instead of RESOLVE poses many challenges. Not only does Java have reference semantics, but many features of RESOLVE that aim to simplify logical reasoning about a program are not present. In particular, Java programs use short-circuit evaluation of logical operators and allow for multiple return statements in a method’s body. These two features, coupled with the fact that expressions can have
side-effects and can be nested, allow Java programs that look simple to have multiple execution paths (each with possibly different variable values depending on what path led us there). Further, the presence of nested expressions complicates the RESOLVE rule of one line of code per state index. As we’ll see, such simplifications designed to make RESOLVE verification more straightforward are (unsurprisingly) not essential, merely convenient.

The possibility of control-flow statements having side-effects and multiple paths leading to the same point of code makes the symbolic tracing tables used in RESOLVE impractical, since the Facts column in the table would need to present each fact as an implication, with coverage of of every possible control expression. That is, the facts right after a control statement would also look like the merged Facts when multiple paths can lead to a common point in the code. Even worse, some state indexes might be skipped by a short-circuit evaluation without explicit \textbf{if} statements. The presence of too many implications in the tracing table, generated by all the alternative execution paths, would make even simple Facts hard to understand (and Obligations possibly harder to prove, too).

We instead create multiple \textit{linear tracing tables}\textsuperscript{57}. Each of the linear tracing tables corresponds to a single execution path through a method. That is, a branch in execution (either due to control-flow statements, or short-circuit evaluations) results in a split of the current tracing table into two different ones (one for each path). In the two tables the indexes before the branch are equal, but the two continue in the different paths. It is worth noting that the tool assumes that the input code is

\textsuperscript{57}The tool generates Linear Paths Facts and Obligation (LPFOs) instead, which look much like a tracing table but without the code interlaced between state indexes. Sometimes this text mentions LPFO generation instead of linear tracing tables since that is the intermediate form from which we can easily generate VCs.
compatible with the Java 1.7 spec (which is easily checked by compiling the code with the standard javac compiler).

Other differences between RESOLVE and Java include the availability in Java of nested expressions, break statements, for loops (including the for-each constructs), static fields\(^{58}\), and the presence of polymorphism (which can prevent a method from accessing the instance fields of any formal parameter other than this). The use of linear tracing tables helps address the problems created by most of the changes in execution paths created by these types of statements. While Java allows for labeled branching statements, we do not attempt to verify programs using them (though this, too, is a convenience decision rather than an essential limitation).

Another difference with the tracing tables in Chapter 2 is the introduction of two new columns to the tracing tables. The Depleted and Encumbered columns are added to the Path, Facts, and Obligations. The Depleted column keeps track of what variables are depleted at a particular point in the execution of a method. The presence of a variable in the column means that the variable is depleted at that state index\(^{59}\). Note that a variable appearing in the Depleted column is not part of the corresponding state’s Facts: we are unable to express the abstract object value of a depleted variable.

Similarly, the Encumbered column keeps track of the dynamically changing encumbers relation. Each pair in the relation is denoted in the same way as in Chapter 4. That is, if variable $a$ encumbers $b$ at a particular point in a method, the tracing table will show $a \mapsto b$.

\(^{58}\) Though this will be limited to basic constants, i.e., final primitive types.

\(^{59}\) For simplicity, uninitialized variables will also be included in the Depleted column, since they are both equally unusable.
5.1 Overview of Tool Design

The tool presented here is meant to generate LPFOs for all the methods in a Java .class file. Its inputs are the properly documented class file, as well as the library of Java components that it uses. This library must include all implemented interfaces as well as all the interfaces used by the code (by default, this also includes the bridge interfaces from java.lang). Figure 5.1 presents a schematic of the tool’s components, its inputs, and its outputs.

This chapter describes the tools within the dashed red rectangle, as well as their inputs and outputs. We have already described what RESOLVE VCs look like in Chapter 2 and, to preserve interoperability with our provers, the VCs generated by this tool look exactly the same. The LPFOs generated by the tool are equivalent to the tracing tables described here, with the rows containing code in our tracing tables removed in the LPFO. That is, there is a one-to-one correspondence between any
state index in a tracing table in this chapter and the generated LPFO. This chapter introduces concepts and rules using linear tracing tables instead of LPFOs. They both present the same information, but the convenience of seeing the code affecting the variables between two state indexes makes linear tracing tables the better choice for explaining the tool.

Another difference between an LPFO and a linear tracing table is that, unlike the former, a split in an LPFO results in a node from which multiple paths are possible. Despite this difference, the two representations are equivalent: every split in a table creates two tables with the exact same rows before the split. The LPFO simplifies this by organizing the possible execution paths into a tree\(^\text{60}\). The condensed information of the LPFO tree makes it clear that it is not necessary to prove the same obligation for each tracing table (it suffices to prove each Obligation once).

LPFOs are similar in nature to the structures used to verify programs in tools using symbolic execution, such as KeY. However, they differ on the treatment of a variables value between states: our tracing tables assume that the values of a variable at each state index are completely independent. Thus the value of each variable at a particular state index needs to be specified by either a method’s contract or some frame function. Symbolic execution behaves the opposite way, assuming the values of a variable at each state index to be the same as in the previous state, except when they are changed by a method’s contract. Both methodologies are equivalent, they just arrive to similar results from opposite directions.

The VC generator creates VCs from the LPFO in the same way as done in Chapter 2. It accumulates into a conjunction the Facts and Path Conditions though a

\(^{60}\)We could do this with the tables too, but it is hard to fit into a page the results nesting two tables into one, so we opted for duplicating the first rows of the original table instead.
public class isPalindromeOneLine{

    /**
     * @ensures isPalindrome = (str = rev(str))
     */
    public static boolean isPalindrome(String str) {
        /**
         * @decreases |str|
         */
        return str.length() <= 1
            || (str.charAt(0) == str.charAt(str.length() - 1)
                && isPalindrome(str.substring(1, str.length() - 1)));
    }
}

Listing 5.1: A one-line implementation of the isPalindrome method for String.

path up to (and including) a state containing an Obligation. It uses this conjunction of Facts as the antecedent of an implication where the Obligation is the consequent. The only difference with the tool used in Chapter 2 is that the tool used here does a depth first search for Obligations though a tree instead of a linear search.

Using expressions as our unit of computation requires us to provide a name for each intermediate result inside of a nested expression. We achieve this by “storing” an expression’s value in a temporary variable with a name of the form @temp#, where # represents a unique number for each temporary variable. In order to make subtle difference between the mathematical symbol for equality (=) and the symbol for assignment (=) clear, we write let @tempj = expr_i in a program code row of a table, whenever we need to store the intermediate value resulting from an expression expr_i.
Listing 5.1 contains a recursive static method that returns `true` if and only if a `String` is a palindrome. Let's use the expression returned by the `isPalindrome` method to briefly illustrate how the different execution paths and expressions form our linear tracing tables and LPFOs. The code presented in Listing 5.1 is not intended to be efficient, but rather to highlight how a Java programmer might use short-circuit evaluation instead of control-flow statements, and how our tool handles that.

Figure 5.2 presents a control-flow diagram of the expression, with each subexpression arranged in its execution order. This example is explained in greater detail in Section 5.2.9.

Let's focus on the two top elements in the flow chart, and use them to clarify what to expect inside the code rows of the tracing tables in this chapter. The top of the returned expression is an `||` operator, with `str.length() <= 1` as its left-hand side subexpression. In the starting box we can see the division of the expression into two subexpressions that appear in the order that they would be evaluated by the JVM. Due to short-circuit evaluation, if the result stored in `@temp2` is `true`, the result of the whole expression is decided without execution of the rest of the expression. This is the execution path ending in A. We create a linear tracing table for each of the possible execution paths. For example, we create a table for path A with only two expressions inside it (Table 5.8). In it, in the state index right after its evaluation, `@temp2` appears as the Path Condition for the table. Another tracing table is created for the execution path that has `¬@temp2` as its Path Condition.

This split into two different tracing tables also serves to illustrate the differences between the structure of a LPFO and a linear tracing table. The actual LPFO closely mimics the shape of the flow chart, with branches in the same places as the diamonds,
Figure 5.2: A complete flow chart (including subexpressions) for the expression in Listing 5.1. The left path from a diamond is taken when the variable inside it evaluates to false, the right path when it evaluates to true.
and with each branch of execution marked by its Path Condition. In the case of linear tracing tables, reaching the diamond with @temp2 forces us to create two tables, one for each possible value of the path condition. It is worth reminding the reader that creating this many redundant rows inside the VC generator is wasteful, but trying to fit tables with 6 columns into Figure 5.2 would be impossible. This work presents concepts in terms of linear tracing tables because they are easier to read, but the implementation creates the more concise LPFOs that better mimic execution paths.

5.2 Expressions

Expressions form the majority of the code in Java. Anything from a variable name to a method call or an assignment is considered an expression in Java. Each expression, by definition, evaluates to a typed result (even if the result type is void). The presence of nested expressions means that the result of such an expression might not have a name. Thus, the approach we use to produce an LPFO for a nested expression needs to introduce its result into the facts by using a variable whose name does not match the name of any declared variable.

There are many types of expressions in Java, and we do not support all of them in this proof-of-concept tool-set. Some—such as bit-wise operators—break the abstractions of the data representations they operate on. Some, such as type-casting expressions, have their use restricted because they are not congruent with contracts that force a strict separation between programming types and mathematical models. The complete list of unsupported or restricted expressions is detailed later in this section.
The presence of nested expressions, each with its own side-effects on the state, requires that the tracing tables resulting from an expression expand subexpressions in the same way the Java compiler would. The expressions presented here are assumed to be part of an abstract syntax tree (AST), from which the order of evaluation for each subexpression is easy to determine. There are many ways to obtain such an AST from a Java program. We use the Java 1.7 language provided by the ANTLR tool [41]; the complete language can be found in the list of supported languages in their website. In it, the simplest form of an expression is a primary.

The presence of these nested expressions also introduces implicit variables where the intermediate results are stored. We use temporary variables in our tracing tables to name such intermediate results. Each temporary variable, while having a programmatic type, is not a declared program variable. To ensure that there are no naming conflicts between the temporary variables that store an intermediate result and the declared variables of a program, the name of a temporary variable starts with the @ symbol in our tracing tables\textsuperscript{61}.

### 5.2.1 Primary

This is the simplest expression in the language, and it is either a variable, a class, or a built-in constant. In all cases, its evaluation does not add a new row to the tracing table, since the use of a name or constant does not have side-effects (i.e., it does not change the value of a variable).

In the case of a literal, the result of the evaluation is a typed constant. In the case of an identifier, the tool tries to assign it meaning the same way the compiler would. It first tries to find a variable whose name matches the identifier, in which

\textsuperscript{61}An identifier is not allowed to start with this symbol in a Java program.
case the evaluated result is a typed variable. If there is no variable with that name, the tool searches for a class by that name, which is returned as a name with the type of `class`. Since our tools do not support reflection, classes in expressions are only used to provide qualified names in a method (such as `Integer.MAX_VALUE` or `ClassName.StaticMethod()`).

A primary can also be the `this` or `super` keyword. The use of `super` is restricted to the `transferFrom` method. It is not hard to evaluate the programmatic type of `this`. However, the meaning of the `this` keyword—i.e., its mathematical value—depends on whether it appears inside a layered implementation. If the `class` where the `this` keyword is being used has instance variables, then the convention explained in Chapter 2 of not being able to refer to the abstract value of a variable that we are representing, prevents the use of `this` as a separate variable in the code. Thus, its use has to be as a name qualifier—such as `this.instanceVar`, making the mathematical type of `this` a tuple of all its instance variables’ types. In the case of layered methods, the mathematical model for `this` is the mathematical model of the type.

### 5.2.2 Equality Operators

An equality operator (`==` and `!=`) always evaluates to a boolean. However, the expressions containing them have different meanings depending on whether the subexpressions are primitives or objects. Figure 5.3 presents the AST of an operator for two expressions. The shape of the tree remains the same regardless of the operator in `OP`. In all cases—save for the short circuit evaluation of the `&&` and `||` operators—the

---

62When the object types of `this` and the formal parameter do not match, the method is to call the implementation in the secondary `abstract class`. 

127
evaluation of the expression will happen after the evaluation of the subexpressions $\text{expr}_1$ and $\text{expr}_2$.

