Engineering Specifications and Mathematics for Verified Software

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Hampton Smith
May 2013

Accepted by:
Dr. Murali Sitaraman, Committee Chair
Dr. Brian C. Dean
Dr. Jason O. Hallstrom
Dr. Roy P. Pargas

Abstract

At the heart of the argument for formal, mathematical methods of software quality assurance is that increased energy spent to develop formal specifications and prove software components against those specifications is amortized over the lifetime of the verified component. Thus, modularity and reuse are central prerequisites of practical verification. Because of this, there are two strategies for reducing effective energy invested in a verified component: 1) decrease the amount of effort required to verify it, and 2) increase its reusability and thus its lifetime.

While many modern verification systems exist, few seem to have been designed with modularity and reuse in mind. On the one hand are systems built on industrial, object-oriented languages, which provide modularity in the programming world, but whose specifications rely on mathematics that do not support these goals. On the other hand are extensible, generic mathematical systems that are not integrated with programming languages that support component reuse. The result, on both sides, is the creation of components insufficiently generic and extensible to be reused.

As we wish to verify components of increasing complexity, we must build these components out of smaller subcomponents. If these subcomponents display poor modularity, they will compose poorly or not at all, resulting in complex interactions and proof obligations that are difficult to satisfy. To date, modern systems have addressed this complexity by placing the onus of verification on a suite of sophisticated, industrial-strength automated theorem provers that are on the bleeding edge of artificial intelligence design.

This seems paradoxical, however, as programmers do not often rely on deep mathematical results when they reason about the correctness of components. We posit that by shifting the burden to the design of good components and specifications, as supported by a flexible mathematical and specification subsystem, proof obligations should become much more obvious and more easily-proved. In addition to influencing the design of our own system, RESOLVE, such exploration would benefit

other systems as well, by informing specification and theory design across the board.

The intent of this proposal is three-fold. First, to develop a flexible mathematical framework for program specification designed with modularity and reuse in mind. Second, to experiment with the design and implementation of a minimalist prover sufficiently flexible to operate on that mathematical framework and determine the actual practical requirements for program verification of well-specified components. Third, to develop a diverse library of components specified using a variety of specification styles in order to identify best-practices in specification engineering by measuring the complexity of resultant VCs.

Dedication

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Acknowledgments

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea

commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

The verifying compiler is a grand challenge in computing, perhaps most famously stated by Tony Hoare in 2003[18], but based on research into program correctness stretching back to King's thesis[21] and research in the 1960s[17]. The goal of the verifying compiler is to prove mathematically that software is correct—i.e., it behaves according to its specification. Such a compiler would ideally eliminate the need for testing to reveal functional bugs by accomplishing what testing cannot: demonstrating the *absence* of any bugs. Unlike testing and informal reasoning, formal verification demonstrates that code behaves as specified under every possible valuation and along every possible path of execution.

For a noteworthy and recent demonstration of the weakness of traditional testing and informal reasoning, we consider Joshua Bloch's blog post on a Java implementation of binary search[4]. Binary search is a simple, well-understood, and widely-implemented algorithm. Yet, Bloch, a Google Researcher, revealed a subtle bug in the standard Java library's implementation of binary search—an implementation that had been in place for nine years (since 1997) and was based on a version of the algorithm "proven" correct (via informal reasoning) by Jon Bentley of Carnegie Melon University in his famous *Programming Pearls*[3]. Certainly, such a straightforward implementation of such a simple algorithm, backed, as it was, by a proof from a respected algorithms guru and in wide deployment for nine years would be considered by most as a *mature*, *well-tested* component—one suitable for use in critical deployments. And yet it contained a subtle, latent overflow bug that was only revealed when the code of a client in the wild broke because the component failed to meet its specification (specifically causing an `ArrayIndexOutOfBoundsException`, which, in a C or C++context, would

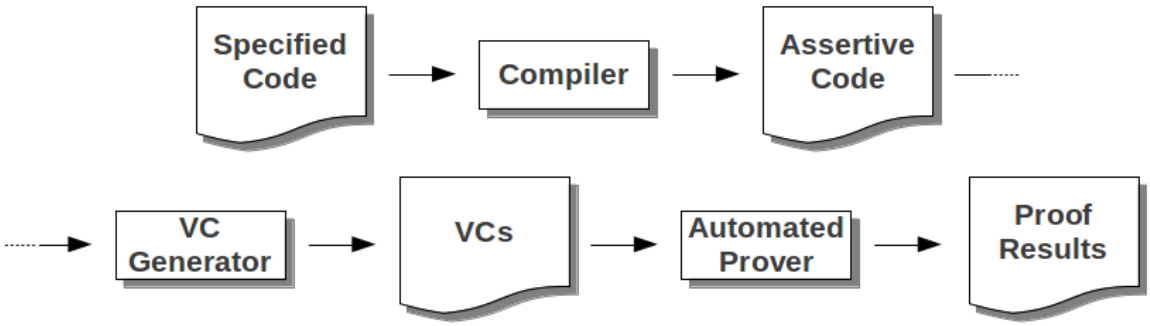


Figure 1.1: The Verification Pipeline

be a recipe for a buffer overflow attack in addition to a potential crash.) This bug, so subtle and resilient against traditional testing and human reasoning, becomes extremely easy to detect under formal reasoning, where bounded, programmatic integers are well-specified in a mathematical way.

Several systems for formal verification exist today, including some built on Java which may have caught this and other bugs. These systems are traditionally built as a pipeline in which code and its associated specification are translated into an intermediate *assertive code*, which is then translated into a series of *verification conditions* (VCs), which are mathematical expression that express proof obligations necessary and sufficient to demonstrating the functional correctness of the code. These VCs are sent to one or more *automated provers* which attempt to dispatch the VCs. This process is illustrated in Figure 1.1.

Despite many successes, many existing systems require a great deal of effort, either in the form of interactive proving or carefully contrived hints to an automated prover, in order to verify correctness. This is counter-intuitive since most programs contain straight-forward logic that the programmer feels assured of without calling upon complex mathematical reasoning. While extensive reuse would amortize this verification effort over time, a number of properties of modern systems prevent wide reuse of verified components. We seek to demonstrate that a well-integrated, flexible, and extensible mathematical and specification subsystem permits specifications that more closely reflect the programer's intuition and enable the usual patterns of reuse, resulting both in more straightforward proofs and more generic, longer-lived components. This dissertation presents both our design and our implementation of a system for testing this hypothesis and evaluates the result via experimentation.

1.1 Alternative Systems: Practical vs. Pure

Existing verification systems largely fall into two categories: those with a focus on practical, automated verification and those with a focus on pure mathematics. Some representative examples of the former include Jahob[25], based on Java; Spec#[1], based on C#; and ACL2[20], based on Lisp. Examples of the latter include Coq[29], based on the Calculus of Inductive Constructions; and Isabelle[32], based on a higher-order, intuitionistic logic. Each of these verification systems is impressive in its own way, but all of them differ from the system discussed in this dissertation. For a more complete overview of modern verifications systems derived from those presented at VSTTE 2010, see [22].

Practical systems often take advantage of a limited, hardcoded mathematical universe that corresponds closely to or is conflated entirely with programming constructs. Pure systems permit an extensible, flexible mathematical framework with clear separation of programming concepts from mathematics. Because of their narrower focus, the former often permit easier mechanical verification than the latter[1], which tend to emphasize interactive user-guided verification.

To illustrate the technical issues in verification for these two kinds of systems, we discuss in detail an illustrative example of each.

