Investigations in Automating Software Verification

Dissertation

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

By

Jason Kirschenbaum, B.S., B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Dissertation Committee:

Bruce W. Weide, Advisor
Paul Sivilotti
Michael Bond
Abstract

In order to have more confidence in software-based systems, we would like to be able to show automatically that a particular software component meets its specification; the component does what it is supposed to do. Testing can increase confidence in software; however, testing alone cannot show that software meets its specification, only that no errors have been found yet. We are building stronger tools for increasing our confidence in software in response to the verifying compiler grand challenge. The vision is that all software will include specifications for all of the operations of a component (including pre- and post-conditions) along with code annotations in a purported implementation (including loop invariants, and progress metrics for the termination of both loops and recursive operations). These annotations allow us to generate verification conditions (VCs) that represent the correctness of the code mathematically. The code meets its specification and is correct if the VCs are proved.

We focus on the problem of automating VC proofs and examining potential obstacles for automated proofs. More specifically, this dissertation defends the following three-part thesis:

1. Modest changes in programming languages may effectively support and facilitate automated verification without introducing an undue annotation burden on programmers.
2. An interactive proof assistant can be an effective back-end prover for automated software verification.

3. Translations between similar formal systems present significant challenges in fully automating software verification.
For my parents, Marc and Robin; and my grandparents, Hy, Honey, Robert and Jo Ann.
Acknowledgments

I would like to thank my parents, Marc and Robin, and my sister Irene. Their support throughout school was invaluable; whenever I was drained from my studies, I could talk to them and be instantly invigorated.

I have had the privileged of excellent mentors throughout my graduate career. Tunc Aldemir, Paolo Bucci, Harvey M. Friedman, Wayne D. Heym, William F. Ogden, Paul A. G. Sivilotti, Murali Sitaraman, Michael P. Stovsky, and Bruce W. Weide have all had insightful contributions and comments on my work. I am extremely grateful for the work that Harvey, Bill, and Wayne have done upon which my dissertation is based and for their help and guidance. I would also like to thank Michael and Tunc for employing me, an unfunded incoming PhD student, as a GRA and my advisor, Bruce, for his guidance, advice, and financial support; I will miss our regular meetings and lunch-time discussions. I would also like to thank Paul for his career advice and all of the members of my committee, Bruce, Paul and Mike, for their helpful comments and suggestions on both the form and content of this dissertation. I would be remiss if I did not thank the NRC, Air Force, NSF and Department of Computer Science for their financial support of my time at Ohio State as a graduate student.

I could not imagine graduate school without the friends I have made while in school. Many research ideas along with philosophical questions were bounced off of Bruce Adcock and Derek Bronish. Matt Lang and Brian Larkins both were seemingly
endless sources of humor and interesting conversation.  I would especially like to thanks Aditi Tagore, Diego Zaccai, Ted Pavlic, Sanket Tavarageri, and Vahid Schwart for their very helpful comments on my practice defense talk.

My girlfriend, Beth Pierson, was both a forcing function and a cheerleader—whichever was needed at the time—throughout my graduate school career. Beth, this dissertation would not be possible without your help; thank you.

Lastly, I would like to thank the Ohio State University. Its faculty, staff and students has shaped me for the past eleven years as both an undergraduate and graduate student. I will always look back fondly at my time here.

Jason Kirschenbaum
Columbus, Ohio
June 11, 2011
Vita

September 9, 1982 .......................... Born – Cleveland, OH

2004 ........................................B.S. Mathematics with Honors, The Ohio State University

2004 ........................................B.S. Computer Science with Honors, The Ohio State University

2008 ........................................M.S. Computer Science, The Ohio State University

2004 — Present ............................ Graduate Research Associate/Graduate Teaching Associate, The Ohio State University

Publications


**Fields of Study**

Major Field: Computer Science and Engineering

Studies in:

- Software Verification Prof. Bruce W. Weide
- Logic Prof. Tim Long
- Distributed Systems Prof. Paolo A.G. Sivilotti
# 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 Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 A Verifying Compiler</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Vision of Verified Software Development</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 The Programming Language Choice</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2 Mathematical Theories and Back-end Prover Choice</td>
<td>10</td>
</tr>
<tr>
<td>1.2.3 Impact of Formal Logic on Verification</td>
<td>11</td>
</tr>
<tr>
<td>1.3 Thesis Statement</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Dissertation Outline</td>
<td>13</td>
</tr>
<tr>
<td>2. Programming Language Support for Automated Verification</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Motivating Example</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Illustrating Example</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Sort Specification</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2 Quicksort Implementation</td>
<td>24</td>
</tr>
<tr>
<td>2.4 Introduction to the Syntax and Semantics of Restrictions: Sorting</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>25</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Restrictions</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Client Usage of Restrictions</td>
</tr>
<tr>
<td>2.4.3</td>
<td>More Uses of Restrictions</td>
</tr>
<tr>
<td>2.5</td>
<td>Full Syntax and Semantics for Restrictions within RESOLVE</td>
</tr>
<tr>
<td>2.5.1</td>
<td>RESOLVE Syntax, Semantics and Proof Rules</td>
</tr>
<tr>
<td>2.5.2</td>
<td>RESOLVE with Restrictions Syntax</td>
</tr>
<tr>
<td>2.5.3</td>
<td>RESOLVE with Restrictions Semantics</td>
</tr>
<tr>
<td>2.5.4</td>
<td>RESOLVE with Restrictions Proof Rules</td>
</tr>
<tr>
<td>2.5.5</td>
<td>A Small Lemma on the Semantics of RESOLVE Programs</td>
</tr>
<tr>
<td>2.5.6</td>
<td>Proof of Soundness and Relative Completeness</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Evaluation</td>
</tr>
<tr>
<td>2.6</td>
<td>Related Work</td>
</tr>
<tr>
<td>2.7</td>
<td>Future Work</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary and Conclusion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Use of Interactive Proof Assistant in Automated Verification</td>
<td>81</td>
</tr>
<tr>
<td>3.1</td>
<td>Requirements of a Back-end Prover</td>
<td>81</td>
</tr>
<tr>
<td>3.2</td>
<td>Categories of Automated Provers</td>
<td>83</td>
</tr>
<tr>
<td>3.2.1</td>
<td>SMT Solvers</td>
<td>84</td>
</tr>
<tr>
<td>3.2.2</td>
<td>First Order Theorem Provers</td>
<td>85</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Interactive Proof Assistants</td>
<td>86</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Special-purpose Decision Procedures</td>
<td>87</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Rationale for Examining Isabelle</td>
<td>88</td>
</tr>
<tr>
<td>3.3</td>
<td>RESOLVE Mathematical Theories</td>
<td>89</td>
</tr>
<tr>
<td>3.4</td>
<td>Embedding RESOLVE theories within Isabelle</td>
<td>91</td>
</tr>
<tr>
<td>3.4.1</td>
<td>String Theory Encoding</td>
<td>93</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Automated Proofs in Isabelle</td>
<td>98</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Additional Lemmas Needed by Isabelle for Automated Proofs</td>
<td>100</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Potential Proof Commands and a Proof Script for VCs</td>
<td>104</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Results</td>
<td>105</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Lessons Learned</td>
<td>108</td>
</tr>
<tr>
<td>3.5</td>
<td>Related Work</td>
<td>114</td>
</tr>
<tr>
<td>3.6</td>
<td>Future Work</td>
<td>117</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusion</td>
<td>118</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Impact of Formula Translation on Provability</td>
<td>119</td>
</tr>
<tr>
<td>4.1</td>
<td>Syntax of Many-Sorted Free Logic</td>
<td>120</td>
</tr>
<tr>
<td>4.2</td>
<td>Semantics of Many-Sorted Free Logic</td>
<td>122</td>
</tr>
<tr>
<td>4.3</td>
<td>Syntax of Many-Sorted Total Logic</td>
<td>125</td>
</tr>
</tbody>
</table>

xii
4.4 Semantics of Many-Sorted Total Logic ........................................ 126
4.5 Logical Notions Independent of the choice of Free or Total Logic ... 128
4.6 Mathematical Machinery Relating Many-Sorted Free and Total Logic ................................................................. 128
   4.6.1 Some Kinds of Total Interpretations of Partial Functions and Relations ............................................................... 129
4.7 Relationship Between Free and Total Many-Sorted Logics ............. 133
   4.7.1 Proof of Lemma .......................................................... 135
4.8 Practical Use of the Method .................................................... 141
   4.8.1 Definition of Logical Signature .......................................... 141
   4.8.2 Intended Model ............................................................ 143
   4.8.3 Model of Objects with Isabelle ......................................... 143
   4.8.4 Translation Rules .......................................................... 144
   4.8.5 Validation of Rules ........................................................ 153
   4.8.6 Limitations .................................................................. 157
4.9 Related Work ........................................................................ 158
4.10 Future Work ........................................................................ 159
4.11 Summary and Conclusion ....................................................... 160

5. Conclusions and Future Work ..................................................... 161
   5.1 Summary ........................................................................... 161
      5.1.1 Programming Language .................................................. 161
      5.1.2 Interactive Proof Assistants as an Automated Back-end Prover for VCs ................................................................. 162
      5.1.3 Translations Between Different Logical Systems ............... 162
   5.2 Future Work ....................................................................... 162
   5.3 Conclusion ......................................................................... 164

Bibliography ............................................................................... 165
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Multipliers for Function Meas Created by Heym [31]</td>
<td>73</td>
</tr>
<tr>
<td>3.1 Results Comparing Isabelle and SplitDecision on VCs</td>
<td>106</td>
</tr>
<tr>
<td>3.2 Results of Isabelle on VCs SplitDecision Does Not Prove Nor Refute</td>
<td>106</td>
</tr>
<tr>
<td>3.3 Results of SplitDecision on VCs Isabelle Does Not Prove Nor Refute</td>
<td>106</td>
</tr>
<tr>
<td>4.1 Formulas Used for Validation of Rules</td>
<td>155</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Envisioned Verified Software Development Infrastructure</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Set Template Facility Specification</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Set Template Intersection Extension and Implementation</td>
<td>9</td>
</tr>
<tr>
<td>1.4</td>
<td>Thesis Areas</td>
<td>13</td>
</tr>
<tr>
<td>2.1</td>
<td>Code to Compute $x^p$</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>QueueTemplate Contract and QueueConcatenateExtension contract</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Math Unit with Mathematical Definitions Used in the Sort Contract</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>QueueSortExtension Extension to QueueTemplate</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Quicksort Implementation of Sort Extension to QueueTemplate</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Dependency Diagram for Quicksort</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>OrderedQueue Restriction</td>
<td>26</td>
</tr>
<tr>
<td>2.8</td>
<td>SmallValueQueue Restriction</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>LargeValueQueue Restriction</td>
<td>28</td>
</tr>
<tr>
<td>2.10</td>
<td>QueueSortExtension Extension to QueueTemplate Using Restrictions</td>
<td>30</td>
</tr>
<tr>
<td>2.11</td>
<td>OrderedQueueConcatenateExtension Restriction</td>
<td>30</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>2.12 Quicksort Implementation of \texttt{Sort} Using Restrictions</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>2.13 Dependency Diagram for \texttt{RestrictedQuickSort}</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>2.14 MemoizingMachine Contract</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>2.15 PartialMap Contract</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>2.16 FunctionalMap Restriction</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>2.17 Implementation Using \texttt{FunctionalMapRestriction}</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>2.18 VC for \texttt{Define From FunctionalMapRestriction} contract</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>2.19 VC for \texttt{Undefine From FunctionalMapRestriction} contract</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>2.20 Definition of Mathematical Machinery</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>2.21 Notation for Environments</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>2.22 Additional Grammar for RESOLVE with Restrictions Added</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>2.23 Modified Grammar for RESOLVE with Restrictions Added</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>2.24 Redefinition of \texttt{alter all} Semantics</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>2.25 Auxiliary Function</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>2.26 Definition of the Semantics of Restrictions</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>2.27 Confirm Restriction Proof Rule</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>2.28 End Restriction Proof Rule</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>2.29 Restriction Declaration Proof Rule</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>2.30 Restricted Procedure Body Proof Rule</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>2.31 Restriction Procedure Call Proof Rule</td>
<td>57</td>
<td></td>
</tr>
</tbody>
</table>
2.32 VC Classification and Diagram of Category Relationships (Adapted From [36]) ................................................................. 75

3.1 String Theory ........................................................................... 90

3.2 Permutation Lemmas for Equivalence Relation ......................... 96

3.3 Permutation Lemmas ................................................................. 96

3.4 Permutation Equivalence Relation ............................................ 97

3.5 Definition of Concatenation ..................................................... 101

3.6 Concatenation Definition in Isabelle ....................................... 102

3.7 Concatenation Interpretation as a Monoid in Isabelle ............... 102

3.8 Additional Lemma for Isabelle ................................................. 103

3.9 Generated Proof Script for VCs with String Variables ............... 105

3.10 Lessons Learned .................................................................... 108

3.11 Essence of the Interesting VC ................................................. 109

3.12 General Theorem for Is_Conformal_w ................................. 111

3.13 Specific Theorem for Is_Conformal_w ..................................... 112

3.14 Envisioned Verified Software Development Infrastructure Revisited . 115

4.1 Signature of a First-Order Many-Sorted Free Logic ................ 120

4.2 Definition of a Free Logic Term ............................................. 121

4.3 Definition of a Free Logic Atomic Formula ............................ 122

4.4 Definition of a Free Logic Formula ....................................... 122

4.5 Definition of $|$ for Free Logic Atomic Formulas ................... 124

xvii
4.6 Definition of $|$ for General Free Logic Formulas . . . . . . . . . . . . 124
4.7 Syntax of a First-Order Many-Sorted Free Logic . . . . . . . . . . . . 125
4.8 Definition of $|$ for Total Logic Atomic Formulas . . . . . . . . . . . 127
4.9 Definition of $|$ for Total Logic General Formulas . . . . . . . . . . 127
4.10 Signature of the Many-Sorted Free Logic Under Translation . . . . 142
4.11 Example Formula in Free Logic . . . . . . . . . . . . . . . . . . . . . . . 145
4.12 Example Formula Given to Isabelle . . . . . . . . . . . . . . . . . . . 145
4.13 Example Formula Simplified . . . . . . . . . . . . . . . . . . . . . . . 146
4.14 Example Formula After Existential Split . . . . . . . . . . . . . . . . 147
4.15 Example Formula With No More obj Variables . . . . . . . . . . . . 147
4.16 Example Formula With obj_of_* Simplified . . . . . . . . . . . . . 148
4.17 Example Formula With Obvious Simplifications Applied . . . . . . 148
4.18 Example Formula With obj_of_real Removed . . . . . . . . . . . . . 149
4.19 Example Formula Without Type Information . . . . . . . . . . . . . 149
4.20 Example Formula in Isabelle With Object Existential Replaced . . . 150
4.21 Split Rule for Division . . . . . . . . . . . . . . . . . . . . . . . . . . 151
4.22 Example Formula After the Split . . . . . . . . . . . . . . . . . . . . 152
Chapter 1: Introduction

Given the ubiquity of software, individuals must have confidence that any software they use is correct. There are many methods and tools that can help software engineers increase their confidence that a software component is indeed correct, for instance code reviews [18]. These activities act as social processes similar to those in mathematics described by DeMillo et al [48]. However, in many instances there are unhandled “corner cases,” subtle flaws in reasoning, or outright wrong rationale that may be missed; any of these can introduce bugs into software. Moreover, when a bug is expressed, it undermines users’ confidence not only in the software, but also in the capabilities of software engineers. Fortunately, many types of tools can aid software engineers in finding and ultimately correcting bugs. These tools include compilers, static analyzers, and test harnesses—each can help achieve a higher level of confidence. However, these tools may not find all bugs in software. More specifically, for non-trivial software components test plans are—necessarily—incomplete, current compilers cannot detect most semantic bugs, and static checkers are unsound and/or incomplete. These tools all focus on small parts of an overarching goal: proving that a software component does exactly what it is supposed to, i.e., proving that the component implements its specification.
1.1 A Verifying Compiler

In contrast with the above tools, a verifying compiler would either prove that code is correct relative to a specification of desired behavior and produce verified object code, or reject that code and refuse to produce any object code. A verifying compiler is envisioned to employ a back-end prover (either integrated or external) that takes Verification Conditions (VCs), or mathematical formulas generated from the source code, and either proves or rejects each VC. These VCs would correspond to the correctness of the source code. Such a tool would offer the software engineering community the assurance of a full formal methods analysis of the software. Once a software component has been verified, it would be reusable without the possibility of executing incorrectly.

Creating a verifying compiler has been reissued as a grand challenge to computer scientists by Hoare [32] and expanded in the software engineering domain by Leavens et al [42]. Indeed after over forty years of research, the current lack of an acceptable solution demonstrates the theoretical difficulties in finding one. A known limitation from Gödel’s famous incompleteness theorem [15] is that any solution will be either unsound or incomplete if the programming language allows sufficiently strong arithmetic operations. However, this theoretical barrier has, in practice, no real impact on software verification; software engineers rarely rely on true, but not provable formulas—the real problems lie elsewhere. Current programming languages, such as C [34], Java [24], and others were not developed to support verification and as such require extension and revision to create the necessary machinery for verification. Research into methods to deal with such issues has had some results [19, 26] although the full goal is still elusive. Finally, it is evident that the current tools for performing
the proofs of the VCs are inadequate; it is still easy to create typical software which is not provable using these tools.

1.2 Vision of Verified Software Development

The long-term vision guiding this research is of a future in which no production software is considered properly engineered unless it has been fully specified and automatically verified as satisfying these specifications. Even when this vision is achieved, it will not imply that software is absolutely correct; formal specifications might not capture informal requirements. However, a significant gap will be bridged as questions about a software product’s correctness will not involve testing. Instead they will be settled by examining only the specifications. Questions about the correctness of code relative to the specifications will be effectively moot, that code having been subjected to far more rigorous reasoning than is practically possible today.

Figure 1.1: Envisioned Verified Software Development Infrastructure

While this is a grand goal, there are many challenges to achieving it. Even though practical software’s correctness will not hinge on answers that are dependent on the
incompleteness of mathematics, there still remain other issues that are serious ob-
stances to automated “push-button” verification for practical software. One reason
correct code is not provably correct is because of weaknesses in specification engineer-
ing and mathematical development. As a practical matter, reasoning soundly about
the behavior of software—even software designed with the utmost care and rigor—
will always remain a huge intellectual task for humans alone; it will be successful only
when human reasoning is carefully augmented by automated reasoning. Conversely,
creating verified software will remain an impossible task for computers alone.

Given these constraints, the precise demands on humans under a verified soft-
ware paradigm, and the new knowledge, skills, and tools that will best help them
meet those demands, have yet to be clearly elucidated by the verified software com-

munity. Our vision of the high-level structure for a verified software paradigm is
presented in Figure 1.1. Software specialists will be responsible for designing and de-
veloping behavioral specifications as well as implementations with annotations that
justify code correctness relative to those specifications. Mathematical logic specialists
will be responsible for developing theory for use in the specifications. Collaborating
with software specialists, they will define mathematical notions to help formalize be-
havioral specifications, and prove highly reusable mathematical results that support
correctness justifications. With such mathematical developments in place, verification
of correct software itself is intended to be a fully automatic “push-button” activity.
Thus, the structure will use two completely different skill-sets to achieve the goal of
automated verification.
1.2.1 The Programming Language Choice

Given any verified software development infrastructure, the programming language choice affects the provability of the VCs. The model of computation, the complexity of representing built-in programming language structures, and the level of support for annotations within the programming language all contribute to the complexity of VCs. In the first category there are many different models of computation, from sequential to concurrent, and imperative to declarative. Since within any block of concurrent code, there are multiple sequential blocks, we focus on the problem of verifying sequential programs. We also focus on the most common kind of programming language: imperative. Within imperative programs, annotations are necessary to denote loop invariants, and progress metrics for loops and recursive procedures are necessary in order to prove correctness. In addition, programming languages not designed for verifiability and reusability may need to have programming language implementation details such as memory addresses on the heap represented in the VCs in order to ensure soundness and relative completeness.

In this dissertation, RESOLVE [52, 14, 10, 33] is the programming language used. This choice has profound implications on any VCs generated in that the language was designed to simplify the latter two aspects of VCs. RESOLVE is component-based and has value semantics (“clean semantics”), absence of aliasing, support for modular reasoning, and was designed with the goal of both automatic verification and usability. This choice of a programming language removes much of the accidental complexity of the problem and focuses on the essential issues of program verification. Value semantics are achieved efficiently by decoupling the notion of data movement from the notion of an actual copy. The data movement operation chosen is the \texttt{swap}
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

Figure 1.2: Set Template Facility Specification
operator, which is superior to the traditional assignment operator [57]; swap allows for value semantics while allowing for efficient implementation using references “under the covers.” The proof rules needed for the generation of the VCs along with the mathematical theories necessary for the specification of the components and language semantics have all been developed by Krone [40], Heym [31], and Ogden [52]. These rules have all been proved sound and relatively complete. Moreover, requirements for the implementation of RESOLVE have been discussed by Harms [29] and prototype verifying compilers exist for two of its dialects [65].

In the RESOLVE language, the behavior of each software component is described by a contract, providing a model of the component’s behavior. There can be multiple implementations of each contract. One example of a contract written in this language is given in Figure 1.2. Within this contract, we see that a variable of type Set has a value modeled by a mathematical finite set. The contracts for operations that can be performed on a Set are specified in terms of their effects on the mathematical set model; VCs are in terms of this mathematical model. One more feature of the RESOLVE language is that every variable always has a value of its mathematical model, i.e., there is no “undefined” value for any variable. The initialization ensures assertion characterizes the initial value of a Set; any value that satisfies the predicate is a possible initial value of a variable of that type.

Another key feature in this contract is the parameter modes given to each parameter. Each of these modes carries a behavioral meaning. The first, and most restrictive, is restores mode. It means that the outgoing value of the parameter equals the incoming value (even though within the implementation, the value may change temporarily, as long as it changes back before the operation ends). The next
is **clears**; it means that the outgoing value is reset to an initial value of its type. The **replaces** parameter mode indicates that the incoming value of that parameter has no effect on the outgoing values of any parameters, *e.g.*, the **RemoveAny** procedure in Figure 1.2. Finally, the **updates** parameter mode simply indicates that its outgoing value is specified by the **ensures** clause. A function returns a value and may return values from already declared user-defined data types or a **control** type that is used to indicate control flow in computations. Moreover, to simplify verifiability, each program function must behave as a mathematical function, *i.e.*, it must **restore** its arguments. This is superficially similar to the restriction that functions be “pure” in JML [43] or Dafny [44]. The difference is that RESOLVE program functions still may not be used in specifications because RESOLVE rigorously separates and distinguishes mathematical entities (including definitions of mathematical functions) from programming entities (including program operations, *i.e.*, program functions, that happen to have functional behavior).

If we examine the contract for the **Add** operation, we see the use of pre- and post-conditions—expressed in terms of **requires** and **ensures** clauses. The **requires** clause here indicates that, in order to add an item to a **Set**, one must first be sure that the item is not in the **Set** already; variables used in the **requires** and **ensures** clauses represent the values of the variables as defined by their mathematical model. The **ensures** clause indicates that the item has been added to the set. The # prefix on a variable name in the **ensures** clause denotes the value of that variable prior to the operation execution.

These component contracts may used in two ways. The first is to add additional functionality to existing components in the form of extensions, *i.e.*, functionality
**contract** IntersectExtension **enhances** SetTemplate

**procedure** Intersect (updates s: Set, restores t: Set)

ensures

\[ s = \#s \text{ intersection } t \]

end IntersectExtension

**realization** Iterative implements IntersectExtension for SetTemplate

**procedure** Intersect (updates s: Set, restores t: Set)

variable tmp: Set

loop

  maintains \( (s \cup \text{tmp}) \text{ intersection } t =\)
  \( (\#s \cup \#\text{tmp}) \text{ intersection } t \) and
  \( s \text{ intersection } \text{tmp} = \text{empty_set} \) and
  \( \text{tmp intersection } t = \text{tmp} \) and
  \( t = \#t \)

  decreases \( |s| \)

  while not IsEmpty (s) do
    variable x: Item
    RemoveAny (s, x)
    if Contains (t, x) then
      Add (tmp, x)
    end if
  end loop

end loop

s :=: tmp

end Intersect

end Iterative

Figure 1.3: Set Template Intersection Extension and Implementation
that is implementable on top of the extended component’s operations. The other is to describe a new type that may be implemented in many ways. Figure 1.3 gives an example of a simple extension and implementation of that extension. This extension includes a procedure to compute the intersection of two sets. The implementation includes a loop invariant or the maintains clause along with a loop progress metric or decreases clause to demonstrate loop termination. Operations are called procedurally, rather than with the usual object-oriented dot notation with an implicit this parameter.

The RESOLVE project has separated the use of mathematical definitions and results for the VC proofs from the machine-checked proofs of those results within each of the theories. Briefly, the theories include results about mathematical functions defined in the theory to be used for the proofs of the VCs; these theorems can be difficult to prove, even for experienced mathematicians. However, any proofs of the VCs never need to include proofs of these reusable theorems; they only need to be able to use them correctly to prove VCs, which are generally quite ad hoc and simple theorems. This simplifies the VC proofs; the tool used to prove the VCs need only be able to make sufficient progress to prove most VCs from relatively simple applications of the available theorems.

1.2.2 Mathematical Theories and Back-end Prover Choice

The choice of tool used to prove the VCs can affect provability. In general there are three broad categories of tools. The first is the truly automated theorem prover. This type of tool is designed to prove theorems, generally in first order logic, and is typically expected to be run for a long time to establish the theorem. The second type
of tool is a satisfiability modulo theory tool (SMT). These tools have been designed for use in the proofs of VCs obtained from the process of verifying software. Such a tool typically includes a propositional satisfiability solver, several decision or semi-decision procedures, and heuristics for eliminating quantifiers. These tools essentially attempt to find satisfying assignments as if the first order formulas were actually propositional formulas and use the decision procedures to confirm or refute the potential satisfying assignments found. Finally, there are interactive proof assistants that are designed to prove possibly deep theorems guided by a mathematician, prove properties about the semantics of programming language semantics, or explore other hard problems. These proof assistants typically provide methods to prove some theorems automatically along with commands to guide the proof and include a rich underlying axiom system, logic, and input language.

The richness of the theory development (the number and strength of the theorems defined within a theory) also has an impact on the provability of VCs. The fewer automated reasoning steps that a tool has to perform before a proof can be found, the quicker and more reliably VCs will be proved. This is also seen quite naturally in even the most introductory mathematical material; proofs are simplified by proving lemmas or corollaries that are used within the proof.

1.2.3 Impact of Formal Logic on Verification

Automated or interactive formal verification requires a fully developed logical framework—a formal logic in which formulas have a specific meaning. The definition of both the syntax and semantics of a formal logic allows for a construction of a proof system for the logic that proves formulas valid [15]. Unfortunately, there is
not just one logic [3, 15]; many formal systems exist, each representing a trade-off of the expressiveness of the logic with the best-case soundness and completeness of any proof systems associated with the logic. Moreover, in the automated verification community programming languages are commonly first translated to an intermediate representation for VC generation [44, 8, 4]. Each of these translations brings the opportunity for error and subtle incompatibilities. Little attention has been paid in the literature to the translation issues; translations such as these are assumed to be correct.

1.3 Thesis Statement

Based on this brief discussion of the issues involved in the process of verification contained in the grand challenge of a verifying compiler, this dissertation defends the following three-part thesis:

1. Modest changes in programming languages may effectively support and facilitate automated verification without introducing an undue annotation burden on programmers.

2. An interactive proof assistant can be an effective back-end prover for automated software verification.

3. Translations between similar formal systems present significant challenges in fully automating software verification.

Reexamining the vision of software development shown in Figure 1.1, we depict the areas of this process that each part of the thesis addresses in Figure 1.4.
Each of these thesis parts contributes to the state of the art. The first part of the thesis both gives an example of such a construct, usable in aiding verification, and also serves as an example for how other, similar constructs can be found. The second part of the thesis aids in the eventual selection of the best back-end prover for VCs while also examining the roles of the mathematician in the envisioned verified software development infrastructure. Finally, the last part of the thesis serves as guidance for the potential problems in translating between related logics, and provides an example of the potential translation issues.

1.4 Dissertation Outline

The remainder of the dissertation is organized as follows: Chapter 2 describes how Part 1 of the thesis is supported by introducing a new programming language construct that simplifies verification. Chapter 3 describes how Part 2 of the thesis is supported by describing how an interactive proof assistant is used as a back-end prover in RESOLVE tools. Chapter 4 discusses how Part 3 of the thesis is supported by examining a specific instance of the translation problem.
Chapter 2: Programming Language Support for Automated Verification

Programming language support for automated verification has its roots in language annotations that allow for the possibility of fully automated verification, e.g., loop invariants. With the construction of sound and relatively complete proof systems for such programming languages, one might infer that there is little need for new programming language constructs of this kind; sound and relatively complete proof systems are quite simply the best that one can achieve. Recent experience with automating the proofs of program correctness demonstrates that even something as simple as removing irrelevant facts can impact effective provability [71, 35]. Thus, while the term “sound” has very practical implications the term “relatively complete” does not give enough information about the qualities of VCs generated by a programming language proof rules. Therefore, it is possible that there are new language constructs that, while they add no theoretical power to a sound and relatively complete proof system and programming language, in practice simplify VCs. In this chapter we present a construct that satisfies this property by both effectively documenting program invariants and exploiting their properties to simplify VCs.
2.1 Introduction

It has long been claimed in some circles that software professionals cannot be expected to write mathematically rigorous descriptions of their code such as formal specifications and loop invariants [20]. This contention arguably underestimates the capabilities of software professionals—after all, most of them have not been taught either why or how to write such annotations, so it is not surprising they are currently unequipped to do so. Nonetheless, the perception has led to exploration of some promising mitigating techniques that might be useful under a verified software paradigm. One approach involves inferring invariants (e.g., loop invariants) either by dynamic or static analysis of code [61, 56, 27, 25, 67]. A complementary approach involves minimizing what needs to be written in mathematical language by providing special syntax for certain situations: syntax that looks more familiar and code-like to software developers. For instance, rather than demanding that the post-condition of an operation include a clause like \( x' = x \), \( x = \text{old}(x) \) or \( x = \#x \) to specify that the value of \( x \) does not change, most specification languages have tailor-made syntax for documenting this. JML [43] uses a modifies clause to list operation parameters whose values the parameters refer to might be changed during the operation body. RESOLVE [52] offers (among others) a restores parameter “mode” to state that an operation parameter, while it might change temporarily during the body of the operation, has the same value at the end of the operation body as it had at the beginning.

Such mechanisms incrementally reduce the mathematical annotation burden for the software professional. It is not yet clear how effective the invariant-inference approach will be under a verified software paradigm for component-based software;
when automated verification does not succeed, it will be critical for a human to understand these invariants in order to repair the code, the annotation or both. This means that inferred invariants should be not only technically correct but also comprehensible to the software professional, who will ultimately be responsible for at least reading and likely for modifying formal mathematical descriptions of software behavior. Some human input into writing invariants and other assertions therefore seems unavoidable.

We describe a language construct to reduce the annotation burden. It focuses on relationships between invariant properties of variables that hold during an entire code segment and loop invariants within that segment. Observe that two kinds of properties must be included in a loop invariant to verify software. The first kind arise from the desire to treat a loop as a single statement in straight-line code for verification and reasoning purposes. Properties of this kind document the behavior of the loop by stating what it does not change; they are intimately tied to the loop and are local to it. The second kind arise from the need to maintain continuity of abstract invariants on variable values. These properties are often incidental to a particular loop yet are critical pieces of the loop invariant. For example, when using memoization to avoid re-computation of a function with a Java \texttt{Map}, one abstract invariant on the \texttt{Map}'s value is that if a key is contained in the \texttt{Map} then the value associated with that key is the function applied to the key. This information must be in the loop invariant for any loop involving the \texttt{Map}, because this property is true before the loop is encountered, is maintained by the loop, and might be intended to persist after the loop has terminated. This restricted set of \texttt{Map} values is known \textit{a priori} by the software developer independently of any loops, and it can and should
be documented. If the documentation is formal, its connection to the code can be verified. In other words, this documentation not only records the software developer’s reasoning but—in a verified software paradigm—also can be used to check that the reasoning is correct.

This chapter presents a programming language construct, restrictions, that can be used to document *abstract invariant properties of individual variables over segments of imperative code* without introducing new abstract data types. This construct allows for reuse of these invariants. Moreover, it implicitly provides guidance to the verifier by factoring potentially complicated verification conditions (VCs) into conceptually simpler VCs. The overabundance of assumptions in VCs has been reported [71, 35] as a problem for back-end provers, and the new construct tends to limit assumptions to only those that are useful in VC proofs. This chapter also presents the formal proof rules along with the semantics of restrictions.

Restrictions are presented in the context of RESOLVE; however, the restriction construct should be adaptable to other programming languages with little change, so long as the specification language can express and ensure frame properties, such as JML [43] or Dafny [44].