In Java, the subexpressions in the tree are evaluated from left to right before the equality is evaluated. Unsurprisingly, our tracing tables evaluate the subexpressions the same way. That is, before the evaluation of the operator, the tracing table includes code to compute $\text{expr}_1$ and $\text{expr}_2$. The results of both of those subexpressions are represented by one of a variable, a constant (a literal), or a temporary variable. Each of those possibilities has a type associated with it. In Table 5.1, the subexpression $\text{expr}_1$ is expanded starting at index $i$, and its value is assumed to be stored at index $j$. Notice that in the row between state indexes $i$ and $j$, both $\text{@temp1}$ and the $=$ sign are not in program font. This serves to remind the reader that the store is not a programmatic assignment, but a way to “remember” the values in the table.

Each subexpression is also replaced by the temporary variable that holds its value in order to make the tables more readable, since the Facts and Obligations refer to
them. That is why, in the last program row of both Table 5.1 and 5.2, only the == operator is in programming font.

The meaning of the equality operator depends on the types of its arguments. In the case where both of them are primitives, as in Table 5.1, the evaluation is intuitive: the == operator evaluates to true iff both arguments have equal abstract values. Similarly, the != operator evaluates to true iff the abstract values of the arguments are not equal. Table 5.1 presents an incomplete linear tracing table for this situation. Notice that the result from evaluating each subexpression is stored in a temporary variable that starts with the @ symbol.
Table 5.1: An incomplete tracing table for an equality expression for primitive types. Notice that the indexes before and after the evaluation of each subexpression are not necessarily contiguous. This is due to the fact that the expansion of each subexpression might, itself, add more than one row to the table. The Facts also include frame facts.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td>1 obligations for expr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td>@temp1 = eval(expr1)</td>
<td>1 obligations for expr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td>@temp2k = eval(expr2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k+1</td>
<td></td>
<td>@temp3k+1 = @temp1k = @temp2k</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: An incomplete tracing table for an equality expression for reference types. The expansion of each expression has been avoided and the results of their evaluations are assumed to be in @temp1 and @temp2. Missing here is the check that both @temp1 and @temp2 cannot be in the Depleted column.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>@temp3 = (@temp1 = @temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td></td>
<td>@temp3i+1 = @temp1i = @temp2i</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After evaluating the subexpressions, the code (and thus our table) can evaluate the equality operator. The LPFO generator checks that the result from each subexpression is not depleted. This is a rare case when the appearance of a variable does not require us to deplete any variables encumbered by it. Since the check does not access the object part of the variable, there is no need to worry about the object values changing due to this expression. The result for the evaluation is presented as the first fact in the $k+1$ row in Table 5.1. Notice that, despite the fact that the result for $\text{expr}_1$ was calculated in row $j$, the value is not used until row $k$ and thus it must be preserved through frame facts during the execution of $\text{expr}_2$. For primitive types, the result for the $!=$ operator is the negation of equality. That is, for primitive types, the Fact added right after the evaluation of $!=$ in our tracing table is $\text{temp}_3^{k+1} = (\neg \text{temp}_1^k \neq \text{temp}_2^k)$. This is a simplification of $\text{temp}_3^{k+1} = \neg(\text{temp}_1^k = \text{temp}_2^k)$.

The $==$ Operator and Reference Types

The meaning of the equality operator for references has been extensively discussed in Chapter 3. Aliases are not possible in our discipline, since using aliased objects as arguments would violate the repeated argument rule. Still, in the case of subexpressions that are of reference types, the Fact generated as a result of the $==$ operator needs to be changed to account for the fact that the operator only works on references$^{63}$.

Consider the three variables in Figure 5.4, and the possible results of using the $==$ operator among them. All three variables have the same abstract value (42), yet the operator only returns $\text{true}$ when its arguments are aliased (such as $m$ and $n$). It

---

$^{63}$This discussion might seem pedantic at first sight, but the meaning of the $==$ operator is problematic enough that it is overridden by the equals method in Scala, even though the language has interoperability with Java code as one of its core goals.
should not be surprising that aliased variables have the same abstract object value, they are the same object after all. However, \( m == k \) produces \texttt{false} as its result, despite both \( m \) and \( k \) having the same abstract value. This is the limitation of the == operator, it only checks for reference equality. As a result, the result of == being \texttt{false} provides us with no information about the abstract object values of its arguments. This is recorded in our tracing table by using an implication. That is, the result of the operator being \texttt{true} implies that the two subexpressions have the same abstract value—i.e., \( \neg \temp3_{i+1} \Rightarrow (\temp1_k = \temp2_k) \); but the implication might not hold in the other direction.

For reference types, the result of the != operator in a tracing table is not the negation of the result from the == operator. The operator can still only tell us about aliases, and the knowledge we can get about the abstract object values of two variables from the fact that they are aliased remains the same. Consider again the example in Figure 5.4. The only information we can obtain about the abstract object values of \( m \), \( n \) and \( k \) is still limited to the alias between \( m \) and \( n \). Thus, the != operator tells us that if it evaluates to \texttt{false}, the two variables have the same abstract value. That is, the row after the evaluation has the fact \((\neg\temp3_{i+1}) \Rightarrow (\temp1_k = \temp2_k)\).
5.2.3 Comparison Operators

Java only allows the use of comparison operators between subexpressions of primitive types—i.e., no reference arithmetic. The AST for such expressions looks very similar to that of Figure 5.3. The \( OP \) root in the case of comparisons is one of \( \leq, \geq, >, \) or \( < \). In all four cases the meaning of the operator matches the meaning of its math equivalent. Similarly, the fact that these operators only apply to primitive types removes the need to check for subexpressions involving or resulting in depleted references.

Thus, the generation of a tracing table for a comparison operator simply evaluates its two subexpressions and then assigns to a temporary boolean variable the result of the comparison (in the same way that was presented in the Facts of row \( k+1 \) in Table 5.1).

5.2.4 Method Calls

A method call in Java can, due to the presence of nested expressions, rely on the evaluation of an expression to find the contract for a method. Figure 5.5 presents a simplified AST for a method call. In it, the subexpression \( expr_A \) can be either a variable—declared or temporary—or the name of a \texttt{class}, distinguishing whether the method is an instance method or a static method.

Note that Java allows for a method call that is missing the \( expr_A \) subexpression. In those cases, the compiler will look for an instance method that matches the name in Identifier and, if it finds one, add an implicit \texttt{this} in front of the call. Tools such as CheckStyle automatically add the implicit receiver upon saving and, for simplicity, we therefore assume that no instance method is called with an implicit receiver.
Figure 5.5: A representation of the AST for a method call, $expr_A$ represents the distinguished parameter or a `class`, the Identifier is the method’s name and the optional $expr_1$ to $expr_n$ represent the list of arguments to the call.

Thus, whenever our VC generator has a method call that does not have the $expr_A$ subexpression in it, it assumes it is a call to a static method of that name in the class for which VCs are being generated.

In order to find what method is being called, the tool needs to evaluate the type of each of the arguments to the call. It first evaluates $expr_A$, adding the side-effects of doing so to the tracing table. It then proceeds to evaluate the other arguments to the call (those in subexpressions $expr_1$-$expr_n$). It does this in the same way as it would happen in program execution: from left to right. Once the types of all subexpressions are determined, the VC generator can find the contract for the method being called, as well as the parameter modes for each argument.

The static checker needs to ensure that the call does not violate the implicit preconditions imposed by our discipline. It is worth noting that the way in which primitive types are passed in Java—by making a fresh copy—and the fact that they have to be restored, makes the checks presented here unnecessary. Thus, the checks
explained below are only required for reference types. Primitives cannot be depleted, encumbered, or aliased; thus the checks here do not apply to them.

The VC generator first marks as depleted each variable encumbered by the result of each subexpression—i.e., all the variables used in the method call. This has the advantage of depleting arguments that are repeated because they are related by the encumbers relation, preventing an object and part of its data representation from being used as two separate arguments to a call. The tool then checks that the result of each subexpression is well defined before the call (i.e., not depleted). Once again, failure on this check prevents the creation of VCs because the program does not conform to our discipline.

The VC generator then checks for repeated arguments: the presence of a repeated argument is a syntactic check at this stage, since it only requires the name of the resulting expression to be equal to or the root of another one (in the presence of variable access though the dot operator). A key exception is the presence of this in a class that has instance variables: the use of the this prefix when accessing an instance variable is a best practice that is enforced. Thus the use of two different instance variables in a method will make them share the this prefix, yet that is not a repeated argument unless the suffix of the variables names have repeated parts. That is, using the arguments this.a and this.b is not a violation of the repeated argument rule, but using this.a and this.a.c is. This rule is conservative. Fortunately, our discipline discourages the use of public fields, making the example an unlikely situation.
The final check of implicit preconditions on a method call is to verify that any variable encumbered at the moment before the method call has the \texttt{@restores} parameter mode. This ensures that no encumbered variable is modified, and prevents the possibly of changing the representation (hence possibly the abstract value) of another object by using an alias instead of going through its interface.

Once all of these checks are made, and satisfied, the VC generator can add to the LPFO the \texttt{@requires} and \texttt{@ensures} clauses for the method. This happens the same way it does in RESOLVE. That is, the \texttt{@requires} clause is added to the Obligations before the call and the \texttt{@ensures} clause to the Facts after the call (with the formal parameters replaced by the arguments to the call).

The VC generator then adds to the Depleted column after the call each variable that is depleted by the method. If the returned variable is encumbered, the encumbers relation is added to the Encumbers column. The frame rule for the method is the same as in RESOLVE: any variable not mentioned in the method call is assumed to be restored—unless depleted because it was encumbered. The reasoning for this will become clearer when we explain how assignment expressions are treated, in Section 5.2.5.

Recursion

In the case of a recursive call, the tracing table must include a progress metric. It is worth noting that while the \texttt{@decreases} clause uses name of at least one formal parameter, a recursive call might use a variable with a different name. It becomes necessary then to state as an Obligation that the incoming value of the \texttt{@decreases} expression is smaller than its value at the time of the original call. An example of this is presented in Obligation 1, at index 10, in Table 5.10.
import components.stack.Stack;
import java.util.Iterator;

public class Test{

    /**
     * @requires |s| > 1
     */
    public static void depleteLocalVariables(Stack<String> s){
        Iterator<String> it = s.iterator();
        String first = it.next();
        String second = it.next();
        first = s.top();
    }
}

Listing 5.2: Sample code to illustrate the depletion of encumbered variables.

Constructors

From the point of view of a client, the constructor works in a very similar way to a method. That is, a constructor has a contract as defined in the Kernel interface, and in it the same kinds of constructs appear. It is not surprising then, that the expression resulting from a constructor is treated the same way as a method call (but without the possibility of returning an encumbered variable).
Table 5.3: A sample tracing table for a call to \texttt{s.length()}. Notice that the previously encumbered \texttt{top} variable becomes depleted as a result of the use of \texttt{Stack} variable \texttt{s}, even though \texttt{s} is restored by the call.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>(s_i \rightarrow \text{top}_i)</td>
</tr>
<tr>
<td></td>
<td>\texttt{let @temp1 = s.length()}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td></td>
<td>(\text{@temp}_{i+1} = \text{</td>
<td>s</td>
<td>})</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: A sample tracing table illustrating both method calls as well as assignments. The expressions are from the first two statements in Listing 5.2.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{let @temp1 = s.iterator()}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>(\text{@temp}_{1} = (\langle\rangle, s_0))</td>
<td></td>
<td>1</td>
<td>(s_1 \rightarrow \text{@temp}_1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s_1 = s_0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\texttt{it = @temp1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>(\text{it}_2 = \text{@temp}_1)</td>
<td></td>
<td>1</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s_2 = s_1)</td>
<td></td>
<td>1</td>
<td>(@temp_2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>(s_2 \rightarrow \text{it}_2)</td>
</tr>
<tr>
<td></td>
<td>\texttt{let @temp2 = it.next()}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>(\text{it}_3\text{.seen} = \text{it}_2\text{.seen} * (@temp2_3))</td>
<td></td>
<td>1</td>
<td>(@temp_3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\langle@temp2_3\rangle \ast \text{it}_2\text{.unseen} = \text{it}_3\text{.unseen})</td>
<td></td>
<td></td>
<td>(s_3 \rightarrow \text{it}_3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s_3 = s_2)</td>
<td></td>
<td>2</td>
<td>(@temp2_3)</td>
</tr>
<tr>
<td></td>
<td>\texttt{first = @temp2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>(\text{first}_4 = \text{@temp2}_4)</td>
<td></td>
<td>1</td>
<td>(@temp4_4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{it}_4 = \text{it}_3)</td>
<td></td>
<td></td>
<td>(s_4 \rightarrow \text{it}_4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{it}_4 = \text{it}_3)</td>
<td></td>
<td></td>
<td>(@temp2_4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s_4 = s_3)</td>
<td></td>
<td>2</td>
<td>(\text{it}_4 \rightarrow \text{first}_4)</td>
</tr>
</tbody>
</table>
5.2.5 Assignment

The AST for the assignment operator looks very similar to the one in Figure 5.3. However, the left-hand side expression (expr$_1$) needs to evaluate to a declared variable (which is checked by the javac compiler, not our tools). Much like equality checks, the meaning of assignment depends on whether we are dealing with primitive types or references. For primitive types, the assignment is straightforward: after the statement, the variable to the left has the same value as the one on the right and no other variables change.