1.1.1 Example Practical System: Jahob

Let's consider a version of Java's `ArrayList` verified in Jahob[2]. Jahob is a system developed at MIT that combines code written in a subset of Java with specifications written in Isabelle. VCs can be output in a format acceptable to a number of popular off-the-shelf automated provers.

Listing 1.1 shows the `contains()` operation of an ArrayList. The list is modeled as a set of $(index, element)$ pairs. Assertions are provided inside Java comments (meaning that Jahob-specified Java programs remain compilable using a standard Java compiler) that begin with a colon. Note that mathematical assertions are set off in quotes, a syntax inherited from Isabelle. In the requires clause, `init` is a predicate defined elsewhere in the class. Jahob encourages the inclusion of inline "hints" targeted at the backend provers[48] (of which it supports many) and we see several of those here interspersed in the Java code.

[1] Some systems even permit efficient decidable verification algorithms for certain domains or properties. See, e.g., information on SplitDecision in [36].

[2] Jahob does not yet work with Java generics and so the version of ArrayList verified is the pre-Java-1.5 version that operates on `Objects`.

Listing 1.1: ArrayList.contains()

```
public boolean contains(Object elem)
    /*: requires "init"
        ensures "(result = (EX i. (i, elem) : content))";
    */
    {
        int index = indexOfInt(elem);
        //: noteThat PosIndex: "0 <= index --> ((index,elem) : content)";
        //: noteThat NegIndex: "index = -1 --> ~(EX i. (i,elem) : content)";
        //: noteThat IndexLemma: "0 <= index | index = -1";
        boolean res = (0 <= index);
        /*: note ResultLemma: "res = (EX i. (i, elem) : content)"
                from PosIndex, NegIndex, IndexLemma;
        */
        return res;
    }
```

Utilizing a suite of provers, the Jahob system is able to dispatch the resultant VCs and yield a fully-verified `ArrayList` implementation suitable for generic use—an impressive feat. In fact, in a recent result, the Jahob team verified a handful of different linked data structures[47]. A number of design decisions support this ability. First, Jahob has a flexible set of syntactic tools for specification, including pre- and post- conditions, auxiliary variables[21], and conceptual definitions that permit an intuitive specification. Second, while Jahob supports higher-order specifications, where possible it uses a standard first-order logical representation that can be translated into the input format of multiple back-end proving systems including CVC3[2] and Z3[8]. Those specifications that cannot be translated down to a first-order representation can still be sent to Isabelle for interactive proving. By using multiple prover backends, Jahob can take advantage of multiple proving paradigms including interactive, algebraic, boolean satisfiability (SAT) solvers, and efficient decision procedures. Third, as already stated, Jahob permits in-line mathematical assertions that allow the programmer to guide backend provers by stating useful intermediate results.

However, despite this success verifying small Java programs somewhat automatically, it has significant shortcomings scaling to long-term, scalable verification of complex, modular applications, which we discuss in the following sections.

Complex Proof Obligations for Simple Methods. When using the Jahob system, VCs that result from even simple programs are often extremely complex. Jahob's intermediate VC syntax is not intended to be human readable, so we do not reproduce any VCs here, but we can get a feel for their complexity via sheer volume: the three-line boolean method `contains()` generates VCs that

span over 150 lines[3].

A number of factors contribute to this VC complexity. First, Jahob is built on top of an existing programming language that includes many features that are not amenable to verification, including null pointers and uncontrolled aliasing, both of which complicate reasoning and introduce additional VCs[44]. Second, Jahob's inline assertions must themselves be verified to preserve soundness. The intent of these assertions is that they simplify the proving process by proving intermediate steps, creating additional (presumably easier) VCs that in turn lower the difficulty of the original proof obligations. Still, why intermediate results should be required for a simple `contains()` method is unclear. Third, while perhaps subjective, the system encourages the use of awkward mathematical models for components. In the list example, the choice of a set as the mathematical model requires additional invariants to establish that, for example, no index in the list appears twice with different elements. Presumably a set was chosen because it was the closest mathematical object that already had a well-developed Isabelle theory, but in an ideal world a verification and specification system would provide a first-class, integrated mechanism for extending the mathematical universe so that a more directly analogous mathematical object could be used—perhaps a function mapping or a finite sequence abstraction.

Lack of Support for Modular Design. Jahob stands nearly alone amongst the practical systems for supporting higher-order mathematics. However, practical issues with the integration of Isabelle into Java hamstring the usefulness of this feature. Among the Java features not supported by Jahob are Java generics and dynamic dispatch. These unsupported features preclude many important patterns of reuse that the mathematics of Isabelle are otherwise ready to support. Components cannot be parameterized from the outside with predicates (this is not supported directly and programmatic work-arounds like the Strategy and the Template design patterns rely on dynamic dispatch.) Data structures cannot make parameterizable guarantees about the properties of contained elements (which would require Java generics.) Indeed, the majority of the polymorphism pillar of object-orientation is precluded, seriously reducing the reusability of components and thus the amortization of verification effort over time.

A particularly illustrative example of this lack of modularity appears in a `Map` data structure verified by the Jahob team in [47]. In a language that supports reference semantics (such as Java),

[3]When formatted to standard 80-character lines.

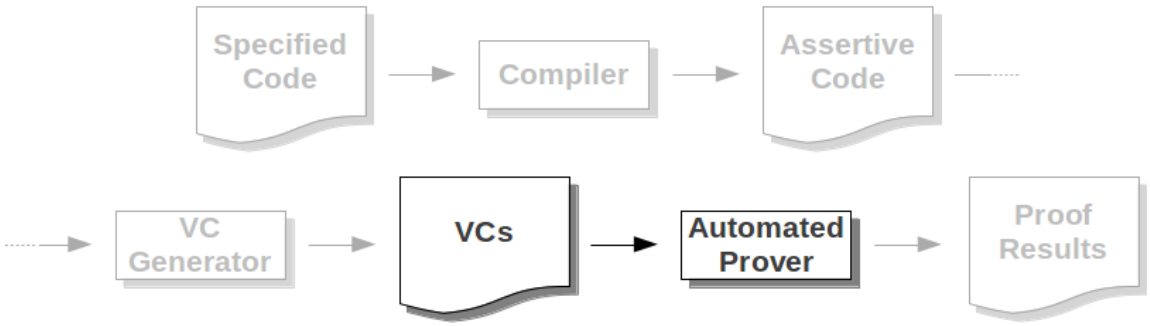


Figure 1.2: Shifting the Burden to the Prover

any efficient implementation of a map data structure must take care that its keys cannot be changed after being inserted (or that, if they are, it happens in a controlled way). In the Jahob example, this trouble is addressed by specifying that *all objects* are immutable with respect to the built in Java `hashcode()` method—a restriction that does not appear in the original `Object` contract and further implies that all objects are immutable with respect to `equals()`. The correctness of the `Map` implementation requires changes to external components, rather than relying on a self-contained specification. We discuss this problem in more detail in [7].

With lengthy, complex VCs, syntax for guiding back-end provers, and a system that encourages a trade-off of mathematical flexibility for prover diversity, the Jahob design seems to assume that the onus of verification is on the provers. Jahob shifts the complexity of verification to the part of the pipeline highlighted in Figure 1.2. However, as the strength of automated provers is the current bottleneck of verification, this requires a great many sacrifices to the limitations of this bleeding-edge part of the verification toolchain, including forcing the programmer to reason about the available provers and what their strengths or weaknesses might be.