The chapter is structured as follows. Section 2.2 presents a simple motivating example (in C++ rather than RESOLVE). Section 2.3 includes a summary of the features and syntax of RESOLVE needed to explain restrictions. An introduction to restrictions in Section 2.4 features an in-depth example using sorting. The formal syntax, semantics and proof rules are presented in Section 2.5 along with an evaluation of the effectiveness of the construct. Related work is discussed in Section 2.6, with future work presented in Section 2.7 followed by conclusions in Section 2.8.
2.2 Motivating Example

Consider code that computes $x^p$ where $x$ is a double and $p$ is a positive integer; see Figure 2.1(a). It computes $x^p$ by first computing $x^{2^k}$ where $k$ is the largest natural number that satisfies $2^k \leq p$ and then making a recursive call to finish the job. In this particular implementation, $q$ always equals $2^{k'}$ where $k'$ is some non-negative integer, and this property holds both as a loop invariant and, more generally, as an invariant on $q$ throughout the code. We argue that this invariant can and should be documented.

```
double Power (double x, int p)
{
    double result = x;
    int q = 1;
    while(q <= p/2)
    /*! updates result, q maintains
       result = x^q and
       q <= p and
       there exists k : integer
       (q = 2^k)
      decreases
      p - q */
    { q *= 2;  // restrict q to be a power of 2 !*/
      result *= result;
    }
    if (p - q > 0)
    { result *= Power(x, p-q); }
    return result;
}
```

(a) Original version

```
double Power (double x, int p)
{
    double result = x;
    int q = 1;
    while(q <= p/2)
    /*! restrict q to be a power of 2 !*/
    { q *= 2;
      result *= result;
    }
    if (p - q > 0)
    { result *= Power(x, p-q); }
    return result;
}
```

(b) Documented with a restriction

![Figure 2.1: Code to Compute $x^p$](image)

One method a software professional can use to document the invariant on $q$ is to add extra assertions in the code. At every line where the invariant holds, she asserts
the invariant. Frame properties allow one to limit the number of such statements needed, by using them only after a modification to a variable under consideration. This documents the invariant on \( q \), but it is rather clumsy and the annotation burden is high. Restrictions (Section 2.4) are a construct to document the claims for this code more clearly and to reduce the annotation burden. Figure 2.1(b) shows what the code might look like in this situation. The loop invariant is simplified and the invariant on \( q \) is explicit.

### 2.3 Illustrating Example

As discussed in Chapter 1, a RESOLVE component defining a new abstract data type specifies a mathematical model of its behavior in a contract, as illustrated in Figure 2.2 by a QueueTemplate contract. The mathematical model is explicit in the type declaration of Queue. Each operation has a formal description of behavior in terms of the mathematical model via standard \texttt{requires} and \texttt{ensures} clauses, as stated in Section 1.2.1. The \texttt{control} return type is used within \texttt{if/while} conditions; it is not a type technically, because one may not declare a \texttt{control} variable.

Behavioral extensions to abstract components, such as the QueueConcatenateExtension extension in Figure 2.2, are specified via contracts and ordinarily are implemented by layering on other components’ contracts. Another extension of a QueueTemplate contract is QueueSortExtension that adds the \texttt{Sort} operation. Sorting has been studied by the computer science community since the field’s inception; in the past few years there has been significant work on inferring loop invariants [61, 56, 27, 25, 67] among other work on verification of sorting algorithms. For the purposes of demonstrating the utility of restrictions,
contract QueueTemplate (type Item)
uses UnboundedIntegerFacility
math subtype QUEUE_MODEL is string of Item
type Queue is modeled by QUEUE_MODEL
exemplar q
initialization ensures
q = empty_string

procedure Enqueue (updates q: Queue, clears x: Item)
ensures
q = #q * <#x>

procedure Dequeue (updates q: Queue, replaces x: Item)
requires
q /= empty_string
ensures
#q = <x> * q

function IsEmpty (restores q: Queue): control
ensures
IsEmpty = (q = empty_string)
end QueueTemplate

contract QueueConcatenateExtension enhances QueueTemplate

procedure Concatenate (updates p: Queue, clears q: Queue)
ensures
p = #p * #q
end QueueConcatenateExtension

Figure 2.2: QueueTemplate Contract and QueueConcatenateExtension contract

sorting therefore serves as an appropriate standard benchmark that naturally involves variables with abstract invariants beyond any of the generic abstract data type (ADT) invariants of its variables.

2.3.1 Sort Specification

Since the QueueTemplate component is generic, i.e., parameterized by a type Item, the contract of a Sort operation should also be generic. Figure 2.3 shows
the requirement on the ordering relation \texttt{ARE\_IN\_ORDER} used in sorting, namely that \texttt{ARE\_IN\_ORDER} is a total pre-order.

\begin{verbatim}
mathematics StringExtras ( 
    type Item, 
    definition ARE_IN_ORDER(x:Item, y : Item) : boolean 
        satisfies 
        for all x, y : Item 
            (ARE_IN_ORDER(x, y) or ARE_IN_ORDER(y, x)) and 
        for all x, y, z : Item 
            where (ARE_IN_ORDER(x, y) and ARE_IN_ORDER(y, z)) 
            (ARE_IN_ORDER(x, z)) 
)

definition OCCURS_COUNT ( s : string of Item, i : Item ) : integer 
    satisfies 
    if s = empty_string 
        then OCCURS_COUNT ( s, i ) = 0 
    else there exists x : Item , 
        r : string of Item 
        ((s = <x> * r) and 
        (if x = i then OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i) + 1 
        else OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i)))

definition IS_PERMUTATION ( s1: string of Item, s2: string of Item ) : boolean 
    is for all i : Item (OCCURS_COUNT (s1, i) = OCCURS_COUNT (s2, i))

definition IS_PRECEDING ( s1: string of Item, s2: string of Item ) : boolean 
    is for all i, j : Item 
    where (OCCURS_COUNT (s1, i) > 0 and OCCURS_COUNT (s2, j) > 0) 
    (ARE_IN_ORDER (i, j))

definition IS_NON_DECREASING ( s: string of Item ) : boolean 
    is for all a, b: string of Item 
    where (s = a * b) (IS_PRECEDING (a, b))
end StringExtras
\end{verbatim}

Figure 2.3: Math Unit with Mathematical Definitions Used in the Sort Contract

Figure 2.3 also shows the mathematical definitions used to specify sorting, given \texttt{ARE\_IN\_ORDER}. \texttt{OCCURS\_COUNT} is the number of times a given \texttt{Item} appears in a string; it is used to construct the other mathematical definitions. This allows the contract to be specific about the value of the outgoing \texttt{Queue}: not only are the values of items in the outgoing \texttt{Queue} the same as in the incoming \texttt{Queue}, but the number of times each
appears is the same. **IS_PRECEDING** is a binary predicate that holds on two strings if and only if every item in the first string is related by **ARE_IN_ORDER** to every item in the second string; intuitively, every item in the first string is “no larger” than every item in the second. **IS_NON_DECREASING** is a unary predicate that is true if and only if every consecutive pair of Items in the string are related by **ARE_IN_ORDER**. Finally, **IS_PERMUTATION** is a binary predicate on strings that is true if and only if the number of occurrences of every Item is the same in the first string and in the second.

---

```
contract QueueSortExtension uses StringExtras (  
  definition ARE_IN_ORDER (x : Item, y : Item): boolean  
  satisfies  
    for all x, y:Item  
      (ARE_IN_ORDER(x, y) or ARE_IN_ORDER(y, x)) and  
    for all x, y, z:Item  
      where (ARE_IN_ORDER(x, y) and ARE_IN_ORDER(y, z))  
      (ARE_IN_ORDER(x, z))  
enhances QueueTemplate  
  uses StringExtras (Item, ARE_IN_ORDER)  
  procedure Sort (updates q: Queue)  
    ensures  
      IS_PERMUTATION (q, #q) and  
      IS_NON_DECREASING(q)  
end QueueSortExtension
```

Figure 2.4: QueueSortExtension Extension to QueueTemplate

The contract specification of a **Sort** operation is given in Figure 2.4. The **Sort** operation takes a **Queue** and returns with the property that the outgoing string q is a permutation of the incoming string and the outgoing string is non-decreasing with respect to **ARE_IN_ORDER**.
realization QuickSort(
  function AreInOrder (restores i: Item, restores j: Item): control
  ensures
    AreInOrder = ARE_IN_ORDER (i, j)
) implements QueueSortExtension for QueueTemplate

uses QueueConcatenateExtension for QueueTemplate(Item)

local procedure Partition (updates qSmall: Queue, replaces qBig: Queue, restores p: Item)
  ensures
    IS_PERMUTATION (qSmall * qBig, #qSmall)
    and IS_PRECEDING(qSmall, <p>)
    and IS_PRECEDING(<p>, qBig)
  variable tmp: Queue
  Clear (qBig)
  loop
    updates qSmall, qBig, tmp
    maintains
      IS_PERMUTATION (qSmall * qBig * tmp, #qSmall * #qBig * #tmp)
      and IS_PRECEDING(tmp, <p>)
      and IS_PRECEDING(<p>, qBig)
    decreases |qSmall|
    while not IsEmpty (qSmall) do
      variable x: Item
      Dequeue (qSmall, x)
      if AreInOrder (x, p) then
        Enqueue (tmp, x)
      else
        Enqueue (qBig, x)
      end if
    end loop
  qSmall :=: tmp
end Partition

procedure Sort (updates q: Queue)
  decreases |q|
  if not IsEmpty (q) then
    variable partitionElement: Item
    variable qBig: Queue
    Dequeue (q, partitionElement)
    Partition (q, qBig, partitionElement)
    Sort(q)
    Sort(qBig)
    Enqueue (q, partitionElement)
    Concatenate (q, qBig)
  end if
end Sort
end QuickSort

Figure 2.5: Quicksort Implementation of Sort Extension to QueueTemplate
2.3.2 Quicksort Implementation

We present an implementation of the Sort operation using quicksort in Figure 2.5. Our implementation partitions a non-empty incoming queue $q$ into two queues ($q$ and $q_{\text{Big}}$) and a partitioning element ($\text{partitionElement}$) with the property that every Item in $q$ is in order with $\text{partitionElement}$ and $\text{partitionElement}$ is in order with every item in $q_{\text{Big}}$. Each of the resulting queues is sorted recursively and $q$, $\text{partitionElement}$, and $q_{\text{Big}}$ are all concatenated to obtain the final, sorted queue. Besides the loop invariant and other mathematical annotations, this code is similar to code in most other languages. A local operation $\text{Partition}$ is used to split a queue according to the quicksort algorithm. The $:=$: operator is the “swap” operator [28, 57]. This exchanges the values of its two arguments, and is a key aspect of avoiding aliasing while preserving efficiency. The loop invariant documents the insight of the algorithm, namely the ordering relationships among the variables $\text{partitionElement}$, $q$ and $q_{\text{Big}}$, as expressed formally via IS_PRECEDING and IS_PERMUTATION. The programmer’s justification for termination is given by the decreases clause (i.e., progress metric).

Figure 2.6 show what a component dependency graph looks like for this implementation. The QueueSortExtension contract depends on the reusable mathematical unit StringExtras by using the mathematical unit (denoted by the ‘u’ edge label), while the specifications of the QueueSortExtension and QueueConcatenateExtension contracts enhance the QueueTemplate contract (denoted by the ‘e’ label). Finally QuickSort implements the QueueSortExtension contract (denoted by the ‘i’ label) and uses the contract of QueueConcatenateExtension. We update this diagram later to show how restrictions fit within these dependencies.
2.4 Introduction to the Syntax and Semantics of Restrictions: Sorting Example

First we examine the issues involved in defining restrictions, and then the issues in the usage of restrictions in client code. Code presented in this section is analogous to the code in Section 2.3.1 except that it uses restrictions.

2.4.1 Restrictions

We create three restrictions for this example, one for each different abstract invariant maintained by specific uses of Queues in quicksort. The first invariant is that a Queue is sorted, i.e., it is an OrderedQueue. The other invariants relate the Items in a Queue to another Item. These invariants arise during the Partition implementation and simply relate qSmall to p and qBig to p by ARE_IN_ORDER; more specifically every Item in qSmall is in order with p and p is in order with every Item in qBig. For each operation that is called on any Queue that satisfies one of these properties,
the programmer reasons that the abstract invariant is not broken by the operation call. The proof boils down to the question: if the operation is executed, does the new value of the variable still satisfy the restriction? With this intuition in mind, we show the contracts for the restrictions corresponding to these ideas in Figures 2.7, 2.8 and 2.9.

```
contract OrderedQueueRestriction {
    definition ARE_IN_ORDER (x: Item, y: Item): boolean satisfies
    for all x, y: Item
        (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
    for all x, y, z: Item
        where (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z))
            (ARE_IN_ORDER (x, z))
) restricts QueueTemplate

uses StringExtras (Item, ARE_IN_ORDER)

restriction OrderedQueue(q: Queue) is (IS_NON_DECREASING(q))

procedure Enqueue(updates q: Queue, clears x: Item)
    under restriction
    OrderedQueue(q)
    also requires
    IS_PRECEDING (q,<x>)

procedure Dequeue(updates q: Queue, replaces x: Item)
    under restriction
    OrderedQueue(q)
    also ensures
    IS_PRECEDING (<x>, q)
end OrderedQueueRestriction
```

Figure 2.7: OrderedQueue Restriction

A restriction is declared relative to one or more existing contracts, for example QueueTemplate. Operations of the underlying contract may be given additional requires and ensures clauses. The restriction is given by a predicate where parameters are of the specified types. Since functions may not “break” the invariant—they
cannot change the abstract value of any argument—they are always available to be used with a program type in any restriction.

```plaintext
contract SmallValueQueueRestriction {
    definition ARE_IN_ORDER (x: Item, y: Item): boolean
        satisfies
            for all x, y: Item
                (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
            for all x, y, z: Item
                where (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z))
                    (ARE_IN_ORDER (x, z))
    ) restricts QueueTemplate

    uses StringExtras (Item, ARE_IN_ORDER)

    restriction SmallValueQueue(q: Queue, max : Item) is (IS_PRECEDING(q, <max>))

    procedure Enqueue(updates q : Queue, clears x : Item)
        under restriction
            SmallValueQueue(q, max)
        also requires
            ARE_IN_ORDER(x, max)

    procedure Dequeue(updates q : Queue, replaces x : Item)
        under restriction
            SmallValueQueue(q, max)
        also ensures
            ARE_IN_ORDER(x, max)
}
end SmallValueQueueRestriction
```

Figure 2.8: SmallValueQueue Restriction

Conceptually, the also requires clauses are conjoined with the original requires clauses for the operation. These are used by the programmer to ensure both that the restriction is maintained by the operation, and to document conditions under which it is safe to call the operation while still maintaining the invariant. The also ensures clauses strengthen the original postconditions. In the OrderedQueueRestriction contract, Dequeue’s also ensures clause gives information about how the dequeued item relates to items that remain in the OrderedQueue.
contract LargeValueQueueRestriction {

definition ARE_IN_ORDER (x : Item, y : Item) : boolean satisfies
  for all x, y : Item
    (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
  for all x, y, z : Item
    where (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z))
    (ARE_IN_ORDER (x, z))
) restricts QueueTemplate

uses StringExtras (Item, ARE_IN_ORDER)

restriction LargeValueQueue(q : Queue, min : Item) is (IS_PRECEDING(<min>, q))

procedure Enqueue(updates q : Queue, clears x : Item)
  under restriction
    LargeValueQueue(q, min)
  also requires
    ARE_IN_ORDER(min, x)
procedure Dequeue(updates q : Queue, replaces x : Item)
  under restriction
    LargeValueQueue(q, min)
  also ensures
    ARE_IN_ORDER(min, x)
end LargeValueQueueRestriction

Figure 2.9: LargeValueQueue Restriction

Since programmers may need some help in making sure that their reasoning pro-
cess is correct, the compiler should generate VCs corresponding to the correctness of
the restriction contract. The contract’s correctness condition is that if an operation
is invoked in a state satisfying the variable restrictions and the requires clause, and
the operation completes successfully, then the restriction is still satisfied by the up-
dated variables; any also ensures clauses must also be satisfied. More concretely,
each operation’s invocation can be assumed to occur in a state in which the restric-
tion, the original requires clause, and the also requires clause hold. By a process
similar to datatype induction, these VCs are generated just once for the contract.
We examine the formal reasons for correctness in Section 2.5.3. (Notice that this
construction leaves the initialization of restrictions to a client-side activity and is discussed in Section 2.4.2.) The general form of the generated VCs, where $s_0$ and $s_1$ are variables of the mathematical model of the type of restriction and $\text{args}$ is the list of arguments to the operation is given by:

\[
\begin{align*}
\text{restriction}[s_0] \land \text{requires}_{\text{original}}[s_0, \text{args}_0] \land \\
\text{requires}_{\text{also}}[s_0, \text{args}_0] \land \\
\text{ensures}_{\text{original}}[s_0, \text{args}_0, s_1, \text{args}_1] \land \\
\text{restriction}[s_1] \land \text{ensures}_{\text{also}}[s_0, \text{args}_0, s_1, \text{args}_1]
\end{align*}
\]

2.4.2 Client Usage of Restrictions

The updated Sort contract, shown in Figure 2.10, is almost the same as the original contract. The difference is that the Queue formal parameter $q$ is constrained to satisfy the restriction OrderedQueue when the operation returns. The formal parameter $q$ must be of type Queue; when Sort returns, $q$ conforms to the restriction OrderedQueue (checked as a proof obligation). We can omit the \texttt{IS\_NON\_DECREASING}(q) from the \texttt{ensures} clause now, since it is subsumed by the restriction.

This reduces the mathematical annotation burden on the programmer. The \texttt{establishes} restriction annotation in the formal parameters need not be checked by the static type system. It is equivalent to having the restriction in the \texttt{ensures} clause for that variable. We examine this issue in more depth in the discussion of the Sort operation from Figure 2.12. There is also the dual of the \texttt{establishes} restriction, \texttt{consumes} restriction that is not used in this example, which is equivalent to having the restriction predicate in the \texttt{requires} clause.
contract QueueSortExtension {
    definition ARE_IN_ORDER (x: Item, y: Item): boolean satisfies
    for all x, y: Item (ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
    for all x, y, z: Item where (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z))
      (ARE_IN_ORDER (x, z))
} enhances QueueTemplate

uses StringExtras (Item, ARE_IN_ORDER)
uses OrderedQueueRestriction (ARE_IN_ORDER) for QueueTemplate(Item)

procedure Sort (updates q: Queue)
  establishes restriction
    OrderedQueue(q)
  ensures
    IS_PERMUTATION (q, #q)
end QueueSortExtension

Figure 2.10: QueueSortExtension Extension to QueueTemplate Using Restrictions

Figure 2.11 shows an additional operation Concatenate defined on Queues that is used by OrderedQueues in the OrderedQueueConcatenateExtension contract. The under restriction slot is used in this contract to indicate that two variables, q1 and q2, satisfy the OrderedQueue restriction as both a pre- and post-condition of the operation.

contract OrderedQueueConcatenateExtension
  enhances QueueConcatenateExtension
  restricts OrderedQueueRestriction for QueueTemplate

procedure Concatenate (updates q1: Queue, clears q2: Queue)
  under restriction
    OrderedQueue(q1) and OrderedQueue(q2)
  also requires
    IS_PRECEDING(q1, q2)
end OrderedQueueConcatenateExtension

Figure 2.11: OrderedQueueConcatenateExtension Restriction
The `Partition` operation uses the `SmallValueQueue` and `LargeValueQueue` restrictions. The code for performing the partition operation is given in Figure 2.12. In the contract of `Partition`, the `ensures` clause and loop invariant are simplified by the use of the `establishes restriction` annotation. Otherwise, the code is similar to the original version in Section 2.3.1.

Recall that in the contracts of restrictions, there were no VCs generated for initialization; that piece is left to the clients or users of the restriction. So, when a variable of a particular type, say `Queue`, has to satisfy a restriction, say a `LargeValueQueue`, a VC is generated to make sure that variable satisfies that restriction. For example, `confirm restriction LargeValueQueue(qBig, p)` generates a VC whose goal is `IS_PRECEDING(<p>, qBig)` and whose assumptions are those facts known at that point in the code, e.g., resulting from path conditions, loop invariants, and contracts of other operations called. We note that a result of the use of restrictions is that the specification of `Partition` is simpler, along with a simpler loop invariant as well. The `confirm restriction SmallValueQueue(qSmall, p)` is what allows us to perform the swap of `qSmall` and `tmp`.

Figure 2.12 also shows a `QueueSortExtension` implementation using restrictions and the modified `Partition` local operation. Except for the `confirm restriction` annotation, the code is exactly the same as the original version. The loop invariant is simplified as two conjuncts removed are implicitly implied by the restrictions.

Figure 2.13 is an update of Figure 2.6 with new nodes colored gold. In this figure, we see that the restrictions `LargeValueQueueRestriction`,
realization QuickSort ( 
  function AreInOrder (restores i: Item, restores j: Item): control 
    ensures
      AreInOrder = ARE_IN_ORDER (i, j) 
  ) implements QueueSortExtension for QueueTemplate 

  uses SmallValueQueueRestriction (ARE_IN_ORDER) for QueueTemplate (Item) 
  uses LargeValueQueueRestriction (ARE_IN_ORDER) for QueueTemplate (Item) 
  uses OrderedQueueConcatenateExtension 
    for OrderedQueueRestriction (ARE_IN_ORDER) for QueueTemplate (Item) 

local procedure Partition (updates qSmall: Queue, replaces qBig: Queue, restores p: Item) 
  ensures
    IS_PERMUTATION (qSmall * qBig, #qSmall) 
  establishes restriction
    LargeValueQueue(qBig, p) and SmallValueQueue(qSmall, p) 

  variable tmp: Queue 
  confirm restriction SmallValueQueue(tmp, p) 
  Clear (qBig) 
  confirm restriction LargeValueQueue(qBig, p) 
  loop 
    updates qSmall, qBig, tmp 
    maintains
      IS_PERMUTATION (qSmall * qBig * tmp, #qSmall * #qBig * #tmp) 
    decreases |qSmall| 
    while not IsEmpty (qSmall) do 
      variable x: Item 
      Dequeue (qSmall, x) 
      if AreInOrder (x, p) then 
        Enqueue (tmp, x) 
      else 
        Enqueue (qBig, x) 
      end if 
    end loop 
    confirm restriction SmallValueQueue(qSmall, p) 
    qSmall :=: tmp 
  end Partition 

procedure Sort (updates q: Queue) 
  decreases |q| 

  variable qtmp: Queue 
  qtmp :=: q 
  confirm restriction OrderedQueue(q) 
  if not IsEmpty (qtmp) then 
    variable partitionElement: Item 
    variable qBig: Queue 

    Dequeue (qtmp, partitionElement) 
    Partition (qtmp, qBig, partitionElement) 
    Sort(qtmp) 
    Sort(qBig) 
    Enqueue (qtmp, partitionElement) 
    Concatenate (qtmp, qBig) 
    q :=: qtmp 
  end if 
end Sort 
end QuickSort 

Figure 2.12: Quicksort Implementation of Sort Using Restrictions
SmallValueQueueRestriction, and OrderedQueueRestriction all use the mathematical unit StringExtras and restricts the extension QueueTemplate (a new relationship that is denoted by the label ‘r’). OrderedQueueConcatenateExtension enhances OrderedQueue and restrict QueueConcatenateExtension. Finally, QuickSort implements Sort and uses each of the restrictions. While there is a cost of using restrictions, as shown by the addition of many new contracts and dependencies, the benefits overcome these potential limitations; see Section 2.5.7.

Finally, to finish an earlier discussion about the implementation of the consumes restriction or establishes restriction annotation for an operation’s formal parameter, one can implement the annotation by automatically translating it into a requires or ensures clause, respectively, in the operation contract. On every client use of the operation, the verification system adds a confirm restriction annotation after the call to reassert the restriction, generating one additional (simple) VC.
This process can be invisible to the client programmer, but simplifies the information needed for restrictions by avoiding carrying it across operation boundaries.

In summary, we have demonstrated and discussed the idea of restrictions, a contract of an invariant over potentially many variables with notation to indicate how procedures interact with the invariant, either maintaining it (under restriction), starting it (establishes restriction) or ending it (consumes restriction). In all of these cases, VCs are generated for each operation to make sure that the reasoning is correct. Finally, annotation to indicate when restrictions start in a block of code is given by confirm restriction. A restriction block is terminated by an end restriction annotation, which has not occurred in our example; the next example will demonstrate this in code.

2.4.3 More Uses of Restrictions

Our second example using restrictions stems from the motivating example, namely the idea of memoizing the result of a function. This can be encapsulated in a component called \texttt{MemoizingMachineTemplate}, shown in Figure 2.14.

\begin{verbatim}
contract MemoizingMachineTemplate (type DItem, type RItem, definition F(x: DItem) : RItem )

  type MemoizingMachine is modeled by boolean
  exemplar m
  initialization ensures
    m = true

  function Compute (restores m: MemoizingMachine, restores d: DItem)
    ensures
      Compute = F(d)

end MemoizingMachineTemplate

Figure 2.14: MemoizingMachine Contract
\end{verbatim}
The **MemoizingMachine** is an interesting component in the sense that it has unique features. The first is that it has no state. Its only operation returns the result of computing a function on a given **DItem**, namely the function given as a parameter.

The implementation uses a component called a **PartialMap** that acts like a partial function, *i.e.*, one can **Define** domain/range (or key/value) pairs into the **PartialMap** and retrieve them later, shown in Figure 2.15. Operations to check if a domain item is defined along with the operation to **Undefine** either a particular mapping or an unspecified mapping (**UndefineAny**) are provided. Similar components can be found in most programming languages.

While a general **PartialMap** may include any valid values of the two types, the specific usage we have in mind has more structure—namely we store already computed domain/range pairs. The representation has a convention to that effect, and the code also includes a restriction in the **Compute** body. Now, in order to do this we must assume that we may make copies of both domain and range values via **Replica** functions; the realization accepts parameters to compute the given function as well as functions to make copies of **RItem**s and **DItem**s.

An implementation of **MemoizingMachine**, shown in Figure 2.17 uses the restriction within the body of **Compute**. The **confirm restriction** is implied by the convention and the **end restriction** (which indicates that we do not expect the restriction predicate to hold anymore) gives us the convention when the operation terminates. After the **Undefine**, we have the new fact that \( F(dCopy) = Compute \), indeed either branch of the **if else** gives us this fact. The **also requires clause** for **Define** requires us to prove that \( F(dCopy) = rCopy \) right before the **Define** call. We also see the first instance of **end restriction** being used; the statement does
contract PartialMapTemplate (type DItem, type RItem)

uses UnboundedIntegerFacility

definition IS_PARTIAL_FUNCTION (m: finite set of (d: DItem, r: RItem)):
  boolean is
  for all d1, d2: DItem, r1, r2: RItem
  where ((d1, r1) is in m and (d2, r2) is in m)
  (if d1 = d2 then r1 = r2)

math subtype PARTIAL_MAP_MODEL is finite set of (d: DItem, r: RItem)
exemplar m
constraint
  IS_PARTIAL_FUNCTION (m)
definition IS_DEFINED_IN (m: PARTIAL_MAP_MODEL, d: DItem): boolean is
  there exists r: RItem ((d, r) is in m)
type PartialMap is modeled by PARTIAL_MAP_MODEL
exemplar m
initialization ensures
  m = empty_set

procedure Define (updates m: PartialMap, clears d: DItem, clears r: RItem)
  requires
  not IS_DEFINED_IN (m, d)
  ensures
  m = #m union {(#d, #r)}

procedure Undefine (updates m: PartialMap, restores d: DItem,
  replaces dCopy: DItem, replaces r: RItem)
  requires
  IS_DEFINED_IN (m, d)
  ensures
  m = #m \ {((dCopy, r)) and
  (dCopy, r) is in #m and
  dCopy = d

procedure UndefineAny (updates m: PartialMap, replaces d: DItem,
  replaces r: RItem)
  requires
  m /= empty_set
  ensures
  (d, r) is in #m and
  m = #m \ {((d, r))

function IsDefined (restores m: PartialMap, restores d: DItem): control
  ensures
  IsDefined = IS_DEFINED_IN (m, d)

function IsEmpty (restores m: PartialMap): control
  ensures
  IsEmpty = (m = empty_set)

function Size (restores m: PartialMap): Integer
  ensures
  Size = |m|

end PartialMapTemplate

Figure 2.15: PartialMap Contract
contract FunctionalMapRestriction(type DItem, type RItem,
    definition F(x: DItem) : RItem
) restricts PartialMapTemplate

restriction AFunction(m: PartialMap) is
    for all x: DItem, y: RItem where ((x, y) is in m) (y = F(x))

procedure Define(updates m: PartialMap, clears d: DItem, clears r: RItem)
    under restriction AFunction(m)
    also requires r = F(d)

procedure Undefine(updates m: PartialMap, restores d: DItem,
    replaces dCopy: DItem, replaces r: RItem)
    under restriction AFunction(m)
    also ensures r = F(d)

procedure UndefineAny(updates m: PartialMap, replaces d: DItem,
    replaces r: RItem)
    under restriction AFunction(m)
    also ensures r = F(d)

end FunctionalMapRestriction

Figure 2.16: FunctionalMap Restriction

not change the values of any variables, so after this statement we may assume the invariant is true of the actual variable.

The additional VCs generated by the FunctionalMapRestriction require reasoning about quantifiers. However once the proofs are done, these VCs need never be examined again, their value is that this factorization simplifies the proofs of the Compute VCs. Moreover, these VCs are actually the essence of why the code is correct; the other VCs are simple bookkeeping. The restriction VCs appear in Figures 2.18 and 2.19.

2.5 Full Syntax and Semantics for Restrictions within RESOLVE

Given the examples before for intuition and explanation, we are now in a position to define the full semantics of restrictions. We do so in the context of the current OSU
realization PartialMapRealization (  
  function Function (restores d: DItem): RItem  
    ensures  
      Function = F(d),  
    function Replica(restores r: RItem): RItem  
      ensures  
        Replica = r,  
    function Replica(restores d: DItem): DItem  
      ensures  
        Replica = d  
  ) implements MemoizingMachineTemplate

uses FunctionalMapRestriction (F) for PartialMapTemplate (DItem, RItem)

type representation for MemoizingMachine is (  
  items: PartialMap  
)

exemplar m

convention
  for all x: DItem (for all y: RItem where ((x, y) is in m)  
  (y = F(x)))

end MemoizingMachine

function Compute (restores m: MemoizingMachine, restores d: DItem)  
variable dCopy: DItem  
variable rCopy: RItem

confirm restriction AFunction(m.items)

if not IsDefined(m.items, d) then  
  dCopy := Replica(d)  
  Compute := Function(d)  
else  
  Undefine(m.items, d, dCopy, Compute)
end if

rCopy := Replica(Compute)
Define(m.items, dCopy, rCopy)
end restriction AFunction(m.items)

end Compute

end PartialMapRealization

Figure 2.17: Implementation Using FunctionalMapRestriction
RESOLVE syntax and semantics for ease of presentation purposes; the semantics of restrictions can easily be “ported” to the other dialects of RESOLVE and to other programming languages.

We use the syntax, semantics and proof rules defined by Heym [31], which in turn use syntax, semantics and proof rules defined by Krone [40]. The only change from Heym’s presentation are minor syntactic changes resulting from the current OSU RESOLVE syntax.

We first review the key notions of the semantics and proof rules for Heym’s dissertation. The purpose of the syntax, semantics, proof rules, and soundness and completeness arguments for this “minor” addition of a language construct is to build confidence that the construct actually does perform the function we have described in the previous chapters.
2.5.1 RESOLVE Syntax, Semantics and Proof Rules

Syntactically, restrictions are another type of contract in RESOLVE. Other types of contracts include definitions of new types, (e.g., QueueTemplate) or an enhancement (e.g., Sort); the grammar for each of these has been codified previously in the contract non-terminal. The difference between these two types of contracts is that an enhancement may not define new types. Essentially, restrictions are contracts that may extend multiple contracts (either types or enhancements). The restrictions themselves, as discussed earlier, may include operations that include also ensures and also requires clauses; newly declared procedures may also use restrictions.

Our definition of syntax is based both on the syntax described by Heym [31] and the current RESOLVE tools developed by the OSU RSRG [62] which we will highlight key components of. The semantics of RESOLVE is denotational and functional, namely the semantics of a program is a function that maps environments to environments. The environment captures the state of execution; it includes information needed for the execution of the program such as the values of all variables and the information needed for calling procedures. The environment includes mathematical and theoretical components, such as whether the environment is unusable or whether something has gone wrong in the execution. The environment also includes auxiliary information needed for the execution of every statement, e.g., remembering the state of variables before a call. Figure 2.20 presents the environmental information needed for RESOLVE.