When dealing with references, the assignment does not copy the abstract object value but rather the reference value. Thus, we need to control for the alias created by it. In my personal experience (and in most of the code presented here), the assignment operator is used primarily to move information from a variable on the right-hand side to the one on the left. This is why our handling of the assignment operator acts like a destructive assignment. That is, when dealing with reference types, the assignment depletes the variable on the right$^{64}$.

In the same way Java requires the expression on the right-hand side to be an initialized variable, we require the expression on the right-hand side to not be depleted. This is not necessary of the variable on the left. As a corollary we can see that the variable on the left is never depleted after an assignment.

The assignment operator also updates the encumbers relation. For example, if we have an assignment expression $x = b$, and we know that $a \mapsto b$ and $b \mapsto c$, then after the assignment $b$ is depleted and $a \mapsto x$ and $x \mapsto c$. All variables encumbered

$^{64}$Note that in the case of a function, such as $x = s$.pop(), our VC generator assigns the object returned by the subexpression $s$.pop() to a temporary variable. Thus, the variable being depleted is not the expression itself, but the temporary variable that does not appear in the program.
by $x$ before the assignment become depleted, too. An example of this can be seen in state index 4 of Table 5.4, where the variable $\text{first}$ takes the place of the temporary variable $\text{@temp2}$ in the Encumbered column while $\text{@temp2}$ is depleted.

The final consideration about the assignment in Java is that, unlike RESOLVE, it is an expression. That is, it evaluates to a typed variable. In this case, the expression evaluates to the variable on the left-hand side. Thus, a chain of assignments for primitive types, such as $x = y = 2$ has the intended result of setting both $x$ and $y$ to 2. However, the similar chain for a reference type, say $x = y = \text{new } \text{NaturalNumber} \ (2)$; leaves $y$ depleted, and the variable $x$ as the one with the value of 2.

**Formal Parameters**

As explained in Chapter 4, our discipline prevents us from assigning to a formal parameter of a method. The rationale for this is that the client is always unaware of this assignment, while the implementer “loses” the value provided by the client. The check to ensure that this does not happens is done, unsurprisingly, when processing an assignment expression. If the left-hand side of the assignment corresponds to a formal parameter, the tool reports an error and does not generate any VCs.

An important result of an implementer being unable to assign to a formal parameter is that if at any point a formal parameter becomes depleted in the body of a method, it remains depleted until the end of the method’s execution. This makes it possible to check at the end of the method’s execution that only parameters with the $\text{@depletes}$ parameter mode are depleted by a method. However, in order to provide more user-friendly error messages, a future version of the tool should check for this in all of the places where a variable can be depleted (method calls and assignment expressions).
5.2.6 Short-Circuit Boolean Connectives

The logical connectives in an expression (&& and ||) use short-circuit evaluation in Java. Thus, the evaluation of a single expression can lead to two different paths of execution. Figure 5.6 presents the possible execution paths for an expression containing the && operator. In it, diamonds represent branching points. The arrow leaving from the left represents a false evaluation and the one to the right represents a true one.

Notice the path that leads to node A terminates the evaluation of the expression without executing the right subexpression (i.e., it short-circuits). The combination of this behavior with the fact that evaluating a Java expression—unlike a function in RESOLVE—can have side-effects, necessitates that expressions that might be short-circuited be treated in the same way as if-else statements.
In order to simplify the tracing tables (and LPFOs), the branches split the table into two: one for the path that leads to A and one for the path that leads to B. It is also worth noting that, by the time we reach A or B, not only do we need to account for the changes to the state caused by expr₁, but also for the fact that we know what the expression expr₁ evaluated to. That is, we can only reach point A if expr₁ evaluated to \texttt{false}, and point B can only be reached if expr₁ evaluated to \texttt{true}.

As a result of this, the evaluation of the expression on the path that ends in A adds the negation of variable \texttt{@temp₁} to its Path Condition, and its evaluation is not an expression but rather the literal \texttt{false}. Similarly, the path that leads to point B adds \texttt{@temp₁} to its Path Condition\footnote{Thus effectively making it a Fact for every VC generated through that path.} before the evaluation of the expr₂ subexpression.

Figure 5.7 presents the possible execution paths for the \texttt{||} operator. The evaluation for this operator works the opposite way of the \texttt{&&} operator: if the subexpression...
Table 5.5: A tracing table for the short-circuit evaluation of the expression `test1 && test2`. The result of the expression is stored in `@temp1` in the table, but the parent expression receives the literal `false` as the result.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>1</td>
<td>¬@temp1</td>
<td>1 (\text{test}<em>{i+1} = \text{test}</em>{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>2 (\text{test}<em>{i+1} = \text{test}</em>{i})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider the expression `test1 && test2`, where both `test1` and `test2` are programmatic `boolean`-typed variables. Since both sides of the operator are variables, there is no need to add rows to the table to account for the execution of the subexpressions. In the table generated from this expression, the result of the expression is stored in the temporary variable `@temp1`. This expression creates two different tracing tables, presented in Tables 5.5 and 5.6.

Table 5.5 is the result of tracing over the expression in the case of short-circuit evaluation. As previously explained, the result of the expression is `false`, while
Table 5.6: A tracing table for the evaluation of the expression test1 && test2 that evaluates both sides. The result of the expression is stored in @temp2 in the table. Notice that test1 is a path condition in index i + 1 of this table. Unlike the paths with short-circuit evaluation, the parent expression receives the @temp2 variable as the result of this expression.

the Path Condition for it is the negation of its first subexpression. In Table 5.6 the evaluation goes through both subexpressions. For this to happen, the variable @temp1 is true, and this is expressed as a Path Condition for state i + 1. Finally, since the value of test1 has been accounted for in the path conditions leading to the evaluation of test2, the final result of @temp1 on this path depends only on the value of test2.

Similar tables can be constructed for the || operator, only with the path conditions and resulting expressions from the first subexpression negated.

5.2.7 Algebraic Operators With Primitive Types

The algebraic operators—i.e., +, −, *, /, and %—also have the same AST as the one in Figure 5.3. As in Java, the type of the two subexpressions exp1 and exp2 determines both the contract used for the operation as well as the type of the result (and thus its mathematical model). In this section we abstain from explaining the concatenate operator for Strings, as that will be explained in Section 5.2.8.
package java.lang;

/**
 * @mathsubtype
 * INT is integer
 * exemplar n
 * constraint Integer.MIN_VALUE <= n and n <= Integer.MAX_VALUE
 */

public interface int_ {

/**
 * This bridge contract specifies the + operator
 *
 * @requires Integer.MIN_VALUE <= this + x and
 *          this + x <= Integer.MAX_VALUE
 * @ensures plus = this + x
 */

public int plus(int x);
...
}

Listing 5.3: Part of the bridge interface for int.

We use the contracts for the int interface in Listing 5.3 to illustrate how the operators are defined. Similar contracts for all the other primitives are necessary in order to support them, but their similarity (in the case of short and long) or the difficulty of concisely defining them (in the case of float and double) leads us to include only the contract for int in this work.

The name in Listing 5.3 ends with an underscore to allow the parser to generate an AST for the interface without the need to change its keywords. However, since this interface is part of java.lang and thus parsed before anything else, the VC generator is able to fix the name of the interface.

The presented interface for the behavior of an int represents a subset of possible behavior. The arithmetic operations presented in it prevent overflow instead of using
modulo or “clock” arithmetic. This is because, while clock arithmetic allows for completeness with respect to the Java Virtual Machine, rarely is there a program that relies on this behavior and using it is usually a sign of an error. This feeling was perhaps best explained by Dijkstra in [17]:

I assume that our programs have been written for the GLM, because that is a machine we can hope to be able to program for. The designer of the actual machine knows—or at least, he should know—that he is not the Good Lord Himself, and that he can hope at most to build a partial simulator of the GLM. While in the GLM, for instance, there is in principle no upper bound on the maximum value of integer variables, the actual machine simulating the behavior of the GLM may be such—usually is such—that it can only cope with integers up to a certain limit. The simulator should check constantly whether it fails, not by virtue of malfunctioning, but by virtue of its designed construction, to simulate the GLM faithfully. As a result a test on overflow of integer capacity is absolutely essential and a machine which in order to remain in range, reduces integer values, for the sake of its own convenience and without warning, modulo something, is a monstrum, unfit for human use.

In any case, the use of an interface makes changing the behavior from an overflowing to a modulo arithmetic as simple as changing a file. Thus for the rare cases where the modulo behavior is needed (such as swapping two int s in place by clever addition and subtraction), simply swapping the interface for the modulo arithmetic one will do the trick.

In Section 5.2.2 we went into detail about the evaluation process of the two subexpressions. Those details remain the same here, but now the types and evaluations of the results change. Table 5.7 presents a tracing table for the addition of two int expressions whose values have been previously calculated in the temporary variables @temp1 and @temp2.

Since all of the variables involved in the expression are primitives, there is no need to worry about checking for depleted or encumbered variables nor repeated arguments.
It is worth noting that the connection between the interface and the tracing tables is very similar to the one explored when dealing with method calls.
<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td>1 @temp1_i + @temp2_i ≤ int.MAX_VALUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 int.MIN_VALUE ≤ @temp1_i + @temp2_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let @temp3 = @temp1_i + @temp2_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td></td>
<td></td>
<td>1 @temp3_{i+1} = (@temp1_{i+1} + @temp2_{i+1})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 @temp1_i = @temp1_{i+1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 @temp2_i = @temp2_{i+1}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: An incomplete tracing table for an add expression of int types.
5.2.8 String Concatenation Operator

The String concatenation operator (+) uses the contract for the `concat` method for String. This is an oversimplification of the operator, as it disallows behavior that automatically calls the `toString()` method on the arguments if they are not Strings. However, this is intended: We could follow Java’s implementation of the operator with the use of the `append` method of the `StringBuilder` class, but this method hides away the change of math model required from the types on both sides of the operator. So instead we opt to force programmers to make both of the arguments to the + operator match whenever one of them is a String.

If we follow the rules presented in the paragraph above, it is not hard to see that the concatenation `expr1 + expr2` when both operands are String-typed is interchangeable with the expression `expr1.concat(expr2)`.

5.2.9 Example

Let’s revisit our example from Listing 5.1 to demonstrate how linear tracing tables are generated in the presence of short-circuit evaluation. Remember, Listing 5.1 contains a recursive static method that returns `true` if and only if a String is a palindrome.

Figure 5.8 presents a condensed version of the control-flow diagram in Figure 5.2, without the subexpressions in it. Here, while `@temp7` is the name for the result of the `str.charAt(0) == str.charAt(str.length() - 1)` expression. The result of the recursive call in line 13 does not cause a branch in execution, thus the temporary variable storing its value is not represented in this graph. Notice that the result of the expression for the path that leads to B will be the result of the recursive
Figure 5.8: A condensed control-flow diagram for the expression in Listing 5.1. As in previous figures, the left path is a result of a false evaluation and the right path is a result of a true one.

call. However, due to short-circuit evaluation, the path that leads to a results in the expression evaluating to true, while the path that leads to c results in false.

For this example, the result from the expression is stored inside the temporary with the highest number for each the execution path. This is not how our tool would name the results, since there is no rule that requires the temporary variables to be numbered in order of expected execution, but it allows for simpler reading. The presence of three possible execution paths necessitates that we generate three different linear tracing tables, each corresponding to one of the paths shown in Figure 5.8.

Table 5.8 shows the linear tracing table for the execution path that ends with the short-circuit evaluation of the || operator. This is the execution path ending in a on Figure 5.8. For this to happen, the expression on the left hand side needs to evaluate to true. This is why the temporary variable @temp2 is part of the path condition of state index 2. Thus, for this execution path to exist, str has to have fewer than two characters (making it a palindrome). In Section 5.6.1, we complete the tracing
Table 5.8: A tracing table for execution path A of Figure 5.8 representing flow chart for Listing 5.1.

table to include the return statement and the obligations created by the method’s postcondition.