1.1.2 Example Pure System: Coq

For an illustration of the workings of a pure system, we now consider a recursive implementation of the integer division operator, specified and implemented in Coq. Despite the fact that Coq is a mathematical system, unable to execute code, we could use its ability to "extract" a program into various target languages[4] for execution. Coq separates specification from implementation; in

[4]At time of writing: OCaml, Haskell, and Scheme.

Listing 1.2 we see a pair of predicates for specifying integer division.

Listing 1.2: Division Predicates

```
Definition divPre (args:nat*nat) : Prop := (snd args)<>0.

Definition divRel (args:nat*nat) (res:nat*nat) : Prop :=
    let (n, d):=args in let (q,r):=res in q*d+r=n /\ r<d.
```

The `divPre` predicate represents the precondition on division—that the second argument is not 0. The `divRel` predicate specifies the relational[5] behavior of division—that the result will consist of two natural numbers, `q` and `r`, such that `q * d + r = n` and `r < d`, i.e., `q` is the quotient and `r` the remainder.

We then implement division recursively as shown in Listing 1.3.

Listing 1.3: Division Implementation

```
Function div (p:nat*nat) {measure fst} : nat*nat :=
    match p with
    | (_,0) => (0,0
    | (a,b) => if le_lt_dec b a
               then let (x,y):=div (a-b,b) in (1+x,y)
               else (0,a)
    end.
```

Coq's `Function` keyword introduces a recursive function with an appropriate implicit fixpoint. The `measure` keyword provides a progress metric—namely, that `fst` is decreasing. Note that, despite the fact that an input matching `(_,0)` is disallowed by the precondition, Coq does not permit non-total functions, so we provide a nonsense return value for this case. The `le_lt_dec` is simply a less-than-or-equal-to predicate on natural numbers. Defining this function immediately raises a termination proof obligation, which one of Coq's built-in proof scripts can dispatch automatically.

To demonstrate the correctness of the implementation, we can assert the theorem given in Listing 1.4. That is, for all inputs, if the inputs meet the precondition, then the behavioral function holds between the inputs and the result of applying `div`.

Listing 1.4: Division Functional Correctness Theorem

```
Theorem div_correct : forall(p:nat*nat), divPre p -> divRel p (div p).
```

[5]Note that while, in general, this technique permits a relation, here the predicate takes the form of a function.

Proving this theorem is more complicated than the termination proof and none of Coq's built-in tactics can dispatch it automatically. Instead we enter interactive mode and prove it live with the sequence of tactics given in Listing 1.5.

Listing 1.5: Division Functional Correctness Proof

```
unfold divPre, divRel.
intro p.
functional induction (div p); simpl.
intro H; elim H; reflexivity.
replace (div (a-b,b)) with (fst (div (a-b,b)),snd (div (a-b,b))) in IHp0.
simpl in *.
intro H; elim (IHp0 H); intros.
split.
change (b + (fst (x,y0)) * b + (snd (x,y0)) = a).
rewrite <- e1.
omega.
change (snd (x,y0)<b); rewrite <- e1; assumption.
symmetry; apply surjective_pairing.
auto.
Qed.
```

A high-level sketch of this proof is that it applies induction by cases, proving that each of the `match` branches and each of the `if` branches maintains the correctness of the implementation. Definitions are repeatedly expanded to take advantage of their hypotheses. This syntax is far from readable without being intimately familiar with Coq and certainly looks nothing like a mathematical proof as it might be conceived by a mathematician.

Systems like Coq have been used to excellent effect verifying complex programs. Coq has been used to verify a C compiler[28], while Isabelle has been used to verify an operating system kernel[23]. This success is due to a number of factors. Unlike in the practical programming systems, Coq and other pure systems provide a rich, extensible mathematical universe permitting higher-order logic and user-created mathematical theories. This enables a hierarchy of abstraction similar to the development of object-oriented code in which complex mathematical objects are composed from smaller mathematical objects and an "interface" of theorems is provided for working with the high level objects. In addition, Coq's programming model is functional, eschewing a number of complexities pervasive in industrial languages—pointers, aliasing, and referential opacity to name a few. Automated proof systems are used to jump small steps, but interactive mechanisms are provided for splitting complex proof obligations into multiple smaller tasks. It is as though the human user becomes part of the assertive code and VC generation step, pointing out useful decompositions of existing proof obligations until they become simple enough that the automated prover can take it

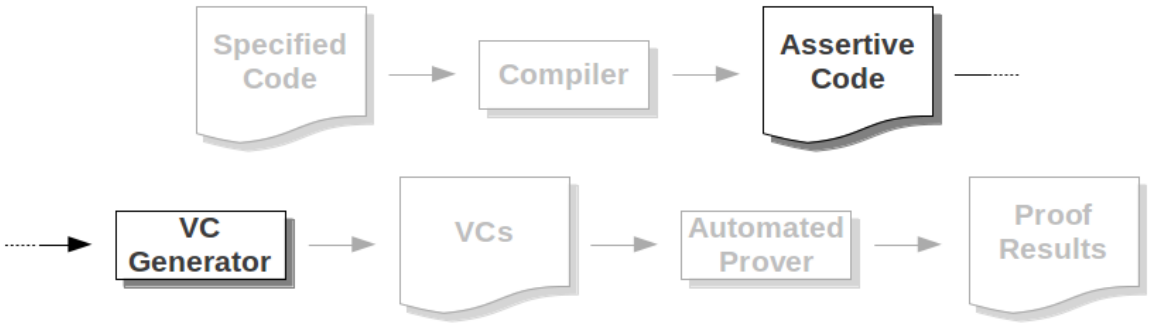


Figure 1.3: Shifting the Burden to the VC Generator

from there. In a sense, such a system shifts the onus of the verification process to the part highlighted in Figure 1.3.

Unfortunately, given the value of a highly-trained mathematical and programming professional's time, a user-guided strategy is likely cost-prohibitive if all but the smallest proof obligations must be discharged by hand. A number of factors contribute to the difficulty of these proof obligations.

One is that, because of the flexibility of the mathematical system, the proof space is much larger and useful simplifications much more difficult to qualify than for provers that are narrowly-applicable to some particular theory. Another is that the divide between the mathematical world of Coq and the programming world of an industrial language is large. We see an example of this in the division example, where time and effort must be expended proving that the behavioral relation is total (even though the method it specifies is not!) and that an irrelevant program branch maintains invariants.

Compounding this problem, pure systems have no awareness of the underlying programmatic structure they are being applied to and thus inherently view verification as operating on procedural rather than object-oriented code. Modular mathematics are therefore not applied to modular code and the result is that, impressive as a verified compiler is, the next complex component must be verified from scratch as it is unlikely that any component from the compiler will be generic enough to be reusable.