The $I$ function denotes the interpretation of programs, for any given program, it returns a function that maps an Environment to another environment or a Boolean.
1. $\mathcal{I}$: Programs $\rightarrow$ (Environments $\rightarrow$ (Environments $\cup$ Boolean))

2. Environments = Abnormal-assert-statuses $\cup$ (Current-states $\times$ Old-states $\times$ Index-states $\times$ Setups $\times$ Declaration-meanings)

3. Abnormal-assert-statuses = \{VT, CF, $\bot$\}

4. Assert-statuses = Abnormal-assert-statuses $\cup$ \{NL\}

5. Current-states = (Current-variable-names $\rightarrow$ Values)

6. Index-states = ($\mathbb{Z} \rightarrow$ Current-states)

7. Setups = Current-states *

8. Augmented-old-variable-names = \{t $\circ$ s $\in$ Current-variable-names $\land$ $|t| > 0$ $\land$ elements(t) = \{#\}\}

9. Old-states = (Augmented-old-variable-names $\rightarrow$ Values)

10. Declaration-meanings = (Identifiers $\rightarrow$ (Type-meanings $\cup$ Predicate-meanings $\cup$ Procedure-meanings $\cup$ Generic-meanings $\cup$ Module-meanings))

11. Values = \{v $\mid$ there exists $t \in$ Type-meanings such that $v \in t$\}

12. Type-meanings = Base-types $\cup$ Defined-types

13. Predicate-meanings = \{p $\mid$ p : $d \rightarrow$ t $\land$ d $\in$ Argument-domains $\land$ t $\in$ Type-meanings\}

14. Argument-domains = \{T_1 \times \cdots \times T_n $\mid$ T_i $\in$ Type-meanings $\land$ 1 $<=$ n $\land$ 1 $<=$ i $<=$ n\}

15. Procedure-meanings = Predicate-meanings $\times$ Predicate-meanings $\times$ Procedure-functions $\times$ Status-functions

16. Procedure-functions = \{f $\mid$ f : $d \rightarrow$ r $\land$ d $\in$ Argument-domains $\land$ r $\in$ Argument-domains\}

17. Status-functions = \{f $\mid$ f : $d \rightarrow$ Assert-status $\land$ d $\in$ Argument-domains \}

Figure 2.20: Definition of Mathematical Machinery
An Environment is returned for the interpretation of programmatic blocks, while a Boolean is returned when a Boolean valued expression is evaluated.

The Environments domain includes several important components of information. It is the union type of two different domains, \textit{i.e.}, an environment value can be value in either of those two disjoint domains. The first (Abnormal-assert-statuses ) domain indicates that an Environment’s value is “abnormal”, denoted by the values within the domain. The value CF indicates “Categorically False”, in other words some assertion is false. The dual of this is the value VT, which means “Vacuously True.” VT indicates that the environment is not in a state in which the program is meaningful. For example, consider a procedure called \textbf{Increment} that takes an \texttt{Integer} parameter and whose contracts states that its behavior is to increment the \texttt{Integer}. An environment is not suitable for this program if the environment is started up with an actual implementation for \textbf{Increment} that decrements that \texttt{Integer}. Any procedures that use \textbf{Increment} will be incorrect, but not because of anything that they have control over. Within the Assert-statuses, there is one more status value, namely NL or “Neutral,” which indicates that everything is normal. In order to simplify the notation used by Heym \cite{31}, an Environment value is either an Abnormal-assert-status or it is a valid value (with an implicit NL status).

The Index-state maps integers to a map of variable names to values. This is used within the indexed method for generating VCs.

The Setups is used to define the meaning of an \texttt{alter all} statement. It is a finite sequence of Current-states. The \texttt{alter all} statement modifies the current state and sets it to the front of the Setup and removes the state from front of the Setup. Heym \cite{31} notes that
the term ‘setup’ is a slang expression for a situation (e.g., a trial jury) that has been ‘rigged’ (arranged in advance). The initial environment contains, in the setup component, a prearrangement of the meeting of each alter all statement that might be encountered during execution.

The Old-states domain is a map from augmented variable names to values, where the variable names are augmented by adding a positive number of #’s to the beginning of them. This domain is used to remember the “old” states of variables, such as for any ensures clauses or loop invariants.

The Declaration-meaning is particularly important to our context. In it the semantics of certain global names is maintained. These global names include types, procedures, restrictions, functions, generics, and modules. Essentially all of requisite information about the semantics of each of these names is maintained in the Declaration-meaning.

The Type-meanings are either those that are predefined, so called Base-types, or those that are defined by users, the Defined-types. The Values domain is simply the union of all of the possible values of any possible type.

Predicate-meanings include all potential functions that are possible given the Type-meanings, i.e., a value p in Predicate-meanings takes a certain number of arguments from Type-meanings and returns a value within some Type-meanings. Syntactically correct programs do not have name conflicts; any procedures are correctly typed.

A value in Procedure-meanings is a tuple of several components, intuitively it contains all of the requisite pieces of the semantics of any procedure. The first component (called the domain predicate) is a Predicate-meaning that indicates the values for which the procedure is callable; it functions like a requires clause. The second
component is also a Predicate-meaning (called the effect predicate) that indicates what the potential outgoing values of the parameters are after the procedure is called given the incoming parameters, like a \texttt{ensures} clause. The next component is the actual semantics of the procedure, \textit{i.e.}, it indicates how the parameters are changed, and is called the procedure function. The last component is a function that indicates the assert-status of the procedure given any input values, called the status function.

1. \texttt{cs} represents the Current-states component
2. \texttt{os} represents the Old-states component
3. \texttt{ns} represents the Index-states component
4. \texttt{se} represents the Setups component
5. \texttt{d} represents the Declaration-meaning component
6. \texttt{dp} represents the Domain predicate of a procedure’s Procedure-meaning
7. \texttt{ep} represents the Effect predicate of a procedure’s Procedure-meaning
8. \texttt{pf} represents the Procedure Function of a procedure’s Procedure-meaning
9. \texttt{sf} represents the Status Function of a procedure’s Procedure-meaning

Figure 2.21: Notation for Environments

Figure 2.21 shows the notation used to represent each of these components. We will either use a component name by itself to indicate an arbitrary value of that component, \textit{e.g.}, \texttt{cs}. We also use dot notation to represent the component of a particular environmental, \textit{e.g.}, \texttt{env.d(h).dp} for the domain predicate of a procedure called “\texttt{h}” in environment \texttt{env}.

A program \texttt{P} is considered valid if for any environment \texttt{env}, where \texttt{env} \notin \texttt{Abnormal-statuses}, \( \mathcal{I}(P)(env) \neq \text{CF} \).
The proof rules defined by Heym [31] based on rules defined by Krone [40] and proved sound and relatively complete [31], can be thought of in two parts. The first is within operation body code and the second is the module and declaration code. The first piece uses the *indexed method* and can be thought of as a symbolic tracing table [63]. For a block of code, any assumptions are written as facts and any proof obligations are written as obligations. Assumptions may arise via *ensures* clauses of just-returned operation calls, control flow, loop invariants, representation invariants, abstraction relations and *requires* clauses at the beginning of an operation body. Obligations can arise from loop invariants, representation invariants, *requires* clauses of called operations, and *ensures* clauses at the end of operation bodies. Looking ahead, restrictions will add two new kinds of statements within this type of code.

The second part includes the definitions, contracts, and realizations. The rules here deal with remembering the definitions, and checking for validity of realizations. Evaluating a procedure declaration, the proof rules simply add the requisite information into the proof context. However, VCs are generated so a realization can be proved, either exactly in the manner mentioned above for an extension, or by additionally proving extra VCs for a data type representation. These VCs check for feasibility of the implementation, and that each operation is implemented in a way that preserves the declaration contract. Looking ahead, restrictions will add a new kind of definition to this type of code.

The proof of soundness and relative completeness by Heym [31] uses the concept of preserving invalidity from the program to the proof rules for the proof of soundness. Heym proves that, if the program is invalid, then each rule of the VC generation
process maintains the invalidity. For the proof of relative completeness, the converse is used, \textit{i.e.}, that reversing each rule preserves invalidity. Our technique for adding restrictions will be to show that adding the extra statements does not invalidate Heym’s proofs, as well as proving the soundness and relative completeness of the requisite additional rules.

2.5.2 RESOLVE with Restrictions Syntax

Our first attempt at adding the syntax for restrictions is given in Figure 2.22. Here we include syntactic support for all of the requisite features, as described earlier in this chapter. We have the constraint that all variables included in \texttt{consumes restriction} or \texttt{establishes restrictions} are parameters to the procedure, while the \texttt{under restriction} parameters may have extra variables, but their value may not change as a result of the call. Unfortunately, the semantics for such a system would be difficult to create without changing the mathematical domains in Krone’s and Heym’s semantics in a deep way. The semantics would need to “remember” the restriction declaration while processing the restricted procedures and be able to switch context on the fly; the same procedure name may actually have different contracts depending on the variables involved in the call. Moreover, the actual intended meaning of restrictions is not dynamic; it is statically determined within the source code. Rather than attempting to redo all of the mathematical machinery for not only procedure declarations but also procedure calls, we will take a different approach—we stipulate that the compiler syntactically rewrites any code written in the form in Figure 2.22 into the form described by Figure 2.23—a source-to-source transformation. The declarations of restrictions are folded into the \texttt{requires} and \texttt{ensures} clauses of any procedures. We
<compilationUnit> := <contract> | <realization> | <restriction>

<statement> := <ifelse> | <loop> | <swap> | <functionAssignment> | <operationCall> | <clear> | <confirm> | <assume> | <adjunctAssignment> | <alterAll> | <remember> | <whenever> | <stow> | <begin_restriction> | <end_restriction>

<restriction> ::= 'contract' <identifier> ('restricts' <identifier> )+ 
                { <uses> } 
                {<restricted_procedure> } 
                'end' <identifier>;

<restricted_procedure> ::= 'procedure' <identifier> '(': <parameters> ? ')' 
                            <under_restrictions> 
                            <also_requires> <also_ensures> ;

<procedure> ::= 'procedure' <identifier> '(': <parameters>? ')' 
              <under_restrictions>? <establish_restrictions>?
              <consumes_restrictions>? ( <precondition> )?
              <postcondition> ;

<localProcedure> ::= 'local' 'procedure' <identifier> '(': <parameters>? ')' 
                   <under_restrictions>? <establish_restrictions>?
                   <consumes_restrictions>? <precondition>?
                   <postcondition> <metric>? <block> 
                   'end' <identifier> ;

<procedureBody> := 'procedure' <identifier> '(': <parameters>? ')' 
                  <under_restrictions>? <establish_restrictions>?
                  <consumes_restrictions>? <metric>? <block> 
                  'end' <identifier> ;

<under_restrictions> ::= ( 'under' 'restriction' <identifier> ) + ;
<establishes_restrictions> ::= ( 'establishes' 'restriction' <identifier> ) + ;
<consumes_restrictions> ::= ( 'consumes' 'restriction' <identifier> ) + ;
<also_requires> ::= { 'also' 'requires' <formula> } ;
<also_ensures> ::= { 'also' 'ensures' <formula> } ;
<begin_restriction> ::= 'confirm' 'restriction' <identifier> '(': <parameters> ')' 
<end_restriction> ::= 'end' 'restriction' <identifier> '(': <parameters> ')'

Figure 2.22: Additional Grammar for RESOLVE with Restrictions Added
also assume that the ‘$’ symbol is reserved by the compiler to allow us to encode where this procedure came from; we could use any symbol that is not allowed in the syntax of an identifier. In that form, we will encode the restrictions (under restriction, establishes restriction, and consumes restriction) into the procedure names along with which parameters are affected by the restrictions. We also assume that the restricted procedure has extra parameters corresponding to the free parameters in any under restriction, consumes restriction, and establishes restriction clauses placed after the original parameters under the restores mode. So, if we were to examine the Concatenate procedure declaration in Section 2.4, the compiler would effectively rewrite that as:

```plaintext
procedure $$O2$$A0$U$OrderedQueue$1$1$U$OrderedQueue$1$2$Concatenate ( 
    updates q1: Queue, 
    clears q2: Queue )

requires
    IS_PRECEDING(q1, q2)
```

Moreover, we also assume that the same renaming process is done to all procedures within a restriction block denoted by confirm restriction and end restriction or by the start or end of a restricted procedure implementation. This same process happens if a procedure body or a local procedure has restriction information, e.g., in the Quicksort implementation and Partition local procedure. The body is then changed to have confirm restriction and end restriction statements at the beginning and end of the block as required, i.e., confirm restrictions are added at the beginning for both under restriction and consumes restriction, and end restrictions added at the end for both under restriction and establishes restriction.

48
Figure 2.23: Modified Grammar for RESOLVE with Restrictions Added
2.5.3 RESOLVE with Restrictions Semantics

We now present the semantics of the restrictions constructs as additions to the semantics of RESOLVE. We augment the definition of the environments with two extra components, $R$ and $CR$. $R$ is a function from Identifiers to Predicate-meanings, that intuitively gives the predicate of every restriction; this is needed when processing the Restriction declaration. We leverage the already defined notion of a conformal procedure meaning to define the semantics of a Restriction definition. A conformal procedure meaning intuitively that the semantics of the implementation does not violate the semantics of the specification. Formally, [17]

...A procedure-meaning is defined to be conformal if the following conditions are true:

let pd = (dp, ep, pf, sf) be the procedure-meaning. Then,

1. the number and types of arguments of the various components of pd are consistent with one another;
2. for all $x_1, \ldots, x_n$ such that $dp(x_1, \ldots, x_n)$ and for all $y_1, \ldots, y_m$ such that
   
   \begin{align*}
   & pf(x_1, \ldots, x_n) = (y_1, \ldots, y_m), \\
   & sf(x_1, \ldots, x_n) = CF \\
   & \text{if } sf(x_1, \ldots, x_n) = NL \text{ then } ep(x_1, \ldots, x_n, y_1, \ldots, y_m)
   \end{align*}

The second component $CR$ denotes the “current” restrictions, i.e., it is a finite set of restriction names and string(variable) pairs. The idea is that anything defined in this set is an actual current restriction on these variables. We use and maintain this extra information for the purposes of defining the semantics of alter all statements. We extend Heym’s definition of the semantics of alter all [31] in Figure 2.24\footnote{Both semantics are equivalent for programs without restrictions.}. The key idea is that any new variable values used in the alter all statement, i.e., those in the head of se, may not change the value of any current restrictions. If we had a
variable \( q \) that is restricted by \texttt{OrderedQueue} and before an \texttt{alter all} statement the value of \( q \) was \( <3,2,3> \) in the head of \( se \) in that environment then the semantics would assign a value of VT, as the \texttt{alter all} modified the value of a restriction predicate. We also modify the semantics of procedure bodies so that the \( CR \) component is assumed to have exactly the current restrictions within a procedure body.

\[
I(\text{alter all})(env) \overset{\text{def}}{=} \begin{cases} 
VT & \text{if } \textit{bogus}(env) \\
[first(se), ns, \text{tail}(se), os, d, R, CR] & \text{otherwise}
\end{cases}
\]

\[
\textit{bogus}(env) \overset{\text{def}}{=} env.se = \epsilon \lor (\exists R. R \in \{x | \exists y (x, y) \in env.CR \} \land
I(env.R(R))(env) \neq I(env.R(R))(first(se), ns, \text{tail}(se), os, d, R, CR))
\]

Figure 2.24: Redefinition of \texttt{alter all} Semantics

The first definition in Figure 2.26 makes sure that if the stored definition of the Restriction does not match the desired predicate given by the source, that the returned environment is VT—it give us no information about the validity of the program. Nothing else can happen to a definition of a restriction, so otherwise, the semantics evaluates the restricted procedure definitions.

The semantics of a restricted procedure declaration are slightly more complicated, but follow the same idea as the semantics of the definition of a restriction as shown in Figure 2.26. The semantics needs to determine if the environment is not suited for the definition (and assign a semantics of VT), or if the definition itself is broken in some way (and assign a semantics of CF), or accept the definition and leave the environment unchanged. The interesting piece of this construct is that we reuse
any already defined semantics ($pf$ and $sf$) for the underlying procedure, therefore the semantics of the restricted procedure declaration must check to make sure the semantics matches\(^2\). There might be a few instances in which the environment is not suitable, namely when the $pf$ or $sf$ components do not match the underlying procedure. The semantics of the declaration may be wrong if the specification given, with the requisite restriction predicates plugged into the `requires` and/or `ensures` clauses, is not conformal. While the “overloading” of an already declared procedure has pitfalls, the definition of `new` procedures, either as extensions, a local procedure or a procedure body, follow the semantics of normal procedure definitions after the source translation.

\[\text{ProcName} : \text{encoded\_restriction\_name} \rightarrow \text{proc\_name} \text{ where} \]
\[\text{ProcName}(\text{rpm}) = \begin{cases} 
\text{rpm} & \text{if rpm is just a procedure\_name} \\
\text{ProcName}(r\text{\_proc\_name}) & \text{if } \text{rpm} = \$$d$$\text{R}_n\$$I_1$$ \ldots $$I_n$$r\text{\_proc\_name} 
\end{cases}\]

Figure 2.25: Auxiliary Function

In order to define the semantics of a restricted procedure, we first must define a helper function to unpack the restrictions from the procedure name. Our helper function is $\text{ProcName}$, shown in Figure 2.25, that, given an encoded restricted identifier, produces the original procedure name that it is based on.

\(^2\)We actually could define the semantics to require only a compatible procedure meaning. However this goes far enough from our intuition of what the semantics should be that we reject it as a potential solution.
\( I \left( \text{contract Rname restricts Cname } \right) \) \\
\{ uses Cname1 \} \\
Restriction R (parameters) is P(parameters) \\
\{ uses Cname1 \} \\
\( \text{Rp} \) \\
\( (env) \text{ def } = \) \\
\[ \begin{cases} 
VT & \text{if } env.\text{Restrictions}(R) \neq env.d(P) \\
I(\text{Rp}) & \text{otherwise} 
\end{cases} \]

\( I \left( \text{procedure e_r_i( parameters )} \right) \) \\
\( (env) \text{ def } = \) \\
\[ \begin{cases} 
VT & \text{if } env.d(e_r_i).pf \neq env.d(\text{ProcName}(e_r_i)).pf \\
VT & \text{else if } env.d(e_r_i).sf \neq env.d(\text{ProcName}(e_r_i)).sf \\
VT & \text{else if } env.d(e_r_i).dp \text{ is not a superset of ar} \\
VT & \text{else if } env.d(e_r_i).ep \text{ is not a superset of ae} \\
CF & \text{else if } env.d(e_r_i) \text{ is not conformal} \\
env & \text{otherwise} 
\end{cases} \]

\( I(\text{confirm restriction R(parameters)}) \) \\
\( (env) \text{ def } = \) \\
\[ \begin{cases} 
CF & \text{if } \neg env.R(R)(\text{parameters}) \\
env' & \text{otherwise, where } env = env' \\
& \text{except } (R, < \text{parameters }> ) \text{ is added to CR} 
\end{cases} \]

\( I(\text{end restriction R(parameters)}) \) \\
\( (env) \text{ def } = \) \\
\( env' \) where \( env = env' \) except \( (R, < \text{parameters }> ) \) is removed from CR

Figure 2.26: Definition of the Semantics of Restrictions
2.5.4 RESOLVE with Restrictions Proof Rules

The proof rules for restrictions are presented in a similar manner to Heym’s presentation, i.e., via two definitions, a program-like and math-like definition. Given a program in one form, the rules allows for the replacement of the other. Within our description, we use \( \lfloor \) and \( \rfloor \) to denote free variables within a formula, \( x \leftarrow y \) to denote that the free variable \( x \) is replaced by \( y \) within a formula.

\[
P \overset{\text{def}}{=} C \setminus \begin{align*}
&\text{prec\_top\_lev\_code} \\
&\text{stow}(i) \\
&ACseq_0 \\
&\text{whenever } Br\_Cd \text{ do} \\
&\text{confirm restriction } R(y^1, \ldots, y^n) \\
&cd\_suffix \\
&\text{end whenever} \\
&\text{fol\_top\_lev\_code}
\end{align*}
\]

\[
M \overset{\text{def}}{=} C \setminus \begin{align*}
&\text{prec\_top\_lev\_code} \\
&\text{stow}(i) \\
&ACseq_0 \\
&\text{whenever } Br\_Cd \text{ do} \\
&\text{confirm } E[x^1 \leftarrow y^1, \ldots, x^n \leftarrow y^n] \\
&cd\_suffix \\
&\text{end whenever} \\
&\text{fol\_top\_lev\_code}
\end{align*}
\]

Additional syntactic assumption:
\( C \) includes \( \text{Restriction } R(x^1:T_1, \ldots, x^n:T_1) \) is \( E[x^1, \ldots, x^n] \) and \( y^1:T_1, \ldots, y^n:T_1 \)

Figure 2.27: Confirm Restriction Proof Rule

We informally argue why the confirm restriction proof rule, shown in Figure 2.27 is sound and relatively complete. Given an invalid math or program environment, we could take the exact same environment for the other rule and it would still be invalid.
\[
P \equiv C \setminus \text{prec\_top\_lev\_code} \\
\text{stow}(i) \\
\text{ACseq}_0 \\
\text{whenever } \text{Br\_Cd} \text{ do} \\
\text{end restriction } R(y^1,\ldots,y^n) \\
\text{cd\_suffix} \\
\text{end whenever} \\
\text{fol\_top\_lev\_code} \\
M \equiv C \setminus \text{prec\_top\_lev\_code} \\
\text{stow}(i) \\
\text{ACseq}_0 \\
\text{whenever } \text{Br\_Cd} \text{ do} \\
\text{assume } E[x^1 \leftarrow y^1_1,\ldots,x^n \leftarrow y^n_i] \\
\text{cd\_suffix} \\
\text{end whenever} \\
\text{fol\_top\_lev\_code} \\
\] 

Additional syntactic assumption: 
\( C \) includes \text{Restriction} \( R(x^1:T_1,\ldots,x^n:T_1) \) is \( E[x^1,\ldots,x^n] \) and \( y^1:T_1,\ldots,y^n:T_1 \)

Figure 2.28: End Restriction Proof Rule

We informally discuss why the end restriction proof rule, shown in Figure 2.28, is sound and relatively complete. Just like the confirm restriction rule, given an invalid math or program environment, we could take the exact same environment for the other rule and it would still be invalid.

We argue informally that the proof rule for a Restriction declaration, shown in Figure 2.29, is sound and relatively complete by simply noting that both the program and math equation are CF under exactly the same conditions, i.e., the semantics of the declaration is essentially the proof rule. If the declaration has already been processed, there is no need to generate VCs for it again.

We take a prototypical procedure body with any restriction in the declaration and show the proof rule in Figure 2.30. The proof rule simply replaces the restricted procedure body with what the procedure body proof rules would have done if it were
\[ P \overset{\text{def}}{=} C \backslash \text{contract } R \text{ restricts } C_1 \ldots \text{ restricts } C_n \]
\[ \text{uses } A_1 \ldots A_m \text{ enhances } B_1 \ldots B_j \]
\[ \text{Restriction } R_1(x_1 : T_1, \ldots, x_n : T_n) \text{ is } E \]
\[ \text{restricted\_procedures} \]
\[ \text{end } R \text{ code confirm } Q \]
\[ M \overset{\text{def}}{=} C \cup \{ \text{contract } R \text{ restricts } C_1 \ldots \text{ restricts } C_n \]
\[ \text{uses } A_1 \ldots A_m \text{ enhances } B_1 \ldots B_j \]
\[ \text{Restriction } R_1(x_1 : T_1, \ldots, x_n : T_n) \text{ is } E \]
\[ \text{restricted\_procedures} \]
\[ \text{end } R \} \backslash \text{code confirm } Q \]

and for each procedure \( W \) in restricted\_procedures
\[ M \overset{\text{def}}{=} C' \cup \backslash \text{assume } (E \land E_{\text{consumes}} \land Req_{\text{also}} \land Req) \text{ W confirm } (Ens_{\text{also}} \land Ens \land E \land E_{\text{establishes}}) \]

Additional syntactic assumptions: Restriction declaration does not occur in \( C \); if it does, the proof rule skips over the declaration.
\( C \) includes the contracts for \( C_1, \ldots, C_n, A_1, \ldots, A_m, B_1, \ldots, B_j \)
\( C' \) includes all of \( C \) and \( x^1 : T_1, \ldots, x^n : T_n \)

We also assume that the procedure VCs will have the bridge rule applied to them.

Figure 2.29: Restriction Declaration Proof Rule

\[ P \overset{\text{def}}{=} C \backslash \text{procedure } RP(\text{parameters}) \]
\[ \text{under restriction } R_1(\text{parameters}) \]
\[ \text{establishes restriction } R_2(\text{parameters}) \]
\[ \text{consumes restriction } R_3(\text{parameters}) \]
\[ \text{metric } M \]
\[ \text{block} \]
\[ \text{end } P \]
\[ \text{code} \]
\[ \text{confirm } Q \]
\[ M \overset{\text{def}}{=} C \backslash \text{assume } (E_1 \land E_2 \land pre \land post) \]
\[ \text{confirm restriction } R_1(\text{parameters}) \]
\[ \text{confirm restriction } R_2(\text{parameters}) \]
\[ \text{block} \]
\[ \text{end restriction } R_1(\text{parameters}) \]
\[ \text{end restriction } R_2(\text{parameters}) \]
\[ \text{confirm } (post \land E_1 \land E_2) \]
\[ \text{code} \]
\[ \text{confirm } Q \]

Additional syntactic assumptions:
\( C \) includes the matching contracts for \( RP \) and all constructs used in the block
\( C' \) includes all of \( C \) and \( x^1 : T_1, \ldots, x^n : T_n \)

We also assume that the body VCs will have the bridge rule applied to them.

Figure 2.30: Restricted Procedure Body Proof Rule
a normal procedure body. Thus, the proof rule follows the semantics of the procedure exactly, making it is sound and relatively complete by construction.

\[ P \overset{\text{def}}{=} C \backslash \begin{align*}
\text{stow}(i) \\
\text{ACseq}_0 \\
\text{whenever } Br\_Cd \text{ do} \\
\text{Proc}(y^1, \ldots, y^n) \\
\text{cd\_suffix} \\
\text{end whenever}
\end{align*} \]

\[ M \overset{\text{def}}{=} C \backslash \begin{align*}
\text{stow}(i) \\
\text{ACseq}_0 \\
\text{confirm } (Br\_Cd) \rightarrow (pre \land ar)[x^1 \leftarrow y^1_i, \ldots, x^n \leftarrow y^n_i] \\
\text{alter all} \\
\text{stow}(j) \\
\text{whenever } Br\_Cd \text{ do} \\
\text{assume } (post \land ae)[\#x^1 \leftarrow y^1_1, \ldots, \#x^n \leftarrow y^n_i, x^1 \leftarrow y^1_1, \ldots, x^n \leftarrow y^n_i] \\
\text{cd\_suffix} \\
\text{end whenever}
\end{align*} \]

Additional syntactic assumption:
- \( C \) includes \( \text{Restriction } R(x^1 : T_1, \ldots, x^n : T_1) \) is \( E[x^1, \ldots, x^n] \) and \( y^1 : T_1, \ldots, y^n : T_1 \) and
- procedure \( \text{Proc}() \) requires \( \text{pre}[x^1, \ldots, x^n] \) ensures \( \text{post}[\#x^1, \ldots, \#x^n, x^1, \ldots, x^n] \) and
- procedure \( \text{Proc}() \) under restriction \( R() \) also requires \( \text{ar}[x^1, \ldots, x^n] \) also ensures \( \text{ae}[\#x^1, \ldots, \#x^n, x^1, \ldots, x^n] \)

Figure 2.31: Restriction Procedure Call Proof Rule

We also informally argue why the Restriction procedure call rule, shown in Figure 2.31, is sound and relatively complete. Here, because of the invariant property, any environment that is not abnormal right after the \texttt{whenever} must have the restriction satisfied, and if that is satisfied, because of the \texttt{Restricted} declaration rule, we already know that \texttt{ae} is implied by the postcondition and the Restriction predicate, therefore this is exactly like the procedure call rule, sound and relatively complete.
2.5.5 A Small Lemma on the Semantics of RESOLVE Programs

Our first step in the proof of the soundness and relative completeness of the proof rules for restrictions is to prove two lemmas about the semantics of RESOLVE programs, namely that if we happen to have a sequence of statements in which all operations preserve an invariant and the invariant held before the sequence of statements was executed, then the invariant holds after the sequence of statements was executed and within any intermediate environment—the foundation of restrictions. This makes sure that restriction predicates are actually preserved by the semantics.

Lemma 2.5.1. For any neutral environment env, sequences of statements SS, and formula R,

if $I(SS)(env)$ are not Abnormal-statues and

for any procedure used within SS and environment env,

if $I(R)(env)$ then $I(R)(I(SS)(env))$

then $I(R)(I(SS)(env))$.

Proof. We proceed by structural induction on SS.

The first base case is when $SS = \epsilon$, where $I(SS)(env) = env$ and therefore $I(R)(I(SS)(env))$

Next, we examine the base statements that have no sub-formulas. We note that remember, assume, confirm, and stow, if the interpretation remains neutral, do not change the state of the variables. Therefore we know that $I(R)(I(SS)(env))$. The alter all statement is assumed to maintain the formula, therefore the theorem goal must hold as well.
The next potential statements include sub-statement sequences. The first we examine is sequential composition, namely SS = S1 SS2. By assumptions, we have that $\mathcal{I}(SS)(env)$ is not abnormal, and S1 maintains $P$ as an invariant, so $\mathcal{I}(R)(\mathcal{I}(S1)(env))$. We also know, by the inductive assumption and earlier facts, that $\mathcal{I}(R)(\mathcal{I}(SS2)(\mathcal{I}(S1)(env)))$.

The next composition operator we examine is the \texttt{whenever} statement. However, the theorem is readily apparent no matter whether $\mathcal{I}(\mathtt{b_p_e})(env)$ holds. In the case where it does not hold, it is equivalent to an empty statement. When it does hold, since the evaluation of any function may not change the variables values, the induction assumption gives us the result. The \texttt{if} and \texttt{if else} constructs use the same reasoning; in either case of the condition, the inductive assumption gives the result; the evaluation of any boolean condition may not change the environment state.

Our last two statements are while loops and operation calls. We start with the while loop and then proceed to the operation calls. The interpretation of a while loop is $\mathcal{I}(\text{loop maintaining } I \text{ decreases } D \text{ while } B \text{ do } SS1 \text{ end loop})(env)$. Since the interpretation of a while loop involves a minimum fixed point operator, we start by using the inductive assumption, namely that $SS1$ satisfies the theorem. Moreover, we know that the while loop terminates by assumption. We also know that all procedures that use any of the free variables of $R$ preserve the value of $R$, so we know that $SS1$ itself preserves the $R$ formula, namely if $R$ holds before $SS1$, then it holds in the corresponding environment after $SS1$ executes. Moreover, by assumption we also know that $\mathcal{I}(R)(env)$. Therefore, $R$ is an invariant of this loop and, as such, must hold of the environment produced by the minimum fixed point of the loop, and therefore $\mathcal{I}(P)(\mathcal{I}(\text{loop maintaining } I \text{ decreases } D \text{ while } B \text{ do } SS1 \text{ end loop})(env))$. 

59
Finally, we examine the last type of statement, namely operation calls. We break any operation call into two cases: either it takes some $x_i$ or it does not. If it does not, the valuation of $R$ may not be changed, so the result holds. Even if the procedure does take some $x_i$, by assumption the valuation of $R$ is true in the environment after the operation call finished, therefore we have shown the lemma. ■

2.5.6 Proof of Soundness and Relative Completeness

In the proofs of soundness and relative completeness, our obligation is to show that the proof rules given for the untranslated source code are sound and relatively complete with respect to the semantics given above, i.e., first translated to new source code and then interpreted.

Our first goal is soundness:

**Theorem 2.5.2.** The proof rules set presented by Heym along with the rules for restrictions are sound for the above semantics.

We use a lemma proved by Heym [31] to simplify our task.

**Lemma 2.5.3.** If each proof rule preserves invalidity going from Programs towards Mathematics, then the proof system is sound.

We note that if we restrict our language to only those constructs presented by Heym [31], the semantics are equivalent. Heym’s proofs of soundness and relative completeness of those same proof rules are also applicable.

We now show directly that two of the three proof rules are sound and relatively complete.
Lemma 2.5.4. The confirm restriction rule preserves semantics (i.e., any environment of \(P\) or \(M\) is an environment of the other) and, consequently, is sound and relatively complete.

Proof. The proof rule replaces a confirm restriction with a confirm where we have substituted the restriction for the restriction’s predicate. Our obligation is to show that they have the same semantics, i.e., for any environment \(\text{env}\)

\[
\mathcal{I}([\text{confirm restriction } R(y^1,\ldots,y^n)](\text{env})) = \mathcal{I}([\text{confirm } E[x^1 \leftarrow y^1,\ldots,x^n \leftarrow y^n]](\text{env}),
\]

where \(E\) is the restriction predicate. The only two possible values for both are either unchanged, or CF. We prove one is CF if and only if the other is. Suppose the left hand side is CF. Then the semantics gives us that \(\neg \mathcal{I}(E)(\text{env})[x^1 \leftarrow y^1,\ldots,x^n \leftarrow y^n]\) where \(E\) is the restriction predicate, but by definition, the right hand side expression must fail as well. Similarly, if the left hand side evaluates to \(\text{env}\), then

\[
\mathcal{I}(E)(\text{env})[x^1 \leftarrow y^1,\ldots,x^n \leftarrow y^n],
\]

and so the right hand side expression must evaluate to \(\text{env}\) as well. Thus, we use the soundness and relative completeness arguments by Heym [31] for the confirm rule for soundness and relative completeness. 