Table 5.9 presents the linear tracing table obtained by the short-circuit evaluation of the && operator. Note that the indexes before the evaluation of the left side of the || operator are identical in both tables. That is, the state indexes 0-1 for both tables are exactly the same. In the path condition at index 2 we can see that the length of the string str was greater than 1. Thus, after state index 2, the tracing table needs to start evaluation of the && subexpression.

Since the right-hand child of the || operator is also a boolean connective, the mechanisms in Table 5.9 are the same ones as in Table 5.8. However, the expansion of the highly nested str.charAt(0) == str.charAt(str.length()- 1) subexpression demands many more state indexes in the table. The expansion follows the same depth-first left-to-right evaluation of the AST that the Java compiler would use.

\*\*\*Some substitutions are required to arrive to this conclusion, but I’ll let the reader follow the trail.\*\*\*
<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>let @temp1 = str.length()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>@temp1 =</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>str1 = str0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp2 = (@temp1 &lt;= 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>@temp2 = (@temp2 \leq 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>str2 = str1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>@temp2 = @temp1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp3 = str.charAt(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>str3 = str2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>@temp[1 - 2]3 = @temp[1 - 2]2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp4 = str.length()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>str5 = str4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>@temp[1 - 4]5 = @temp[1 - 4]4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp5 = @temp4 - 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp6 = str.charAt( @temp5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>str6 = str5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>@temp[1 - 5]6 = @temp[1 - 5]5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let @temp7 = ( @temp3 == @temp6 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>¬(@temp2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>¬(@temp7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>@temp[1 - 6]7 = @temp[1 - 6]6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.9: A tracing table for execution path C of Figure 5.8 representing the flow chart for Listing 5.1. Here @temp[1 - k] stands for each of the temporary variables 1 through k, in order to condense the table.
After state index 6, we are able to evaluate the first half of the `&` expression. For the code execution to be able to end without evaluating the subexpression `expr3`, the one on line 13, we need both `@temp2` and `temp7` to be `false`. This is provided by the path conditions at state indexes 2 and 7 respectively.

Table 5.10 presents just the bottom portion of the tracing table that executes the complete expression. From index 7 we can see that this execution path requires the first and last characters of the string `str` to be equal. It is also worth noting that, since `char` is a primitive type, the result of the evaluation of `@temp3 == @temp6` is an equality check and not an implication. The obligation in state index 10 stems from the `@decreases` clause of the method. We need to ensure that the length of the first formal parameter of the method is smaller than the one at the beginning of execution (regardless of their names). Finally, it is worth noting that this is the only execution path for which the result of the expression is the result of another expression (the recursive call to the `isPalindrome` method) and not a literal. This differs from the tables for paths `A` and `B`, where we know the result of the expression to be `true` and `false` respectively.
<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>let <code>@temp7 = @temp3 == @temp6</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>¬(temp2)</code></td>
<td>1</td>
<td><code>@temp7 = (@temp3 = @temp6)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>@temp7</code></td>
<td>2</td>
<td><code>str7 = str6</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td><code>@temp[1-6] = @temp[1-6]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let <code>@temp8 = str.length()</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>¬(temp2)</code></td>
<td>1</td>
<td>`@temp8 =</td>
<td>str8</td>
<td>`</td>
</tr>
<tr>
<td></td>
<td><code>@temp7</code></td>
<td>2</td>
<td><code>str8 = str7</code></td>
<td>2 ( @\text{temp8} - 1 \leq Integer.MAX_VALUE )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td><code>@temp[1-7] = @temp[1-7]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let <code>@temp9 = @temp8 - 1</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>¬(temp2)</code></td>
<td>1</td>
<td><code>@temp9 = @temp8 - 1</code></td>
<td>1 ( 0 \leq 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>@temp7</code></td>
<td>2</td>
<td><code>str9 = str8</code></td>
<td>2 ( 1 \leq @\text{temp9} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td><code>@temp[1-8] = @temp[1-8]</code></td>
<td>3 ( @\text{temp9} \leq</td>
<td>str9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let <code>@temp10 = str.substring(1, @temp9)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>¬(temp2)</code></td>
<td>1</td>
<td><code>@temp10 = str10[1, @temp9]</code></td>
<td>1 (</td>
<td>@\text{temp10}</td>
</tr>
<tr>
<td></td>
<td><code>@temp7</code></td>
<td>2</td>
<td><code>str10 = str9</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td><code>@temp[1-9] = @temp[1-9]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let <code>@temp11 = isPalindrome(@temp10)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><code>¬(temp2)</code></td>
<td>1</td>
<td><code>@temp11 = (@temp10[11 = rev(@temp10[11])])</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>@temp7</code></td>
<td>2</td>
<td><code>str11 = str10</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td><code>@temp[1-10] = @temp[1-10]</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10: An incomplete tracing table for execution path B of Figure 5.8 representing the flow chart for Listing 5.1. Refer to Table 5.9 for state indexes 0-6.
5.3 Statements

If expressions are the words of the Java language, then statements are its sentences. They provide both structure and direction to a program. They introduce variables—through variable declaration statements—as well as group instructions—through blocks and control-flow statements. In fact, most of the structures that decide what part of a program will be executed are a result of statements.

Much of the groundwork on how we can analyze a program follows directly from our discussions about expressions. Thus, the treatment of statements does not introduce many new concepts, just new ways of generating tracing tables.

As was the case with expressions, the scope of this work allows us to “postpone” into future work certain types of statements. Some convenient statements, such as do-while and switch statements, are not addressed in this work because they are redundant, providing functionality that can be obtained with other constructs. These statements cause no technical difficulties. Statements related to exceptional behavior also are not treated in this work, because they would impose demands on our verification system that are currently not addressed in RESOLVE. These do cause technical difficulties.

As previously discussed, our belief that go-to statements can have absolutely corrupting influences on code clarity, leads us to omit any structure that behaves in a similar way to a jump. Thus, this work does not introduce reasoning rules for the break label; and continue label; statements or for the labels introducing their targets. It does, however, include breaking statements in their simpler form, without
a target label, i.e., \texttt{break}; and \texttt{continue};. It is worth noting that this stance a philosophical one and not technical one, and that this work could be expanded to include these constructs in future work.

With the exception of the \texttt{return} statement, a statement does not evaluate to anything. Thus it is easier to understand an expression, since all of its possible effects are shown in the tracing table: there is no discussion of what a statement “evaluates” to, nor of how its result is seen by its parent statement. The simplest statement—a single ;—would add a new row to our tracing tables while preserving the entire variable state. That is, it has no effect on the variable state. Thus, it will be removed from all of our ASTs.

5.3.1 The Assert Statement

When programming by contract, as we do in our methodology, an implementer is to assume that the client was responsible for meeting the method’s precondition. Thus, she is never supposed to check a precondition in the method’s body. This is a very sensible norm when dealing with verified code. But we do not always write code after all the contracts are finalized, and in the process of exploring a system’s requirements we might need more flexibility.\(^{67}\)

Whenever we are working on unverified software, we would like problems to be detected and reported as soon as possible. This is called a \textit{fail fast} approach. However, the practice of not checking preconditions means that an erroneous call to a method might not be detected until its results create a bigger problem later in the program’s execution. Assertions allow us to bridge the gap between the programming by contract

\(^{67}\text{We would love to always start coding a project after we have its full formal specification, allowing us to verify our code before it ever runs, but sometimes ideas need to be refined in a more incremental way.}\)
conventions and our desire for a program to fail fast. They do this by introducing inside the method’s body code that checks whether the preconditions of a method are met. Since assertions can be turned on/off before a program is executed, they can be used whenever an unverified program is being developed, and turned off once the final product is verified.

Since assert statements are not intended to affect program state, they are not part of our software verification effort. However, a version of the tool might need to check that the expressions used in them do not change the abstract values of any variables. Similarly, it could check that the assertions are implied by the requires clause of the method, reporting an error when this is not true. An advanced tool also might check that the @requires clause is implied by the conjunction of the assert statements, warning the developer of unchecked preconditions when this is not the case.

5.3.2 Blocks

Blocks provide both a way of grouping statements, as well as a scope in which variables are declared. Inside of a block we find an ordered sequence of statements that are added in order to our tracing tables. Much like the empty statement (;), the start of a block does not have any effect on the variable state. Yet, since it provides important insights into our understanding of a program, the opening of a block is represented in our tables with a row. Table 5.11 presents a template of how the opening of a block is treated in a tracing table. In it, $f_i$ represents the facts at state index $i$, while $v_{i+1} = v_i$ is short-hand for the equivalence of all the variables in state index $i$ and $i + 1$. The sets $d_i$ and $d_{i+1}$ contain the same variables (with each index

157
updated to match its state index). Similarly, the pairs presented in $e_i$ and $e_{i+1}$ differ only in their subscripts.

The closing of a block does not change the values of any variable either. However, a variable going out of scope can have implications in our Encumbered column. Once a variable is out of scope it can no longer encumber, nor be encumbered by, another variable. The solution for this is to, whenever possible, maintain the encumbers chains created by the program. That is, suppose $b$ is going out of scope after state index $i$ while encumbered by $a$ and encumbering $c$ ($a_i \mapsto b_i$ and $b_i \mapsto c_i$). Then in the state after $b$ goes out of scope ($i+1$), the encumbers relation involving the remaining variables is updated so that the remaining chains are maintained. Thus, with the variable in the middle of the chain going out of scope, we are left with $a_{i+1} \mapsto c_{i+1}$ after $b$ goes out of scope in our example. If, as in the case of $f_i \mapsto g_i$ in Table 5.12 no variable encumbers $f$, then the variable encumbered by it ($g$) becomes unencumbered. This is represented by the absence of $g_{i+1}$ in the Encumbers column.

The variable $e$ presents us with a final case. When an encumbered variable goes out of scope while not encumbering another variable, as $e$ does in our example, the encumbering variable becomes unencumbered (as represented by the absence of $d_{i+1}$ in the Encumbers column.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$1$</td>
<td>$f_i$</td>
<td>$d_i$</td>
<td>$e_i$</td>
<td></td>
</tr>
<tr>
<td>$i+1$</td>
<td>$1$</td>
<td>$v_{i+1} = v_i$</td>
<td>$d_{i+1}$</td>
<td>$e_{i+1}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.11: A template tracing table for the opening of a block statement.
Table 5.12: A sample tracing table for the closing of a block statement in which the variables b, e, and f go out of scope.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>$f_i$</td>
<td>1 $a_i \mapsto b_i$</td>
<td>1 $b_i \mapsto c_i$</td>
<td>1 $d_i \mapsto e_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 $f_i \mapsto g_i$</td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td></td>
<td>$v_{i+1} = v_i$</td>
<td>1 $a_{i+1} \mapsto c_{i+1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thankfully the situation is much easier for the depleted variables. Once a depleted variable goes out of scope, it no longer appears in our tracing tables. This makes sense if we consider that a variable becomes depleted if it is a reference to the data representation of another variable, because there is no need to worry about altering said data representation through an aliased reference after the reference is no longer in scope.

5.3.3 Variable Declaration Statements

The variable declaration statements discussed here, while syntactically similar to field declarations, are semantically different. A variable declared in a variable declaration statement does not outlive the scope of the block it is declared in (and thus, it cannot not outlive the execution of a method either). Java allows for multiple variables to be declared in a single statement with the use of a variable declaration sequence, such as `final int x=3, y;`. This is equivalent to having the sequence of statements `final int x=3; final int y;`\(^{68}\), in which a single variable is declared per sequence of modifiers and type. Our tool simplifies combined declarations into

\(^{68}\)This is not technically equivalent if done at the source code level, since replacing one statement with two could change the nesting of statements inside control-flow statements. However, the differences in behavior can be accounted for by grouping the two statements in the AST.
sequences of single declarations in our ASTs, so that a single variable is declared in each variable declaration statement. Our decision to not support arrays considerably simplifies what can appear in both the type and init subtrees of Figure 5.9.

Java allows for a variable to be modified in its declaration by either the final keyword or an annotation. An annotation has no effect on how the variable can be used, while the final keyword prevents assigning it a value more than once. The final modifier considerably affects the ways in which the declared variable can be used\(^\text{69}\), yet the tool does not need to be concerned about this. This is because the requirement that programs can be compiled with the javac compiler ensures that those restrictions are met beforehand.