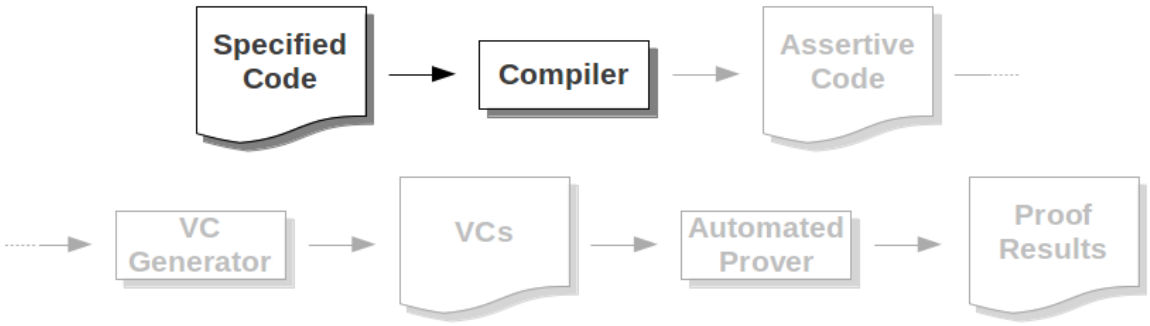


Figure 1.4: Shifting the Burden to Specification

1.2 Best of Both Worlds?

At the core of this research is the question of whether or not a system that combines the best parts of practical systems and the best parts of pure systems might permit specifications that are more amenable to automatic verification. A programming system that eschews certain complexities of industrial languages and avails itself of a tightly-integrated, flexible and extensible mathematical system designed to work with it could be used to create modular components based on modular mathematics. We hypothesize that in such a system, a focus on modularly verified components would yield less complex proof obligations while at the same time encouraging the creation of reusable components that amortize the verification effort invested in them over time. Such a system would place the onus of the verification on the design of modular specifications supported by a flexible compiler, i.e., that part of the pipeline highlighted in Figure 1.4. Highly trained programmers and mathematicians are still required, but the fruits of their labors will be generic and reusable in new environments, unlike one-time proofs of correctness for specialized code.

Developments such as these would also benefit existing systems of all types. Increased understanding of the importance and techniques of modular reasoning can be applied to systems like Jahob to increase the provability of large components via better-engineered subcomponents. A richer understanding of the importance of levels of mathematical abstraction to long-term verification goals would assist those working with a pure system like Coq in making better up-front choices to pay long-term dividends.

1.3 Problem Statement

This dissertation explores if the mechanical verifiability of software components can be improved by better-engineered mathematics and specifications and, if so, which techniques best support these goals. In the process we address the following open problems in the area of formal methods:

- Design and implementation of an extensible, flexible mathematical framework for a practical verification system that permits reuse as well as the development of a rich set of models and assertions.
- Design and implementation of a well-integrated specification framework that is explicitely designed to work with the mathematical system, supporting verifiability by allowing simple, flexible specifications and supporting scalability by encouraging verified component reuse.
- Validation of our central hypothesis via the architecture and implementation of a minimalist rewrite prover to support reasoning in the above framework and explore those prover capabilities practically necessary to mechanically verify well-engineered, modular components.

1.4 Research Approach

Building on previous work on specification language design, VC generation, and automated prover development, this research augments the existing RESOLVE[38, 36] system with a flexible mathematical subsystem and modular specification subsystem in order to test our hypothesis that a minimalist prover should be sufficient for reasoning about well-specified programs.

Unlike current practical systems, where bringing modular mathematics to bear in support of modular programming is difficult at best and where special-case constructs like references cause complex reasoning even in situations where they ought not be relevant, the mathematical system described here is tightly integrated and based on an extensible Morse-Kelley Set Theory, extended to include higher-order definitions, and permits specification of programmatic constructs (e.g., references) only as modeled in that pure mathematical system.

Unlike pure systems, where the specification system that bridges between mathematics and programming is ad-hoc and not designed to take advantage of the structure of a program,

the specification system described here cooperates with the mathematical system by design, thus permitting it to take advantage of the same modularity and genericity used in a well-designed component.

Armed with such a system, we present our findings applying it to the design, specification, and implementation of a set of verification benchmarks selected to exercise the system and demonstrate its features over a diverse set of challenges, including reasoning about integers, functions, arrays, and abstract data types. We present analysis of resulting VCs to justify which prover capabilities are strictly necessary, presenting our design and implementation of a minimalist rewrite-based prover based on a plug-in architecture to support only those capabilites necessary for practical VCs.

Finally, we present an evaluation of the resulting system that includes classification of VCs by the minimalist prover based on a number of proof metrics including number of required proof steps, steps spent in certain expensive portions of the proof search, and proof time; as well as subjective, qualitative metrics derived from our own experience.

1.4.1 Contributions

This system address several open problems in the area of formal methods, both practical and theoretical:

First, a flexible, extensible, and intuitive mathematical subsystem would significantly improve on a key component of the verification tool chain. The day when an AI can infer the intent of a program and verify it without rigorous specification and significant mathematical development is extremely distant. Until that time, formal verification will be a joint effort between highly trained programmers and mathematicians. While improvements are being made[46], the current generation of specification systems (both practical and pure) largely use ad-hoc mathematical syntax and esoteric mathematical foundations that programmers may find convenient but mathematicians find confusing and unweildy. Beyond simply engendering collaboration between programmers and mathematicians, such a math-centered language design will encourage the full body of modern mathematical developments to be used directly. Additionally, since it is not grounded in any programming constructs, it will not be tied to any particular language and may itself be used as a component outside of RESOLVE.

Second, a well-integrated and flexible specification and mathematical system would open up the development of verified components to modular, reusable techniques by providing first-class

syntax and flexible mathematical semantics for mapping such programmatic ideas into the mathematical realm. As an example, the lack of useful higher-order logic in practical systems prevents components from being designed to take mathematical assertions or abstract operations as parameters, severely limiting one dimension of reuse: genericity[7]. Our hypothesis is that a well-designed, modular system should better exploit programmer intuition and allow for more straightforward proof obligations, permitting slower, but more expressive, automated provers to be used. The development of such techniques would be a boon to existing systems, as well. For example, as part of our research so far, we have explored the concept of quantifier elimination and techniques that can be used to specify and verify in their absence. In a recent paper from the Jahob team, we find a quote highlighting the need for such research in the context of an attempt to verify an implementation of a Java `ArrayList`: "Unfortunately, the provers are unable to automatically prove the post-condition of remove. What makes the problem...difficult is that the assumptions contain universally quantified formulas while the post-condition contains an existentially quantified formula." Our research on engineering mathematics addresses this and other issues and may be helpful in elliminating such complications.

Thirdly, such a system will permit the above hypothesis to be tested and demonstrated in a rigorous, mechanical environment. The current state of the art in verification suggests that verification is hard because programs are complicated. We believe that well-engineered programs are not complicated and that, by extension, if augmented with well-engineered specifications, the resulting proof obligations should not be difficult. Thus, a system designed to support well-designed programs is more important to successful verification than one designed to support the limitations of a state-of-the-art prover. What such a design entails is an open questions addressed by this research.

1.5 Thesis Statement

In a verification system, an extensible, flexible mathematics and specification subsystem enables better-engineered component specifications and thus more straightforward proof obligations that are more easily dispatched by even minimalistic automated provers.

1.6 Dissertation Organization

XXX

Chapter 2

Verification Background and Related Work

In order to justify the presented research as well as place it in context, it is important to consider the full breadth of existing mathematical systems, provers, and verification libraries. In this chapter, we discuss related work and necessary background in these areas.

2.1 Specification and Mathematical Systems

Early systems for program specification include Z[9] and Larch[13]. Both are first-order predicate logics not grounded in any particular programming system, though each has been adapted for many different programming langauges. Larch provides a two-tiered specification style to ease adapting it to a new language. Because it is based on Zermelo-Fraenkel set theory, Z is one-sorted (i.e., it has only one "type": the *set*). Larch, on the other hand, is multi-sorted, but assumes all sorts are disjoint (i.e., it does not permit sub-"types", among numerous other "type" relationships.) These restrictions complicate the mapping of programming concepts in modern languages to their mathematical universes. Modern systems for specification in practical systems often address the limitations on sorts, but, as we will see, many are restricted to first-order logics, severely restricting the genericity (and thus the reusability) of mathematical theories and specifications, as discussed in detail in Section 5.3.1.