Lemma 2.5.5. The end restriction rule preserves semantics and, consequently, is sound and relatively complete.

Proof. The proof rule replaces an end restriction with an assume where we have substituted the restriction for the restriction’s predicate. Our obligation is to show that they have the same semantics, i.e., for any environment \(\text{env}\)

\[
\mathcal{I}([\text{end restriction } R](\text{env})) = \mathcal{I}([\text{assume } E[x^1 \leftarrow y^1,\ldots,x^n \leftarrow y^n]](\text{env}),
\]

where \(E\) is the restriction predicate. We do have one extra complication, namely that not all environments are possible. Since the restriction is ending and all other rules...
preserve all restriction predicates at the beginning of `whenever` statements (a fact that is proved in lemma 2.5.8), we also have the fact that the restriction invariant is true in `env` before the end restriction. Therefore, the `assume` may not change the environment (as it always evaluates to true), and both sides of the equation are always the same value. ■

Now we turn our attention to the meat of the new proof rules, namely the Restricted procedure call rule.

**Lemma 2.5.6.** The Restricted procedure call rule preserves invalidity in the mathematical direction.

**Proof.** We suppose that $P$ is an invalid program and we show that $M$ is invalid. Let $env^P_I$ be a witness to $P$’s invalidity, and let $env^P_F = \mathcal{I}(P)(env^P_I)$. We note that $env^P_I$ is not an abnormal assert status (because $P$ is invalid only if it starts in a some normal environment, but ends up in a CF environment) but $env^P_F = CF$. Our goal is to show the existence of an environment $env^M_I$ such that $env^M_I$ is not an abnormal assert status, but $env^M_F = CF$, where $env^M_F = \mathcal{I}(P)(env^M_I)$.

We define several environments that will be used to examine where the execution of the program entered a CF state.

\[
\begin{align*}
env^P_i &\overset{\text{def}}{=} \mathcal{I}(\text{proc}_\text{top}_\text{lev}_\text{code} \text{ alter all stow}(i) ACseq_0)(env^P_I) \\
env^P_j &\overset{\text{def}}{=} \mathcal{I}(e_r_i \text{ stow}(j))(env^P_i) \\
env^{P_{eu}} &\overset{\text{def}}{=} \mathcal{I}(cd_suffix)(env^P_j)
\end{align*}
\]

We now examine the program equation to determine the total number of cases based on where in the code the environment enters the CF state.
1. $env_i^p = CF$

2. $env_i^p \neq CF$ and $\neg I(Br_Cd)(env_i^p)$. We note that this means that
   $I(fol_top_lev_code)(env_i^p) = CF$

3. $env_i^p \neq CF$ and $I(Br_Cd)(env_i^p)$ and $env_j^p = CF$

4. $env_i^p \neq CF$ and $I(Br_Cd)(env_i^p)$ and $env_j^p \neq CF$ and $env_{ew}^p = CF$

5. $env_i^p \neq CF$ and $I(Br_Cd)(env_i^p)$ and $env_j^p \neq CF$ and $env_{ew}^p \neq CF$. We note that this means that $I(fol_top_lev_code)(env_{ew}^p) = CF$

For each of these different cases, using Heym’s procedure call proof rule proofs [31] as a guide, we will provide environments $env_i^M$ such that the mathematical program is invalid under that environment. First, we define:

\[
env_i^M \overset{def}{=} I(proc_top_lev_code alter all stow(i)ACseq_0)(env_i^M)
\]

\[
env_F^M \overset{def}{=} I(P)(env_i^M)
\]

For Case 1, set $env_i^M = env_i^p$, then $env_i^p = env_i^M = CF$, and since CF is a fixed point, $env_F^M = CF$.

For Case 2, we choose $env_i^M = [cs_i, ns_i, se_i \circ (cs_i) \circ se, os, d, R, CR]$, where $env_i^p = [cs_i, os, ns, se, d, R, CR]$, and $ns(i) = cs_i$, basically we are choosing an environment in which $se$ behaves how we would like. Based on the semantics,

\[
env_i^M = [cs_i, ns_i, (cs_i) \circ se, os, d, R, CR]
\]
Now, the value of $I(\text{confirm } Br\_Cd \rightarrow pre \land ar)(env^M_i) = env^M_i$, and the value of $I(\text{alter all stow}(j))(env^M_i) = [cs_i, ns_i, se, os, d, R, CR] = env^M_i$ where:

$$ns_i(h) = \begin{cases} 
    ns_i(h) & \text{if } h \neq j \\
    cs_i & \text{otherwise}
\end{cases}$$

Now the only difference from $env^P_i$ and $env^M_i$ is in the state index. Therefore, we know the formula $Equal\_except\_at(\{j\}, env^P_i, env^M_i)$. By chaining reasoning, we have $Equal\_except\_at(\{j\}, I(\text{fol_top_lev_code})(env^P_i), I(\text{fol_top_lev_code}env^M_i))$, and therefore $Equal\_except\_at(\{j\}, env^P_i, env^M_i)$. We also have that $env^P_j = env^P_F = CF$.

For Case 3, we know that $env^P_i \neq CF$ and $I(\text{Br\_Cd})(env^P_i)$ and $env^P_j = CF$. In other words, the environment allows us to enter the `whenever` statement, but executing the restricted procedure is what sends us to CF. We let

$$env^M_j = I(\text{confirm } Br\_Cd \rightarrow (pre \land ar))(env^M_i) \text{ and } env^M_j = env^P_j$$

Therefore, $env^M_j = env^P_j$. Now, we first use (yet-to-be-proved) lemma 2.5.8 that all proof rules preserve restriction predicates. Hence, the only way that the $env^P_j = CF$ is if $ar$ or $pre$ were violated. Thus, $env^M_j = CF$.

For Case 4, we know that `cd_suffix` is responsible for the CF, so we use the setup from case 2 as the environments, i.e., $env^M_i = [cs_i, ns_i, se_i \circ (cs_i) \circ se, os, d, R, CR]$. Then $env^M_i = [cs_i, ns_i, (cs_i) \circ se, os, d, R, CR]$. We reuse the same reasoning process as in case 2 and arrive at the fact that $Equal\_except\_at(\{j\}, env^P_j, env^M_j)$, where $env^M_{ew} = I(\text{alter all assume post } \land \text{ae cd\_suffix})(env^M_j)$. Since we know proof

\[\text{\textsuperscript{3}Introduced by Heym, this indicates that only ns differs between these environments, and ns only differs at } j\]
rules preserve restrictions by lemma 2.5.8, we know that the restriction predicate actually held before the **confirm** and, critically, because of the proof rules, we know that \( ae \) (and \( E \)) is already implied by assuming \( post \). So, the assume statement does not change the environment and therefore \( Equal\_except\_at(\{j\}, env^P_{ew}, env^M_{ew}) \). We then obtain \( env^M_F = env^P_F = CF \).

Finally, for Case 5, we use the environments from the last case, except we know that \( env^P_{ew} \) is not abnormal. So, \( Equal\_except\_at(\{j\}, env^P_{ew}, env^M_{ew}) \). And, since \( env^P_F = CF \), we know \( env^M_F = CF \).

We now turn our attention to the lemma-to-be-proved. We first prove a simple, though helpful lemma that allows us to “back up” the environment value over certain statements:

**Lemma 2.5.7.** For any formula \( I \), and any environment \( env \),

if \( stmt \) is an assume, confirm, confirm restriction, end restriction or stow, and \( I(I(stmt)(env)) \)

then \( I(I)(env)) \)

**Proof.** We note that \( env \) is normal, *i.e.*, its value is not in Abnormal-status.

1. **Assume:** Since \( I(assume\ H)(env) = env \), we have the lemma.

2. **Confirm:** Since \( I(confirm\ H)(env) = env \), we have the lemma.

3. **Stow:** Since \( I(stow(i))(env) = env' \), where \( env' = env \) except the \( ns \) component differs, the lemma is true.

4. **Confirm Restriction:** Since \( I(confirm\ restriction\ R)(env) = env \), again we have the lemma.
5. End Restriction: Since \( I(\text{end restriction } R)(env) = env \), yet again we have the lemma.

We note that we may apply this theorem to any sequence of these statements. We now turn our attention to the lemma we are interested in.

**Lemma 2.5.8.** *For any proof rule introduced by Heym that is not the Bridge rule,*

if \( E \) is an restriction predicate and the program satisfies the restriction hypothesis, i.e., no variables involved in it are ever in any non-restricted procedure call,

and for any normal environment \( env \). \( E \) holds at the first statement in the first whenever in the program

then for all normal environments \( env \). \( E \) holds right before the \( cd\_suffix \), or \( cd\_kern \) of all modified/new whenever bodies.

It suffices for us to prove the contrapositive of the inner statement, namely:

**Lemma 2.5.9.** *For any proof rule introduced by Heym that is not the Bridge rule,*

if \( E \) is an restriction predicate and the program satisfies the restriction hypothesis, i.e., no variables involved in it are never in any non-restricted procedure call, and

there is a environment \( env \). \( env \) is normal and \( \neg E \) holds right before the \( cd\_suffix \), or \( cd\_kern \) of some modified/new whenever body

then (there is an environment \( env \). \( env \) is normal and the \( \neg E \) holds at the first statement of the first whenever in the program).

**Proof.** We must prove this for all proof rules. Our proofs will follow a certain structure: we will assume that we have a witness to \( \neg E \) at the specified point in the code, i.e., an environment that, if we run it through the code, gets to the specified point in
the code and $\neg E$ is true. We then use the starting point to remember the environment at specific points in the execution. Due to the semantics of certain operations, we will be able to conclude that $\neg E$ actually had to hold in a prior math environment until we can conclude that it held at a point where we can take the programmatic environment to be the same as the mathematical environment.

- Assume rule: We assume $env_i^M$ is the witness to the assumption and let $env_i^P = env_i^M$. Let

$$env_i^M \overset{\text{def}}{=} \mathcal{I}(\text{prec}_\text{top}_\text{lev}_\text{code\, alter all\, stow}(i)\, ACSeq_0))(env_i^M)$$

and

$$env_i^P \overset{\text{def}}{=} \mathcal{I}(\text{prec}_\text{top}_\text{lev}_\text{code\, alter all\, stow}(i)\, ACSeq_0))(env_i^P)$$

So $env_i^P = env_i^M$. Since $env_i^M$ is a witness to the assumption, we know that it has the property $\mathcal{I}(Br\_Cd)(env_i^M)$. We also know then that

$$env_i^M \overset{\text{def}}{=} \mathcal{I}(\text{assume\, Br\_Cd} \rightarrow H)(env_i^M)$$

has the property $\neg \mathcal{I}(E)(env_j^M)$. By lemma 2.5.7, we know $\neg \mathcal{I}(E)(env_i^M)$. Therefore, $env_i^P$ is our witness.

- Confirm rule: We assume $env_i^M$ is the witness to the assumption, we let $env_i^P = env_i^M$. Let

$$env_i^M \overset{\text{def}}{=} \mathcal{I}(\text{prec}_\text{top}_\text{lev}_\text{code\, alter all\, stow}(i)\, ACSeq_0))(env_i^M)$$

and

$$env_i^P \overset{\text{def}}{=} \mathcal{I}(\text{prec}_\text{top}_\text{lev}_\text{code\, alter all\, stow}(i)\, ACSeq_0))(env_i^P)$$

So $env_i^P = env_i^M$. Since $env_i^M$ is a witness to the assumption, we know that it has the property $\mathcal{I}(Br\_Cd)(env_i^M)$. We also know then that

$$env_j^M \overset{\text{def}}{=} \mathcal{I}(\text{confirm\, Br\_Cd} \rightarrow H)(env_i^M)$$

67
has the property $\neg \mathcal{I}(E)(env^M_j)$. By lemma 2.5.7, we know $\neg \mathcal{I}(E)(env^M_i)$. Therefore, $env^P_I$ is our witness.

• Confirm restriction rule: We assume $env^I_M$ is the witness to the assumption, we let $env^P_I = env^M_I$. Let

$$env^M_i \overset{\text{def}}{=} \mathcal{I}(\text{prec\_top\_lev\_code alter all stow}(i)ACSeq_0))(env^M_I)$$

$$env^P_i \overset{\text{def}}{=} \mathcal{I}(\text{prec\_top\_lev\_code alter all stow}(i)ACSeq_0))(env^P_I)$$

So $env^P_i = env^M_i$. Since $env^M_i$ is a witness to the assumption, we know that it has the property $\mathcal{I}(Br\_Cd)(env^M_i)$. We also know then that

$$env^M_j \overset{\text{def}}{=} \mathcal{I}(\text{confirm E})(env^M_i)$$

has the property $\neg \mathcal{I}(E)(env^M_j)$. By lemma 2.5.7, we know $\neg \mathcal{I}(E)(env^M_i)$. Therefore, $env^P_I$ is our witness.

• End restriction: We assume $env^I_M$ is the witness to the assumption, we let $env^P_I = env^M_i$. Let

$$env^M_i \overset{\text{def}}{=} \mathcal{I}(\text{prec\_top\_lev\_code alter all stow}(i)ACSeq_0))(env^M_I)$$

$$env^P_i \overset{\text{def}}{=} \mathcal{I}(\text{prec\_top\_lev\_code alter all stow}(i)ACSeq_0))(env^P_I)$$

So $env^P_i = env^M_i$. Since $env^M_i$ is a witness to the assumption, we know that it has the property $\mathcal{I}(Br\_Cd)(env^M_i)$. We also know then that

$$env^M_j \overset{\text{def}}{=} \mathcal{I}(\text{assume E})(env^M_i)$$

has the property $\neg \mathcal{I}(E)(env^M_j)$. By lemma 2.5.7, we know $\neg \mathcal{I}(E)(env^M_i)$. Therefore, $env^P_I$ is our witness.
• Procedure call. We assume $env_M^I$ is the witness to the assumption, we let $env_P^I = env_M^I$. Let 

$$env_M^I \triangleq \mathcal{I}(prec\_top\_lev\_code \text{ alter all stow}(i)ACSeq_0)(env_M^I)$$

and 

$$env_P^I \triangleq \mathcal{I}(prec\_top\_lev\_code \text{ alter all stow}(i)ACSeq_0)(env_P^I)$$

So $env_P^I = env_M^I$. Since $env_M^I$ is a witness to the assumption, we know that 

$$env_c^M = \mathcal{I}(\text{confirm}(Br\_Cd) \rightarrow (pre \land ar)[x^j \leftarrow x^j_i] \text{ alter all stow}(j))(env_M^I)$$

has the property $\mathcal{I}(Br\_Cd)(env_c^M)$. We also know then that 

$$env_j^M \triangleq \mathcal{I}(\text{assume} \ (pos \land ar)[x^k \leftarrow x^k_i])(env_c^M)$$

also has the property $\neg \mathcal{I}(E[x^k \leftarrow x^k_j])(env_j^M)$. By lemma 2.5.7, we know $\neg \mathcal{I}(E[x^k \leftarrow x^k_j])(env_j^M)$. Moreover, if we use the fact that we have a restriction, then either the procedure does not change those variables and $x^k_i = x^k_j$. We have $\neg \mathcal{I}(E)(env_i^P)$. We then use $env_P^I$ as our witness in the program and obtain that $\neg \mathcal{I}(E)(env_P^I)$. Therefore, $env_P^I$ is our witness.

• Selection with an else rule: Here we have three possible places where the restriction predicate may not hold. We first reduce the number of cases by one. Assuming that the place where the negation of the restriction holds is the last whenever, we know that $Br\_Cd$ is true, therefore exactly one of the assumes adds a fact. Therefore, either $x_n = x_k$ or $x_n = x_m$. Since $cd\_kern_i$ have not been processed, we know that the restrictions are invariants of the inside blocks of the whenever bodies. We perform a rewrite of the invariant variables to un-indexed within the body and determine that exactly one of the other whenever statements has the negation of the restriction predicate as well.
Without loss of generality, we examine the first `whenever` as the witness; the other case is analogous. We assume $env^M_I$ is the witness to the assumption, we let $env^P_I = env^M_I$. Let

$$env^M_I \triangleq \mathcal{I}(\text{prec_top_lev_code alter all stow}(i)ACSeq_0)(env^M_I)$$

and

$$env^P_I \triangleq \mathcal{I}(\text{prec_top_lev_code alter all stow}(i)ACSeq_0)(env^P_I)$$

We know, by assumption, that, if

$$env^M_j = \mathcal{I}(\text{alter all stow}(j) \text{ assume } x_j = x_k)(env^M_i)$$

then $\neg \mathcal{I}(E)(env^M_j)$. Since we have the `assume` clause, we actually also know that $\neg \mathcal{I}(E)(env^M_i)$. Therefore $env^P_I$ is our witness.

- Selection without else rule: We leave it as a special case of the previous rule.

- Loop while rule: While it is possible for either of the `whenever` statements to contribute an environment to as a witness, we first argue that they are one and the same witness. First, because it is a restriction predicate, the value of $b_p_e$ has no effect on the restriction predicate. Therefore, the loop invariant is the only connection that would contribute to the witness. We then chose one of the two potential witnesses, namely the first one. We assume $env^M_I$ is the witness to the assumption, and let $env^P_I = env^M_I$. Let

$$env^M_i \triangleq \mathcal{I}(\text{prec_top_lev_code alter all stow}(i)ACSeq_0)(env^M_i)$$

and

$$env^P_i \triangleq \mathcal{I}(\text{prec_top_lev_code alter all stow}(i)ACSeq_0)(env^P_i)$$

So $env^P_i = env^M_i$. We now use the fact that `alter all` statements do not change the values of any current restriction predicate. Therefore, we have that $\neg \mathcal{I}(E(env^M_i))$, and that $env^P_i$ is the witness.
• Restricted procedure call rule: We assume $env^I_M$ is the witness to the assumption, we let $env^P_I = env^I_M$. Let

$$env^M_i \equiv I(prec\_top\_lev\_code\ alter\ all\ stow(i)ACSeq_0)(env^M_i)$$

and

$$env^P_i \equiv I(prec\_top\_lev\_code\ alter\ all\ stow(i)ACSeq_0)(env^P_i)$$

So $env^P_i = env^M_i$. Since $env^M_i$ is a witness to the assumption, we know that

$$env^M_c = I(confirm(Br\_Cd) \rightarrow (pre \land ar)[x^j \leftarrow x^j_i] alter\ all\ stow(j))(env^M_i)$$

has the property $I(Br\_Cd)(env^M_c)$. We also know then that

$$env^M_j \equiv I(assume\ (pos \land ar)[x^k \leftarrow x^k_i])(env^M_c)$$

also has the property $\neg I(E[x^k \leftarrow x^k_j])(env^M_j)$. By lemma 2.5.7, we know

$$\neg I(E[x^k \leftarrow x^k_j])(env^M_i)$$

Moreover, if we use the fact that we have a restriction, then either the procedure does not change those variables (and $x^k_i = x^k_j$) or the variable are changed, but the invariant could not have held before the confirm (as it is an invariant). In either case, we have $\neg I(E)(env^M_i)$. We then use $env^P_i$ as our witness in the program and obtain that $\neg I(E)(env^P_i)$. Therefore, $env^P_i$ is our witness.

Since every proof rule has this property of preserving the negation for the predicate in the program direction, we have our original lemma. ■

Our next goal is relative completeness of the Restricted procedure call rule.

**Lemma 2.5.10.** The Restricted procedure call rule preserves invalidity in the programmatic direction.
Proof. The proof uses the same arguments as the corresponding procedure call rule proof from Heym [31] augmented with the fact that \texttt{alter} all statements do not change the state of the restriction predicate values. ■

We now turn our attention to the restriction declaration soundness and relative completeness proofs.

**Lemma 2.5.11.** The Restricted declaration rule is sound and relatively complete.

Proof. We first prove soundness by assuming that there is some environment \( env \) that, upon running the program, enters the CF state. We argue that, based on the semantics, the only way that this can happen is if some translated procedure declaration has the following properties:

1. \( env.d(e_r_i).pf = env.d(ProcName(e_r_i)).pf \)
2. \( env.d(e_r_i).sf = env.d(ProcName(e_r_i)).sf \)
3. \( env.d(e_r_i).dp \) is not a superset of \( ar \)
4. \( env.d(e_r_i).ep \) is not a superset of \( ae \)
5. \( env.d(e_r_i) \) is not conformal

where \( ar \) is the entire precondition, and \( ae \) is the entire postcondition. The only way that these can occur is if \( env.d(e_r_i) \) is not conformal and its domain and effect predicates are correct with respect to their specification. We also know that \( ProcName(e_r_i) \) has already been processed and it is conformal, so the procedure and status functions associated with \( ProcName(e_r_i) \) match its specification. Therefore, the only way that \( env.d(e_r_i) \) is not conformal is if there is some input
that matches the domain predicate and either the status function returns CF or the status function returns NL and the effect predicate does not hold of the outputs. The first case cannot happen as the domain predicate of the restriction is a subset of the domain predicate of \( \text{ProcName}(e_{r_i}) \). Therefore, some value is returned that does not match the specification. We then choose our mathematical environment \( env_i^M \) to have these values. Therefore the mathematical environment is invalid also.

For completeness, we simply invert the argument: if the given program is invalid, and \( env_i^M \) is its witness (i.e., interpreting the mathematical program gives CF), then any environment will enter CF when processing the corresponding procedure declaration, so we set \( env_i^P = env_i^M \). Therefore, \( env_i^P \), when interpreting the restricted procedure declaration, gives CF. ■

<table>
<thead>
<tr>
<th>multiplier</th>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>whenever</td>
</tr>
<tr>
<td>1</td>
<td>alter all</td>
</tr>
<tr>
<td>1</td>
<td>stow</td>
</tr>
<tr>
<td>1</td>
<td>assume at top level</td>
</tr>
<tr>
<td>1</td>
<td>confirm at top level</td>
</tr>
<tr>
<td>2</td>
<td>assume inside whenever</td>
</tr>
<tr>
<td>2</td>
<td>confirm inside whenever</td>
</tr>
<tr>
<td>5</td>
<td>procedure call</td>
</tr>
<tr>
<td>11</td>
<td>loop</td>
</tr>
<tr>
<td>12</td>
<td>if</td>
</tr>
</tbody>
</table>

Our last task is to modify the proof of rules termination to take these new rules into account. We take Heym’s \( \text{Meas} \) function, as tabulated in Table 2.1 and extend it
to take care of the new constructs. We define the measure of a confirm restriction and an end restriction as 3, and a restricted procedure call as 5. If we examine either the confirm restriction or end restriction rule, \(\text{Meas}(M) = \text{Meas}(P) - 3 + 2\) therefore, \(\text{Meas}(M) < \text{Meas}(P)\). Finally, if we examine the restricted procedure call rule. \(\text{Meas}(M) = \text{Meas}(P) - 5 + 2 + 2\) and therefore \(\text{Meas}(M) < \text{Meas}(P)\).

While the procedure declarations are not handled by this metric, the Restriction declaration rule completely processes the declaration, so it will not cause a cycle in the proof rules.

### 2.5.7 Evaluation

Based on the discussion of Figures 2.13 and 2.6, we see that there is a cost to using restrictions, namely the proliferation of new contracts. Even though there is this cost, we will examine the benefits and show that the use does actually simplify proof of correctness and also has value simply because of the re-usability of restriction contracts.

Appropriate use of restrictions may also help simplify proofs of VCs by making the VCs easier to prove. We examine the impact of restrictions on the difficulty of VCs as defined by [36]. In that work, VCs are categorized according to the number of hypotheses (\(H_0, H_1, \ldots\)) and whether only logical rules (L), theory-specific knowledge (M) or local mathematical definitions (D) are needed to prove a VC. VCs that use fewer assumptions or require less mathematical knowledge are considered less difficult. The metrics are summarized in Figure 2.32.

The code presented in Section 2.3 without restrictions was compared to the code in this section with restrictions. The original quicksort implementation’s most difficult
## What is needed in the proof

<table>
<thead>
<tr>
<th>Label</th>
<th>What is needed in the proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Rules of mathematical logic</td>
</tr>
<tr>
<td>$H_n$</td>
<td>At most $n$ hypotheses from the VC ($n &gt; 0$)</td>
</tr>
<tr>
<td>M</td>
<td>Knowledge of mathematical theories used in the specifications</td>
</tr>
<tr>
<td>D</td>
<td>Knowledge of programmer-supplied definitions based on mathematical theories above</td>
</tr>
</tbody>
</table>

(a) VC Classification

![Diagram of VC Classification]

(b) Lattice of the VC Classification

Figure 2.32: VC Classification and Diagram of Category Relationships (Adapted From [36])

VC was categorized as $MH_6$, while the restrictions version has VCs of difficulty at most $MH_3$. The $MH_6$ VC is particularly difficult; it arises from proving the second conjunct in the `ensures` clause of `Sort`:

The proof requires hypotheses 3 through 8, and is fairly involved; mathematical lemmas are needed, for instance, to conclude that hypotheses 3, 5 and 6 imply $IS\_PRECEDING(q_6, \langle partitionElement_4 \rangle)$. The corresponding VCs from restrictions are easier. The direct analog of the above VC, in particular, is in category $LH_1$, 

75
$1: \text{is\_initial}(\text{partitionElement}_2)$

$2: \wedge \text{IS\_PERMUTATION}(q_5 \circ q_{Big5}, q_4)$

$3: \wedge \text{IS\_PRECEEDING}(q_5, \langle \text{partitionElement}_4 \rangle)$

$4: \wedge \text{IS\_PRECEEDING}(\langle \text{partitionElement}_4 \rangle, q_{Big5})$

$5: \wedge \text{IS\_PERMUTATION}(q_6, q_5)$

$6: \wedge \text{IS\_NON\_DECREASING}(q_6)$

$7: \wedge \text{IS\_PERMUTATION}(q_{Big7}, q_{Big5})$

$8: \wedge \text{IS\_NON\_DECREASING}(q_{Big7})$

$9: \wedge \text{is\_initial}(\text{partitionElement}_8)$

$10: \wedge (\text{partitionElement}_4) \circ q_4 \neq \Lambda$

$\rightarrow \text{IS\_NON\_DECREASING}(q_6 \circ (\text{partitionElement}_4) \circ q_{Big7})$

\[i.e., \text{the goal is one of the hypotheses. The proof of a VC arising from the call to }\]
\text{Concatenate} \text{ in the body of Sort} \text{ is the most difficult: it is in the category MH}_3.

\text{Another VC in MH}_3 \text{ arises from the also requires clause for Concatenate:}

$1: \text{IS\_PRECEEDING}(q_{1\text{original}}, q_{2\text{original}})$

$2: \wedge \text{IS\_NON\_DECREASING}(q_{1\text{original}})$

$3: \wedge \text{IS\_NON\_DECREASING}(q_{2\text{original}})$

$\rightarrow \text{IS\_NON\_DECREASING}(q_{1\text{original}} \circ q_{2\text{original}})$

\text{The one-time, reusable proof of this VC is also in MH}_3. \text{ However, it is a relatively easy proof to discharge; it is an algebraic lemma of string theory. This is the essence of the proof of the original VC. Proving these VCs with Isabelle [51] using a version of RESOLVE’s string theory in an automatic mode [35] confirms that the MH}_6 \text{ VC is hard to prove—Isabelle does not prove it automatically. The two MH}_3 \text{ VCs are proved automatically. For this example, restrictions are able to simplify the code annotations and reduce the maximum difficulty of VCs generated from the resulting code.}

\text{We expect this empirical result to generalize; restrictions have the effect of adding}

“way-points” \text{ in the proofs of the VCs from code using restrictions. These way-points}
are created from input from the programmer; the requires and ensures clause are both modified to preserve the requisite invariant, thus ensuring that the way-point is useful for the justification of correctness. Moreover, the VCs generated from the declaration of a restriction should be syntactically simple with few assumptions and highly targeted—excellent candidates for proofs from general, reusable theorems.

We also expect that many restrictions will be reusable. For example, OrderedQueue may be used for any sorting algorithm implementation or client. Restrictions presented in this paper could be generalized to be usable in selection problems, e.g., via a predicate parameter to SmallValueQueue. Even if we assume that restrictions turn out to not be reusable, there is still value in using them; restrictions document the reasoning behind why a particular block of code is correct, and, as such, aid readability by humans.

2.6 Related Work

The idea of restrictions is similar to a core idea expressed in predicate subtypes, dependent types, refinement types and contract types [59, 37] (PDRCT). Depending on the exact setup of PDRCT used, proof obligations may be generated (such as Type Correctness Conditions (TCCs) in PVS) when converting from a type to a predicate subtype. In other setups the type checking system can infer many of the requisite properties. These ideas have been applied both to mathematical and programmatic domains. In any case, a restriction is different in that it entails modifying pre/post-conditions of operations to maintain the user-supplied invariant. Moreover, no new executable code need be emitted as a result of a restriction, which documents invariants and simplifies proofs of resulting VCs rather than defining a new type;
a restriction is *not* a new type. However, the type inference and other algorithms used in contract and refinement types are largely absent; these could be added in the future using some of the existing work to alleviate some of the annotation burden on programmers.

The Jahob system [71] uses annotated Java source code as its source language. The annotation language has support for a proof language, with essentially full first-order prover functionality. There are first order proof commands, such as applying modus ponens, along with commands to perform local proofs. Invariants can be expressed as well. While Jahob’s proof language is powerful, the proof commands are not natural for a software professional. Rather than learning a proof system, software professionals using restrictions think in terms of contracts and component invariants, concepts that are used in the normal course of programming.

Behavioral subtyping [46] uses a set of rules to ensure that a subtype can always be used in place of a supertype without violating a behavioral property of the client program. Contractually, the preconditions of any subtype operation may not be strengthened, postconditions may not be weakened, and invariants must be preserved. Restrictions impose different requirements; in particular, preconditions may be strengthened. The goal of restrictions is not to allow for substitution, but rather to indicate that during specific code segments (i.e., not necessarily for the entire lifetimes of variables) stronger abstract invariants hold for specific variables.

Object invariants [5] are defined over the *concrete* representation of the object; they denote consistency or other properties that relate specific fields or ownership of a particular field or object. Restrictions instead are over the *abstract* state of the objects, their cover story as represented in mathematics, rather than over any
particular representation of the object’s abstract state space. This feature ensures that restrictions can be used with any correct implementation of their underlying type, making them more reusable.

2.7 Future Work

While the restriction constructs have formal semantics, its semantics is based on top of already defined semantics. As was mentioned before, the semantics of RESOLVE could be reworked to handle both relational semantics and a semantics of restrictions without source-to-source translations. This would provide some additional value by natively supporting restrictions; the current semantics in this dissertation provides a framework to ensure that the semantics are implemented faithfully.

Also, the restrictions currently only handle a definite number of arguments; one would need to have a two \texttt{Set} version of a \texttt{AreDisjoint} restriction as well as a three \texttt{Set} version, along with every other number of \texttt{Sets} desired. Moreover, restrictions do not exploit commutative properties of the restriction to simplify the required annotations for each procedure, \textit{e.g.}, each \texttt{Set} in each \texttt{AreDisjoint} formal argument slot would require a separate contract. We envision adding additional annotation to handle both of these issues, the first would be handled by a \texttt{Reduce} keyword to apply a binary mathematical operator to a string of variable names, the second by \texttt{Property} annotations with rewrite rules for restriction names, \textit{e.g.}, \texttt{AreDisjoint(s, t)} = \texttt{AreDisjoint(t, s)}.

Also, one drawback of using RESOLVE as a language is that there are few examples of source code to which any new constructs can be applied. Thus more examples of code would allow for a more in-depth evaluation of the effectiveness of restrictions.
Along those same line, investigations into more representative and finer grained metrics for the difficulty of VCs are also needed for a more effective comparison.

2.8 Summary and Conclusion

We have presented the formal syntax and semantics of a programming language construct, restrictions, that helps address a limitation in current verification languages, namely the clumsiness of formally documenting client code, especially with loops. This construct, when applied to code similar to that shown in Section 2.4, provides a mechanism to separate out two uses of loop invariants, namely an abstraction of the behavior of a loop and a mechanism to maintain abstract invariants on variables. This approach not only can simplify VCs generated in client code, but also can result in reasoning reuse. This reuse happens both when restrictions are reused across clients, and even when there are multiple calls to a single restriction operation by a particular client. Since this construct helps with proofs of VCs, it validates the first part of the three-part thesis statement.
Chapter 3: Use of Interactive Proof Assistant in Automated Verification

This chapter discusses the efficacy of Isabelle as a back-end prover for the automated proofs of VCs. The chapter begins with a discussion of the requirements of an ideal back-end prover in Section 3.1. It then continues with discussions of the types of automated provers in Section 3.2, a discussion of the embedding of RESOLVE theories in Isabelle and a comparison of Isabelle with another back-end prover on VCs from the current catalog of RESOLVE components and extensions in Section 3.4, and finally concludes with related work in Section 3.5 and conclusions in Section 3.7.