The effect of a variable declaration in our tracing table is the introduction of a new symbol into the tracing table that remains in scope until the block in which it was declared is closed. The initial value of the new variable depends upon the expression

\(^{69}\text{Since the variable can only be assigned to once, a final variable of a primitive or immutable type is a constant. However, that is not the case for a variable of a mutable reference type, since keeping a reference value constant does not prevent the object value of the variable from changing.}\)
in the *init* subtree. In the absence of an initializer expression, the variable is marked as depleted—thus it appears in the Depleted column after the statement, and is not in any Fact or Obligation. If there is an initializer, the expression is expanded *before* the introduction of the new variable as in Section 5.2, and the new variable is assigned the result of the expression when it is introduced.

Let’s use the block `{NaturalNumber x = s.pop(); NaturalNumber y;} to show by example how a variable declaration can affect a tracing table. Table 5.13 presents the complete tracing table for this block statement. In it, we can see how the code that evaluates the initializer expression for *x* appears before the introduction of the variable. Similarly, the assignment is treated as if it were an assignment expression, depleting the temporary variable from which we obtain its value. Note that the uninitialized variable *y* appears in the Depleted column. Being marked as depleted requires *y* to be assigned a value before it can be used, and this has the same effect on it as being uninitialized

5.3.4 The Single Expression Statement

A single expression statement is of the form `expression;` and is incorporated into our tracing table by expanding the evaluation of the expression as described in Section 5.2. This means that the semicolon ending the statement is missing from our table. Since an expression evaluates to something—i.e., a variable of a specific type—and a statement does not, in cases where the expression is not an assignment,

---

70We could have added an Uninitialized column to our table. However, since the only way for a variable to stop being uninitialized is the same way in which a variable stops being depleted, we felt this would have been overly pedantic. Conversely, labeling depleted variables as uninitialized would have been confusing, since they actually were initialized at one point.
the temporary variable created to store the value of the expression is never used in our tracing table.

5.3.5 Branching Statements

This section addresses both if and if-else statements. The syntax of these differs from RESOLVE’s in two main aspects: the control expression can have side-effects, and the controlled code is a statement instead of a block. The former creates the same problems as the ones addressed when dealing with short-circuit evaluation. The latter allows for extra flexibility in the source code. In its simplest use, it allows a programmer to avoid using the {} brackets when the control is used over a single statement—we generally still prefer the use of brackets in this case. However, it also
allows the formation of \texttt{if-else if} chains, where all of the options appear in the same level of indentation in the code\textsuperscript{71}.

Regardless of whether there is an \texttt{else} statement, the execution of an \texttt{if} statement splits our tracing table into two different paths. One of the paths is the result of \texttt{cond_expr} evaluating to \texttt{true}—i.e., it has the temp variable from the result of its evaluation as part of its path condition—and it includes the evaluation of the \texttt{then_stmt} statement. The second path is the result of \texttt{cond_expr} evaluating to \texttt{false}, and continues with the evaluation of the \texttt{else_stmt} statement (if present), before moving into the statement that follows it. Pictorial representations of these possible execution paths are provided in Figures 5.11 and 5.12.

From our previous discussion on short-circuit evaluation, we know that the presence of the boolean operators \&\& or || inside of \texttt{cond_expr} results in the creation of two new paths just from their evaluation. In turn, each of those paths then splits into

\textsuperscript{71}Since the \texttt{if} statement is a statement, this is equivalent to having the second \texttt{if-else} nested within a block of the first’s else, however the chain-like presentation is preferred due to its improved readability.
Figure 5.11: A flow chart for an if-else statement.

Figure 5.12: A flow chart for an if statement.
two: one that executes the *then_stmt* statement, and one that doesn’t. However, the paths created from a short-circuit evaluation return a *boolean* literal instead of a variable\(^{72}\). This means that the path condition for one of the tracing tables is known to be *false*. A *false* path condition implies that the code is unreachable under the circumstances that led us to that point, and thus there is no need to continue generating facts for that table. This means that in the presence of short-circuit evaluation of the conditional *cond_expr*, we only generate three tracing tables, one for each possible execution path, discarding the unreachable one.

This type of simplification is used in other control-flow statements, too. Whenever the result of the control expression (*cond_expr*) is known—such as in a *while(true)*; statement—that knowledge is used to reduce the number of tracing tables. It is also worth noting that this simplification does not change the meaning of a tracing table. A *false* path condition makes all of the VCs generated from a tracing table after that point trivially valid, since all of them include the literal *false* in their givens.

Table 5.14 presents the template for the evaluation of the *then_stmt* statement in a tracing table. Tables 5.16 and 5.15 present the templates for evaluation of the *if* statement when the *cond_expr* is false when there is an *else_stmt* statement or *not*, respectively.

### 5.3.6 While Statements

As in RESOLVE, verification of code containing looping constructs requires a loop invariant. The AST for a *while* statement is presented in Figure 5.13; notice that the *inv* subtree is not optional. The possibility that the loop condition can

\(^{72}\)The result of an expression that short-circuits, skipping its second half, is known regardless of the expression. That is, a short-circuited *&&* will always evaluate to *false*, while a short-circuited *||* will always be *true*. 

165
Table 5.14: A template tracing table for the execution of the `then_stmt` statement in a linear tracing table. Note that the result of `cond_expr` is assumed to be `true` on the path condition of state index $j$. The other columns would be filled in as appropriate.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.15: A template tracing table for skipping the `then_stmt` statement in the absence of an `else_stmt` statement in a linear tracing table. Note that the result of `cond_expr` is assumed to be `false` on the path condition of state index $j$. The other columns would be filled in as appropriate.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.16: A template tracing table for the execution of the `else_stmt` statement in a linear tracing table. Note that the result of `cond_expr` is assumed to be `false` on the path condition of state index $j$. The other columns would be filled in as appropriate.
change the variable state raises the question: should the invariant apply before or after the evaluation of the loop condition? Another way of looking at this is: where are the side-effects of evaluating cond accounted for? The answer determines where in the execution path of a program the invariant must be confirmed, and where it can be assumed. The changing of state as a result of the conditional test also has repercussions on the variant; in particular, we need to decide where to check for its lower bound\textsuperscript{73}.

These questions leave us with two reasonable yet different answers. We could consider the evaluation of the conditional to be the first thing in the loop body, and assert the invariant before the cond expression is ever evaluated. This leaves us with the execution paths presented in Figure 5.14. Alternatively, we could require the invariant to be true after the conditional expression is checked (at points A or B of our diagram). Checking the invariant only after we first enter the loop (at point

\textsuperscript{73}I.e., we need to show not only that the @decreases clause is reduced by the body of the loop, but also that it is greater than zero whenever we enter the loop.
B) might seem to be the more flexible approach, since we would not have to prove
the invariant for states where the loop body is never executed. However, this has
the disadvantage of creating two possible paths out of the loop\(^74\). In the presence
of short-circuit evaluation, this means that the split in execution paths due to the
conditional expression would happen twice: once before we enter the loop and, for
each of those paths, again when we trace the code back to the loop invariant\(^75\). The
added tracing tables would not be a problem if we truly gained flexibility from this
approach, but in reality the two methods are equally flexible: we can prove the same
programs with both approaches by carefully choosing a loop invariant.

Enforcing the invariant as it is done in Figure 5.14 or point A is also equivalent.
Our preference for enforcing it before the execution of the conditional expression stems
from the fact that our reasoning system advances the same way as the program.
This makes the state before the test begins execution the one that is favored in
our reasoning system. However, if our system were to start proofs at the end of a
program, as Hoare Logic does [24], it might be sensible to enforce the invariant at
point A instead.

It is worth noting that, unlike most other statements, there are no frame facts after
the invariant. However, there are modes that provide a similar function. The declared
variables whose values are changed by a loop are singled out in the \texttt{@updates} clause
of the loop, while all other declared variables in scope are assumed to be restored.
This means that the invariant is not just the expression after the \texttt{@maintains} clause,

\(^{74}\)One in which the loop never executes, and one after the loop body is executed at least once.

\(^{75}\)This means, for example, that for a conditional with an `||` operator, where two paths would
enter the loop body, each of them would be split into two again as we try to prove the invariant is
maintained.
Figure 5.14: A control-flow diagram for a \texttt{while} statement in our discipline. Note that the invariant is checked \textit{before} the start of the evaluation of the conditional expression.
but also the fact $x = \#x$ for each $x$ that is a variable in scope but not mentioned in the `@updates` list\textsuperscript{76}.

The execution path in Figure 5.14 forces us to confirm that the invariant is true before the evaluation of `cond` begins. This is done in our tracing table by having one of the “programmatic” rows contain the JavaDoc with the invariant, with the expression of the invariant as the obligation before it. After this check is done, the only fact that remains is the invariant. In our template tracing table, the appearance of $Inv(i, j)$ refers to the conjunction of the `@maintains` clause with all the results of the implicit `@restores` clause, where a variable preceded with a `#` is replaced by the value of variable at state index $i$, and one without the `#` sign is replaced by the value of the variable at state index $j$\textsuperscript{77}.

There are some interesting consequences of this system. If a variable has a scope larger than the body of the loop, then the loop body may not deplete it. This is because it is impossible to prove the loop invariant at the end of the loop body if it depends on a depleted value. Remember, there are no Facts about a depleted variable in our tracing table, making it impossible to know anything about the value of said variable. Thus, whenever a variable is depleted in the body of a loop, we report an error when we are asserting the loop’s invariant.

Since we are focused on full verification, we need to prove termination of each loop. This is done by using a variant, or progress metric, that is introduced by the `@decreases` clause. As in Chapter 2, the `@decreases` clause is a function that\textsuperscript{76}Note that, due to the lack of a frame fact and since temporary variables are not part of the invariant, we are unable to express anything about the values of any of the temporary variables after we assert the invariant.

\textsuperscript{77}For example, if the invariant is `@maintains x + y = \#x + \#y`, then $Inv(i, j)$ would be equivalent to $x_j + y_j = x_i + y_i$. 

170
evaluates to a mathematical natural number ($\mathbb{N}$), that is decreased by the execution of the loop body\(^{78}\). Since our tools treat natural numbers as non-negative integers ($\mathbb{Z}$), we want to show that the \texttt{@decreases} clause is a non-negative integer whenever we enter the loop body. Thus, in the state right after the invariant is asserted, we add an obligation stating that the \texttt{@decreases} clause evaluates to a non-negative number (for example, $Dec(i) \geq 0$, for state index $i$). Similarly, at the point where the loop body ends, we need to check that we made progress. This is done by checking that the progress metric is smaller than it was. That is, we add the obligation $Dec(j) < Dec(i)$ to state index $j$, where $i$ and $j$ are the state indexes before and after the loop body, respectively. A sample application of these template tables can be found in the verification presented in Section 5.6.2.

\(^{78}\)Our tools currently do not handle the general case that the progress metric is an ordinal-valued function, though this is necessary for relative completeness.
### Table 5.18: A template tracing table for the execution of path through a *while* statement’s body in a linear tracing table. Note that state indexes $i$ and $i+1$ are the same as in Table 5.17, and that the table ends at state index $k$.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$1 \text{ Inv}(i,i)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$@\text{updates}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$@\text{maintains}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$@\text{decreases}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i+1$</td>
<td>$1 \text{ Inv}(i,i+1)$</td>
<td>$1 \text{ Dec}(i+1) \geq 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{let } @\text{temp}x_{j} = \text{eval( cond_expr )}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j$</td>
<td>$1 @\text{temp}x_{j}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{let } @\text{temp}x_{j} = \text{eval( body_stmt )}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$k$</td>
<td>$1 @\text{temp}x_{j}$</td>
<td></td>
<td>$1 \text{ Inv}(i,k)$</td>
<td>$2 \text{ Dec}(k) &lt; \text{Dec}(i+1)$</td>
<td></td>
</tr>
</tbody>
</table>

#### 5.3.7 Break and Continue Statements

The unlabeled *break* statement stops execution of the nearest loop, letting execution continue after the looping statement. It should not be surprising, then, that a tracing table containing a *break*; closes all of the blocks opened after we entered the body statement before continuing execution on the same path as the table that exits the loop.

The unlabeled *continue* statement ends the execution of the loop body, returning to the conditional check. Thus, a tracing table that encounters a *continue*; statement treats it the same way we treated the end of the loop body. That is, the state index before it contains the invariant and variant obligations. Further, since no new statements are executed after the *continue*; statement, there is no state index after it; a similar thing happens with *return* statements.
5.3.8 For Loops

These loops provide a compact way of describing loops aimed at iterating over a fixed dataset. They do not, however, introduce any new functionality. That is, for any for-statement, an equivalent while loop can be generated. Thus, our approach is to translate the AST of a for into a while loop instead of creating new rules for them. Java allows for two different types of for loops: the standard for loop (with three statements in its header) and the for-each loop (intended to simplify code that uses an iterator). The translation techniques for both of them are presented below.