A straightforward method of specification in practical systems is to permit them to be written programmatically in the target language–i.e., the specification is *executable*. This technique was introduced by the Eiffel language[30] in 1985 and served as the progenitor of modern contract-based programming[1].

An exectuable specification has the immediate advantage of permitting dynamic checks to be compiled into the code if static verification is not possible. However, it raises a number of disturbing possibilities including specifications that have side effects or that require proofs of "termination". Additionally, such a specification can introduce so-called "implementation bias" in which the implementer chooses an implementation that matches the specification closely rather than the best implementation. A final complications is that the specification is only as clear as our understanding of the meaning of the underlying language. If the language in question does not have a formal semantic, we're not any better off.

A number of specification systems use this method as their basis (though in some cases they offer non-executable extensions). Eiffel, for example, persists as an important language to this day. Another prominent example is the Java Modeling Language (JML)[26]. The majority of functions used in a JML specification are, in actuality, Java methods. However, in order to defeat the side-effecting problem, they are required to be declared *pure*, i.e., they must be demonstrably side-effect free (this does not, however, speak to their termination.) "Model methods" may also be provided, which define a method for the purposes of specification, but contain no code—however, while JML provides some features for non-code-based specification, model methods are often still expressed in terms of (non-executable) Java code (i.e., in the formal comment there is commented-out Java code expressing the behavior of the model method.) The logic of JML is first-order (i.e., we cannot quantify over methods or types), but we are able to take advantage of the flexibility of including "code" in our mathematical specification to achieve higher-order methods for the purpose of, for example, passing them as a parameter[2].

Because of the tight integration between "specification" and "code" in JML, the mathematical realm becomes restricted to ideas expressible in Java. E.g., the only type-relationship are Java type-relationships. While this simplifies the mathematical realm and makes it consistent with the programmer's understanding, we posit that the necessary complexity of mathematical develop-

[1] Indeed, the phrase "Design by Contract" remains a registered trademark of Eiffel Software.
[2] We can effectively include an instance of the *Strategy* design pattern in our specification.

ment required for true mathematical modularity necessitates a mathematical specialist. Thus, the mathematical realm should target the universe familiar to such a specialist, rather than cater to a programmer. Many restrictions necessitated by the computability requirements of code are irrelevant in the mathematical sphere and lead to specifications that are less general or less descriptive.

Another system with executable specifications is Why[5]. Why is ML-like and focussed on reasoning about built-in structures like arrays rather than general data abstractions. By using a variety of translation front-ends, Why can be an intermediate specification and programming representation for different programming languages including Java, C, and ML[5].

JML and Why are not alone in using this specification style: additional systems in this style include the specification language of Spec#[1], built on C#, and ACL2[20], built on a dialect of Common Lisp.

Another popular style of specification in practical systems relies heavily on separation logic[34]. Because it models the heap directly, separation logic provides mechanisms for reasoning directly about aliasing and pointer arithmetic. In order to bring these complex problems into the range of modern theorem provers, separation logic allows operations to explicitly state which areas of the heap they will modify, along with which arbitrary properties they expect the heap to maintain while they operate. This requires the heap to be modeled in its entirety, along with specialized logical systems for reasoning about this model. A notable system focussed on this style is Verifast[19], a system for verifying Java and C programs. It uses separation logic expressed in a custom specification language and targets the Z3 prover[19].

A third style of specification in practical systems is to use a strictly separate mathematical language for specification. Such systems often provide more generality than others, since the mathematical language can usually be arbitrarily extended to permit new bases for mathematical models and provide generalized machinery for reasoning about these new classes of mathematical objects. Examples of this kind of system are Jahob, the RESOLVE system developed here at Clemson, and the RESOLVE system being developed at The Ohio State University. As explored in Section 1.1.1, Jahob uses Isabelle as its specification language, granting it higher-order logic and a rich theory of sorts for free. However, the usefulness of these features is hamstrung somewhat by a lack of support for the associated programmatic features in Java, such as generics and inheritence. RESOLVE strives to provide a sorted, higher-order mathematical language, through the complete design and implementation of such a language as detailed in this dissertation. OSU's RESOLVE

implementation does make a clear delineation between mathematics and programming, and permits higher-order definitions, but does not permit the mathematical universe to be extended at this time. As a result, its reasoning is limited to a small library of built-in sorts.

An interesting comparison of some additional practical systems with respect to their specification and reasoning systems can be found in [16].

Amongst the pure systems, the style of specification and mathematical representation becomes more uniform, at least in their almost-universal inclusion of the features crucial for a component specification platform, including higher-order definitions, first-class types, and a rich, extensible theory of sorts. By far the most popular pure systems are Coq, described in detail in Section 1.1.2, and Isabelle.

Isabelle is notable for providing a very small logical core from which all other theories and logics are built, permitting the system to be soundly extended with diverse logics for which it was not designed. For example, the most popular variant, Isabelle/HOL, Isabelle with Higher Order Logic, provides multi-sorted higher-order logic, expressed in a reasonably intuitive mathematical syntax called Isar[46].

2.2 Automated Theorem Provers

We may broadly partition automated theorem provers into three categories—those based on decision procedures, those based on boolean satisfiability (SAT)[3], and those based on term rewriting. The distinction between these is not clear-cut, however, as many (indeed, most) systems combine all three to some extent, using term rewriting as a preprocessing step before passing the problem off to a SAT-solver or a decision procedure, or using a feedback loop between a term-rewrite prover and a SAT-solver.

2.2.1 Decision Procedures

The decision procedures are the least flexible but often the most efficient. They take advantage of information about a specific domain to arrive at conclusions quickly. Examples include linear arithmetic solvers of the kind found in Z3[8] and OSU's SplitDecision[36] system for reasoning about

[3]Technically, SAT is a decision problem with finite solution space, and thus its solvers are decision procedures. However, since this problem is NP-hard, we use "decision procedure" throughout to mean an algorithm that is *efficiently decidable*.

finite sequences. Because they exploit domain-specific information, these systems are necessarily hard-coded and cannot be generalized to new domains.

2.2.2 SAT-Solvers

Provers in this class attempt to find valuations of boolean variables to satisfy a given formula. While in general this is NP-hard, a number of branch-and-bound techniques can be used to drastically reduce the problem space for practical formulae. Almost universally such provers use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a constellation of specific heuristics and refinements that reduce the search space considerably in many cases. For a good overview of this algorithm along with some recent refinements, the interested reader is directed to [31].

A number of SAT-solvers exist. Yices[10] and Z3 are popular provers based at their core on SAT-solvers. It may seem difficult, at first, to apply provers for boolean formulae to complex programmatic proof obligations, but many techniques exist for translating complex domains into equisatisfiable boolean representations. Indeed, because of the blazing speed and mature implementations of SAT, such translations are an active area of research (see, for example, [12] and [35].) SAT-solvers extended to include the Maximum Satisfiability (Max-SAT) problem are able to provide useful debugging information in the form of counter-examples: indicating precise valuations under which a theorem does not hold. Yices is an example of a SAT-solver with this capability. The general applicability of boolean logic (particularly in the field of computing) makes SAT-solvers significantly more general, since they may be applied to any theory for which a translation into boolean logic exists.