3.1 Requirements of a Back-end Prover

Back-end provers in the context of a verifying compiler have specific requirements that differ from provers in other applications. For example, the automatic inference of induction hypotheses is usually not needed for VCs from RESOLVE code while string and finite set theory are especially important for these VCs. We now endeavor to describe requirements for an ideal (though realistic) back-end prover. A back-end prover should:
1. be sound and preferably produce proof certificates of every proof; \textit{i.e.}, it may not prove results that are false, and it should be able to generate machine-checkable proof certificates corresponding to any proofs the prover constructs,

2. be fast; whatever reasoning process it employs must be able to determine quickly if a VC generated from typical software components is true or false,

3. be able prove “obvious VCs,” \textit{i.e.}, VCs for which human readers can quickly construct a \textit{completely correct} proof,

4. be able to construct counterexamples to invalid VCs,

5. be readily extensible with additional mathematical theories and lemmas for use in its internal proof process without undue configuration or coding burden.

We expound on these requirements in order to justify them. The first requirement, namely that the prover is sound, is necessary to ensure that VCs are only proved if they are valid. We do not want to prove components correct that really are not; this would be worse than no verification. A weaker version, in which there is just a proof certificate for each claimed proof, allows the prover to be \textit{potentially} unsound, but its inferences can, in principle, be checked by a simpler proof checker.

The requirement that the prover is fast comes from the potential for a large number of VCs generated from moderately sized operation bodies and the interactive nature of compilation; programmers simply cannot wait an excessively long time to determine if their code is correct. Literature from programming experiences in the 1970s where long compilation times are discussed as detriments to high-quality software, such as Brooks’ \textit{Mythical Man-Month} \cite{9}, gives evidence that long waiting periods for compilation are not acceptable.
Provers must be able to prove “obvious” VCs; if the back-end prover cannot prove VCs that programmers can readily convince themselves of (with a fully correct proof), then the system does not inspire confidence. Moreover, its utility is questionable if few realistic and obvious programs are verifiable.

Counterexamples to invalid VCs are useful in aiding software engineers in debugging the buggy code. Adcock [2] describes the potential debugging process using these counterexamples that avoids the use of test cases as a tool for debugging.

Finally, the last requirement covers the unavoidable instances in which new mathematical theories or lemmas are needed in either the specification of software components or proofs of VCs. While ideally this would not be necessary, experience suggests that no fixed set of theories with associated theorems is enough, so our methodology must be able to account for this possibility. This requirement is potentially too strong; it is possible, though we believe unlikely, that at some point the mathematical theories developed will be enough to specify all software that people would like built. We leave this requirement as-is to ensure that we identify this as a desirable feature until there is evidence that it is not needed.

3.2 Categories of Automated Provers

Automated provers fall into several categories. There are satisfiability modulo theory (SMT) solvers, first order theorem provers, interactive proof assistants, and special-purpose decision procedure tools. Each of these has merits, relative to automated VC theorem proving, along with potential limitations. While the ultimate back-end prover has not yet been found, we are in a position to present, based on the
current literature, a qualitative discussion of the relative merits of each of the types of tools.

### 3.2.1 SMT Solvers

In order to examine SMT solvers, we first briefly review satisfiability (SAT) solvers [11]. SAT solvers determine the satisfiability of propositional formulas. For example, a SAT solver would return “sat” with a model of \( P \leftarrow \top \) and \( Q \leftarrow \top \) if given \( P \land Q \). The problem of deciding whether a formula is satisfiable is hard, assuming \( P \neq NP \), but it is decidable and, in practice for “typical” formulas, quite fast.

SMT solvers extend this idea by treating every quantifier-free first order logic formula propositionally, \( e.g. \), by treating each atomic formula (\( e.g. \), \( x + y = 45 \)) as a proposition. This propositional instance is conceptually given to a SAT solver and any potentially satisfying assignments to the atomic formulas are then fed to a decision procedure for the underlying theories. If the assignment is possible in that theory, then it is a model for the satisfiability of the original formula; otherwise the tool asks for another satisfying assignment from the SAT solver. This process is continued until no satisfying assignment remains or one is found. As an example, consider the original example of \( P \land Q \) where \( P \equiv i < 0 \) and \( Q \equiv i > 0 \). While the SAT solver’s only possible satisfying assignment is \( P \leftarrow \top \) and \( Q \leftarrow \top \), because of the definition of \( P \) and \( Q \), this potentially satisfying assignment cannot happen, so the formula is “unsat.”

While this is how an SMT solver works conceptually, there are many extensions to this process. First, the SMT solver may use different decision procedures for multiple theories if the theories “interact” in only rudimentary ways, through the
Nelson-Oppen [50] procedure or through other, more sophisticated methods. Secondly, quantifiers may be allowed, but either completeness or soundness may be compromised [45]. Finally, some theories may not have a decision procedure. These theories may still be used, but soundness or completeness must be sacrificed.

We use Z3 [12] as the exemplar SMT solver for the purposes of this comparison. Others include Yices [13], and Nitpick [6] (which is in Isabelle). Z3 is able to generate a proof certificate, is reasonably fast at finding witnesses, and is extensible by “pushing” new axioms onto the context. Its effectiveness at finding counterexamples to VCs is highly dependent on their encoding, e.g., sets are encoded as their characteristic function. Particular encodings make some results easy to derive, while obfuscating others. Z3’s “built-in” types are relatively few and are treated differently than user-defined types or functions. The theory of uninterpreted functions is used to generate counterexamples for any user-defined functions. The counterexamples may not make sense in a given context due to incomplete axiomatizations; they only satisfy the given axioms. Moreover, the extensibility of Z3 is not as powerful as other tools since it is restricted to first order logic; interactive proof assistants (discussed later) allow for more powerful abstractions. The input format is not designed for readability not expressibility. Finally, Z3 includes the ability to generate proof certificates.

### 3.2.2 First Order Theorem Provers

First order theorem provers, such as Vampire [58], SPASS [68], and E [60] are designed to prove only first order formulas. Many of the techniques are designed to “saturate” the derivable facts from both the assumptions and the negation of the goal, in an attempt to find a contradiction. Some of these tools, such as E, have
excellent support for equality and/or equivalence relations. While these tools are designed to be automated, they are as restrictive as SMT solvers in terms of their inputs. Extensibility is achieved by adding new axioms, similar to a SMT solver. These provers are not designed to find counterexamples, nor are their input formats especially human-friendly or expressive. Any higher-order features must be “pushed” into first order logic.

3.2.3 Interactive Proof Assistants

Proof assistants focus on the problem of formalizing portions of mathematics for human consumption. As such, they are interactive and use very expressive underlying logics such as HOL [30] (Higher Order Logic). The main difference between first order logic and HOL is the ability to quantify not only over the universe of values, but the universe of objects, functions and predicates in a manner subject to technical limitations that avoid logical inconsistencies.

Examples of these types of provers include Isabelle [51], PVS [53], HOL light [30] and Coq [1]. For our discussion, we will use Isabelle as an exemplar. Isabelle [51] is an interactive proof assistant that supports both automated and interactive proof methods and combinations of the two. Isabelle is implemented using a simple typed logic (called Pure) that can be used to define other axioms systems and syntax. The Isabelle tool includes, but is not limited to, embeddings of Higher Order Logic [51], First Order Logic, and ZF Set Theory [54]; Isabelle also allows for definition of new syntax. Isabelle also has support for using automated first order logic theorem provers to prove any theorems or lemmas as desired by using the sledgehammer command.

4 We are commenting on the relative human-readability; ugly syntax seems to be a common theme among all theorem provers.
The combination of the ability to prove a VC automatically, to find where the prover has problems in VC proofs, and to add additional lemmas easily makes Isabelle an extremely useful tool for the exploration of the issues relating to the automatic proofs of VCs.

Also, Isabelle is first and foremost designed to model phenomena and prove the correctness of theorems with the help of a user. As such, it includes features that are useful for mathematicians and the formal methods community for expressing desired properties; relative to other tools, its input is both human readable and extensible. For example, Isabelle includes the ability to make local assumptions, prove properties about those assumptions and later prove specific types satisfy those assumptions. This aids in reusing theories while also meshing with the RESOLVE theory development style. These features enable users to apply theorems that have been verified by Isabelle in the proofs of VCs. It is, of course, not a good idea to allow a programmer to simply assert the truth of an arbitrary “lemma” without proof.

In terms of the categories presented in Section 3.1, Isabelle may produce proof certificates (and has a small inference core that is relatively easy to analyze for soundness), and has excellent support for extensible theories given with or without proof. Isabelle’s weaknesses are that it may be slower than the other tools, it is unclear that obvious VCs are always proved, and it was not originally designed to give counterexamples, though other internal commands, such as quickcheck may be able to.

### 3.2.4 Special-purpose Decision Procedures

Special-purpose decision procedures are, by definition, specialized. However, since they are decision procedures, they are typically implemented using data-structures
appropriate for speed. Assumptions about the types of formulas received, such as the underlying theory, occurrence of quantifiers, etc., also allow for performance increases. Soundness is usually not as easily justified, though proof certificates may be generated. A specialized decision procedure will prove obvious and non-obvious VCs within its domain and guarantees little outside of that domain. The decision procedure may be able to give back counterexamples, though it usually is not readily extensible to handle other theories. For example, PVS [53] includes many decision procedures. Each must be invoked by a different command, and they are not used together directly. SplitDecision is a decision procedure tool developed by Adcock [2], based on a decision procedure for a fragment of string theory invented by Friedman [21]. It does include rules for processing some other theories besides just strings, though no guarantees are provided.

### 3.2.5 Rationale for Examining Isabelle

As a result of the qualitative analysis of each of these different types of automated (or semi-automated) theorem provers based on published information and anecdotal experience, it is clear none of them is a silver bullet. Moreover, VCs resulting from programs may be entirely different than the types of formulas the provers are designed to handle, so their performance on benchmarks chosen by their developers may not be indicative of the tool performance on VCs.

For the verifying compiler project, the goal is to identify and choose, or build the best tool for the job. In this vein, there are multiple threads of research into each of these different types of systems. We choose to examine Isabelle because the interactive nature of the proof process allows us to gain a better understanding of
proof process pitfalls as well as allows us to interact with several first order theorem provers. It also allows us to experiment with our vision of software development; Isabelle allows us to fill both the mathematician and software engineer roles. Since VCs resulting from typical software components tend to be mathematically simple, our thesis is that Isabelle is an effective back-end prover for these types of VCs.

We now describe the setup of RESOLVE mathematical theories.

3.3 RESOLVE Mathematical Theories

As described by Edwards et al. [14], RESOLVE uses a model-based specification approach. The models used are described by the RESOLVE mathematical theory units, which include both descriptions of the theory including definitions, theorems, and lemmas as well as proofs of the theorems and lemmas. So far, string, tuple, finite set, and integer theory have been developed, among others. This sub-section describes an exemplar theory, the theory of strings, that illustrates the use of theories in specifications.

String theory can be defined, as in Figure 3.1, through two functions: the empty string ($\Lambda$), and the extension function (ext($s, x$)) which adds a new element to a given string to produce a new string. The axioms of string theory state the properties of strings that one would consider intuitive: the empty string is not the result of a string extension by any element, two extended strings that are equal implies the components are equal, and an inductive axiom for strings. Other functions such as length, $\circ$, and reverse may be defined inductively. Some lemmas, as shown in Figure 3.1, involve algebraic properties of the functions. For example, $\circ$ is associative. Also, the relationships between the operators are captured by these lemmas.
Type Signature 1 (String).

string \overset{\text{def}}{=} \text{string(obj)}

\Lambda : \text{string}

ext : \text{string} \times \text{obj} \rightarrow \text{string}

String Axioms 1.

1. \text{ext}(s, x) \neq \Lambda

2. \text{ext}(s_1, x_1) = \text{ext}(s_2, x_2) \Rightarrow s_1 = s_2 \land x_1 = x_2

3. \forall S \in \mathcal{P} (\text{string}) : (\Lambda \in S \land \forall x, s : (s \in S \Rightarrow \text{ext}(s, x) \in S)) \Rightarrow S = \text{string}

Function Definitions 1.

1. \langle \_ \rangle : \text{obj} \rightarrow \text{string} \overset{\text{def}}{=} (x) = \text{ext}(\Lambda, x)

2. |\_| : \text{string} \rightarrow \mathbb{N} \overset{\text{def}}{=} |\Lambda| = 0 \land |\text{ext}(s, x)| = |s| + 1

3. \_ \circ \_ : \text{string} \times \text{string} \rightarrow \text{string} \overset{\text{def}}{=} s \circ \Lambda = s \land s_1 \circ \text{ext}(s_2, x) = \text{ext}(s_1 \circ s_2, x)

4. reverse : \text{string} \rightarrow \text{string} \overset{\text{def}}{=} \text{reverse}(\Lambda) = \Lambda \land \text{reverse}(\text{ext}(s, x)) = \langle x \rangle \circ \text{reverse}(s)

Useful Lemmas 1.

1. lemma EmptyNotSingle: \Lambda \neq \langle x \rangle

2. lemma IdofEmpty : \Lambda \circ \alpha = \alpha

3. lemma LenofSingle : |\langle x \rangle| = 1

4. lemma LenofCat : |\alpha \circ \beta| = |\alpha| + |\beta|

5. lemma AssocCat : \alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma

6. lemma ReverseofReverse : \text{reverse}(\text{reverse}(\alpha)) = \alpha

7. lemma ReverseofCat : \text{reverse}(\alpha \circ \beta) = \text{reverse}(\beta) \circ \text{reverse}(\alpha)

8. lemma LenofReverse: |\text{reverse}(\alpha)| = |\alpha|

Figure 3.1: String Theory
We now turn our attention to the problem of importing those mathematical theories into Isabelle; one of the contributions of this chapter.

3.4 Embedding RESOLVE theories within Isabelle

In this section we discuss the embedding of RESOLVE theories within Isabelle. Isabelle provides many ways of importing new theories. Theories can be added either via a direct axiomatization, built on-top of already defined theories in Isabelle, or defined in terms of “locales” or ”classes.” Direct axiomatization is the simplest method to import theories. A user simply states some type exists and then proceeds to give axioms about that type. Unfortunately axiomatization is dangerous because the axioms cannot be checked by the tool for consistency. Thus one could prove many results in an inconsistent theory that has contaminated the logical framework. Neither typos nor bugs in the theory can be found directly using this method, though one can attempt to prove “False” as a rudimentary check of consistency. Moreover, the inconsistency is not local, in that it may impact other theories added later.

One can also build new types and theories on top of existing theories in Isabelle. A common way to do this is to define a new type from a predicate on existing types. Then new theorems can be proved about this new type. Inconsistency is now less of a concern as presumably the existing theory has already been proved correct. However, we have another potential pitfall: the proofs must “unfold” the definition of the new type. Therefore, we might prove facts about the underlying type rather than the type we were actually interested in—we become tied to the witness to the theory rather than the theory’s abstract properties.
“Locales” and/or “classes” allow us to parameterize our theories by constants, along with local assumptions about the constants, and prove theorems relative to the constants and assumptions. In order to use the theory, we would need to provide a witness, i.e., prove that the type we are interested in, along with given constants, satisfies the assumptions. We may still create an inconsistent theory like we could in the axiomatization. However, we would be unable to find a type and constants that satisfy the assumptions; the potential inconsistency is local and does not pollute all of Isabelle unlike a direct axiomatization.

Given this characterization, the locales/classes method seems to be the best. However, this is our first instance of hitting a tool limitation. Locales/classes do not support multiple type parameters. One such example is a string of arbitrary kinds of items, in which, contrary to intuition, there are two type parameters: strings of items and items. This limitation makes these “locales” or “classes” a non-starter for embedding RESOLVE finite sets, and strings. Given this constraint, we use the following methodology: we implement our types in terms of existing types, but we insist on using the underlying type representation to prove only the assumptions of the RESOLVE theory and then we require that the underlying type is never used after the axioms are proved. In other words, we prove the axioms of whatever new theory we are embedding, say string theory, then once that is done, the proofs never unfold the type definitions but rather rely on the new theory’s axioms and definitions instead of those in the witness.
3.4.1 String Theory Encoding

We now describe the boot-strap process of importing a RESOLVE theory into Isabelle using string theory as an example. The main points along the way will be the definition of the new type, any initial constants and axioms.

We start the embedding of RESOLVE’s string theory by first defining a new type ‘a string. This new type must be represented by something, we choose the obvious, Isabelle ‘a list type, and we allow for every ‘a list to represent one string. This is done by the command typedef ‘a string = "{x::‘a list. True}" by auto. The last part requires some explanation. Isabelle’s underlying logic assumes that all types are non-empty. Thus, anytime a new type is defined, a proof obligation is generated to make sure something is in the type. Here it is obvious since ‘a list is a non-empty type and Isabelle proves it using the auto proof method.

The next piece of infrastructure is a set of lemmas that relate the underlying type, in this case ‘a list, to the new type, in this case ‘a string. With the type declaration, Isabelle automatically defines functions called Abs_string that is a function from ‘a list to ‘a string and Rep_string that is a function from ‘a string to ‘a list. Properties already known include that Rep_string is 1-to-1. In our case, we strengthen these results, namely we also know that Abs_string is 1-to-1, and that Rep_string and Abs_string are inverses.

RESOLVE string theory is built on two fundamental constants, an empty string and an extension function. The representations for each of these

5The ‘a must be explained, it indicates an arbitrary type; readers familiar with ML [49] will recognize this notation.
constants is obvious, we use the corresponding ’a list constant. Formally, we define the empty string by \textbf{definition empty_string :: ”’a string” where ”empty_string == Abs_string []}^6 and the extension function as \textbf{definition ext :: ”’a string => ’a => ’a string” where ”ext a x == Abs_string (x # (Rep_string a))”}^7 Once we have these constants, we now must perform the last “representation” task; we must prove the axioms of string theory where we are allowed to use the representation. The axioms are defined from Figure 3.1; the proofs of axioms 1 and 2 are simple and consist of burrowing into the representation as ’a list and reasoning within that context. We must break the proof of axiom 3 into 2 parts, first by proving the result in the ’a list context, then lifting it to the level of ’a string.

We now run into the second modification of string theory because of tool limitations; rather than defining length using strings, we define the length function using the representation. If we did not, the recursive function definition infrastructure within Isabelle will not work because we would be unable to show that the recursive function definitions terminate (to define \textit{length}, we need a measure). This does compromise our criteria slightly; the RESOLVE string theory defines the length function by induction on the string. We make this one exception because the alternative is to be unable to define recursive functions; we would be unable to develop the theory. Thus we define the \textit{len} function from ’a string to nat as the size of the underlying ’a list representation. Our VCs will be expecting a function into the integers, since that is how the RESOLVE theory defines it, so we also define the \textit{length} function

^6 [] is the empty list.

^7 # is the extension function for Isabelle lists.
from 'a string to int using a built-in conversion function from nat to int. Our last axiom is over the len function, and it characterizes the function recursively, namely len empty_string = 0 & len (ext a x) = 1 + len a and is proved using a combination of axiom 3 and the definitions of all of the constants.

Now, we are finally ready to step back from the witness to RESOLVE string theory and reason using only the given axioms (namely 1-4). We now add some helper lemmas to aid Isabelle is reasoning about length. The reason is that because while length’s range is the integers, it really only maps into the positive integers. We add rules to allow Isabelle to reason about the length function with this additional property, simple ones such as !!a. length a >= 0 along with more complicated ones such as !!y c s. y < c ==> y < c + (length s). Here the !! indicates an meta-level universal quantification rather than a quantification within HOL.

The many theorems, lemmas and definitions within string theory such as concatenation, reverse, occurs count, and substring are all defined within this framework, translated more or less directly from the original RESOLVE theories. However, there are some definitions that are not as easy to create lemmas for, such as IS_PERMUTATION. We next describe the problem with this definition and its solution in Isabelle.

Definition of IS_PERMUTATION within Isabelle

The IS_PERMUTATION definition is a predicate on two strings that returns true if and only if both strings have the exact same occurrence counts for every item—one string can be reordered to be exactly the other one. In RESOLVE mathematical theory, the main lemmas for reordering within string theory are shown in Figure 3.2.
Corollary IsPermutation1Refl:
∀α (α)IS_PERMUTATION(α)

Corollary IsPermutation1Sym:
∀α, β (α)IS_PERMUTATION(β) ⇔ (β)IS_PERMUTATION(α)

Corollary IsPermutation1Trans:
∀α, β, γ (α)IS_PERMUTATION(β) ∧ (β)IS_PERMUTATION(γ) → (α)IS_PERMUTATION(γ)

Figure 3.2: Permutation Lemmas for Equivalence Relation

Extra lemmas are added for the IS_PERMUTATION predicate. These extra lemmas were not strictly necessary for the completeness of the theory development, but rather are more useful for Isabelle. Figure 3.3 shows the original corollaries; essentially with the associativity of concatenation, this is enough theoretically to prove that any reordering of concatenations results in permutation of the original string.

Corollary IsPermutation2:
∀α, β (α ◦ β)IS_PERMUTATION(β ◦ α)

Corollary IsPermutation3:
∀α, β, γ, δ (β)IS_PERMUTATION(γ) ⇔ (α ◦ β ◦ δ)IS_PERMUTATION(α ◦ γ ◦ δ)
∀α, β, γ, δ (β)IS_PERMUTATION(γ) ⇔ (α ◦ β)IS_PERMUTATION(α ◦ γ)
∀α, β, γ, δ (β)IS_PERMUTATION(γ) ⇔ (β ◦ δ)IS_PERMUTATION(γ ◦ δ)

Figure 3.3: Permutation Lemmas

Isabelle, however, cannot use this form. The main issue is that Isabelle applies simplification rules in one direction, left to right. Therefore, the associativity order restricts the applications of this lemma. This would require new lemmas to be added
for specific forms of permutations similar to those in Figure 3.3 for each syntactic categorization. However it is possible to fix this issue by using other features of Isabelle. By the already proved theorems, IS_PERMUTATION is an equivalence relation. Therefore it is possible to create equivalence classes based on the permutation equivalence and perform manipulations on those equivalence classes. First, the equivalence class is defined as in Figure 3.4 by appealing to the definition of IS_PERMUTATION.

Permutation Equivalence Class Definition:
\[ a \simeq b \equiv a \text{ IS } _{-} \text{PERMUTATION } b \]

Figure 3.4: Permutation Equivalence Relation

The solution defines rules that take these permutations equivalence classes and perform operations on them that preserve equivalence, but allow for the creation of a normal form. A new function, that represents concatenation that syntactically occurs in an argument of the IS_PERMUTATION predicate, is defined. In most situations, the concatenation function is not commutative and commutativity is a property of this new function. Then lemmas are proved to create the simplification rules to put any permutation into this normal form. Finally, lemmas are created to return the simplified equivalence class version of the IS_PERMUTATION predicate with a rewritten concatenation operator back to the original form. This form is readable by humans and usable by the other proof rules. While specific proof invocations are needed to accomplish this, to avoid simplifier loops an automated tool generating input for Isabelle may insert the requisite proof statements when needed.
3.4.2 Automated Proofs in Isabelle

Isabelle [51] provides several methods for proving a mathematical formula automatically, with varying degrees of power. All of the methods use the formalism of a natural deduction proof system, so we describe it here. Natural deduction [23] is a formal proof system that fairly closely mirrors how humans naturally perform proofs. The setup is as follows. Each step of the proof includes a set of local proofs. Each local proof has a set of assumptions that are implicitly conjoined together, and one goal. Proof rules either add, remove, or modify local proofs.

As an example, consider an attempt to prove that the set of integers is infinite. One statement of this fact is, for all integers $z$, there is an integer $i$ that is bigger than $z$. Formally, this is expressed as: $\vdash \forall z. z \in \mathbb{Z} \rightarrow \exists i. i \in \mathbb{Z} \land i > z$. Any assumptions are written to the left of the $\vdash$ symbol and the goal is written after; in this case, we have no assumptions. On this proof statement the proof system allows us to modify the left or right side of the $\vdash$ symbol. We may take any already proved theorem or axiom and either match the goal of this theorem with the local goal, or match the assumptions of the theorem with some assumptions in the local assumptions. In this case, an axiom of the proof system is that $P(y) \vdash \forall x. P(x)$ where $y$ is a fresh variable (not free in any local proof). This allows us to replace the goal of the first proof with the assumptions of the theorem. We then obtain this proof statement: $\vdash y \in \mathbb{Z} \rightarrow \exists i. i \in \mathbb{Z} \land i > y$. After one more axiom application of $(P \vdash Q) \vdash P \rightarrow Q$, we obtain: $y \in \mathbb{Z} \vdash \exists i. i \in \mathbb{Z} \land i > y$. The proof of this, which we will not discuss, involves choosing $i$ to be $y + 1$. Notation-wise, this axiom states that, in order to prove $P \rightarrow Q$ it is enough to show $P \vdash Q$. 

98
So far we have seen only applications of matching the goal of the purported theorem with the goal of an already proved theorem. We may also match the assumptions of the purported theorem with the assumptions of an already proved theorem. We use the same example, \( \forall z. z \in \mathbb{Z} \rightarrow \exists i. i \in \mathbb{Z} \land i > z \), to show this. Instead of proving theorem directly, we will prove it via contradiction. The axiom for proof by contradiction is \( \neg Q \rightarrow Q \). After applying this axiom by matching the goal, we obtain \( \neg (\forall z. z \in \mathbb{Z} \rightarrow \exists i. i \in \mathbb{Z} \land i > z) \rightarrow (\forall z. z \in \mathbb{Z} \rightarrow \exists i. i \in \mathbb{Z} \land i > z) \). We can then simplify the form of the equation to \( \exists z. z \in \mathbb{Z}; \forall i. i \in \mathbb{Z} \rightarrow i \leq z \vdash \bot \).

Here, we write \( \bot \) to indicate that the goal was simplified to false, because its negation was an assumption. We now can remove the \( \exists \) by this theorem: \( \exists i. P(i) \vdash P(i') \) where \( i' \) is a fresh variable. Now we match the assumptions of the already proved theorem with some assumptions of the purported theorem. So we have \( z' \in \mathbb{Z}; \forall i. i \in \mathbb{Z} \rightarrow i \leq z' \vdash \bot \). We now also need to choose the particular value of \( i \) that will give us a contradiction, namely \( z' + 1 \). The axiom that allows us to do this is \( \forall x. P(x) \vdash P(x^t) \) where \( x^t \) is some term of the same type as \( x \). The resulting purported theorem is \( z' \in \mathbb{Z}; z' + 1 \in \mathbb{Z} \rightarrow z' + 1 \leq z' \vdash \bot \). Now, we can simplify the formula to \( z' \in \mathbb{Z}; 1 \leq 0 \vdash \bot \) which is the same as \( z' \in \mathbb{Z}; \bot \vdash \bot \) and fortunately, our last axiom needed is \( \vdash (Q \vdash Q) \) for any predicate \( Q \), which closes out the proof.

These examples were designed to give some intuition of the proof steps that Isabelle performs by matching against the goal, some assumptions, or both. Isabelle also has a directional simplifier that rewrites terms; we used this in the examples implicitly e.g., by replacing \( z' + 1 \leq z' \) with \( 1 \leq 0 \). There is also another implicit way of categorizing theorems—those that should not be used. For the goal of using Isabelle as an automated theorem prover for VCs, we must categorize all theorems into eight
categories. The five main categories are “not used”, “simplifications”, “introduction rules” (match the goals), “destruction rules” (match some of the assumptions), or “elimination rules” (match both the goal and some of the assumptions). For each of the latter three, we can sub-categorize them into “safe” or “unsafe” where “unsafe” is akin to a “cut” rule in logic programming.\footnote{There are other categorizations, but they can be thought of as subsets of these categories.}

We also have one more option to use in the automated proofs: we may prove that the new types are actually instances of existing, generic theories and automatically import the reasoning from those theories. For example, there is a general theory of symmetric groups that can be instantiated in this way.

### 3.4.3 Additional Lemmas Needed by Isabelle for Automated Proofs

While a given theory, \textit{e.g.}, string theory, developed by mathematicians, may be adequate for proofs created by humans, the form of theorems may complicate formal proofs. We now describe the process of attempting to prove a VC, and modifying the theory to be more amenable to verification.

We take the theory surrounding the concatenation operator in string theory and show how it is both integrated into Isabelle and the extensions needed for automated VC proving. The theory for the concatenation operator is given in Figure 3.5. The function is inductively defined based on value of the second parameter. The first corollary indicate that $\lambda$ is an identity for the operator, \textit{i.e.}, $\alpha \circ \lambda = \alpha$ and $\lambda \circ \beta = \beta$. The second corollary shows that $\circ$ is also associative; all parenthesizations are equivalent. The third corollary indicates that if we have a fact that $\alpha \circ \beta = \delta \circ \beta$ then we can conclude that $\alpha = \delta$. 
Inductive Definition for \( _\circ_ \): \( \alpha : \Gamma^* \times \beta : \Gamma^* \rightarrow \Gamma^* \) is

1. \( \alpha \circ \lambda = \alpha \)
2. \( \alpha \circ \text{ext}(\beta, x) = \text{ext}(\alpha \circ \beta, x) \)

Corollary 1: \( \text{Is\_Identity\_For}(\circ, \lambda) \);
Corollary 3: \( \text{Is\_Associative}(\circ) \);
Corollary 4: \( \text{Is\_Right\_Cancellative}(\circ) \);

Figure 3.5: Definition of Concatenation

Our first order of business is to define the concatenation operator within Isabelle, shown in Figure 3.6. We define it using the \texttt{function} infrastructure, which is designed for recursive function definitions. We name the operator \texttt{cat}, but we also make it an infix operator with the name \texttt{concat}\textsuperscript{9}. The first sequence of proof steps establishes the consistency of the function, \textit{i.e.}, that no input is mapped to two function values and that every possible input value is handled. The second justifies termination of the recursion via a measure that decreases on every recursive call. The measure is given by the \texttt{measure} keyword; here the measure is simply the length of the second string. Here is where \texttt{len} is needed.

The infrastructure in Isabelle automatically adds simplification rules based on the formulas in the \texttt{where} clauses. So, we have the first corollary essentially for free. Here is where we diverge slightly from the RESOLVE theory. Isabelle has a notion of a monoid, \textit{i.e.}, a structure with an associative binary operator and an identity. The characteristic properties include both corollaries 1 and 3. So, instead of proving the properties as corollaries, we instead prove that concatenation is an instance of

\textsuperscript{9}We would like to name concatenation \( \circ \), or \( \ast \). Unfortunately, both of these symbols lead to parsing problems, especially with Integer theory.
function cat :: 'a string => 'a string => 'a string" (infix "concate" 55) 
where 
   'cat a empty_string = a'
   | 'cat a (ext b x) = ext (cat a b) x'
apply atomize_elim
apply clarsimp
apply (rule ALLNotExEmptyL)
applyclarsimp+
apply (drule Axiom2)
apply clarsimp
done

termination
apply (relation "measure (%(x,a). (len(a)))")
apply auto
done

Figure 3.6: Concatenation Definition in Isabelle

this general mathematical theory. We show the Isabelle source for this in Figure 3.7. 

The first line of the proof substitutes the empty_string and cat functions into the 
corresponding monoid parameters. The second is a general proof command, while 
the third adds the induction axiom as an introduction rule.

interpretation monoid_cat : monoid_mult "empty_string" "cat"
apply unfold_locales
apply auto
apply (auto intro: Axiom3)
done

Figure 3.7: Concatenation Interpretation as a Monoid in Isabelle

Our last task is to show the last corollary, namely that Is_Right_Cancellative(◦). 
We show this directly, without appealing to other aspects of Isabelle. Now that we 
have proved all of the aspects of the RESOLVE theory for concatenation, we expect 
to be able to prove all of the VCs. Unfortunately, we will quickly see that there are
additional theorems that should be added. Consider the formula \( b \circ a = b \rightarrow a = \lambda \).

This is a theorem, provable from our definition, with a proof sketch as follows: we modify the formula using corollary 1 to obtain \( b \circ a = b \circ \lambda \rightarrow a = \lambda \) and then apply corollary 4 to obtain \( a = \lambda \rightarrow a = \lambda \). Unfortunately, this reasoning process is not one that Isabelle will do; more syntactically complicated formulas are not generated because of the possibility of making the formula harder to prove. So, we add a new lemma to make this reasoning step obvious to the verifier, as shown in Figure 3.8. An examination of the proof indicates that this proof is long enough to warrant this new lemma.

```
lemma CatEmpty [dest]:
  \( b \text{ concate } a = b \Longrightarrow a = \text{ empty_string} \)
apply (induct b a rule: cat.induct)
apply clarsimp
apply (rule classical)
apply simp
apply (subgoal_tac "length (ext (a concate b) x) = length a")
defer
apply clarsimp
apply (erule contrapos_pp)
apply (simp only: LengthExt)
apply clarsimp
apply (subgoal_tac "length b < 0")
apply (drule LengthGZ1)
apply auto
done
```

Figure 3.8: Additional Lemma for Isabelle
3.4.4 Potential Proof Commands and a Proof Script for VCs

Isabelle also has several methods of automated proof, all varying in power and riskiness. The first, and simplest, is the `clarify` command. This command performs only substitution of equals for equals and “safe” rules that do not cause any splits. It is a very safe command to apply in that it will terminate quickly—there is no backtracking.