Equivalence for Standard For Loops

A standard for loop has the form presented in Figure 5.15. The optional «init» can be either a statement or an expression sequence. In either case, it is always executed before the invariant is asserted. If it is an expression sequence, then each expression is treated as an individual statement. Thus, in the translation presented in Figure 5.16, the contents of the «init» text are moved ahead of the loop’s documentation. Note that, in the cases where «init» is a variable declaration, the variable scope is to be restricted to the for loop. The entire result of the translation is within its own block to ensure that this is the case.
The «cond_expr» is the same conditional expression as the one in our equivalent loop. Though this is technically optional, we disallow missing conditionals in order to improve code readability. Finally, the optional «update_expr» expression list is executed every time after «body_stmt». This means that it is moved right after «body_stmt» in our translation, and inside a new block to keep both of them grouped together.

Finally, «body_stmt» of the for loop is the first statement in the block of our equivalent while statement. The introduction of blocks ensures that all the statements and expressions execute in the correct order, as well as keeping the scopes of the variables declared in an «init» statement limited to the loop.

**Equivalence for For-Each Loops**

For-each loops are specifically designed to facilitate iteration over objects that implement the java.lang.Iterable<T> interface. They do this by hiding from the source code any mention of the iterator obtained from that interface. That is, the
/* *
 * Documentation of invariant and variant *
 */

for(«type» «varName» : «iterable_expr») «body_stmt»

Figure 5.17: A template of for-each loop statements.

code using these loops does not have calls to the iterator() method, nor to hasNext
() and next() on the iterator that was obtained. However, the differences are just
cosmetic, and the code generated from these loops does exactly what one would expect
it to do.

It is difficult to document what a loop does when its main variable is hidden away.
Since the code is mostly using an Iterator, it is most likely that the invariants and
variants of the loop involve it. Yet, this Iterator variable is not named in our code,
leaving us with a need to create a naming convention for it. Luckily, we already had a
similar problem with the Kernel interfaces that extended the java.lang.Iterable<T>
interface, and our solution borrows heavily from that.

Consider the for-each loop declaration in Figure 5.17, where «iterable_expr» is
an expression that evaluates to an Iterable variable. Our documentation uses the
textual representation of this expression (most likely an identifier) preceded by the
~ symbol to represent the iterator in the documentation for the loop\textsuperscript{79}. This allows
the documentation to address the values of the “hidden” iterator. Further, since no

\textsuperscript{79}In the cases where the text representing the expression involves method calls and loops, the iden-
tifier created must have no spaces so that it can be parsed correctly. Remember, we are presenting
a “programmatic” equivalent, but we will not attempt to compile the translated code.
variable name in Java can start with a ~ character, there cannot be a declared variable in scope with the same name.

However, it is still possible to have code that uses two iterators from the same variable (in which case there is a possibility of a naming conflict). The solution to this problem depends on whether the second iterator is used inside the loop’s `body_stmt`, or by another for-each loop after the current loop terminates. Obtaining a second iterator from the collection object that is being iterated over, like calling any other method on it, depletes the original iterator because the iterator variable is encumbered by the collection variable. For obvious reasons, depleting an iterator within the body of a for-each loop is reported as an error, thus preventing the original naming conflict. The name conflicts that could arise from multiple non-overlapping iterations are addressed by restricting the scope of the iterator. The translation of a for-each loop into a while statement happens within its own block. This ensures that the ~-named variable is only in scope for the length of the iteration, and a second variable from that name would be completely independent from the previous one.
Figure 5.18 presents the result of the translation of the template presented in Figure 5.17 into a while loop. Note that, while slightly more cumbersome than the code for a regular for loop, it is still a straightforward matter of placing a part of the original into parts of the new loop\textsuperscript{80}.

5.4 Methods

The process of verifying that a method does what its contract says it should do works exactly the same way as in RESOLVE. That is, we assume the precondition of the method at the beginning of the code, by adding it to our Facts. Similarly, we ensure the postcondition of the method by adding it as an Obligation whenever a method’s execution terminates by adding it to the Obligations of the last state index of our linear tracing table.

Unlike RESOLVE, however, execution of a method can end in multiple places in Java. In a method that does not return a value—i.e., void—the Obligation is added to the last reachable state index, that is, the one right after the last expression in our tables. In a method that returns a value, every path terminates by reaching a return statement, and the postcondition of the method becomes an Obligation for the state index before the return\textsuperscript{81}. This is why, in our previous discussion of statements, we deferred talking about the return statement: it is more related to our treatment of methods than to other statements.

For instance methods, the distinguished formal parameter this can have more than one meaning. In cases where no instance variables are declared inside of the

\textsuperscript{80}A very familiar task for those familiar with MadLabs\textsuperscript{TM}.

\textsuperscript{81}Unlike the last statement of a method, there is no state index after it, since the execution of the method body stops at that point. There is effectively a return; statement at the end of every method body that is optional in Java.
class, the implementation of the method is considered to be layered. Thus, the formal parameter this refers to its abstract mathematical model value. The distinguished parameter is no different in this case from any other formal parameter.

In cases where the class implementing the method has instance variables, the formal parameter this can also refer to a tuple containing all of the instance variables.\footnote{This is, in effect, a representation record in the same vein as the one discussed in Section 2.1.3.}

The presence of instance variables also indicates that the class containing the methods is implementing an abstraction, invoking the same rules for our @convention and @correspondence clauses introduced in Section 2.1.3.

In cases where an instance method is in a class containing instance variables, the @convention clause implicitly includes assertions that no instance variable is depleted or encumbered. Thus, the tracing tables for the method start with the assumption that all variables are neither encumbered nor depleted, and check that this remains so at the end of the method’s execution. This is a rather onerous requirement that often prevents the direct implementation of “linked data structures” that require direct manipulation of references, but those are not the focus of this work. Further, they could be independently verified and still used with our tools with appropriate interface specifications.

### 5.5 Verifying a class

The tool presented here is not capable of verifying data structures that rely on pointers, such as linked lists. However, in our discipline, those components are limited to a small Kernel class, from with other data structures can be implemented in a layered fashion. For example, there are 8 classes in Figure 5.19 that provide different
functionality (some of them even have different math models). The tool introduced in this Chapters is able to verify all of them but one: List2. That is, though the use of encapsulation, we are able to verify multiple data structures that can be efficiently implemented on top of a small set of components that rely on pointer-based behavior.

5.6 Sample Verification of Methods

This section provides various tracing tables for different examples. The goal is to demonstrate how the rules of the discipline are applied for different types of programs.

5.6.1 Completing the Tracing Tables for isPalindrome

Let’s start by finishing the tracing tables of a method for which tracing tables have already been (mostly) completed: isPalindrome from Listing 5.1. The three tracing tables for the expression before the return statement have been expanded
in Section 5.2.9. We expand Table 5.8 to include the code row corresponding to the

return statement in Table 5.19.
Table 5.19: A complete tracing table for execution path \( \alpha \) of Figure 5.8 representing flow chart for Listing 5.1.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let ( @temp1 = \text{str.length()} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 ( @temp1 =</td>
<td>str_1</td>
<td>)</td>
<td></td>
<td>( @temp1 \leq 1 )</td>
</tr>
<tr>
<td></td>
<td>2 ( \text{str}_1 = \text{str}_0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let ( @temp2 = (@temp1 &lt;= 1) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 ( @temp2_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 ( \text{str}_2 = \text{str}_1 )</td>
<td></td>
<td>( @temp2_2 = (\text{str}_2 = \text{rev} (\text{str}_2)) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 ( @temp2_1 = @temp1 )</td>
<td></td>
<td>( @temp2_2 = @temp1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \text{str}_0 = \text{str}_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>return ( @temp2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \text{public static boolean isPalindrome(String str)} \)
Note that the only differences between Table 5.8 and Table 5.19 are the additional `return` statement after state index 2, and Obligations 2.1\textsuperscript{83} and 2.2. Obligation 2.1 is a direct application of the `@ensures` clause `isPalindrome = (str = rev(str))`, with the name of the function replaced by the returned variable and the parameter `str` replaced by the properly indexed value. Note that, in the same way as in RESOLVE, had the `@ensures` clause used `#str` instead, Obligation 2.1 would have used `str_0`.

Finally, Obligation 2.2 is the result of `str`'s parameter mode. Since no parameter mode was specified for it, it has the default parameter mode of `@restores`. That is good news, since any parameter mode that requires the value of `str` to change would have made the method unverifiable (Strings are immutable, after all).

The tracing tables for the execution paths ending in B and C add the same Obligations to the last state index of the table, while updating their returned variable to match the temporary variable that has the result of the expression for each execution path. That is, Table 5.10 would return `@temp11` after state index 11, with the same obligations now in state index 11. Similarly Table 5.9 would return `@temp7`.

5.6.2 A Layered Implementation of transferFrom

This example introduces the tracing tables for the `transferFrom` method from `QueueSecondary` in Listing 4.13. It is interesting for two reasons: it is an instance method and it uses a `while` loop. Figure 5.20 presents the flow chart for the method. Notice that, as explained in Section 5.3.6, the specification of the loop is included as a part of the program, with the expressions of the loop’s conditional as part of the loop’s body.

\textsuperscript{83}This is the same shorthand notation from Chapter 2, standing for Obligation 1 at state index 2.
Figure 5.20: A complete flow chart (including subexpressions) for the layered implementation of `transferFrom` for `QueueSecondary` as presented in Listing 4.13.

Table 5.20 presents the tracing table generated from the execution path that exits the method at A. Notice that, since the execution path goes through the loop invariant, it is irrelevant how many iterations of the loop happened before the conditional became `false`. That is, even though this tracing table does not have a single iteration though the loop’s body, it (combined with Table 5.21) is sufficient to prove that the code in the method (including the loop’s body) satisfies the contract for the method\(^{84}\).

In both Table 5.20 and 5.21, state index 2 is the branching point generated by the `while` loop. Table 5.20 exits the loop with the path condition \(\neg@temp2\). Since there are no statements after the loop body, the only thing left in the table is to check

\(^{84}\text{As long as the loop invariant is maintained by the loop’s body.}\)
that the postcondition of the method is satisfied. Obligations 4.1 and 4.2 are a direct result of the @ensures clause for the method.

Table 5.21 proceeds with the path condition @temp2 (i.e., it enters the loop body). In state index 2 we can see the requirement that the @decreases clause remains non-negative. Satisfying the clause at this point makes sure that its value will never be a negative integer (the invariant is the only Fact the proof can use, and it remains the same through all executions). State index 6 introduces two Obligations: maintaining the loop invariant (6.1) and making progress (6.2). These two Obligations would be the same Obligations that would appear before a break; statement. Unsurprisingly, adding a break; as the last line of our body would only alter Table 5.21 in that a new code row would be added (nothing about the variable states would change).
### Table 5.20: A complete tracing table for execution path A of Figure 5.20.

<table>
<thead>
<tr>
<th>Index</th>
<th>Path Cond.</th>
<th>Facts</th>
<th>Obligations</th>
<th>Depleted</th>
<th>Encumbered</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\text{@Override}$</td>
<td>$\text{transferFrom(Queue&lt;T&gt; source)}{}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>$\text{this}_1 = ()$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$\text{source}_1 = \text{source}_0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{@updates this, source}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{@maintains this * source = #this * #source}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{@decreases</td>
<td>source</td>
<td>}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>$\text{this}_2 * \text{source}_2 = \text{this}_1 * \text{source}_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{</td>
<td>source}_2</td>
<td>\geq 0$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$\text{@templ}_3 = \text{</td>
<td>source}_3$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$\text{source}_3 = \text{source}_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>$\text{this}_3 = \text{this}_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{~@temp}_{24}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{@temp}<em>{24} = \text{@temp}</em>{14} &gt; 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{@temp}<em>{14} = \text{@temp}</em>{13} &gt; 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{this}_4 = \text{this}_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{source}_4 = \text{source}_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{</td>
<td>source}_4</td>
<td>\geq 0$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{this}_4 = \text{source}_0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>$\text{source}_4 = ()$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Table 5.21: A complete tracing table for execution path that goes through the loop in Figure 5.20.
5.6.3 A Layered Implementation of contains

This example introduces the tracing tables for the contains method implemented in QueueSecondary in Listing 4.13. Unlike the contains methods in KeY—introduced in Listings 3.1, 3.2, and 3.4—the method presented here uses the equals method to check for equality, yielding the proper result.