2.2.3 Term-Rewrite Provers

These provers are the most flexible, but also the most difficult to optimize. They are by far the most similar to how a human would proceed with a proof—by matching theorems against a set of known facts and goals, transforming them until we derive the result we want. This general pattern-matching approach means that they can be applied to any domain about which a set of theorems has been established. An excellent example of this kind of prover being used in a practical-style system is the prover used in the ACL2 system. It utilizes a complex series of hints, provided by the person developing the theory, about which theorems should be applied in which way, then applying them

sequentially. This differs from the approach of our minimalist prover—where we eschew encoding information about how the prover should apply theorems, as we would prefer mathematicians reason about mathematics rather than about how a particular prover will apply it. Additionally, while our rewrite-prover strives to operate over our distinct mathematical subsystem, ACL2's mathematics are much more tightly bound with its underlying language, Lisp. A noteable feature of the ACL2 prover is its ability to automatically discover inductive proofs over certain restricted domains.

Many SAT-solver-based provers, including Yices and Z3, employ term-rewriting as a pre-processing step to simplify formulae or translate them into a heuristically-convenient form.

2.3 Benchmarks and Specification Engineering

A number of verification benchmarks and libraries exist, and at least one verification effort has tangentally acknowledged the specification engineering question.

The Jahob team has made a concerted effort to verify a number of data structures in a form as close as possible to how they appear in the Java standard library. In a recent paper [47], they discussed a subset of these which were linked data-structures, including `ArrayList`, `HashMap`, and `PriorityQueue`. While small (on the order of ten components), this library represents the sort of effort we would like to make here—a library of components geared toward an exploration of a specific facet of verification. The Jahob library is geared towards linked data structures and aims to maintain the component from modern industrial systems.

A recent paper[6] deals directly with specification engineering, noting that quantifiers consistently complicate verification. It attempts to engineer the specifications of a set of data structures without quantifiers. This enables the specifications to be translated down to a first-order logic, then passed to an efficient first-order prover. This result, however, focusses on the translation process rather than the importance of specification style. To our knowledge, the team has not embarked on a systematic exploration of this facet of verification.

Since 2010, the Verified Software: Tools, Theories, and Experiments conference has run a verified software competition. Attendees are permitted to enter and are given a short amount of time to complete a handful of challenges. Afterwards, the various solutoins are made public along with discussion by their implementers.

These challenges have explored varied domains, ranging from averaging an integer array

to implementing a binary tree data structure. The intent has been to get verification projects communicating and sharing ideas rather than to compare different styles of specification.

This competition has revealed some interesting realities about modern verification systems *vis-á-vis* modularity. For example, in 2010, despite the fact that an earlier challenge required teams to create a list data structure, no team used that data structure in a subsequent challenge to implement an amortized queue with a linked list. They instead re-implemented a different linked list for subsequent challenge problem solutions, presumably because queues required different properties to achieve verification or the original implementation of linked lists was unable to take advantage of the necessary modular verification mechanisms.

In 2008, researchers here at Clemson and at the Ohio State University compiled and published[45] a set of incremental benchmarks intended to be representative of the breadth of verification complexity, starting with simple integer addition and spanning system I/O, design patterns such as Iterator, and finally an integrated application. The focus of these benchmarks was on demonstrating the capabilities considered essential to any verification system that was to be useful in practice. At a subsequent VSTTE conference, the Microsoft verification team published their solutions to some of these benchmarks as implemented in Dafny[27], which revealed a number of interesting properties of that system. This research contains our own implementation of some of these benchmarks.

Chapter 3

RESOLVE Background and Specification Engineering

This chapter contains necessary background information for understanding our verification technique. We begin with an overview of the RESOLVE system, and then discuss how we expect to leverage our techniques toward more easily-verified software using specification engineering.

3.1 RESOLVE Verification System

The RESOLVE[37] Verifying Compiler is an attempt to create a full verificatin pipeline, beginning with an integrated specification and programming language and continuing through to a back-end prover. The focus of RESOLVE is on modular verification and scalability. These are addressed by ensuring each component is verified in isolation and need not be re-verified regardless of the context of deployment. This means that encapsulation must be strictly assured, which requires that certain treacherous programmatic constructs such as aliases must be tightly controlled.

As an example, let's consider a Stack abstraction. We are interested in designing a system for complete verification—including constraints like memory, so we will use a bounded stack that takes as a parameter its maximum depth. Listing 3.1 shows part of a RESOLVE concept describing such a component.

Listing 3.1: A Stack Concept

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
    uses String_Theory;
    requires 1 <= Max_Depth;

    Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = empty_string;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| + 1 <= Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| /= 0;
        ensures #S = <R> o S;

    (* Other operations omitted for brevity. *)
end;
```

RESOLVE permits parameterized types. In this case, `Entry` is a generic type, permitting `Stack`s to be created to contain any other RESOLVE type. `Max_Depth` is an integer indicating the maximum number of elements that can ever be on a stack. The `evaluates` parameter mode indicates that the parameter is pass-by-value (parameters are ordinarily pass-by-reference.)

Applying modular lessons from the world of programming to mathematics, we store theories in separate files with their own scopes that can be imported individually[41]. The *uses* line includes one such theory—the theory of strings, i.e., finite sequences.

The `Family` clause introduces a family of abstract types called `Stack`s, modeled conceptually by strings of `Entry`s. The `exemplar` clause introduces a name to be used for an example stack, which is then used in the `constraint` clause to indicate that not *all* strings of `Entry`s are valid stacks, but rather only those of length `Max_Depth` and less. The `initialization` clause notes that all `Stack`s are assured to be empty when they are first created—it specified the constructor operation.

We then define some operations on `Stack`s—`Push()` and `Pop()`, with their associated pre- and post-conditions, introduced by `requires` and `ensures` clauses, respectively. Parameters to operations are preceded by *parameter passing modes*, which summarize the effect the operation will

have on a parameter—a parameter that's in `alters` mode is passed a meaningful value that might assume an arbitrary value by the end of the call; an `updates` parameter is given a meaningful value that will be changed to a new meaningful value as specified in the `ensures` clause; the input value of a `replaces` parameter is ignored and replaced with a meaningful value by the end of the call as specified in the `ensures` clause.

The `requires` and `ensures` clauses are mathematical assertions and their variables refer to the mathematical values of the parameters. So, while `Push` operates on programmatic `Stack`s, `S` in its `ensures` clause refers to the mathematical value of the passed stack—a string of `Entry`s. Because mathematical variables can have only one, unchanging value, we use the pound sign to indicate the value at the beginning of the call. Thus, `#S` refers to the value of the `Stack` at the beginning of the call and `S` refers to its value at the end. Inside a `requires` clause, all variables refer to the values of parameters at the beginning of the call.

Angled braces and the "`o`" operator come from `String_Theory` and represent the singleton-string constructor and the concatenation operator respectively.

Once a concept has been described, an implementation can be provided in a *realization*. Listing 3.2 provides a realization of a `Stack` using an array. Note that while we provide some syntactic sugar for arrays, a pre-processing step translates all array manipulation into interactions with a normal component and thus all reasoning is accomplished via a specification that models arrays as functions from integers to entries rather than hard-coded or specialized array reasoning. This contrasts with every other practical programming verification system we are aware of that supports arrays and provides an example of how often-primitive structures can be modelled normally within the framework of RESOLVE's flexible mathematical subsystem[1].

Listing 3.2: An Array Realization

```
Realization Array_Realiz for Stack_Template;

    Type Stack is represented by Record
            Contents: Array 1..Max_Depth of Entry;
            Top: Integer;
        end;
        convention
            0 <= S.Top <= Max_Depth;
```

[1] In addition to this array-based implementation, we have experimented with a linked-structure component in order to provide, in this case, a linked-list based implementation. The details of this exploration can be found in [24].