The next command is the `simp` command. It is different than the previous command in that it applies any registered simplification rule. Each simplification rule is a directional rewrite rule. Unless the user has added a non-trivial loop in the simplifier, this command will also terminate quite quickly. This command also has attributes that allow the user to control which simplification rules to use.

The next command, `clarsimp`, applies both of the previous two commands sequentially as many times as possible.

The next command is the `safe` command. It is like `clarify`, but it also applies splitting rules.

The final command is the `auto` command. It performs all of the previous commands and then proceeds to perform a best-first search through the proof tree. This is, obviously, the most powerful command, but it is the most dangerous; this command has the greatest chance of taking a unusable amount time to complete.

In order to use Isabelle as an automated theorem prover, commands must be synthesized to control the proof process. Empirically, we have found that just using `auto` may not prove simple VCs that `clarsimp` would. We would like to solve the simple cases with simple methods before trying the “heavy guns” on the harder proofs. Thus, we use the script shown in Figure 3.9. The script first attempts to use the weakest
commands with fewer than normal rules, *i.e.*, the first line only uses propositional
simplifications. The *algebra_simps* attribute is defined by the Isabelle library and
is used for algebraic simplifications, *e.g.*, from group theory that are applied to both
the integers and strings. The lines with *Permutation* and *PerCatRevert* are used
to canonicalize permutations. The first line with *auto* is generic, while the second
one includes some rewrite rules from Harvey Friedman’s decision procedure for a re-
stricted set of universal string formulas [21], proved correct via Isabelle relative to
our formalization of string theory. The last line performs linear arithmetic.

```isar
apply ((simp only: simp_thms), clarify?)
apply clarify?
apply (simp only: algebra_simps)?
apply (simp add: algebra_simps)?
apply (simp only: Permutation)?
apply (simp add: PerCatRevert)?
apply (auto simp add: algebra_simps)?
apply (auto dest!: Elements18 Elements19 HarveyRule2)?
apply arith?
```

Figure 3.9: Generated Proof Script for VCs with String Variables

### 3.4.5 Results

Here we compare the performance of Isabelle to the performance of SplitDecision
on VCs from RESOLVE programs to validate that Isabelle is an effective back-end
prover of VCs. We take VCs from the current set of RESOLVE programs, both those
with known bugs and known correct implementations; the percentage proved by either
does not change significantly if only VCs from non-buggy implementations are used.
The total number of VCs is 3796, all of which are given to both SplitDecicision and
Isabelle. SplitDecision has no timeout, while Isabelle times out after 3 seconds per
VC. The evaluation is performed on modest hardware, an Intel Atom N280 with 2 Gigabytes of RAM.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Percent Proved</th>
<th>Percent Refuted</th>
<th>Average Time Per VC</th>
<th>Maximum Time Per VC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SplitDecision</td>
<td>84.09%</td>
<td>0%</td>
<td>0.166 seconds</td>
<td>1.01 seconds</td>
</tr>
<tr>
<td>Isabelle</td>
<td>75.00%</td>
<td>0%</td>
<td>1.11 seconds</td>
<td>3.00 seconds</td>
</tr>
</tbody>
</table>

SplitDecision itself fails to prove 604 VCs. If we examine just those VCs and look at how Isabelle performs on them, we see that it performs quite well in that it is able to prove an additional 137 VCs.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Percent Proved</th>
<th>Percent Refuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isabelle</td>
<td>22.68%</td>
<td>0%</td>
</tr>
</tbody>
</table>

SplitDecision also performs well on VCs Isabelle is unable to prove. Of the 953 VCs that Isabelle does not prove, SplitDecision is able to prove 486, or just over half.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Percent Proved</th>
<th>Percent Refuted</th>
</tr>
</thead>
<tbody>
<tr>
<td>SplitDecision</td>
<td>51.00%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Based on these results, while Isabelle is not a particularly fast back-end prover, it does provide considerable added value. It is able to prove many VCs that Split-Decision does not. Recall the criteria of Section 3.1, namely that a back-end prover should:

1. be sound and preferably produce proof certificates of every proof; \textit{i.e.}, it may not prove results that are false, and it should be able to generate machine-checkable proof certificates corresponding to any proofs the prover constructs,

2. be fast; whatever reasoning process it employs must be able to determine quickly if a VC generated from typical software components is true or false,

3. be able prove “obvious VCs” \textit{i.e.}, VCs for which human readers can quickly construct a completely correct proof,

4. be able to construct counterexamples to invalid VCs,

5. be readily extensible with additional mathematical theories and lemmas for use in its internal proof process without undue configuration or coding burden.

Isabelle completely satisfies items 1 and 5, while partially satisfying item 2 and 3. Thus, it is an effective back-end prover, but it remains far from the ideal one. It is less efficient than SplitDecision, but that is to be expected. Isabelle is a general logic framework, as such, it has more overhead than SplitDecision, but its average time taken per VC is only a factor of 10 different (and the difference is from less than one second to roughly one second per VC). Moreover, both tools can be forced to time out to ensure that neither run indefinitely. By contrast, SplitDecision satisfies items 1, and 2 and partially satisfies items 3 and 4.
3.4.6 Lessons Learned

We now endeavor to take our experience with Isabelle and identify some lessons learned for use as guidance in the choice of a back-end theorem prover. The lessons learned are summarized in Figure 3.10. Briefly the lessons follow three categories, properties of the prover, properties of the VCs and properties of the mathematical theory.

1. Sound proof tools are important when adding new lemmas.

2. A proof language to enter in new theorems is important for extensibility, though there should be little to no extra annotation needed for the theorems to be incorporated into the proof process.

3. Specific instantiations of theorems for extremely common cases should be included.

4. The inability to exploit general properties of particular functions is a significant practical limitation for theory development.

5. Quantifiers are hard for provers to handle.

Figure 3.10: Lessons Learned

The first lesson learned is that it is extremely important to have a sound proof tool; one that makes it very hard to enter in wrong theorems. Of course, most tools have ways around this, usually by simply adding new axioms to the theory. So, an ambitious person may decide to add not only the definitions, but also the theorems as axioms. We will show an example of why this is fraught with danger and should be avoided. We examine the \texttt{IS\_CONFORMAL\_W} predicate in string theory. This predicate takes a binary relation on items and a string of items and returns true if and only if
every consecutive pair of items in the string is ordered by the binary relation. For example, one could pass in the empty relation and never be able to prove that any string with more than one item in it is conformal to that relation, or one could pass in the complete relation and prove that any string is conformal to the relation. We now examine an attempt to prove a VC involving combining three different conformal strings, the essence of which is given by Figure 3.11.

... 
\[ IS\_TOTAL\_PREORDER(R) \] 
\[ \land \ IS\_CONFORMAL\_W(R, a \ast b) \] 
\[ \land \ IS\_CONFORMAL\_W(R, b \ast c) \] 
\[ \longrightarrow \ IS\_CONFORMAL\_W(R, a \ast b \ast c) \] 

Figure 3.11: Essence of the Interesting VC

One might, as this author did after a long session of adding new theorems, examine the VC for several minutes and conclude that it is “obviously” a theorem left out of the theory. Naturally, one adds the theorem to the theory immediately to rectify the situation. Since this is operating within Isabelle, once the theorem is entered our obligation is to prove it. The proof starts with structural induction on the value of \( b \).

If \( b = ext(a', x) \), then we can use case analysis on the values of \( a \) and \( c \) to prove the theorem. What of the case when \( b = \lambda \)? Now we are in trouble: if \( a =< 1, 2 > \) and \( c =< -2, -1 > \), the assumptions are all satisfied, but yet the goal is quite false! It turns out the proof assistant was stuck on the one corner case that caused trouble. We are actually missing the assumption that \( b \neq \lambda \). With that assumption, the theorem is proved easily and the VC it is needed for is easily provable.
What this example shows is that humans are typically quite good at “big picture” ideas with proofs, but yet can be stymied by (ignoring) corner cases. Proof assistants, conversely, are very good at finding just these cases, but yet do not see the big picture. Since the verified software community is attempting to show that software is completely correct, bugs resulting from wrong theorems are especially troublesome. The sound proof system gives us one more level of safety by helping to avoid these kinds of mistakes.

Our next lesson learned is that the ability to add new theorems incrementally via a dedicated proof language is a very good feature. Isabelle’s proof language is much easier to use than its corresponding PolyML [47] code implementation. Moreover, based on discussions and collaboration with the implementer of SplitDecision [2], it is quicker to add a theorem via a proof language such as the one in Isabelle than with code. Isabelle’s proof language is especially good at the quick entry of theorems, especially when experimenting with potential theorems when the proofs are not yet needed. However, there is a pitfall, namely that Isabelle’s proof language not only requires the theorems, but also annotations to indicate how the theorem should be used in the proof process. There are relatively few annotations, but the explicit use of annotations is a barrier to the envisioned verified software infrastructure (even though it is essentially required by SplitDecision, Z3 [12] and other provers as well via the use of “triggers” or code to apply the proof rules). Moreover, as a practical matter, there are relatively few mathematicians in the world (compared to software developers), but there are even fewer people who have experience with proof tools of this nature. Thus, the ultimate back-end prover should have a proof language similar to Isabelle’s for entering new theorems, yet it should not require many annotations.
Even though general theorems are very useful, in many cases smaller instances of
the theorems are useful if these smaller instances are used very frequently. The reason
for this is if the occurrences of these simpler instances are frequent, we can reasonably
expect that this type of reasoning will be performed in the process of proving other
VCs as small steps. Rather than forcing the theorem prover to reprove the specific
instance of the theorem from the general instance every time, the specific instance
allows the prover to reuse reasoning. Figure 3.12 shows the general theorem shown
earlier, namely that, if two strings are conformal with a relation \( R \) and overlap in
a non-empty way, that the concatenation is also conformal with that same relation.
The overlap is represented as \( c \) and we see the obligation that \( c \) is not equal to \( \lambda \).

\[
\text{lemma Con4 [intro]:}
\]
\[
\begin{align*}
\text{Is Conformal}_w R (a \text{ concate } c); \\
\text{Is Conformal}_w R (c \text{ concate } b); \\
c \neq \text{empty_string} \\
\Rightarrow \\
\text{Is Conformal}_w R ((a \text{ concate } c) \text{ concate } b)
\end{align*}
\]

Figure 3.12: General Theorem for \text{Is Conformal}_w

It turns out that this general theorem is used in relatively few cases in VCs
generated from the components in the repository. In the vast majority of those cases,
the code is removing from one end and adding on the other. In this case, we can
create a specific instance of the theorem for our context.

This specific instance, shown in Figure 3.13, has one fewer assumption, but is
specific to the context in which the overlap is exactly one item. This theorem requires
fewer reasoning steps than the general version, as the prover no longer needs to establish that the middle piece is not empty. In other words, the more specific the theorem, the fewer steps it might take to be useful. However, a too-specific theorem is also a detriment as it will not be applicable in many situations. A lesson learned from this work is that the general theorems must be a part of the theory, but that specific theorems for common cases may also be included, a mantra akin to an architecture maxim “make the common case fast and all cases correct.” Here we note that this is a case where the tools’ limitations are front and center. Isabelle is not effective at applying very general lemmas without getting stuck within the details of the side-proofs needed to apply the general lemma, e.g., the obligation that $c \neq \lambda$. While other provers might be able to handle some of these more general lemmas without becoming bogged down in the generated proof obligations, at some point the prover will have to give up the proof process and the extra steps needed for the side proof can be the difference in the proof succeeding or failing.

We return to our example of IS_PERMUTATION and reflect on the problems it suggests. Isabelle represents all formulas logically as trees; associativity and commutative functions might present an issue. Fortunately, Isabelle has an internal metric that
allows it—given the requisite lemmas—to reorder the formulas as needed into a canonical form. The potential problem is when certain lemmas expect a particular form, e.g., cancellation lemmas, and the representation is not conducive to the form of the lemma. This was an issue within the RESOLVE theory development within Isabelle, however there were always different ways to structure the lemmas to avoid the issue. However this particular problem is something that neither a mathematician nor software engineer is likely to have experience with; people simply do not think in terms of formula trees if a function is associative and commutative. Thus, a lesson learned is that the more common algebraic properties should be supported natively within the back-end prover to better facilitate the verified software development process.

The last lesson learned is about VCs as they relate to proofs. One of the difficulties in automation has to do with universal instantiation in assumptions and, conversely, existential instantiation in goals as the result of quantifiers in \texttt{requires} and \texttt{ensures} clauses. For example; in a proof of a universally quantified statement in the conclusion of a VC (e.g., $A \Rightarrow \forall x. P(x)$), a prover can simply use a fixed (but arbitrary) element of the universe of the quantification for $x$, and prove the statement true for that element.

However, when an assumption in a VC involves universal quantification, the natural question is “what term should be used to instantiate the quantified variables?” With this issue, a general proof approach (without the use of a specialized decision procedure) must either never need to instantiate a quantified variable (avoiding the issue), or use a method that instantiates the quantifier “correctly” in many cases (possibly tuned for the types of VCs that are likely to occur). SMT solvers, in particular, have an algorithm called E-matching that uses “triggers” to identify places
where quantified formulas should be instantiated and how the instantiation should look. [22], we discuss this in related work in Section 3.5.

The dual of this issue is the use of existential quantification. An existential quantification in the assumptions does not cause any problems, whereas an existential quantification in the goal does. Our experience with Isabelle has indicated that Isabelle performs quantifier reasoning very poorly. It will try to prove the theorem, but it usually must be forcibly stopped because it makes no evident progress. Basically, the quantifiers make any proof much harder, as would be expected. Thus, our last lesson learned is to avoid quantifiers in VCs, in favor of mathematical definitions that encapsulate the quantification. The proved lemmas then act as guides to the prover for how to reason using the quantifiers, similar to e matching algorithms. While some quantifiers are easy to reason about, as discussed before, the dual nature of requires and ensures clauses assures us that the back-end prover will eventually have to deal with the problematic cases if quantifiers are ever used, either on the implementer or client side.

3.5 Related Work

We have already mentioned prototypical examples of each of the types of theorem provers in Section 3.2 along with the benefits and limitations of each. We will extend that material in this section with a review of different integrated programming language/automated theorem prover systems. While the automated theorem prover is important, the total package of the programming language, proof language, VC generator and associated theorem provers gives a better indication of the picture of the impact on the software verification process from Section 1.1 reprinted in Figure 3.14.
We first consider Dafny [44], a research language that is, at a high level, similar to RESOLVE. A main difference is that it models the heap explicitly (references are integral to Dafny). Also the Dafny infrastructure is tied to both the Boogie [7] VC generator and the Z3 [12] back-end prover. The explicit modeling of the heap requires that quantifiers are used. In order to deal with this, Z3 allows for “triggers,” i.e., syntactic expressions for how the quantifiers should be instantiated, similar in spirit to specific lemmas. In some cases, these are just heuristics that are designed for the semantics of Dafny/Boogie to help guide the proofs by adding hints for how to process the semantics of structures within the programming language, e.g., the heap. These trigger may also be used as hints to let the prover know how particular general theorems should be instantiated. RESOLVE has no need for this because no general VCs include heap properties. Moreover, these triggers can be brittle and may not work as expected [22, 45]; both papers mention that distribution axioms are fragile. Also, Leino and Monahan [45] mention that supporting different types of comprehensions are problematic if there are specific properties of the particular
kind of comprehension that is needed. Also, these triggers are not refutationally complete [22] (a slightly weaker notion than complete).

Zee et al. [70, 8, 69, 71] have used a hybrid approach of applying both specialized decision procedures and a general proof assistant to prove that code purporting to implement certain data structure specifications is correct. However, the use of Java as a starting language requires that the specifications use reference equality or comparison. Thus, all Jahob VCs must include models of the heap while RESOLVE VCs do not. Our approach proves properties that depend on the values of the objects instead. Jahob also includes a complete proof language as part of its annotations. Our approach does not do this because including a proof language would merge the two roles of software engineer and mathematician; something that we do not consider tenable in either the short- or long-term.

The Why methodology [19] involves a simplified programming language, annotated with logical definitions, axioms, preconditions, post conditions and loop invariants, for which VCs can be generated. A subset of both C (with annotations) and Java (with JML specifications) can be translated into the simplified programming language, such that the VCs generated are claimed to represent the correctness of the original C or Java code. The translation process from C or Java must explicitly capture the memory model of the original source language (C or Java); as a result of using RESOLVE, we do not need an explicit memory model, simplifying the generated VCs. As with Jahob, Why allows for the use of multiple theorem provers, with a fall-back to an interactive version of Isabelle if a VC is not proved. As with Jahob’s proof language, our vision of verified software development does not include this as a job description for a software engineer, so we explicitly avoid this approach.
The Clemson University RSRG group has developed a RESOLVE prover [64] that has a simple design principle—be as simple as possible. The prover works by expanding the set of facts until the goal is a simple consequence of the facts. The underlying proof system uses Gentzen’s sequent calculus [23]. While the prover can prove many VCs, based on discussions with the authors, it can become bogged down with many extraneous facts. Fundamentally, the goals of the minimalist prover project are the same as for our project; the approach is different. We take an off-the-shelf prover to study properties of back-end provers for VCs, while the Clemson prover starts from scratch.

3.6 Future Work

In the future, we plan to extend the current theory base within Isabelle to handle more mathematical theory. One promising avenue of research is to prove the decision procedure underlying SplitDecision correct within the RESOLVE string theory embedding within Isabelle. This would ensure consistency between the decision procedure and RESOLVE string theory. We also plan on extending the Isabelle VC prover infrastructure to maintain faithful translation of ordering relationships, e.g., a total preorder, between the RESOLVE source code and Isabelle VCs. Essentially, there is not enough information to translate these properties accurately. Isabelle maintains ordering relationships on the object types via the $<$, $\leq$, $\geq$ and $>$ symbols, so a linear order would be represented as a type `obj::linorder`. This is inconsistent with RESOLVE’s convention to use assumptions on named relations; moreover it does not allow for the use of multiple linear ordering relations within the same component. We also plan to take advantage of our lessons learned from proving VCs using Isabelle to
develop our own theorem prover that is “tuned” to the requirements of proving VCs from RESOLVE programs (*i.e.*, without heap properties).

### 3.7 Conclusion

In conclusion, we have demonstrated empirically the second part of the thesis, that an interactive proof assistant can be an effective back-end prover. We have also discussed lessons learned from this process and have explored mathematicians’ role in the verified software development process.
Chapter 4: Impact of Formula Translation on Provability

In this chapter we address the question, “how hard it is, practically, to translate between different logics” In other words, there are many different types of logics such as modal, first order, second order, and higher order each with relative merits and limitations. Logics may also allow for undefined terms—\textit{free logic}—or require that all terms are defined—\textit{total logic}—and may include more than one sort.\footnote{Akin to a type in programming languages.} Simply examining this very coarse categorization of some different kinds of logics, it is clear that there are many different kinds. Even if we fix a particular kind of logic, say first order, there are four kinds (many-sorted/one-sorted and total/free). For instance, one particular back prover may support one logic while the proof rules might be based in another. Concretely, Z \cite{Z} is a specification language that has partial functions, if we wished to process Z assertions using Isabelle \cite{Isabelle}, we would need to be able to perform the translation from a free logic into a total logic; we return to this in Section 4.9. Therefore, we examine the question: what are the practical difficulties a tool builder encounters when implementing machine-checked translations between any two?

Since this is a question with many different facets, we address a small piece of it by simplifying the problem into a concrete translation problem. This piece is the
translation between a many-sorted free first-order logic and a many-sorted total first-order logic, with the same question, “What is the impact of the translation process both theoretically and practically?”

This chapter starts out with definitions of a many-sorted free first-order logic and many-sorted total first-order logic in Sections 4.1, 4.2, 4.3, and 4.4. It then analyzes the problem of translating from the former to the latter theoretically by establishing mathematical infrastructure and proving a translation theorem in Sections 4.5, 4.6, and 4.7. Finally, a practical application of this theoretical result is presented in Section 4.8 that is verified in Isabelle. Related work is discussed in Section 4.9, and future work in Section 4.10 along with conclusions in Section 4.11

4.1 Syntax of Many-Sorted Free Logic

We start first by defining the syntax of a many-sorted free logic. The main difference between a free and total logic is that a free logic allows undefined terms. Terms represent potential values within a universe of discourse, e.g., the symbol 0, and the expression $2 \times 10 + 5 - x$, are terms.

$$
\Sigma = (I, C, R, F, E) \text{ where}
$$

1. $I$ is a set of sorts,
2. Each element in $C$ is written $C^i_j$, for the $j$-th constant symbol for the sort $i$,
3. Each element in $R$ is written $R^{i_1 \times \cdots \times i_n}_j$ as the $j$-th relation symbol that has domain $i_1 \times \cdots \times i_n$,
4. Each element of $F$ is written $F^{i_1 \times \cdots \times i_n: i_i}_j$ as the $j$-th function symbol that has signature $i_1 \times \cdots \times i_n \rightarrow i$,
5. Each element of $E$ has the form $=i, \approx_i, \neq_i$, one for each $i \in I$.

Figure 4.1: Signature of a First-Order Many-Sorted Free Logic
A \( \Sigma \) signature is a 5-tuple that is written in Figure 4.1. Briefly, the signature include a slot for the set of sorts. The signature also has slots for constant, relation, function, and equality symbols. Each of these symbols is parameterized by the sort of its signature. A many-sorted free logic also has variables of each sort, where the \( j \)-th variables of sort \( i \) is denoted \( v^i_j \). There are term connectives, \( \downarrow \) (read as defined), \( \uparrow \) (read as undefined), and propositional connectives \( \land, \lor, \rightarrow, \leftrightarrow \), and \( \neg \), and quantifiers \( \exists^i \) and \( \forall^i \), or for each sort \( i \).

A term is defined by structural induction and is given in Figure 4.2. Essentially a term is either a variable, a constant, or a function applied to terms with the correct sorts. The sort of term is given by its superscript.

A term \( t \) is either

1. a variable \( v^i_j \),
2. a constant \( C^i_j \), or
3. a function application \( F^i_{1}^{i_1}, \ldots, i_n: i_j(t_1, \ldots, t_n) \) where each \( t_k \) is a term and the sort of each \( t_k \) is \( i_k \).

Figure 4.2: Definition of a Free Logic Term

The next order of business is to define an atomic formula: something that is either true or false. Each atomic formula may either be an equality proposition, a proposition that a term is either defined or undefined, or a proposition that some terms satisfy a given relation. The sorts act as types, so the syntax ensures that all terms have the correct sort. The definition is presented in Figure 4.3. Two atomic formulas that are not in standard first-order logic are \( \downarrow t \) and \( \uparrow t \); intuitively the semantics of these two symbols will be that \( t \) is defined and \( t \) is undefined respectively.
An atomic formula is either

1. \( t_1 =_i t_2 \) where \( i \in I \) and the sort of \( t_1 \) and \( t_2 \) is \( i \),
2. \( t_1 \approx_i t_2 \) where \( i \in I \) and the sorts of \( t_1 \) and \( t_2 \) is \( i \),
3. \( t_1 \neq_i t_2 \) where \( i \in I \) and the sorts of \( t_1 \) and \( t_2 \) is \( i \),
4. \( R_{j_1}^{i_1} \ldots R_{j_n}^{i_n}(t_1, \ldots, t_n) \) where each \( i_j \in I \) and the sorts of each \( t_i \) is \( i_j \),
5. \( \downarrow t \) (for any sort \( i \) of \( t \)), or
6. \( \uparrow t \) (for any sort \( i \) of \( t \)).

Figure 4.3: Definition of a Free Logic Atomic Formula

Finally, we are in a position to define the syntax of a general many-sorted free logic formula. A formula \( \theta \) is defined by induction in Figure 4.4.

A formula \( \theta \) is either

1. an atomic formula \( \alpha \),
2. \( \neg \theta_1 \),
3. \( \theta_1 \land \theta_2 \),
4. \( \theta_1 \lor \theta_2 \),
5. \( \theta_1 \rightarrow \theta_2 \),
6. \( \theta_1 \leftarrow \theta_2 \),
7. \( \forall v_i^{j_i} \theta \), or
8. \( \exists v_i^{j_i} \theta \).

Figure 4.4: Definition of a Free Logic Formula

4.2 Semantics of Many-Sorted Free Logic

The semantics of a many-sorted free logic signature must also be presented before we discuss the theorem. The semantics of a \( \Sigma \)-signature will have definitions for
each of the symbols used within the given \( \Sigma \)-signature. More formally, a model \( M \) for a given \( \Sigma \)-signature has components determined by \( \Sigma \), namely for each \( i \in I \) a nonempty domain \( M^i \), for each relation symbol \( R_j^{i_1,\ldots,i_n} \) a predicate \( R_j^{i_1,\ldots,i_n}^{M^i} \) which is a subset of \( M^{i_1} \times \cdots \times M^{i_n} \), for each constant symbol, \( C_j^i \) a value \( C_j^i^{M^i} \) in \( M^i \), and for each function symbol \( F_j^{i_1,\ldots,i_n : i} \) a partial function \( F_j^{i_1,\ldots,i_n : i}^{M^i} \) from \( i_1 \times \cdots \times i_n \rightarrow i \).

We next describe what it means for a model to satisfy a given formula under a particular interpretation. The interpretation will give values to any free variables in the formula. First, let \( \text{Var} \) be the set of all potential variable symbols, and let \( \text{Terms} \) be the set of all well-formed terms. So, given a function \( s : \bigcup_{i \in I} \text{Var}^i \rightarrow \bigcup_{i \in I} M^i \) such that for any \( i \in I \) \( s(v_j^i) \in M^i \), we define \( M \models \Sigma \theta[s]^{11} \).

Given \( s \), we define \( \bar{s} : \text{Terms} \rightarrow \bigcup_{i \in I} M^i \) by induction on the structure of terms to extend \( s \) to handle all terms. So,

\[
\bar{s}(t) = \begin{cases} 
  s(v_j^i) & \text{if } t \text{ equals } v_j^i \\
  C(v_j^i) & \text{if } t \text{ equals } C_j^i \\
  F_j^{i_1,\ldots,i_n : i}^{M^i}(\bar{s}(t_1),\ldots,\bar{s}(t_n)) & \text{if } t \text{ equals } F_j^{i_1,\ldots,i_n : i}(t_1,\ldots,t_n)
\end{cases}
\]

In the last case for the definition of \( \bar{s} \), we note that \( F_j^{i_1,\ldots,i_n : i}(t_1,\ldots,t_n) \) may not be defined for some values of the parameters, because \( F \) is a partial function.

Next, we define \( M \models \Sigma \alpha[s] \) where \( \alpha \) is an atomic formula in Figure 4.5. An intuitive consequence of this definition is that atomic formulas that are relations can be true only if all terms are defined; undefined terms can only give rise to atomic formulas that have the semantics true if the formula has \( \approx_i \) or \( \uparrow \).

Finally, we define \( M \models \Sigma \theta[s] \) where \( \theta \) is a formula in Figure 4.6.

\(^{11}\)Pronounced \( M \) satisfies \( \alpha \) in a structure \( \Sigma \) under an interpretation \( s \)
1. \( M, t_1 = t_2 \[s\] \) iff \( \pi(t_1) = \pi(t_2) \) and both \( \pi(t_1) \) and \( \pi(t_2) \) are defined
2. \( M, t_1 \neq t_2 \[s\] \) iff \( \pi(t_1) \) does not equal \( \pi(t_2) \) and both \( \pi(t_1) \) and \( \pi(t_2) \) are defined
3. \( M, t_1 \approx t_2 \[s\] \) iff \( \pi(t_1) \) equals \( \pi(t_2) \) or both \( \pi(t_1) \) and \( \pi(t_2) \) are undefined
4. \( M, t_1 \downarrow \[s\] \) iff \( \pi(t_1) \) is defined
5. \( M, t_1 \uparrow \[s\] \) iff \( \pi(t_1) \) is undefined
6. \( M, R_{i_1,\ldots,i_n}^s(t_1,\ldots,t_n) \[s\] \) iff \( \pi(t_k) \) is defined for all \( k \in [1,\ldots,n] \) and \( (\pi(t_1),\ldots,\pi(t_n)) \in R_{i_1,\ldots,i_n}^s \)

**Figure 4.5: Definition of \( = \) for Free Logic Atomic Formulas**

1. \( M, \alpha \[s\] \) is determined by the atomic formula rules
2. \( M, \neg \theta \[s\] \) iff \( M, \theta \[s\] \) is false
3. \( M, \theta_1 \land \theta_2 \[s\] \) iff \( M, \theta_1 \[s\] \) and \( M, \theta_2 \[s\] \)
4. \( M, \theta_1 \lor \theta_2 \[s\] \) iff \( M, \theta_1 \[s\] \) or \( M, \theta_2 \[s\] \)
5. \( M, \theta_1 \rightarrow \theta_2 \[s\] \) iff \( M, \theta_1 \[s\] \) is false or \( M, \theta_2 \[s\] \) is false
6. \( M, \theta_1 \leftrightarrow \theta_2 \[s\] \) iff \( M, \theta_1 \[s\] \) is false or \( M, \theta_2 \[s\] \) is true and \( M, \theta_1 \[s\] \) is true or \( M, \theta_2 \[s\] \) is false
7. \( M, \forall v_i^j \cdot \theta \[s\] \) iff for any \( d \) in \( M^i \), \( M, \theta[v_j^i \leftarrow \theta] \)
8. \( M, \exists v_i^j \cdot \theta \[s\] \) iff there is \( d \) in \( M^i \), \( M, \theta[v_j^i \leftarrow d] \)

**Figure 4.6: Definition of \( = \) for General Free Logic Formulas**
In accordance with first-order logic semantics [15], we write \( M \models \theta \) if for any function \( s : U_{i \in I} \text{Var}^i \Rightarrow U_{i \in I} M^i \), \( M \models \theta[s] \); in other words, \( \theta \) is true under any interpretation.

### 4.3 Syntax of Many-Sorted Total Logic

We next turn our attention to the definition of a many-sorted total logic. The syntax is similar to the syntax of the free logic presented earlier.

\[ \Sigma = (I, C, R, F, E) \]

1. \( I \) is a set of sorts,
2. Each element in \( C \) is written \( C_j^i \), for the \( j \)-th constant symbol for the sort \( i \),
3. Each element in \( R \) is written \( R_{i_1, \ldots, i_n}^j \) as the \( j \)-th relation symbol that has domain \( i_1 \times \cdots \times i_n \),
4. Each element of \( F \) is written \( F_{i_1, \ldots, i_n}^j \) as the \( j \)-th function symbol that has signature \( i_1 \times \cdots \times i_n \rightarrow i \),
5. Each element of \( E \) has the form \( =_i, \neq_i \), one for each \( i \in I \).

Figure 4.7: Syntax of a First-Order Many-Sorted Free Logic

A \( \Sigma \) signature is a 5-tuple that is written in Figure 4.7. Briefly, the signature includes slots for the set of sorts. The signature also has slots for constant, relation, function, and equality symbols. Each of these symbols is parameterized by the sort of its signature. A many-sorted free logic also has variables of each sort, where the \( j \)-th variables of sort \( i \) is denoted \( v_j^i \). There are the propositional connectives \( \wedge, \vee, \rightarrow, \leftrightarrow \), and \( \neg \), and quantifiers \( \exists^i \) and \( \forall^i \) for each sort \( i \).
The definitions of the syntax of terms and general formulas is exactly the same as free logic; the only change is in the atomic formulas. Unlike free logic atomic formulas, a total logic atomic formula does not have \( \approx, \downarrow \) or \( \uparrow \) as possible symbols.

Formally, an atomic formula \( \alpha \) is

1. \( t_1 =_i t_2 \) where \( i \in I \) and the sort of \( t_1 \) and \( t_2 \) is \( i \),

2. \( t_1 \neq_i t_2 \) where \( i \in I \) and the sort of \( t_1 \) and \( t_2 \) is \( i \), or

3. \( R^{i_1,\ldots,i_n}_{j}(t_1,\ldots,t_n) \) where each \( i_j \in I \) and the sorts of each \( t_i \) is \( i_j \).

### 4.4 Semantics of Many-Sorted Total Logic

A model \( M \) for a given \( \Sigma \)-signature has the following components, for each \( i \in I \) a nonempty domain \( M^i \), for each relation symbol \( R^{i_1,\ldots,i_n}_{j} \) a predicate \( R^{i_1,\ldots,i_n}_{M,j} \) which is a subset of \( i_1 \times \cdots \times i_n \), for each constant symbol, \( C^i_j \) a value \( C^i_j \) in \( M^i \), and for each function symbol \( F^{i_1,\ldots,i_n;i}_{j} \) a total function \( F^{i_1,\ldots,i_n;i}_{M,j} \) from \( M^{i_1} \times \cdots \times M^{i_n} \to i \).

We fix the interpretation of the equality symbols in the formal semantics.