This method is also of interest due to its use of a for-each loop. As explained in Section 5.3.8, the loop is translated by the tool into an equivalent while loop, with the implicit iterator for this declared as the temporary variable ~this, and initialized by making a call to the iterator() method. Figure 5.21 presents the flow chart for the method after the loop has been translated.

As explained before, the loop’s body has the statement \( T = y = ~\text{this}.next() \) expanded into the first two expressions in the loop body. As in the previous example, the specification of the loop is included as a part of the program, with the expressions of the loop’s conditional as part of the loop’s body.

In Figure 5.21 we can see that there are three possible execution paths for this program. The first one ends when the loop terminates due to its condition, with the method evaluating to false. Table 5.22 presents the linear tracing table for this execution path. In it, we can see that the Encumbered column records the relationship between this and its iterator, ~this.

Table 5.23 is created from the execution path that leads to the loop body terminating. This table also includes the loop invariant, that it is maintained and that progress is made, as shown in Obligations 7.1-4 and 7.5 respectively. In state index 6 we can see how the assignment expression replaces the name of the variable on the
Figure 5.21: A complete flow chart (including subexpressions) for the layered implementation of contains for QueueSecondary as presented in Listing 4.13.
right (@temp3) with a variable on the left (y). This allows for the Encumbers column to show the relationships among y, this, and the ~this iterator.

Finally, Table 5.24 results from reaching the return true statement inside of the loop. Note that, since the return statement effectively ends the loop’s execution, we are not required to show that the loop invariant is maintained, nor that we make progress. However, the tracing table does contain the Obligations that the @ensures clause of the method is met at the point when execution terminates.

Notice that Tables 5.23 and 5.24 differ only on the Path Condition 7.2 and the Obligations in state index 7. This should not be surprising, because from Figure 5.21 we can see that the execution paths only split in the last state index, rendering most of the tables equal\textsuperscript{85}.

\textsuperscript{85}State indexes 3 to 6 were included in Table 5.24 for context, but could have easily been omitted in the same way as the first two state indexes.
Table 5.22: A complete tracing table for execution path ending in `return false` in Figure 5.21. Notice that Obligations 2.3 and 2.4, while trivially true, are the result of the `@restores` clauses added to the loop invariant.
Table 5.23: A complete tracing table for execution path repeating the loop in Figure 5.21. The rows corresponding to state indexes 1 and 2 can be found in Table 5.22.
Table 5.24: A complete tracing table for execution path ending in \texttt{return true} in Figure 5.21. The rows corresponding to state indexes 1 and 2 can be found in Table 5.22.
5.6.4 Methods For Classes Implementing Data Representations

All the methods the previous examples were layered, acting as a client of all the variables involved. That is, none of them were part of a class implementing the representation of an object. Now, we introduce two examples that do that on this Section. The class Queue3 is an implementation of a Queue on top of a Sequence. The complete contracts for them can be found in Appendix B. The Sequence family of components behaves in similar fashion to the List in java.util: a Sequence is a string of T in which methods allow for access to an element inside it, regardless of position in the string. The Kernel interface, in Listing B.3, provides the add method that adds an element to the Sequence at the specified location; as well as the remove method that removes an element from the specified location.

The abstraction function is this = $this.entries. Here, the dollar sign before the this keyword allows us to distinguish between the abstract value of the object being implemented and its representation. The clause states that the value of the mathematical string model of this is the abstract value of the entries Sequence.

Table 5.25 presents the tracing table for the enqueue method for Queue3. Table 5.26 is the tracing table for the implementation of the dequeue method, which is interesting because the method has a @requires clause. In both tables, the first Fact of the first state index, as well as the last Fact of the last state index, are a direct result of the @correspondence clause. In both tables we can see how the contract of the implemented method refers to the abstract value of this. Conversely, we can see that the contracts for the called methods only refer to $this, the name of the implicit record holding the instance variables of the representation.
Index | Path Cond. | Facts | Obligations | Depleted | Encumbered
--- | --- | --- | --- | --- | ---
| @Override | public final void enqueue(T x) { | 0 | 1 $this_0 = $this_0.entries | | |
| let @temp1 = this.entries.length() | 1 | @temp1 = $this_1.entries.length() | | 1 $0 \leq @temp1 | 0 $| |
| | 2 $this_1.entries = $this_0.entries | 2 $@temp1 \leq | $this_1.entries | |
| | 3 $x_1 = x_0$ | } $\ast$ | $this_1.entries$ | |
| this.entries.add(@temp1, x) | 2 | $this_2.entries = $this_1.entries(0, @temp1) \ast$ | $x_1 \ast$ $this_1.entries[@temp1, | |
| | (x_1) \ast$ $this_1.entries[@temp1, | | $this_1.entries]$ | |
| | @temp1 = @temp1 | 1 $this_3 = this_0 \ast (x_0)$ | 1 $x_2$ | |
| } |

Table 5.25: A complete tracing table for the enqueue method for Queue3.

Index | Path Cond. | Facts | Obligations | Depleted | Encumbered
--- | --- | --- | --- | --- | ---
| @Override | public final T dequeue() { | 0 | 1 $this_0 = $this_0.entries | 1 $0 \leq 0$ | |
| let @temp1 = this.entries.remove(0) | 2 $this_0 \neq ()$ | 2 $0 < |$this_0.entries|$ | |
| | 1 $this_1.entries = $this_0.entries(0, 0) \ast$ | $this_0.entries[0 + 1, | |
| | $\ast$ $this_0.entries]$ | | $this_1.entries$ | |
| | @temp1 = @temp1 | 1 $this_0 = @temp1 \ast this_1$ | | |
| | this_1 = $this_1.entries$ | } return @temp1 |

Table 5.26: A complete tracing table for the dequeue method for Queue3.
In Chapter 3, we introduced the shortcomings of current reasoning systems that do not adequately incorporate abstraction into their reasoning. These systems address what their authors saw as a problem when verifying code in languages with reference semantics. Our work presented in Chapter 4 to address this shortcomings does not introduce a fully general approach, either, since it remains incapable of verifying pointer-based data structures. We leave the verification of pointer-based data structures to tools that use separation logic and dynamic frames, which are nicely complementary. The goal of the discipline and tools introduced in this work is to prove client code that uses those lower-level data structures via nicely abstracted interface specifications.

As with RESOLVE, the discipline introduced in this work works on all possible levels of software by modularization and abstraction. Thus, even though the strategy might not work at the lowest level with pointer-based data structures, the verifier introduced here is one that can verify arbitrarily deep layering of client code on top of the low-level data structures that use pointers. That is, since the client code only needs to consider one level of interfaces below, the client is able to abstract away the complexity of the components the client code might be manipulating. Verification is
thereby modular, hence possibly tractable; evidence of actual successful verification demonstrates tractability in practice.

Chapter 4 introduced RESOLVE Java, an adaptation of RESOLVE contracts to Java that allows a language with reference semantics to be verified in terms of the abstract object values of the variables it manipulates. The main contribution of this chapter is the adaptation of contracts from RESOLVE (a language with value semantics) to Java (a language with reference semantics), so that a disciplined approach to the treatment of variables that might be aliased by a method permits modular verification of the Java code. The use of the two new parameter modes—@depletes and @encumbers—and the rules associated with them constitute a novel approach to advertising aliasing.

Chapter 5 presented the steps necessary to construct a proof of correctness for a program following the discipline in Chapter 4. Through the use of linear tracing tables, we can account for the various complexities introduced by Java’s nested expressions as well as its use of short-circuit evaluations. In addition, the use of @depletes and @encumbers annotations allows tools to perform static checks of compliance with the discipline that supports modular verification. What’s more, the verification conditions generated by this tool abstract away the complexities of Java’s reference semantics, allowing us to use the same provers that have been developed for RESOLVE.

6.1 Future Work

Section 3 presented shortcomings in current reasoning tools for Java—in particular expressiveness limitations of JML*—and discussed how hard these problems are to
solve given the current state of the tools. Yet it is possible that by embracing a
more cautious approach in the treatment of references, many of the behaviors that
prevent modular reasoning could be restricted. This could allow for documentation
that deals not only with references, but also with object values. It is likely that
such modifications would make the resulting code look much more like it follows the
discipline presented in this work. Yet, the ability to directly talk about references
would allow for verification of the lower-level pointer-based data structures that our
discipline does not currently address. It remains to be determined whether JML* can
be improved in this direction.

Conversely, the reasoning tools presented here focus only on the abstract values
of variables, completely ignoring dynamic types and reference values. We are able
to verify programs under these conditions by restricting what a programmer can do
with a reference. Yet, this approach is incompatible with data structures that rely
on certain uses of aliased pointers to realize a component—such as linked-lists. In a
similar way to what has been done in RESOLVE, we can efficiently implement the
behavior of most such components by layering on top of a List abstraction. Yet this
still leaves one low-level component that requires less-restricted reference behavior in
order to successfully verify it.

The verification of such a component would most likely heavily borrow from the
work presented in Chapter 3. Yet the combination with our discipline would require
the specification language to be expanded to account for abstract object values, and
the semantic rules to be relaxed to account for the fact that a reference cannot be
used to modify a variable’s abstract or concrete object value in a way that would
affect the abstract value of another variable.
The tool presented in Chapter 5 needs to be expanded to account for Java exceptions. The tools presented in Chapter 3 already do this but, as explained in [27], we find this methodology unsatisfactory. This leaves us with the future requirement of incorporating and adapting the error-management solutions presented in [27] to our tool. This will be especially important as we try to verify enterprise software, or even components with methods that can fail even if the preconditions of a method are satisfied (such as those dealing with file systems, networking, or parallel interfaces).

Much work has been directed into verifying parallel programs. The restrictions imposed on references by the discipline introduced in Chapter 4 would greatly simplify the effort of showing that two method executions do not share any variables—and thus cannot interfere with one another. This would make the detection of potential data-races much simpler, and potentially greatly expand both the number of methods that can be provably executed in parallel, as well as the quality of data-race detection tools. However, the amount of work needed to both introduce parallel constructs into our specification system, as well as developing tools that support them, puts this firmly in the realm of future work.
Appendix A: Referenced RESOLVE Contracts and Realizations

Listing A.1: UnboundedIncrement enhancement for UnboundedNaturalBaseFacility.

```plaintext
contract UnboundedIncrement
  enhances UnboundedNaturalBaseFacility
  procedure Increment(updates n: UnboundedNatural)
    ensures
    n = #n + 1
  end UnboundedIncrement
```

Listing A.2: UnboundedAdd enhancement for UnboundedNaturalBaseFacility.

```plaintext
contract UnboundedAdd
  enhances UnboundedNaturalBaseFacility
  procedure Add(updates n: UnboundedNatural,
                restores m: UnboundedNatural)
    ensures
    n = #n + m
  end UnboundedAdd
```
realization Recursive
  implements UnboundedIncrement for UnboundedNaturalBaseFacility

procedure Increment (updates n: UnboundedNatural)
  decreases n
  variable d, radix: Natural
  radix := Radix(n)
  DivideByRadix(n, d)
  Increment(d)
  if AreEqual(d, radix) then
    Clear(d)
    Increment(n)
  end if
  MultiplyByRadix(n, d)
end Increment
end Recursive

Listing A.3: Recursive implementation of the UnboundedIncrement enhancement for UnboundedNaturalBaseFacility.
contract NaturalFacility
  definition MAX : integer
    satisfies restriction 0 < MAX

  math subtype NATURALMODEL is integer
    exemplar n
    constraint 0 <= n and n <= MAX

  type Natural is modeled by NATURALMODEL
    exemplar n
    initialization ensures n = 0

  function AreEqual (restores m: Natural, restores n: Natural)
    : control
    ensures
      AreEqual = (m = n)

  procedure Decrement (updates n: Natural)
    requires
      n > 0
    ensures
      n = #n - 1

  procedure Increment (updates n: Natural)
    requires
      n < MAX
    ensures
      n = #n + 1

  function IsGreater (restores m: Natural, restores n: Natural)
    : control
    ensures
      IsGreater = (m > n)

  function Max ()
    : Natural
    ensures
      Max = MAX

  function Replica (restores n: Natural)
    : Natural
    ensures
      Replica = n

end NaturalFacility

Listing A.4: The contract for NaturalFacility used in UnboundedNaturalBaseFacility.
realization SecondQueueRealization (  
    function AreEqual (restores i: Item,  
        restores j: Item) : control  
        ensures  
            AreEqual = (i = j))  
    implements SetTemplate  
    uses QueueTemplate  
    uses IsPositive for UnboundedIntegerFacility  
    facility QueueFacility is QueueTemplate(Item)  
        enhanced by Concatenate  
    type representation for Set is (items: Queue)  
        exemplar s  
        convention |s.items| = |elements(s.items)|  
        correspondence function elements(s.items)  
    end Set  
    procedure Add (updates s: Set,  
        clears x: Item)  
        Enqueue(s.items, x)  
    end Add  
    function Contains (restores s: Set,  
        restores x: Item) : control  
        variable temp: Queue  
        loop  
            maintains #temp * #s.items = temp * s.items and x = #x  
                and Contains = (x is in elements(temp))  
            decreases |s.items|  
            while not IsEmpty(s.items) do  
                variable y: Item  
                Dequeue(s.items, y)  
                Contains := Contains or AreEqual(x, y)  
                Enqueue(temp, y)  
            end loop  
            s.items :=: temp  
        end Contains  
    end SecondQueueRealization  