Again, we let \( Vars \) be the set of all variable symbols, and \( Terms \) be the set of all well-formed terms. Given an interpretation \( s : \cup_{i \in I} Var^i \to \cup_{i \in I} M^i \) such that for any \( i \in I \) and \( s(v^i_j) \in M^i \), we define \( M \models_{\Sigma} \theta[s] \).

First, we define \( \bar{s} : Terms \to \cup_{i \in I} M^i \) by induction on the structure of terms. So,

\[
\bar{s}(t) = \begin{cases} 
  s(v^i_j) & \text{if } t \text{ equals } v^i_j \\
  C(v^i_j) & \text{if } t \text{ equals } C^i_j \\
  F^{i_1,\ldots,i_n;i}_{M,j}(\bar{s}(t_1),\ldots,\bar{s}(t_n)) & \text{if } t \text{ equals } F^{i_1,\ldots,i_n;i}_{j}(t_1,\ldots,t_n)
\end{cases}
\]

We note here that all terms are always defined, since all functions are assumed to be total.
Next, we define $M \models_{\Sigma} \alpha[s]$ where $\alpha$ is an atomic formula in Figure 4.8 in a similar manner to free logic.

1. $M \models_{\Sigma} t_1 = t_2[s]$ iff $\pi(t_1)$ equals $\pi(t_2)$
2. $M \models_{\Sigma} t_1 \neq t_2[s]$ iff $\pi(t_1)$ does not equal $\pi(t_2)$
3. $M \models_{\Sigma} R_{i_1 \cdots i_n}(t_1, \ldots, t_n)[s]$ iff $(\pi(t_1), \ldots, \pi(t_n)) \in R_{i_1 \cdots i_n}^M$

Figure 4.8: Definition of $\models$ for Total Logic Atomic Formulas

Finally, we define $M \models_{\Sigma} \theta[s]$ where $\theta$ is a formula in Figure 4.9. This is exactly the same semantic setup as the one for free logic.

1. $M \models_{\Sigma} \alpha[s]$ is determined by the above rules
2. $M \models_{\Sigma} \neg \theta[s]$ iff $M \models_{\Sigma} \theta[s]$ is false
3. $M \models_{\Sigma} \theta_1 \land \theta_2[s]$ iff $M \models_{\Sigma} \theta_1[s]$ and $M \models_{\Sigma} \theta_2[s]$
4. $M \models_{\Sigma} \theta_1 \lor \theta_2[s]$ iff $M \models_{\Sigma} \theta_1[s]$ or $M \models_{\Sigma} \theta_2[s]$
5. $M \models_{\Sigma} \theta_1 \rightarrow \theta_2[s]$ iff $M \models_{\Sigma} \theta_1[s]$ is false or $M \models_{\Sigma} \theta_2[s]$
6. $M \models_{\Sigma} \theta_1 \leftarrow \theta_2[s]$ iff $(M \models_{\Sigma} \theta_1[s]$ is false or $M \models_{\Sigma} \theta_2[s]$) and $(M \models_{\Sigma} \theta_1[s]$ or $M \models_{\Sigma} \theta_2[s]$ is false)
7. $M \models_{\Sigma} \forall v^i_j. \theta[s]$ iff for any $d$ in $M^i$, $M \models_{\Sigma} \theta[s(v^i_j \leftarrow d)]$
8. $M \models_{\Sigma} \exists v^i_j. \theta[s]$ iff there is $d$ in $M^i$, $M \models_{\Sigma} \theta[s(v^i_j \leftarrow d)]$

Figure 4.9: Definition of $\models$ for Total Logic General Formulas

Notationally, $M \models_{\Sigma} \theta$ is for any interpretation $s$ $M \models_{\Sigma} \theta[s]$
4.5 Logical Notions Independent of the choice of Free or Total Logic

In this section, we define the notion of free variables of a formula, \textit{i.e.}, those variables that are not bound by a quantifier, as defined by Enderton [15]. We first create a function to determine the variables used in a term, and then extend it to a general formula.

Let the function $\text{Free} : \text{Terms} \rightarrow \bigcup_{i \in I} \text{Vars}^i$ be defined as follows:

$$\text{Free}(t) = \begin{cases} \{v^i_j\} & \text{if } t \text{ equals } v^i_j \\ \{\} & \text{if } t \text{ equals } C^i_j \\ (\text{Free}(t_1) \cup \cdots \cup \text{Free}(t_n)) & \text{if } t \text{ equals } F^{i_1,\ldots,i_n}(t_1,\ldots,t_n) \end{cases}$$

Let the function $\text{Free} : \text{Formulas} \rightarrow \text{Set(Vars)}$ be defined as follows:

$$\text{Free}(\theta) = \begin{cases} \text{Free}(\sigma) & \text{if } \theta \text{ is } \neg \sigma \\ \text{Free}(\theta_1) \cup \text{Free}(\theta_2) & \text{if } \theta \text{ is } \theta_1 \land \theta_2 \\ \text{Free}(\theta_1) \cup \text{Free}(\theta_2) & \text{if } \theta \text{ is } \theta_1 \lor \theta_2 \\ \text{Free}(\theta_1) \cup \text{Free}(\theta_2) & \text{if } \theta \text{ is } \theta_1 \rightarrow \theta_2 \\ \text{Free}(\theta_1) \cup \text{Free}(\theta_2) & \text{if } \theta \text{ is } \theta_1 \leftarrow \theta_2 \\ \text{Free}(\theta_1) \setminus \{v^i_j\} & \text{if } \theta \text{ is } \forall v^i_j. \theta_1 \\ \text{Free}(\theta_1) \setminus \{v^i_j\} & \text{if } \theta \text{ is } \exists v^i_j. \theta_1 \end{cases}$$

4.6 Mathematical Machinery Relating Many-Sorted Free and Total Logic

We now turn our attention towards the relationship between these two logics; in particular, we are interested in total signature and models that "conform" to a
given a free logic signature and model. This analysis is needed before we can create
translation rules. Our first observation is that any $\Sigma$-signature for free logic is also
a signature for total logic. The only place where there is a substantive difference in
semantics is in the semantics of function and relation symbols.

4.6.1 Some Kinds of Total Interpretations of Partial Functions and Relations

Definition Given a partial function $F : D_1 \times \cdots \times D_n \rightarrow R$, a total function
$F' : D_1 \times \cdots \times D_n \rightarrow R$ is admissible for $F$ iff for any $d_1 \in D_1, \ldots, d_n \in D_n$ if
$F(d_1, \ldots, d_n)$ is defined, then $F(d_1, \ldots, d_n) = F'(d_1, \ldots, d_n)$.

With this definition, $F'$ may send any of $F$’s undefined parameters to any value. This
fact makes it hard to determine where $F$ is undefined, given $F'$. We next add a new
value to preserve this information.

Definition Given a partial function $F : D_1 \times \cdots \times D_n \rightarrow R$, a total function
$F'' : D_1 \times \cdots \times D_n \rightarrow R \cup \{\perp\}$, where $\perp \notin R$, is faithfully admissible for $F$ iff $F$ is admissible for $F$ and for any $d_1 \in D_1, \ldots, d_n \in D_n$ if $F(d_1, \ldots, d_n)$ is undefined
then $F''(d_1, \ldots, d_n) = \perp$.

A faithfully admissible total function for a given partial function adds an extra element
to the range and sends any of $F$’s undefined parameters to that extra element. We
now prove that it is unique.

Lemma 4.6.1. Given a partial function $F$, a faithfully admissible function $F''$ exists
and is unique.
Proof. First we tackle the existential part; given \( F \), let \( F'' = F \cup \{(d_1, \ldots, d_n, \perp) \mid \uparrow F(d_1, \ldots, d_n)\} \). Clearly \( F'' \) is a function and satisfies the properties.

Assume that \( F : D_1 \times \cdots \times D_n \rightarrow R \) is a partial function. Assume that \( F' \) and \( F'' \) are two faithful admissible functions for \( F \). We show \( F' = F'' \) by showing that for any \( d_1 \in D_1, \ldots, d_n \in D_n \). \( F'(d_1, \ldots, d_n) = F''(d_1, \ldots, d_n) \).

Let \( d_1, \ldots, d_n \) be arbitrary domain elements. There are two cases:

1. \( F(d_1, \ldots, d_n) \) is defined. In this case, \( F'(d_1, \ldots, t_n) = F(d_1, \ldots, d_n) = F''(d_1, \ldots, d_n) \) by definition.

2. \( F(d_1, \ldots, d_n) \) is not defined. In this case, \( F'(d_1, \ldots, d_n) = \perp \) by definition.

Similarly, \( F''(d_1, \ldots, d_n) = \perp \), therefore \( F'(d_1, \ldots, d_n) = F''(d_1, \ldots, d_n) \).

In all cases, \( F'(d_1, \ldots, d_n) = F''(d_1, \ldots, d_n) \), therefore \( F' = F'' \) and faithfully admissible functions of \( F \) are unique. ■

Our next observation is that faithful admissible function of \( F \) preserves undefinedness.

Lemma 4.6.2. For all partial functions \( F : D_1 \times \cdots \times D_n \rightarrow R \) and the faithfully admissible function \( F'' \) for \( F \),

\[
F''(d_1, \ldots, d_n) = \perp \iff F(d_1, \ldots, d_n) \text{ is undefined.}
\]

Also, \( F''(d_1, \ldots, d_n) = F(d_1, \ldots, d_n) \) iff \( F(d_1, \ldots, d_n) \) is defined.

Proof. Assume that \( F : D_1 \times \cdots \times D_n \rightarrow R \) is a partial function and \( d_1, \ldots, d_n \) are values.

Case \( F(d_1, \ldots, d_n) \) is undefined. Then by definition of \( F'' \),

\[
F''(d_1, \ldots, d_n) = \perp
\]
Our next observation extends the notion of faithfully admissible to allow for the composition of these total functions. The issue is that the domains of faithfully admissible functions are assumed to be exactly the same as those for the partial functions, however if we start composing these faithfully admissible functions we could end up in trouble. Consider a case of two functions $f$ and $g$, where $g(0)$ is undefined. If we try to rewrite $f(g(0))$ using the corresponding faithfully admissible functions, the domains do not match. We remedy this with yet one more definition.

**Definition** Given a partial function $F: D_1 \times \cdots \times D_n \rightarrow R$, a total function $F'': (D_1 \cup \{\perp_1\}) \times \cdots \times (D_n \cup \{\perp_n\}) \rightarrow R \cup \perp$ where each $\perp_i$ is not a member of each $D_i$ and $\perp$ is not a member of $R$, is an extended faithful admissible function iff $F'''|_{D_1 \times \cdots \times D_N}$ is faithfully admissible for $F$ and $F'''(d_1, \ldots, d_n) = \perp_i$ when some $d_i = \perp_i$.

**Lemma 4.6.3.** Given $F$ a partially defined function, there is an extended faithfully admissible function $F'''$ and it is unique.

**Proof.** First we tackle the existential part; given $F$, let $F'' = F'' \cup \{(d_1, \ldots, d_n, \perp) | \exists i. \uparrow d_i\}$. Clearly $F''$ is a function and satisfies the properties.

Assume that $F: D_1 \times \cdots \times D_n \rightarrow R$ is a partial function. Assume that $F'''$ and $F'''$ are two extended faithful admissible functions for $F$. We show $F''' = F'''$ by showing for any $d_1, \ldots, d_n$. $F'''(d_1, \ldots, d_n) = F''''(d_1, \ldots, d_n)$. Fix $d_1, \ldots, d_n$. There are three cases:
1. All components of $d_1, \ldots, d_n$ are not $\perp$ and $F(d_1, \ldots, d_n)$ is defined. In this case, $F''''(d_1, \ldots, d_n) = F(d_1, \ldots, d_n) = F''''(d_1, \ldots, d_n)$ by definition.

2. All components of $d_1, \ldots, d_n$ are not $\perp$ and $F(d_1, \ldots, d_n)$ is not defined. In this case, $F''''(d_1, \ldots, d_n) = \perp$ by definition. Similarly, $F''''(d_1, \ldots, d_n) = \perp$, therefore $F''''(d_1, \ldots, d_n) = F''''(d_1, \ldots, d_n)$.

3. There is some $i \in \{1, \ldots, n\}$ such that $d_i = \perp^i$. In this case, $F''''(d_1, \ldots, d_n) = \perp$ by definition. Similarly, $F''''(d_1, \ldots, d_n) = \perp$, therefore $F''''(d_1, \ldots, d_n) = F''''(d_1, \ldots, d_n)$.

In all cases, $F''''(d_1, \ldots, d_n) = F''''(d_1, \ldots, d_n)$, therefore $F'''' = F''''$ and faithful interpretations are unique. ■

**Lemma 4.6.4.** For all partial functions $F : D_1 \times \cdots \times D_n \rightarrow R$, for the extended admissible function $F''''$, $F''''(d_1, \ldots, d_n) = \perp$ iff $F(d_1, \ldots, d_n)$ is undefined. Also, $F''''(d_1, \ldots, d_n) = F(d_1, \ldots, d_n)$ iff $F(d_1, \ldots, d_n)$ is defined.

**Proof.** Assume that $F : D_1 \times \cdots \times D_n \rightarrow R$ is a partial function and $F''''$ is the extended admissible function associated with it. Assume that $d_1, \ldots, d_n$ are within the domain of $F$.

First assume that $F(d_1, \ldots, d_n)$ is undefined, then by definition of $F''''$, $F''''(d_1, \ldots, d_n) = \perp$.

Lastly assume that $F(d_1, \ldots, d_n)$ is defined, then by definition of $F''''$, $F''''(d_1, \ldots, d_n) = F(d_1, \ldots, d_n)$. ■

Next we examine the requisite extensions for relations. We need only extend the domains to handle $\perp^i$ for each domain $D_i$. 132
Definition. Given a partially defined relation $R$ with domain $D_1 \times \cdots \times D_n$, a relation $R'$ with domain $D_1 \cup \{ \bot_1 \} \times \cdots \times D_n \cup \{ \bot^n \}$ is a total extension of $R$ if for any $d_1 \in D_1, \ldots, d_n \in D_n$, $R(d_1, \ldots, d_n) = R'(d_1, \ldots, d_n)$ and if there is a $d_i = \bot_i$, then $R'(d_1, \ldots, d_n)$ is false.

Lemma 4.6.5. For any relation $R$, there is a total extension $R'$ and it is unique.

Proof. Let $R' = R$. Then it satisfies the properties.

Fix $R$, and assume that both $R'$ and $R''$ are total extensions of $R$.

For any parameters, either $R'$ and $R''$ have the same value as $R$ or are false (when given a $\bot_i$). ■

Now that the infrastructure has been built up to examine a given partial function and produce a unique total function that captures the partial functions intuitive semantics, we move to the translation theorem.

4.7 Relationship Between Free and Total Many-Sorted Logics

Theorem 4.7.1 (Equivalence of free and total many-sorted first-order logic). For any $\Sigma$-signature in many-sorted free logic where one constant per sort is not used, and any model $M$, there is a $\Sigma'$-signature, a model $M'$ and operator $\prime: M \rightarrow M'$ such that for any formula $\theta$, $M \models_{\Sigma} \theta$ in free logic if and only if $M' \models_{\Sigma'} \theta'$ in total logic.

First, we state a lemma used in the proof of the this theorem.

Lemma 4.7.2 (Construction of $\Sigma'$, $M'$, and $\prime$). For any $\Sigma$-signature in many-sorted free logic where one constant per sort is not used, and any model $M$, there is a $\Sigma'$-signature, a model $M'$ and operator $\prime: M \rightarrow M'$ such that for any
formula $\theta$ and any interpretation $s$ for many-sorted free logic, $M \models_{\Sigma} \theta[s]$ in free logic if and only if $M' \models_{\Sigma'} \theta''[s]$ in total logic.

We defer the proof of this lemma until after the proof of the main theorem.

Proof of the equivalence of free and total many-sorted first-order logic. Our first order of business is to find the required model, signature and formulas. We obtain $\Sigma'$, $M'$ and $''$ from lemma 4.7.2. We now create $'$ directly.

First, given a finite set of variables, we create a formula that ensures that any interpretation $s$ that interprets any variable to $\bot$ to be false.

Let $\text{fix} : \text{Set}(\text{Vars}) \rightarrow \text{Formula}$ be defined by induction on the cardinality of the finite set.

$$\text{fix}(s) = \begin{cases} \text{True} & \text{if } s = \{\} \\ v^i_j \neq \bot^i \land \text{fix}(t) & \text{if } v^i_j \notin t \land s = \{v^i_j\} \cup t \end{cases}$$

Now we define the $'$ operator as $\theta' = \text{fix}(\text{free}(\theta'')) \rightarrow (\theta'').$

It remains to be shown that $M \models_{\Sigma} \theta$ in free logic if and only if $M' \models_{\Sigma'} \theta'$ in total logic.

1. Assume $M \models_{\Sigma} \theta$ is true. We must show $M' \models_{\Sigma'} \theta'$ is true; equivalently, we must show for any $s$, $M' \models_{\Sigma'} \theta'[s]$ is true. Now, by assumption, we know that for any interpretation $s$ where the range of $s$ is the union of all $M'$s, $M \models_{\Sigma} \theta[s]$ is true. Fix $s'$. We have two cases.

- First, if $s'(v^i_j) \neq \bot^i$ for all free variables $v^i_j$, then we can choose some $s$ such that $s = s'$. By lemma 4.7.2, we have $M' \models_{\Sigma'} \theta''[s]$ is true and $M' \models_{\Sigma'} \text{fix}(\text{free}(\theta'')) \rightarrow \theta''[s]$ is true from the semantics of total many-sorted first-order logic. Therefore $M' \models_{\Sigma'} \theta'[s]$ is also true.
• Otherwise, \( s' \) assigns \( \bot^i \) to some variable \( v^j_i \). However, in this case, \( M' \models_{\Sigma'} \theta'[s'] \) is true, as \( M' \models_{\Sigma'} fix(free(\theta'')) \) is false.

Therefore, we have \( M \models_{\Sigma} \theta \) implies \( M' \models_{\Sigma'} \theta' \).

2. Assume \( M \models_{\Sigma} \theta \) is false. We must show \( M' \models_{\Sigma'} \theta' \) is false; equivalently, we must show there is an \( s' \) such that \( M' \models_{\Sigma'} \theta''[s'] \) is false. Now, by assumption, there is an \( s \) where \( M \models_{\Sigma} \theta[s] \) is false. Take that \( s \), we note that no variables are assigned to \( \bot^i \) by definition, therefore \( M' \models_{\Sigma'} fix(free(\theta'')) \) is true. By lemma 4.7.2, we know \( M' \models_{\Sigma'} \theta''[s] \) is false, and therefore, \( M' \models_{\Sigma'} fix(free(\theta'')) \rightarrow \theta''[s] \) is false and \( M' \models_{\Sigma'} \theta'[s] \) is false as well. Therefore, \( M \models_{\Sigma} \theta \) is false implies \( M' \models_{\Sigma'} \theta' \) is false, completing the theorem. ■

### 4.7.1 Proof of Lemma

We first must create definitions of \( \Sigma', M' \) and \( '' \) and then show that they have the requisite properties.

First, without loss of generality assume the first constant in each sort is the unused constant.

**Construction of \( \Sigma' \)**

We construct \( \Sigma' \) from \( \Sigma \), by adding new constants to \( \Sigma \), one for each \( i \in I \), as the first constant \( C^i_0 \). We will denote this constant by \( \bot^i \), to more represent its intended interpretation.

**Construction of \( M' \)**

Second, we construct \( M' \) from \( M \) by taking, for each \( i \in I \), \( M'^i \) to be \( M^i \cup \{ \bot^i \} \) where \( \bot^i \) is not a member of \( M^i \). For example, if \( M^{sets} \) is sets of objects, and \( M^{obj} \)
is the real numbers, then $M'_{\text{sets}} = M_{\text{sets}} \cup \{ \perp_{\text{sets}} \}$, not the set of all objects and $\perp_{\text{obj}}$ union $\{ \perp_{\text{sets}} \}$. Each constant and relation definition is in $M'$ if it is a part of $M$. We interpret the constant $C^i_0$ as $\perp^i$.

For every function definition $F_{1,\ldots,i_n}^{i_1,\ldots,i_n}$ in $M$, $F_{1,\ldots,i_n}^{i_1,\ldots,i_n}_{M'}$ is the extended faithful admissible function for $F_{1,\ldots,i_n}^{i_1,\ldots,i_n}$. For every relation definition $R_{1,\ldots,i_n}$, $R_{1,\ldots,i_n}^{i_1,\ldots,i_n}$ is the total extension of $R$.

Construction of $''$ operator

We first define a $'$ operator that maps terms to terms. We define this operator by induction on the structure of terms.

Definition

$$t' = \begin{cases} 
  v^j_i & \text{if } t \equiv (v^j_i)' \\
  c^j_i & \text{if } t \equiv (c^j_i)' \\
  F_{1,\ldots,i_n}^{i_1,\ldots,i_n}(t^M_1,\ldots,t^M_n) & \text{if } t \equiv F_{1,\ldots,i_n}^{i_1,\ldots,i_n}(t_1,\ldots,t_n)
\end{cases}$$

We now turn our attention to atomic formulas $\alpha$; we define $'$ directly on atomic formulas.

Definition

$$\alpha'' = \begin{cases} 
  (t_1)' = (t_2)' \land (t_1)' \neq \perp \land (t_2)' \neq \perp & \text{if } \alpha \equiv t_1 = t_2 \\
  (t_1)' \neq (t_2)' \land (t_1)' \neq \perp \land (t_2)' \neq \perp & \text{if } \alpha \equiv t_1 \neq t_2 \\
  (t_1)' = (t_2)' & \text{if } \alpha \equiv t_1 \approx t_2 \\
  t' \neq \perp & \text{if } \alpha \equiv \downarrow t \\
  t' = \perp & \text{if } \alpha \equiv \uparrow t \\
  R_{1,\ldots,i_n}(t^M_1,\ldots,t^M_n) & \text{if } \alpha \equiv R_{1,\ldots,i_n}(t_1,\ldots,t_n)
\end{cases}$$

We define $'$ on general formulas by induction on the structure of the formula.
Definition

\[
\theta'' = \begin{cases} 
\alpha'' & \text{if } \theta \equiv \alpha \\
-\theta_1'' & \text{if } \theta \equiv -\theta_1 \\
\theta_1'' \land \theta_2'' & \text{if } \theta \equiv \theta_1 \land \theta_2 \\
\theta_1'' \lor \theta_2'' & \text{if } \theta \equiv \theta_1 \lor \theta_2 \\
\theta_1'' \rightarrow \theta_2'' & \text{if } \theta \equiv \theta_1 \rightarrow \theta_2 \\
\forall x. x \neq \bot^i \theta_1'' & \text{if } \theta \equiv \forall x. \theta_1 \\
\exists x. x \neq \bot^i \land \theta_1'' & \text{if } \theta \equiv \exists x. \theta_1 
\end{cases}
\]

Finally, we extend ' to take a model statement in the intuitive way; we apply the ' operator to all arguments (except the formula, there we apply the " operator).

Definition \((M \models_{\Sigma} \theta)' \overset{\text{def}}{=} M' \models_{\Sigma'} \theta''\)

Proof that \(\Sigma', M', \) and " satisfy the properties in the theorem

Proof. We proceed by induction on the structure of a formula \(\theta\). We fix an interpretation \(s\) for \(M \models_{\Sigma} \theta\). We note that we need \((M \models_{\Sigma} \theta)[s] \equiv (M \models_{\Sigma} \theta)'[s]\), and that by definition \((M \models_{\Sigma} \theta)'[s] \equiv M' \models_{\Sigma'} \theta''[s]\).

1. \(M \models_{\Sigma} \alpha[s]\). We proceed by case analysis:

(a) Case: \(M \models_{\Sigma} t_1 =_i t_2[s]\). Therefore the corresponding atomic formula in \(M'\) is \((t_1)^T =_i (t_2)^T \land (t_1)^T \neq _i \bot^i \land (t_2)^T \neq _i \bot^i\) We proceed by assuming \(M \models_{\Sigma} t_1 =_i t_2[s]\) is true. From the semantics of many-sorted free logic, we know that \(t_1\) and \(t_2\) are defined, and \(\pi(t_1) =_i \pi(t_2)\). By lemma 4.6.4, we know \(\pi(t_1)^T =_i \pi(t_2)^T\), \(t_1' \neq _i \bot^i\) and \(t_2' \neq _i \bot^i\). Therefore\(M' \models_{\Sigma'} (t_1)^T =_i (t_2)^T \land (t_1)^T \neq _i \bot^i \land (t_2)^T \neq _i \bot^i[s]\) by definition.

(b) Case: \(M \models_{\Sigma} t_1 =_i t_2[s]\) is false. Now, there are two cases to consider.

- First, consider if \(t_1\) or \(t_2\) is undefined. In that case, by lemma 4.6.4, either \(t_1'\) or \(t_2'\) evaluates to \(\bot^i\). In that case,
\[ M' \models \Sigma' (t_1)^T = t_2^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i[s] \text{ is false.} \]

- Second, both \( t_1 \) and \( t_2 \) are defined, but \( \overline{\exists}(t_1) \) does not equal \( \overline{\exists}(t_2) \). By lemma 4.6.4, this means \( \overline{\exists}((t_1)^T) \) does not equal \( \overline{\exists}((t_2)^T) \). In that case, \( M' \models \Sigma' (t_1)^T = (t_2)^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i[s] \) is false.

(c) Case: \( M \models t_1 \neq_i t_2[s] \). Therefore the corresponding atomic formula in \( M' \) is \( M' \models \Sigma' (t_1)^T = (t_2)^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i \). We proceed by assuming \( M \models t_1 \neq_i t_2[s] \) is true. From the semantics of many-sorted free logic, we know that both \( t_1 \) and \( t_2 \) are defined, and \( \overline{\exists}(t_1) \neq_i \overline{\exists}(t_2) \). By lemma 4.6.4, we know \( \overline{\exists}((t_1)^T) \neq_i \overline{\exists}((t_2)^T) \), \( t'_1 \neq_i \bot^i \) and \( t'_2 \neq_i \bot^i \). Therefore \( M' \models \Sigma' (t_1)^T \neq_i (t_2)^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i[s] \) by definition.

(d) Case: \( M \models t_1 \neq_i t_2[s] \) is false. Now, there are two cases to consider.

- First, consider if \( t_1 \) or \( t_2 \) is undefined. In that case, by lemma, either \( t'_1 \) or \( t'_2 \) evaluates to \( \bot^i \). In that case, \( M' \models \Sigma' (t_1)^T \neq_i (t_2)^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i[s] \) is false because one of the last two conjuncts is false.

- Second, both \( t_1 \) and \( t_2 \) are defined, but \( \overline{\exists}(t_1) \) equals \( \overline{\exists}(t_2) \). By lemma 4.6.4, this means \( \overline{\exists}((t_1)^T) \) equals \( \overline{\exists}((t_2)^T) \). In that case, \( M' \models \Sigma' (t_1)^T = (t_2)^T \wedge (t_1)^T \neq_i \bot^i \wedge (t_2)^T \neq_i \bot_i[s] \) is false because the first conjunct is false.
(e) Case: $M \models t_1 \approx_i t_2[s]$. Then $\bar{s}(t_1)$ equals $\bar{s}(t_2)$. By lemma, this means $\bar{s}((t_1)^T)$ equals $\bar{s}(t_2^T)$. In that case, $M' \models (t_1)^T =_i (t_2)^T[s]$.

(f) Case: $M \models t_1 \approx_i t_2[s]$ is false. Then $\bar{s}(t_1)$ does not equal $\bar{s}(t_2)$. By lemma, this means $\bar{s}(t_1^T)$ does not equal $\bar{s}(t_2^T)$. In that case, $M' \models (t_1)^T =_i (t_2)^T[s]$ is false.

(g) Case: $M \models R^1_{j_1\ldots j_n}(t_1, \ldots, t_n)[s]$. In this case, we know that $M' \models R^1_{j_1\ldots j_n}(t_1^T, \ldots, t_n^T)[s]$ by definition.

(h) Case: $M \models R^1_{j_1\ldots j_n}(t_1, \ldots, t_n)[s]$ is false. Now there are two cases:
   
   i. All $t_i \neq \bot_i$. Then, we know that $M' \models R^1_{j_1\ldots j_n}((t_1)^T, \ldots, (t_n)^T)[s]$ is false.

   ii. Some $t_i = \bot$. Then, we know that $M' \models R^1_{j_1\ldots j_n}((t_1)^T, \ldots, (t_n)^T)[s]$ is false.

2. Case: $M \models \neg \theta[s]$. By induction hypothesis, we have $M \models \theta[s]$ if and only if $M' \models \theta'[s]$, therefore we have, by definition of $M'$, $\Sigma'$, and $\theta'$, $M \models \neg \theta[s]$ if and only if it is false that $M \models \theta[s]$ if and only if it is false that $M' \models \theta'[s]$ if and only if $M' \models \neg \theta'[s]$ if and only if $M' \models (\neg \theta)[s]$.

3. Case: $M \models \theta_1 \land \theta_2[s]$. By induction hypothesis, we have $M \models \theta_1[s]$ if and only if $M' \models \theta_1'[s]$ and $M \models \theta_2[s]$ if and only if $M' \models \theta_2'[s]$. By definition of the semantics of many-sorted total logic, we have $M \models \theta_1 \land \theta_2[s]$ if and only if $M' \models \theta_1' \land \theta_2'[s]$ if and only if $M' \models (\theta_1 \land \theta_2)[s]$.

4. Case: $M \models \theta_1 \lor \theta_2[s]$. By induction hypothesis, we have $M \models \theta_1[s]$ if and only if $M' \models \theta_1'[s]$ and $M \models \theta_2[s]$ if and only if $M' \models \theta_2'[s]$. By definition
of the semantics of many-sorted total logic, we have $M \models_{\Sigma} \theta_1 \lor \theta_2[s]$ if and only if $M' \models_{\Sigma'} \theta_1'' \lor \theta_2''[s]$ if and only if $M' \models_{\Sigma'} (\theta_1 \lor \theta_2)''[s]$.

5. Case: $M \models_{\Sigma} \theta_1 \rightarrow \theta_2[s]$. By induction hypothesis, we have $M \models_{\Sigma} \theta_1[s]$ if and only if $M' \models_{\Sigma'} \theta_1'[s]$ and $M \models_{\Sigma} \theta_2[s]$ if and only if $M' \models_{\Sigma'} \theta_2'[s]$. By definition of the semantics of many-sorted total logic, we have $M \models_{\Sigma} \theta_1 \rightarrow \theta_2[s]$ if and only if $M' \models_{\Sigma'} \theta_1'' \rightarrow \theta_2''[s]$ if and only if $M' \models_{\Sigma'} (\theta_1 \rightarrow \theta_2)''[s]$.

6. Case: $M \models_{\Sigma} \theta_1 \leftarrow \theta_2[s]$. By induction hypothesis, we have $M \models_{\Sigma} \theta_1[s]$ if and only if $M' \models_{\Sigma'} \theta_1'[s]$ and $M \models_{\Sigma} \theta_2[s]$ if and only if $M' \models_{\Sigma'} \theta_2'[s]$. By definition of the semantics of many-sorted total logic, we have $M \models_{\Sigma} \theta_1 \leftarrow \theta_2[s]$ if and only if $M' \models_{\Sigma'} \theta_1'' \leftarrow \theta_2''[s]$ if and only if $M' \models_{\Sigma'} (\theta_1 \leftarrow \theta_2)''[s]$.

7. Case: $M \models_{\Sigma} \forall^i x. \theta[s]$. By induction hypothesis, we have for all interpretations $s$, $M \models_{\Sigma} \theta[s]$ if and only if $M' \models_{\Sigma'} \theta''[s]$. Now, by the semantics of many-sorted free logic, $M \models_{\Sigma} \forall^i x. \theta[s]$ if and only if for any variable assignment $s'$ and any $d \in M^i$ $M \models_{\Sigma} \theta[s'(x \leftarrow d)]$ if and only if for any variable assignment $s$ and any $d \in M^i$ where $d \neq \bot^i$ $M' \models_{\Sigma'} \theta''[s'(x \leftarrow d)]$ if and only (by the semantics of many-sorted total logic), $M' \models_{\Sigma'} \forall^i x. x \neq \bot^i \rightarrow \theta''[s]$ if and only if $M' \models_{\Sigma'} (\forall^i x. \theta)''[s]$.

8. Case: $M \models_{\Sigma} \exists^i x. \theta$. By induction hypothesis, we have for all $s'$ $M \models_{\Sigma} \theta[s]$ if and only if $M' \models_{\Sigma'} \theta''[s']$. Now, by the semantics of many-sorted free logic, $M \models_{\Sigma} \exists^i x. \theta[s]$ if and only if there exists $d \in M^i$ such that $M \models_{\Sigma} \theta[s(x \leftarrow d)]$ if and only if there exists $d \in M^i$ such that $d \neq \bot^i \land M' \models_{\Sigma'} \theta''[s(x \leftarrow d)]$ if and only (by the semantics of many-sorted total logic), $M' \models_{\Sigma'} \exists^i x. x \neq \bot^i \land \theta''[s]$ if and only if $M' \models_{\Sigma'} (\exists^i x. \theta)''[s]$.■
4.8 Practical Use of the Method

In this section, we implement this translation method within Isabelle specifically in the context of translating free first-order formulas over objects which include the reals, rationals, integers and naturals along with special values such as infinity and many other non-number values, to total first-order formulas over the same set of objects.