Listing A.5: The RESOLVE equivalent of the VCC Set.
**Listing A.6: Sample contract: SetTemplate.**

```plaintext
contract SetTemplate (type Item)
  uses UnboundedIntegerFacility

  math subtype SET_MODEL is finite set of Item

  type Set is modeled by SET_MODEL
    exemplar s
      initialization ensures s = empty_set

  procedure Add(updates s: Set,
    clears x: Item)
    requires
      x is not in s
    ensures
      s = #s union {x}

  procedure Remove(updates s: Set,
    restores x: Item,
    replaces xCopy: Item)
    requires
      x is in s
    ensures
      s = #s \ {x} and
      xCopy = x

  procedure RemoveAny(updates s: Set,
    replaces x: Item)
    requires
      s /= empty_set
    ensures
      x is in #s and
      s = #s \ {x}

  function Contains (restores s: Set,
    restores x: Item) : control
    ensures
      Contains = (x is in s)

  function IsEmpty (restores s: Set) : control
    ensures
      IsEmpty = (s = empty_set)

  function Size (restores s: Set) : Integer
    ensures
      Size = |s|

end SetTemplate
```
contract ListTemplate (type Item)
  uses UnboundedIntegerFacility

  math subtype LIST_MODEL is (left: string of Item,
      right: string of Item)

  type List is modeled by LIST_MODEL
    exemplar 1
      initialization ensures 1.left = empty_string
      and 1.right = empty_string

    procedure Insert(updates s: List, clears x: Item)
      ensures
      s.left = #s.left and
      s.right = <#x> * #s.right

    procedure Remove(updates s: List, replaces x: Item)
      requires
      s.right /= empty_string
      ensures
      s.left = #s.left and
      #s.right = <x> * s.right

    procedure Advance(updates s: List)
      requires
      s.right /= empty_string
      ensures
      s.left * s.right = #s.left * #s.right and
      |s.left| = |#s.left| + 1

    procedure Reset(updates s: List)
      ensures
      |s.left| = 0 and
      s.right = #s.left * #s.right

    procedure AdvanceToEnd(updates s: List)
      ensures
      |s.right| = 0 and
      s.left = #s.left * #s.right

    function LeftLength (restores s: List) : Integer
      ensures
      LeftLength = |s.left|

    function RightLength (restores s: List) : Integer
      ensures
      RightLength = |s.right|
end ListTemplate

Listing A.7: ListTemplate contract.
package components.queue;

import java.util.Iterator;

import components.sequence.Sequence;
import components.sequence.Sequence1L;

/**
 * @correspondence this = $this.entries
 */
public class Queue3<T> extends QueueSecondary<T> {

    /**
     * Private members --------------------------------------------
     */

    /**
     * Entries included in {@code this}.
     */
    private Sequence<T> entries;

    /**
     * Create initial representation.
     */
    private void createNewRep() {
        this.entries = new Sequence1L<T>();
    }

    /**
     * Constructors -----------------------------------------------
     */

    Listing B.1: The Queue3 class implementing QueueKernel. (Cont.)
Listing B.1: (Cont.)

```java
/**
 * Default constructor.
 */
public Queue3() {
    this.createNewRep();
}

/*
 * Standard methods ------------------------------------------
 */

@SuppressWarnings("unchecked")
@Override
public final Queue<T> newInstance() {
    try {
        return this.getClass().newInstance();
    } catch (ReflectiveOperationException e) {
        throw new AssertionError(
            "Cannot construct object of type " +
            this.getClass());
    }
}

@Override
public final void clear() {
    this.createNewRep();
}

@Override
public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    if (source instanceof Queue3<?>) {
        Queue3<T> localSource = (Queue3<T>) source;
        this.entries = localSource.entries;
        localSource.createNewRep();
    } else {
        super.transferFrom(source);
    }
}
```

Listing B.1: (Cont.)
Listing B.1: (Cont.)

```java
/*
 * Kernel methods ---------------------------------------------
 */

@Override
public final void enqueue(T x) {
    assert x != null : "Violation of: x is not null";
    this.entries.add(this.entries.length(), x);
}

@Override
public final T dequeue() {
    assert this.length() > 0 : "Violation of: this /= <";
    return this.entries.remove(0);
}

@Override
public final int length() {
    return this.entries.length();
}

@Override
public final Iterator<T> iterator() {
    return this.entries.iterator();
}

/*
 * Other methods (performance improvements) -------------------
 */

@Override
public final T front() {
    assert this.length() > 0 : "Violation of: this /= <>";
    return this.entries.entry(0);
}

@Override
public final T replaceFront(T x) {
    assert this.length() > 0 : "Violation of: this /= <>";
    return this.entries.replaceEntry(0, x);
}
```
package components.naturalnumber;

import components.standard.Standard;

/**
 * @mathmodel type NaturalNumberKernel is modeled by NATURAL
 * *
 * @initially
 * @default:
 * ensures
 * this = 0
 * int i:
 * requires
 * i >= 0
 * ensures
 * this = i
 * NaturalNumber n:
 * ensures
 * this = n
 */

public interface NaturalNumberKernel
    extends Standard<NaturalNumber> {

    /**
     * A constant, with value 10.
     */
    int RADIX = 10;

    /**
     * @updates this
     * @requires 0 <= k and k < 10
     * @ensures this = 10 * #this + k
     */
    void multiplyBy10(int k);

    /**
     * @updates this
     * @ensures
     * #this = 10 * this + divideBy10 and
     * 0 <= divideBy10 and divideBy10 < 10
     */
    int divideBy10();

    /**
     * @ensures isZero = (this = 0
     */
    boolean isZero();

}

Listing B.2: The NaturalNumberKernel interface.
package components.sequence;

import components.standard.Standard;

/**
 * @mathmodel type SequenceKernel is modeled by string of T
 * @initially default:
 * @ensures this = <>
 * @iterator ~this.seen * ~this.unseen = this
 */
public interface SequenceKernel<T> extends Standard<Sequence<T>>, Iterable<T> {

    /**
     * @depletes x
     * @updates this
     * @requires 0 <= pos and pos <= |this|
     * @ensures this = #this[0, pos) * <#x> * #this[pos, |#this|)
     */
    void add(int pos, T x);

    /**
     * @updates this
     * @requires 0 <= pos and pos < |this|
     * @ensures this = #this[0, pos) * #this[pos+1, |#this|) and
     * <remove> = #this[pos, pos+1)
     */
    T remove(int pos);

    /**
     * @ensures length = |this|
     */
    int length();
}

Listing B.3: The SequenceKernel interface.
package components.sequence;

public interface Sequence<T> extends SequenceKernel<T> {

    /**
     * @encumbers entry via this
     * @requires 0 <= pos and pos < |this|
     * @ensures <entry> = this[pos, pos+1)
     */
    T entry(int pos);

    /**
     * @depletes x
     * @updates this
     * @requires 0 <= pos and pos < |this|
     * @ensures <replaceEntry> = #this[pos, pos+1) and
     *         this = #this[0, pos) * <#x> * #this[pos+1, |#this|)
     */
    T replaceEntry(int pos, T x);

    /**
     * @updates this
     * @clears s
     * @ensures this = #this * #s
     */
    void append(Sequence<T> s);

    /**
     * @updates this
     * @ensures this = rev(#this)
     */
    void flip();

    Listing B.4: The Sequence interface. (Cont.)
Listing B.4: (Cont.)

```c
/**
 * @updates this
 * @clears s
 * @requires 0 <= pos and pos <= |this|
 * @ensures this = #this[0, pos) * #s * #this[pos, |#this|)
 */
void insert(int pos, Sequence<T> s);

/**
 * @updates this
 * @replaces s
 * @requires 0 <= pos1 and pos1 <= pos2 and pos2 <= |this|
 * @ensures
 * this = #this[0, pos1) * #this[pos2, |#this|) and
 * s = #this[pos1, pos2)
 */
void extract(int pos1, int pos2, Sequence<T> s);
}
package components.sequence;

import java.util.Iterator;

public abstract class SequenceSecondary<T> implements Sequence<T> {

/*
 * Public members ---------------------------------------------
 */

/*
 * Common methods (from Object) -------------------------------
 */

@Override
public final boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (!(obj instanceof Sequence<?>)) {
        return false;
    }
    Sequence<?> t = (Sequence<?>) obj;
    if (this.length() != t.length()) {
        return false;
    }
    Iterator<T> it1 = this.iterator();
    Iterator<?> it2 = t.iterator();
    while (it1.hasNext()) {
        T x1 = it1.next();
        Object x2 = it2.next();
        if (!x1.equals(x2)) {
            return false;
        }
    }
    return true;
}

@Override
public int hashCode() { ... }

@Override
public String toString() { ... }

Listing B.5: The SequenceSecondary abstract class partially implementing the secondary methods for Sequence. (Cont.)
Listing B.5: (Cont.)

```java
/*
 * Other non-kernel methods ------------------------------
 */

@Override
public T entry(int pos) {
  assert 0 <= pos : "Violation of: 0 <= pos";
  assert pos < this.length() : "Violation of: pos < |this|";
  int step = 0;
  Iterator<T> it = this.iterator();
  T entry = it.next();
  /**
   * @updates entry, it, step
   * @maintains it.seen * it.unseen = #it.seen * #it.unseen
   * and step = |it.seen| - 1 and
   * entry = it.seen[step, step + 1)
   * @decreases pos - step
   */
  while (step < pos) {
    entry = it.next();
    step ++;
  }
  return entry;
}

@Override
public T replaceEntry(int pos, T x) {
  assert x != null : "Violation of: x is not null";
  assert 0 <= pos : "Violation of: 0 <= pos";
  assert pos < this.length() : "Violation of: pos < |this|";
  T oldEntry = this.remove(pos);
  this.add(pos, x);
  return oldEntry;
}

@Override
public void append(Sequence<T> s) {
  /**
   * @decreases |this|
   */
  assert s != null : "Violation of: s is not null";
  assert s != this : "Violation of: s is not this";
  if (s.length() > 0) {
    T x = s.remove(0);
    this.add(this.length(), x);
    this.append(s);
  }
}
```

Listing B.5: (Cont.)

213
@Override
public void flip() {
    /**
     * @decreases |this|
     */
if (this.length() > 1) {
    T x = this.remove(0);
    this.flip();
    this.add(this.length(), x);
}
}

@Override
public void insert(int pos, Sequence<T> s) {
assert 0 <= pos : "Violation of: 0 <= pos";
assert pos <= this.length() : "Violation of: pos <= |this|";
assert s != null : "Violation of: s is not null";
assert s != this : "Violation of: s is not this";
int i = pos;
    /**
     * @updates this, i, s
     * @maintains this[0, pos) * this[i, |this|) = #this and
     * this[pos, i) * s = #s and
     * |s| + i = |#s| + #i;
     * @decreases |s|
     */
    while (s.length() > 0) {
        T x = s.remove(0);
        this.add(i, x);
        i++;
    }
}
Listing B.5: (Cont.)

```java
@Override
public void extract(int pos1, int pos2, Sequence<T> s) {
    assert 0 <= pos1 : "Violation of: 0 <= pos1";
    assert pos1 <= pos2 : "Violation of: pos1 <= pos2";
    assert pos2 <= this.length() :
        "Violation of: pos2 <= |this|";
    assert s != null : "Violation of: s is not null";
    assert s != this : "Violation of: s is not this";
    s.clear();
    int j = pos2;
    /**
     * @updates this, j, s
     * @maintains #this = this[0, pos1) * s * this[pos1, |this|)
     *           and pos1 + |s| = #pos1 + |#s|
     * @decreases |this|
     */
    while (pos1 < j) {
        //T x =
        s.add(0, this.remove(j - 1));
        j--;
    }
}
```
Bibliography