We apply this method in the context of translating automatically between an instruction-friendly free-logic framework used in teaching discrete mathematics and logic to sophomore/junior level undergraduates and a total-logic framework that is processable by an automated theorem prover. This project is called Syrus and the goal is for students to achieve perfect practice on a virtually unlimited supply of logic problems. Syrus would ensure that students would only perform valid inference steps, thus ensuring perfect practice. Currently students examine a mathematical sentence and answer true or false. The software then tells the student if they were correct. Since the goal is for students to be able to practice their skills \textit{ad infinitum}, the sheer number of requisite sentences forces the tool to both generate and prove/refute the sentences automatically. The free-logic model also include fewer sorts than the automated prover does, \textit{e.g.}, as in most mathematics courses, the natural numbers are treated as a subset of the integers—they are not treated as a distinct sort.

4.8.1 Definition of Logical Signature

The logical signature of the many-sorted free-logic language is as shown in Figure 4.10. Briefly, the signature includes objects, sets and many of the expected functions on both sets and objects, \textit{e.g.}, union and addition.
Sorts:

1. Obj
2. Set

Constants

1. $\emptyset, N, Z, Q, R$, of Sort Set
2. All base-10 integer constants, and $\infty$, of sort Obj

Relations

1. $<$, $>$, $\leq$, $\geq$, of sort Obj $\times$ Obj
2. $\subseteq$, $\supseteq$, of sort Set $\times$ Set
3. $\in$, $\notin$, of Sort Obj $\times$ Set

Functions

1. $+$, $-$, $\ast$, $/$, DIV, MOD, $\ast$ of sort Obj $\times$ Obj $\rightarrow$ Obj
2. $-$ and $\mid\_\mid$, of sort Obj $\rightarrow$ Obj
3. min, max, of sort Obj $\times$ Obj $\rightarrow$ Obj
4. $\cup, \cap, \setminus$, of sort Set $\times$ Set $\rightarrow$ Set
5. $\{\ldots\}$: of sort Obj$^k$ $\rightarrow$ Set

Figure 4.10: Signature of the Many-Sorted Free Logic Under Translation
4.8.2 Intended Model

The intended model for Obj is a superset of the reals numbers. This set of objects also includes infinity and other objects that are not real numbers. The intended model for Set are subsets of the set of objects.

The intended interpretation of any of the arithmetic functions on objects is the indicated real number operation, e.g., for $+$ the operation is addition. If one of the arguments is not a real number, then the result of the function is undefined. The similar idea holds for any of the set operations: it denotes the indicated set operation unless one of the arguments is undefined—in that case the result is also undefined.

The intended interpretation of the set and object constants is exactly what one would expect, i.e., the interpretation of 0 is the natural number zero.

4.8.3 Model of Objects with Isabelle

Our model or universe is represented in Isabelle as $\mathbb{R} \times \mathbb{N}$. This gives us many more values than we actually need, so we represent the actual real numbers as those with the natural number 0 as the second component. We model $\infty$ with the value $(0, 1)$ and the undefined value $\bot_{\text{obj}}$ as $(0, 2)$. We define the result of the other operators ($+, \times, \div, -$) as $\bot$ if any of the arguments are undefined or are not real numbers, otherwise exactly what would be expected. The functions $\max$, $\min$, $\mod$ and $\div$ are all defined similarly. For the operators with limited ranges, e.g., $\div$, $\mod$, any places where the function would not be defined, like $x/0$ are defined to be $\bot$ as well.

A new equality operator $\hequals$ is used in place of $\equiv_{\text{obj}}$ and it is defined to be true only if both arguments are defined and equal. Other new functions are $\hnequals$ (for $\neq_{\text{obj}}$), $\hkleenequal$ (for $\approx_{\text{obj}}$), $\defined$ (for $\downarrow$) and $\undefined$ (for
hnequals returns true only if both arguments are defined, finite and not equal while hkleenequal returns true if either both arguments are undefined or both are equal. A value is defined if it is not equal to \( \bot^{\text{obj}} \). A value is undefined if it equals \( \bot^{\text{obj}} \). The objects 0, 1 and all integral constants are defined to be (0, 0), (1, 0), etc. Less than and Less than or equals are also defined, and only return true if both arguments are defined, numbers and are in the correct order.

Sets are also a sort, and the sets \( \mathbb{R}, \mathbb{Q}, \mathbb{Z} \) and \( \mathbb{N} \) are all defined as sets of objects. Sets of objects are a distinguished type with the symbol \( \bot \) also representing the undefined value in that sort as well. The model for these sets is an Isabelle set along with one of two values. One of those two values is used to represent \( \bot^{\text{set}} \), while the other is used for all actual sets with the constraint that \( \bot^{\text{obj}} \) never appears in the representation. The relation \( x \text{ setin } S \) holds if both \( x \) and \( S \) are defined, and \( x \) is actually in the representative set. The other set functions and relations are defined similarly.

### 4.8.4 Translation Rules

These translation rules are based on mostly one-way rewrite rules. There are many rules to take care of the cases that may occur. Rather than listing the rules, we will go through representative examples to show how the examples are simplified and to demonstrate many of the rules.

The example we use is the formula in Figure 4.11. It involves both universal quantification over objects and the use of set constants as “types.” This formula in the free-logic signature and model discussed before is true. If \( x \) and \( y \) are both real numbers, one can always find a real number that is equal to \( x \ast y \). We will now show
how the translation rules within Isabelle are applied to push the formula into a total logic formula that Isabelle can prove.

\[ \forall x, y : \text{obj} (x \in \mathbb{Z} \rightarrow \exists z : \text{obj} (x + y = \text{obj} z \lor (y \in \mathbb{R} \rightarrow x \times y = \text{obj} z))) \]

Figure 4.11: Example Formula in Free Logic

The first step is actually a pre-processing step. Each quantified variable gets a guard to make sure if it gets the value \( \bot \), that does not affect the formula. For example \( \forall x : \text{obj} \ P(x) \) is translated to \( \forall x : \text{obj} \ (x \neq \bot) \rightarrow P(x) \). Since the free-logic variables may not be assigned an undefined value, this translation preserves the semantics. Conversely, \( \exists x : \text{obj} \ P(x) \) is translated to \( \exists x : \text{obj} \ (x \neq \bot \land P(x)) \). This step mirrors the first step of the main translation theorem. Again, this translation preserves the semantics by not allowing \( x \) to be assigned \( \bot \). We also replace equality by the Isabelle representation of free-logic equality. The resulting formula is shown in Figure 4.12.

\[ \forall x, y : \text{obj} \ (x \neq \bot \land y \neq \bot) \rightarrow (x \in \mathbb{Z} \rightarrow \exists z : \text{obj} \ z \neq \bot \land ((\text{hequals}(x + y, z) \lor (y \in \mathbb{R} \rightarrow \text{hequals}(x \times y, z)))))) \]

Figure 4.12: Example Formula Given to Isabelle

Next, the rules inherited from Isabelle’s HOL rules remove the \( \forall \) and move the antecedents of the first implication to above the line. The first rule specific to the
translation is then applied. This class of rules, one for each of the constant sets, replaces assumptions of the form \( x \in S \) with \( \exists x^R : S(x = \text{obj}_o f _S(x^R)) \) where there are \( \text{obj}_o f _\) for Isabelle natural, integer, rational, and real types. Essentially, these assumptions act as types and are very common; the translation rules include special rules to handle them as an optimization. Also, we replace \( \text{hequal}s(s, t) \) by its definition, namely \( s = t \land s \neq \perp \land t \neq \perp \). The resulting proof form is in Figure 4.13.

\[
\begin{align*}
(x : \text{obj}) \neq \perp \\
(y : \text{obj}) \neq \perp \\
\exists x^R : \text{int} (x = \text{obj}_o f _\text{int}(x^R)) \\
\hline
\exists z : \text{obj} z \neq \perp \land ((x + y = z \land x + y \neq \perp \land z \neq \perp) \lor \\
(y \in \mathbb{R} \rightarrow (x \ast y = z \land x \ast y \neq \perp \land z \neq \bot)))
\end{align*}
\]

Figure 4.13: Example Formula Simplified

Our next proof step applied is to notice that there are two cases with an existential in the goal, satisfied by a real number or by one of the other objects. We split the \( \exists z : \text{obj} \) into the “or" of two existential formulas. Without loss of generality, we move the negation of one of the two into the assumptions. The resulting proof state is shown in Figure 4.14.

The next rules applied are from a class that replace obvious consistencies, \( \text{e.g., } \text{obj}_o f _\text{real}(x) \neq \perp \), with \text{true}. Isabelle infers that the \( y \in \mathbb{R} \) in the goal, since it does not mention \( z \), can be moved outside of the \( \exists \), to the assumptions and can have the previous type-like rules applied to the formula. The simplified proof state is shown in Figure 4.15. Now we have no more variables of the \text{obj} type.
(y : obj) \neq \bot
\forall z^N : \text{not\_real} \ is\_not\_real(z^N) = \bot \lor ((\text{obj\_of\_int}(x^R) + y \neq \text{is\_not\_real}(z^N)) \land
(y \in \mathbb{R} \land \text{obj\_of\_int}(x^R) \neq \text{is\_not\_real}(z^N)))
\exists z^R : \text{real} \ \text{obj\_of\_real}(z^R) \neq \bot \land
((\text{obj\_of\_int}(x^R) + y = \text{obj\_of\_real}(z^R)) \lor
(y \in \mathbb{R} \rightarrow \text{obj\_of\_int}(x^R) \neq \text{obj\_of\_real}(z^R)))

Figure 4.14: Example Formula After Existential Split

\forall z^N : \text{not\_real} \ is\_not\_real(z^N) = \bot \lor
((\text{obj\_of\_int}(x^R) + \text{obj\_of\_real}(y^R) \neq \text{is\_not\_real}(z^N)) \land
(\text{obj\_of\_real}(y^R) \in \mathbb{R} \land
\text{obj\_of\_int}(x^R) \neq \text{is\_not\_real}(z^N))))
\exists z^R : \text{real} \ ((\text{obj\_of\_int}(x^R) + \text{obj\_of\_real}(y^R) = \text{obj\_of\_real}(z^R)) \lor
\text{obj\_of\_int}(x^R) \neq \text{obj\_of\_real}(y^R) \neq \text{obj\_of\_real}(z^R)))

Figure 4.15: Example Formula With No More obj Variables
The next set of rules we apply modify formulas like \( \text{obj\_of\_real}(x) + \text{obj\_of\_int}(y) \) to look like \( \text{obj\_of\_real}(x + \text{of\_int}(y)) \). The rules try to avoid casting directly to Isabelle reals, so there are rules that handle any permutation of types and operations. The proof state after these rules is shown in Figure 4.16.

\[
\forall z^N: \text{not\_real} \text{ is\_not\_real}(z^N) = \bot \lor \\
(\text{obj\_of\_real}(\text{of\_int}(x^R) + y^R) \neq \text{is\_not\_real}(z^N) \land \\
\text{obj\_of\_real}(y^R) \in \mathbb{R} \land \text{obj\_of\_real}(\text{of\_int}(x^R) \ast y^R) \neq \text{is\_not\_real}(z^N)) \\
\exists z^R: \text{real} \ (\text{obj\_of\_real}(\text{of\_int}(x^R) + y^R) = \text{obj\_of\_real}(z^R) \lor \\
\text{obj\_of\_real}(\text{of\_int}(x^R) \ast y^R) = \text{obj\_of\_real}(z^R))
\]

Figure 4.16: Example Formula With obj\_of\_ * Simplified

Now we use more rules to deal with obvious facts involving \( \text{is\_not\_real} \) and \( \text{obj\_of\_} \)-variants. For example, it is always true that \( \text{obj\_of\_real}(x) \neq \text{is\_not\_real}(y) \), and that \( \text{obj\_of\_real}(y^R) \in \mathbb{R} \). The result after these rules are applied and obvious simplifications are done is shown in Figure 4.17.

\[
\exists z^R: \text{real} \ (\text{obj\_of\_real}(\text{of\_int}(x^R) + y^R) = \text{obj\_of\_real}(z^R) \lor \\
\text{obj\_of\_real}(\text{of\_int}(x^R) \ast y^R) = \text{obj\_of\_real}(z^R))
\]

Figure 4.17: Example Formula With Obvious Simplifications Applied

Finally, we use our last set of rules for this example, those that remove \( \text{obj\_of\_real} \) and the like. Once formulas look like \( \text{obj\_of\_real}(E_1) = \)
of real (E2), the only way that can be true is if \( E_1 = E_2 \). This is the essence of the rules, though they take care of all of the different permutations of \( obj\_of\_\) variants. The result of applying the rules is shown in Figure 4.18. This formula is now a native Isabelle formula which happens to be true and is proved easily.

\[
\exists z^R: \text{real} \ (\text{of} \ \text{int} (x^R) + y^R = z^R \lor \text{of} \ \text{int} (x^R) * y^R = z^R)
\]

Figure 4.18: Example Formula With obj_of_real Removed

Based on this example, we can see that there are a variety of rules, most to simplify the formula by removing some free-logic symbol and replacing it by a total logic symbol that Isabelle understands. If the formula is well-behaved, i.e., it is something that a teacher would ask of a student, then usually the typing information is carried along and potential corner cases simply drop out. Our next example demonstrates what happens if that is not the case, namely there is no typing information.

The example formula in Figure 4.19 is slightly subtle; it is true because if \( x \) or \( y \) is not a real number, the addition is undefined and therefore the equality is false.

\[
\exists x, y : \text{obj} \ \forall z, w : \text{obj} \ \neg ((x + y) =_{\text{obj}} (z / w))
\]

Figure 4.19: Example Formula Without Type Information
As before, when translated to Isabelle, the translator adds $\neq \text{bot}$ for all quantified variables, either in an implication or conjoined depending on the quantifier. The equality operator is also named to $\text{hequals}$ to distinguish it from Isabelle’s $=$ operator. After applying the rule to replace $\exists x : \text{obj}$, the proof state is shown in Figure 4.20 and we seem to be stuck.

\[
\forall x^n, y^n : \text{not\_real is\_not\_real}(x^n) = \bot \lor \text{is\_not\_real}(y^n) = \bot \lor \\
\exists z, w : \text{obj} z \neq \bot \land w \neq \text{bot} \land (\bot = z/w) \land \bot \neq \bot \land z/w \neq \bot \\
\exists x^r, y^r : \text{real} \forall z, w : \text{obj} z \neq \bot \land w \neq \text{bot} \rightarrow \neg (\text{obj\_of\_real}(x^r + y^r) = (z/w) \land z/w \neq \bot)
\]

Figure 4.20: Example Formula in Isabelle With Object Existential Replaced

The next proof steps involve the “big guns.” These rules are called split rules because they create case splits. Consider the division of any two objects, $o_1$ and $o_2$. The value of $o_1/o_2$ is one of two values: either it is a real number (because $o_1$ and $o_2$ are real and $o_2$ is not zero), or it is $\bot$. Actually, there are five cases, based on whether $o_1$ and $o_2$ are real numbers or not and the case when both $o_1$ and $o_2$ are real numbers and $o_2$ equals zero. These five cases can be applied to any $/$ operator for objects appearing in the formula and the rule decreases the number of functions with object parameters appearing in the formula. Isabelle has a hard time with quantifiers, so rules like this represent the fact that a term is actually a real number by the valid predicate. We define valid$(x)$ as the same as $\exists x^r : \text{real} \ x = \text{obj\_of\_real}(x^r)$. A function real\_of\_obj that maps from objects to the Isabelle real numbers is an
inverse of \textit{obj\_of\_real} if the object given as a parameter is valid—this avoids the use of a quantification; if the object given is not valid then no other simplifications are possible. This rule is shown in Figure 4.21.

\[
P(x/y) \equiv ((\text{valid}(x) \land \neg \text{valid}(y) \land P(\text{bot})) \lor \\
(\neg \text{valid}(x) \land (\text{valid}(y) \land P(\text{bot})) \lor \\
(\neg \text{valid}(x) \land \neg \text{valid}(x) \land P(\text{bot})) \lor \\
(\text{valid}(x) \land \text{valid}(y) \land y = 0 \land P(\text{bot})) \lor \\
(\text{valid}(x) \land \text{valid}(y) \land y \neq 0 \land \\
P(\text{obj\_of\_real}(\text{real\_of\_obj}(x)/\text{real\_of\_obj}(y))))
\]

Figure 4.21: Split Rule for Division

The result of applying the rule is shown in Figure 4.22. While the rule removes all traces of the / operator on objects, the formula is significantly larger—which does not look good. However, in this case, we can see that any one of the first four disjuncts of the goal is true, because of the negation of \( \bot \neq \bot \). Therefore the formula is true.

Generally, the proof rules follow a particular format to translate formulas.

1. The \textit{hequals}, \textit{defined}, and the like are rewritten to use Isabelle equality and \( \bot \).

2. Rewrite any \( x \in \mathbb{R} \) in the assumptions as \( \exists x^R : \text{real } x = \text{obj\_of\_real}(x^R) \)

3. Rewrite all object relations (<, etc.) using \textit{valid}

4. Split on all object functions, using \textit{valid}.

5. Rewrite all object quantifiers to use \textit{obj\_of\_reals} and \textit{is\_not\_real}.
∀x^n, y^n : not_real is_not_real(x^n) = ⊥ ∨ is_not_real(y^n) = ⊥

(valid(x) ∧ ¬valid(y) ∧
∃z, w : obj z ≠ ⊥ ∧ w ≠ bot →
¬((obj_of_real(x^r) + obj_of_real(y^r)) = ⊥ ∧
((obj_of_real(x^r) + obj_of_real(y^r) ≠ ⊥ ∧ ⊥ ≠ ⊥)) ∨
¬valid(x) ∧ valid(y) ∧ ∃x^r, y^r : real obj_of_real(x^r) ≠ ⊥ ∧
obin_of_real(y^r) ≠ ⊥ ∧ ∀z, w : objz ≠ ⊥ ∧ w ≠ bot →
¬((obj_of_real(x^r) + obj_of_real(y^r)) = ⊥ ∧
((obj_of_real(x^r) + obj_of_real(y^r) ≠ ⊥ ∧ ⊥ ≠ ⊥)) ∨
¬valid(x) ∧ ¬valid(y) ∧
∃x^r, y^r : real obj_of_real(x^r) ≠ ⊥ ∧ obj_of_real(y^r) ≠ ⊥ ∧
∀z, w : obj z ≠ ⊥ ∧ w ≠ bot →
¬((obj_of_real(x^r) + obj_of_real(y^r)) = ⊥ ∧
((obj_of_real(x^r) + obj_of_real(y^r) ≠ ⊥ ∧ ⊥ ≠ ⊥)) ∨
(valid(x) ∧ valid(y) ∧ y = 0 ∧
∃x^r, y^r : real obj_of_real(x^r) ≠ ⊥ ∧ obj_of_real(y^r) ≠ ⊥ ∧
∀z, w : obj z ≠ ⊥ ∧ w ≠ bot →
¬((obj_of_real(x^r) + obj_of_real(y^r)) = ⊥ ∧
((obj_of_real(x^r) + obj_of_real(y^r) ≠ ⊥ ∧ ⊥ ≠ ⊥)) ∨
valid(x) ∧ valid(y) ∧ y ≠ 0 ∧
∃x^r, y^r : real obj_of_real(x^r) ≠ ⊥ ∧ obj_of_real(y^r) ≠ ⊥ ∧
∀z, w : obj z ≠ ⊥ ∧ w ≠ bot →
¬((obj_of_real(x^r) + obj_of_real(y^r)) =
obj_of_real(real_of_obj(z)/real_of_obj(w))) ∧
((obj_of_real(x^r) + obj_of_real(y^r) ≠ ⊥ ∧
obj_of_real(real_of_obj(z)/real_of_obj(w)) ≠ ⊥))

Figure 4.22: Example Formula After the Split
6. Rewrite \(-valid(x)\) (which may appear only in the assumptions) as
\[\exists x^n: not\_real. x = is\_not\_real(x^n).\]

7. Rewrite \(valid(x)\) (which may appear only in the assumptions) as
\[\forall x^R: real. x = obj\_of\_real(x^R).\]

8. Rewrite \(\exists x: not\_real. \bot = is\_not\_real(x)\) as true, and
\[\exists x: not\_real. \bot \neq is\_not\_real(x)\) as true.

9. Rewrite \(\forall x: not\_real. \bot = is\_not\_real(x)\) as false, and
\[\forall x: not\_real. \bot \neq is\_not\_real(x)\) as true.

Rules to simplify occurrences of \(valid\) and \(obj\_of\_real\), as discussed above, are applied between any of these steps along with rules that simplify constants and functions when possible without splitting. Each rule removes one kind of symbol from the formula completely, potentially adding new ones. Any new symbols added, \(i.e.,\) \(\bot\) or \(valid\) are removed by later applications of rules if the formula is true. All rules have been proved correct with respect to the model within Isabelle.

4.8.5 Validation of Rules

We validate the rules by applying them to formulas obtained from an undergraduate discrete mathematical logic book by Epp [16]. We also apply them to “test” formulas generated by Dr. Harvey Friedman for use in the classroom. Unfortunately, because of limitations within Isabelle, we are unable to compare the formulas quantitatively, as we will see. However, we can give a qualitative comparison. In this analysis, we consider a formula \(translated\) if all symbols of the added infrastructure are
removed. The translation rules, for the most part, do a good job translating the formulas. The modulo operator, in particular, is tough. There are many side-conditions on its input that lead to translated formulas that are complicated. Formulas marked as "Proved" are both translated and proved automatically by Isabelle.

Formula names prefaced by "Epp" indicate that the formula is from Epp’s book [16]. The first digit indicates the chapter, the second the section, the remaining ones the problem number. The other problems are variants of test formulas generated by Dr. Harvey Friedman. These reports are for only the true variants. False variants may not be translated completely to Isabelle-native formulas; the extra symbols may be the very reason why the formula is not true. Moreover, a false formula will not be proved! However any false variants in the test suite have their negation reported in Table 4.1.

The results are disappointing in some sense; Isabelle is rarely able to prove translated formulas. In most cases, it is a limitation of the theory development of Isabelle, e.g., for Epp2126, the translated formula is \( \exists y : \text{real}. \ y \not\in \mathbb{Z} \), an obviously true formula. In other cases, the translation itself may not be optimal for Isabelle, e.g., for Epp2340g, the translated formula is \( \forall y, yb : \text{real}. \ \text{real}_\text{of_int}(ya) - \text{real}_\text{of_int}(yb) \in \mathbb{Z} \). If this formula is translated into the integer type rather than the real type, it is true by construction.

The instances in which the translation process leaves some symbols used in the translation process are not surprising. Consider HT4v3, which starts as \( \exists xy. \ x \in \mathbb{Z} \land y \in \mathbb{Z} \land \neg((x * y) \geq (y/x)) \). After some translation, we have a term \( \text{real}_\text{of_obj}(ya/\text{obj}_\text{of_real}(y)) \) in which a split is performed on the / operator.
After the split, we will have terms with $real\_of\_obj(\bot)$, something that is not removable on its own. However, the context in which these terms appear eventually be thrown out as not possible through interactions with other clauses. Indeed, the translated formula

$$\exists y, ya : real. (y = 0) \land (y \in \mathbb{Z}) \land (ya \in \mathbb{Z}) \land$$

$$(y \neq 0 \land \neg real\_of\_obj(\bot) \leq y \ast ya \lor y = 0) \lor$$

$$y \neq 0 \land y \in \mathbb{Z} \land ya \in \mathbb{Z} \land (y \neq 0 \land \neg ya/y \leq y \ast ya \lor y = 0)$$

already has a valid assignment for $y$, namely 0, that renders the $real\_of\_obj$ piece immaterial, even though Isabelle cannot find that assignment.

Table 4.1: Formulas Used for Validation of Rules

<table>
<thead>
<tr>
<th>Formula Identifier</th>
<th>Proved by Isabelle automatically</th>
<th>Translated to Isabelle native formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>HT1v1</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v21</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v3</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v4</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v5</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v6</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT1v7</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>Formula</th>
<th>Proved</th>
<th>Translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>HT2</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT2v3</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT2v5</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT2v6</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT2v7</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT3v1</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT3v2</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT3v3</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT4v1</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT4v2</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT4v3</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>HT5v1</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT5v2</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT5v3</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT5v4</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>HT5v4T</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>HT55</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2126</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2130</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215a</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215b</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

continued on next page
4.8.6 Limitations

A limitation of these rules is that naive applications of them may cause the formula to grow exponentially. Each application of a “split” rule doubles the size of a formula in the best case. Our steps and simplification rules between steps are designed to simplify the formula as much as possible before splitting so that any splits have minimal impact.

Another potential problem is the formula produced by these rules as mentioned in the analysis. Consider the following formula: $\forall x : \text{obj } x \in \mathbb{Z} \rightarrow x \mod 2 \leq 2$. After applying the rules, we obtain the goal (among others) that $2 : \text{real } \in \mathbb{Z}$. This is an Isabelle formula, but yet is not provable directly by Isabelle using its built-in rules and tactics. One potential solution is to augment the split rules with more possibilities. Instead of just having a real number, we could include the rationals, integers and naturals as well. Unfortunately, this also has its limitations; it increases the number of case splits and thus makes the first limitations worse.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Proved</th>
<th>Translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epp2215c</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215d</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215e</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215f</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215g</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215h</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp2215i</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp315</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp3115</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp3127</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Epp3127v2</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Percentage</td>
<td>46.2%</td>
<td>97.4%</td>
</tr>
</tbody>
</table>
Finally, the last major limitation of this work is on the proof of the Isabelle formula; Isabelle may not be able to prove it. This limitation is quite apparent in the data of Section 4.8.5. A usual occurrence is for the translation rules to push the formula faithfully into a form that Isabelle understands, yet Isabelle is unable to prove. This limitation exists to some extent for all provers; each one will be tuned to handle particular kinds of formulas. However, because we have implemented the rules within Isabelle, we are tied to it as a prover.

4.9 Related Work

To our knowledge, no other work has examined the practical implications of translations between different logics as we have done. Enderton [15] presents a proof that many-sorted first-order logic is reducible to one-sorted total first-order logic but does not include any analysis of the practical complexity of the reduction. There has been some work on proving Z specifications within Isabelle. Kraan [39] discusses some of the issues of translating between partial and total functions, but does not do so in depth. Kolyang [38] discusses a structure preserving embedding of Z in Isabelle. However, it does not take into account partial functions completely faithfully, \emph{i.e.}, one of their examples is that $1/0 = 1/0$ is provable in their encoding.

Jahob [8] and Why [19] are examples of program verification frameworks that translate to multiple provers. Why takes input translated from C or Java, while Jahob takes input from a Java program. These tools all generate verification conditions for many different provers. Jahob and Why both internally implement rules to translate between logics and, at least for Jahob [41], have pencil and paper proofs of the rules’ correctness. Our work differs in that these projects’ goal is to prove verification
conditions while our goal is to examine the difficulties in translation. Thus we use
machine-checked translation rules for a simpler translation.

Isabelle [51] include the \texttt{sledgehammer} command that calls first-order automated
provers to try to prove a given higher order logic theorem. This command essen-
tially encodes the purported HOL theorem within first-order logic, though it may
not be encoded faithfully. After a proof is found, Isabelle then must reconstruct the
proof using hints from the first-order prover. The implementers of Isabelle examined
both the theoretical and implementation complexity of such an approach [55], as we
have here, though because Isabelle essentially redoes the work, the potential for poor
translations is not a factor. Our work differs in that we are interested in examining
a verified translation process rather than the proof results.

\section*{4.10 Future Work}

Since the rules have been proved correct within Isabelle, the next step of the work
is to implement a tool that can perform the translations described in this chapter
quickly and directly. The architecture for this new tool to process the formulas into
total-logic formulas. These total-logic formulas can then be translated to any of the
many potential theorem provers in the hope that other theorem provers are better at
these kinds of formulas. Moreover, this would allow us to develop a prover to handle
these particular kinds of formulas.

Also, this technique may be applicable to instances in which one would like to
“type check” mathematics, but yet allow for natural expressions of type inclusions.
For example, type checking could involve determining whether a particular expression
could every be defined. The translation rules would need to be much more elaborate to
handle new mathematical theories; the current work is fixed within a given structure and model.

4.11 Summary and Conclusion

In summary, we have validated the third part of the thesis by demonstrating that, even in simple cases, practical difficulties of machine-checked translation between different logical systems present surprising challenges to automated verification.
Chapter 5: Conclusions and Future Work

In this chapter, we summarize the research conducted for this dissertation, discuss open issues and future work, and finally present conclusions from this work.

5.1 Summary

We summarize each of the chapters in the following subsections.

5.1.1 Programming Language

In Chapter 2, we presented a programming language construct that aids both automated verification and human comprehension of source code. The construct helps decouple the dual purposes of loop invariants by separately describing abstract invariants on variables. These abstract invariants are maintained by the construct automatically with additional requires and ensures clauses for certain procedures, with one-time proofs of invariant and procedure annotation correctness. This construct was given syntax, semantics and proof rules as an addition to the RESOLVE programming language. The proof rules were also proved sound and relatively complete. Since this language construct helps with proofs of VCs, it supports the first part of the three part thesis statement.
5.1.2 Interactive Proof Assistants as an Automated Back-end Prover for VCs

In Chapter 3, we demonstrated how an interactive proof assistant, Isabelle, can be used as an automated back-end prover for VCs from RESOLVE programs, and how mathematical theories used for specification of abstract datatypes can be embedded into an interactive proof assistant. In the process, we explored the role of mathematicians within our vision of the verified software development paradigm discussed in Chapter 1. Based on our experience, we presented some lessons learned from using an interactive proof assistant in these two roles. Since Isabelle was able to function effectively as an automated back-end prover of VCs, it supports the second piece of the thesis statement.

5.1.3 Translations Between Different Logical Systems

In Chapter 4, we presented a proof of a translation method between free and total many-sorted first order logics. The method was then applied to translate a “classroom” presentation of free logic to a format amenable to automated proof. While the proof of the translation method is straight-forward, the actual realization of the translation method is much more complicated. This complication supports the third part of the thesis by demonstrating that, even in simple cases, practical difficulties of machine-checked translation between different logical systems present challenges to automated verification.

5.2 Future Work

We have discussed how the restriction construct appears to factor off the essence of abstract invariant maintenance of blocks of code. These harder VCs, e.g., those
with quantifiers in a restriction predicate, are proved once within the restriction declaration. Procedures called on variables under restrictions usually perform small, “local” updates that do not need knowledge of the full quantified formula. A natural question is how to exploit this more effectively for automating proofs.

Some ideas may bear fruit. The first is to develop mathematical theories for certain types of quantification with some programming language constructs to indicate that a formula should be treated specially by only following the instantiations allowed by the theory. One simple kind of quantification would be $\forall x : \text{obj } x \in S \rightarrow Q(x)$. Rules to manipulate $S$ for updates and removals would include extra obligations and assumptions. This is similar to triggers for E-matching, but would be more easily analyzed for correctness.

The second idea is that the programming language itself might have special syntax for certain common kinds of quantification, just as it might have syntax for functions with certain kinds of algebraic properties. The syntax should be extensible; programmers should be able to add their own. These commands would both provide guidance to the prover (the VC generator could direct the prover towards a goal) and would provide guidance to people reading the source code about the properties and important features of formulas and functions.

Back-end theorem provers also have some threads of research to be explored. While others have looked at SMT solvers, specialized decision procedure, first order provers and interactive proof assistants, the automated use of many tools has not been explored completely. For example, Jahob allows for the use of many prover back-ends, however each back-end must be manually specified. In the future, analysis of the VCs along with heuristics may allow for the automated selection of potential
back-ends based on the information gleaned from the VC and the known strengths and limitations of the back-end provers. Also, a completely new theorem prover tuned to RESOLVE VCs, designed with the lessons learned both in this dissertation and in others’ experience with back-end provers, would be extremely valuable.

Finally, study into methods of automating portions of verified translation between different logic systems would also be extremely helpful because of the difficulties encountered in Chapter 4. For example, tools that aid in visualization of the remaining symbols in a formula and/or the structure of that formula would aid in creating effective translation rules.

5.3 Conclusion

In conclusion, we have presented a vision of software engineering in which no production software is considered properly engineered unless it has been fully specified and automatically verified as satisfying these specifications. The envisioned tool to achieve that vision is a verifying compiler, a compiler that also checks for behavioral correctness. The basic obstacle toward full adoption of such a tool is that current verifying compilers simply cannot prove non-trivial code correct automatically without heavy code annotations. In this thesis we have explored a possible route towards a realized verifying compiler. Specifically, we have presented three contributions towards implementing a verified software paradigm. We have demonstrated that additional programming language constructs can be helpful in verification without a large annotation burden. We have shown that an interactive proof assistant can be an effective back-end prover. Finally, we have demonstrated that verified translations between different logic systems represent a challenge for automated software verification tools.
Bibliography