```
    correspondence
        Conc.S = Reverse(Concatenation i: Integer
        where 1 <= i <= S.Top, <S.Contents(i)>);

Procedure Push(alters E: Entry; updates S: Stack);
    S.Top := S.Top + 1;
    E :=: S.Contents[S.Top];
end Push;

Procedure Pop(replaces R: Entry; updates S: Stack);
    R :=: S.Contents[S.Top];
    S.Top := S.Top - 1;
end Pop;

(* Other operations omitted for brevity *)
end;
```

We represent our Stack programmatically as a *record* (similar to a *struct* in C) containing an array of contents and a top index. A `convention` is a representation invariant that must hold after each method except finalization, and may be assumed before each method except initialization. In this case, the top may be assumed to be at a valid index or 0, where it is permitted to reside if the Stack is empty.

A `correspondence` clause provides a relation between the programmatic `Stack` and its mathematical conceptualization. In this case, we use a mathematical definition `Concatenation`, which is to string concatenation what big Σ is to integer addition. Here, this clause states that the mathematical value of a `Stack` is equal to the reverse of the concatenation of the elements of the `S.Contents` array from 1 to `S.Top`.

A procedure is provided for accomplishing both `Push()` and `Pop()`, taking advantage of standard integer and array operations.

Using the specifications of these operations, the RESOLVE compiler is able to generate VCs establishing the correctness of this `Stack` implementation. The details of this generation process is the topic of [15]. As an example, this is the VC establishing the convention at the end of a call to `Push()`:

```
Goal:
(0 <= (S.Top + 1)) and ((S.Top + 1) <= Max_Depth)
```

Given:

```
1: (min_int  <= 0)
2: (0  < max_int)
3: (Last_Char_Num  > 0)
4: (1  <= (Max_Depth  + 1))
5: (min_int <= Max_Depth) and (Max_Depth <= max_int)
6: (min_int <= 1) and (1 <= max_int)
7: (1  <= Max_Depth)
8: (min_int <= Max_Depth) and (Max_Depth <= max_int)
9: (0 <= S.Top) and (S.Top <= Max_Depth)
10: (|Reverse(Concatenation i:Integer
      where (1 <= i) and (i <= S.Top), <S.Contents(i)>)|  + 1) <= Max_Depth)
11: E' = S.Contents((S.Top  + 1))
12: S'.Contents = lambda j: Z ({{E if j = (S.Top  + 1)
S.Contents(j) otherwise
}})
```

Note that in Given #10, the length of the result of the concatenation is clearly equal to `S.Top`, thus we know that `S.Top + 1` is less than `Max_Depth`, satisfying half the goal. Then, by Given #9, we see that `0 <= S.Top`, so clearly `0 <= (S.Top + 1)`, satisfying the other half.

Once generated, VCs are passed to RESOLVE's integrated minimalist prover. This prover was built as an extensible prover platform for verification experimentation[39] and is discussed more thoroughly in Chapter 4.

It is possible to define *extension operations* to a concept. These are secondary operations that permit useful additional functionality. As an example, `Stack_Template` has an extension operation called `Flip`, presented here as an enhancement:

```
Enhancement Flip_Capability for Stack_Template;
        Operation Flip(updates S: Stack);
                ensures S = Reverse(#S);
end;
```

Just as with a concept, enhancements may have multiple associated realizations, for which the VC-generation and proving process is the same. Consider the realization of `Flipping_Capability` given in Listing 3.3.

Listing 3.3: A Realization of Flip

```
Realization Obvious_Flip_Realiz for Flipping_Capability of Stack_Template;
    Procedure Flip(updates S: Stack);
        Var S_Flipped: Stack;
        Var Next_Entry: Entry;

        While (Depth(S) /= 0)
            changing S, S_Flipped, Next_Entry;
            maintaining #S = Reverse(S_Flipped) o S;
            decreasing |S|;
        do
            Pop(Next_Entry, S);
            Push(Next_Entry, S_Flipped);
        end;

        S_Flipped :=: S;
    end;
end;
```

This `Flip` procedure creates a local `Stack` called `S_Flipped`. It then iteratively pops each element off the original stack and into the local one. Having done so, the `:=:` operator is used to exchange the value of `S_Flipped` with `S`. Using swapping as its basic data movement operator enables RESOLVE to minimize undesirable aliasing[14].

Notice that the while loop requires programmer-supplied annotations. The *changing* clause establishes a frame property: only those values explicitly listed may change. The *maintaining* clause establishes a loop invariant expressing the logic of the loop. The *decreasing* clause establishes a termination metric to allow us to prove termination.

Such annotations are standard operating procedure for verification systems and can be seen across the board at, for example, the first VSTTE verification competition[22]. Some work has been done on automatically discovering, e.g., loop invariants on the programmer's behalf[11], however for the time being such automatic inferences remain limited, especially in the presence of data abstractions. Regardless, RESOLVE's explicit invariants do not preclude such a method from being employed to fill them in.

Using RESOLVE's integrated minimalist prover, a component of this research, we are able to fully and mechanically verify this procedure, as will be discussed further in Chapter 6.

3.2 Specification Engineering

In modern efforts to capture the behavior of existing software components and systems, specifications are often quite complicated, even for simple operations. Consider Listing 3.4 for the JML spec of the `add()` operation on a `List` data structure.

Listing 3.4: JML Specification for `add()`

```
public behavior
        requires !this.containsNull ==> o != null;
        requires (o == null)||\typeof(o) <: this.elementType;
        assignable theCollection;
        ensures \result ==>
                this.theCollection.equals(\old(this.theCollection.insert(o)));
        ensures \result &&\old(this.size() < 2147483647) ==>
                this.size() == \old(this.size()+1);
        ensures !\result ==> this.size() == \old(this.size());
        ensures !\result ==> \not_modified(theCollection);
        ensures this.contains(o);
        ensures_redundantly !\result ==> \old(this.contains(o));
        signals_only java.lang.UnsupportedOperationException,
                java.lang.NullPointerException, java.lang.ClassCastException,
                java.lang.IllegalArgumentException;
        signals (java.lang.UnsupportedOperationException);
        signals (java.lang.NullPointerException);
        signals (java.lang.ClassCastException);
        signals (java.lang.IllegalArgumentException;
```

The JML specification library seeks to formalize the existing informal specification of the Java Runtime Library, so it is bound by design decisions made agnostic of formal specification. This contributes to the complexity of the specifications in a number of ways:

- The original informal specification provides no guidance about how to deal with collections potentially containing more than `Integer.MAX_VALUE` elements.
- Language complexities like null elements and integer bounds must be taken into account.
- Without a correspondence established at the class level between the abstract value of the structure as a whole and its abstract fields, information that would otherwise be redundant

must be encoded. For example, certainly the fact that `this.contains(o)` follows from `old(this.theCollection.insert(o))`, but no correspondence exists between these two values.

Beyond these concrete issues demonstrated in this particular example, complexity arises from other areas as well:

- Capturing alias behavior, including the repeated argument problem.
- Using separation logic to describe allowable changes in memory while the method call runs.
- Poorly designed APIs.

Because specifications form the basis of the reasoning that becomes VCs, a complex specification such as this one necessarily leads to complex VCs. Because Java's `List` is intended as a reusable component, many clients will become victim to this complexity, increasing the difficulty of verifying code over the entire lifetime of this component.

The design of RESOLVE, on the other hand, attempts to minimize such complexities. There are no nulls to reason about. While integers are bounded, their bounds and constraints are asserted once in a first-class component and reasoned about at that level. A class-level correspondence establishes how the actual fields of the representation relate to an abstract value. RESOLVE is designed to have clean semantics[**?**], minimizing aliasing, marshaling reference behavior through a pointer data structure, and providing a well-defined semantic for repeated arguments.

Though no language can force its users to write good APIs, the constraints and rigor of the language contribute to encapsulated, modular design. Consider this specification for the comparable `insert()` operation on the RESOLVE list component:

ensures P.Prec = #P.Prec **and** P.Rem = <#New_Entry> o #P.Rem

While these design decisions encourage reuse and simplify *human* reasoning, they correspondingly simplify automated reasoning. Consider the VC given in Listing 3.5 derived from a stack reversing procedure.

Listing 3.5: VC for the Inductive Case of Loop Invariant

Goal:

```
(Reverse(S_Flipped')  o S'') = (Reverse((<Next_Entry'>  o S_Flipped'))  o S')
```

Given:

```
(((((((min_int <= 0) and
(0 < max_int)) and
((Last_Char_Num > 0) and
((Max_Depth > 0) and
((min_int <= Max_Depth) and
(Max_Depth <= max_int))))) and
(|S| <= Max_Depth)) and
S = (Reverse(S_Flipped') o S'')) and
|S''| /= 0) and
S'' = (<Next_Entry'> o S'))
```

The VC is easily solvable by expanding the variable S", then applying a few relatively straightforward transformations on strings. The presence of these sorts of VCs originally caused us to suspect we could get away with a very simple prover—one that first expanded variables, then substituted equivalent expressions until either the goal matched some given or the expression reduced to `true`. To test this hypothesis, we built a first version of the automated prover, discussed in the next chapter.

Chapter 4

Minimalist Automated Prover

At the core of any mechanical verification system is an automated prover responsible for discharging verification conditions for correctness (VCs). Because of that role, the prover determines whether of not a particular technique is increasing or decreasing the number of VCs that can be proved. As a result of this, most practical systems have focused on incorporating the latest and greatest provers into their retinue to piggyback on the breakthroughs at the bleeding edge of proving and artificial intelligence and thus increase provability.

While improved prover technology is certainly desirable and benefits many domains of computer science and mathematics, we hypothesize that in many cases *flexibility* may trump raw performance with respect to mechanically verifying well-engineered software. A flexible prover would permit the system user to focus on the task at hand—writing good specifications or code—in the terms most comfortable for them, rather than encouraging them to reason about the limitations of the prover. We believe this results in software that exposes the programmer's intuition and thus yeilds shallow VCs.

In order to experiment with this hypothesis and identify those prover capabilities and performance tunings required to verify well-engineered software, we have set out to create a *minimalist automated prover*, starting with only the bare essential capabilities and expanding them only when the present iteration left a significant number of correct VCs remained unprovable. The result of this effort was RESOLVE's integrated rewrite prover. As we've refined our design, it has become a platform for prover experimentation and a yardstick against which to measure our success using our new mathematical system to verify components.

4.1 Version 1 Prover

The first prover's design was extremely straightforward. As a preprocessing step, it expanded any variables (so for the VC in Listing 3.5, appearances of both `S` and `S''` would be replaced with their full values), then read in theorems from any available theories. These theorems were applied in alphabetical order by theorem name (in order to ensure consistency between experimental results) in a depth-first manner, with the search tethered at 6 steps to prevent infinite cycles.

Initially, only equality theorems (i.e., those of the form `A = B`) were considered, and they were permitted to be applied in either direction (i.e., replacing `A` with `B` or replacing `B` with `A`). This alone turned out to be sufficient to prove a surprising number of VCs arising from various contexts. However, we quickly found VCs like the one given in listing 4.1.

Listing 4.1: VC for the Requires Clause of Push

```
Goal:
(|S_Flipped'|  < Max_Depth)

Given:
(((((((min_int  <= 0) and
(0  < max_int)) and
((Last_Char_Num  > 0) and
((Max_Depth  > 0) and
((min_int  <= Max_Depth) and
(Max_Depth  <= max_int))))) and
(|S|  <= Max_Depth)) and
S = (Reverse(S_Flipped')  o S'')) and
|S''| /= 0) and
S'' = (<Next_Entry'>  o S'))
```

The proof for this VC is relatively straightforward: note that `|S| <= Max_Depth` and `S` itself is made up of `Reverse(S_Flipped')` and `S''`. Since we know that `|S''| /= 0`, the length of `Reverse(S_Flipped')` must be strictly less than `Max_Depth`. Since reversing the string does not affect its length, the length of `S_Flipped'` must also be strictly less than `Max_Depth`.

However, note that no amount of variable expansion and the application of equality theorems on the consequent will arrive at the solution. Indeed, even if we expand our process to permit equality theorems to act on the antecedent of the VC, we can't solve it. What we need is a theorem like:

```
Theorem Plus_Non_Zero_N:
    For all i : Z,
    For all n, m : N,
        n + m <= i and m /= 0 implies
            n < i;
```

Since developing the VC's antecedent with a theorem like this is considerably more expensive[1], this was implemented as a preprocessing step, with all available implication theorems applied in three rounds to the antecedent before continuing with the normal proof search using only equality theorems.

This permitted us to dispatch VCs like the one that appeared in Listing 4.1. However, at this point we had begun to amass a considerable number of theorems and the time required to successfully search all proofs of length no more than six for a solution was becoming untenable (over a minute for eventually successful proofs) and we began searching for heuristics to speed up this process.

The first thing we noticed was that *most* VCs required proofs of much shorter length (one, two, or three steps). So the prover was retrofitted to operate in phases, first searching proofs of length no more than three before trying those of length no more than four and finally trying those no more than six. This significantly reduced the time to prove many VCs, but still left others taking far too long.

Our second heuristic was to implement a greedy best-first search algorithm by creating a fitness function to determine which theorems were most likely to be helpful on a per-VC basis. We save discussion of this fitness function for Section 4.3.3, where we discuss optimization of the final version of the prover.

4.1.1 Implementation

This version of the prover was based on our existing abstract syntax tree data structure. A recursive loop implemented the main proof search using available equality theorems. However, a separate, hardcoded pre-processing step performed variable expansions and theory development.

[1]Each conjunct of the antecedent of the theorem must be matched against some given in the antecedent of the VC, and all possible matchings must be considered—making it exponential in the number of VC antecedents, with an order equal to the number of theorem antecedents.

4.2 Version 2 Prover

As the first prover continued to mature and found its way into our web-integrated environment, we continued to experiment with more complex components, including those derived from our burgeoning ideas on specification engineering[40] and one born out of our collaboration with the Intelligent River project here at clemson, that required a routine for averaging the integers in a queue[33].

These new domains yielded VCs that exposed fundamental weaknesses in the first prover and motivated the design and creation of the second prover. As an example of this sort of VC, consider Listing 4.2.

Listing 4.2: VC for the Requires Clause of Advance

```
Goal:
(|(Left_Substring(S.List, 0) o Right_Substring(S.List, (0 + 1)))| <
        |((Left_Substring(
                Reverse((Left_Substring(S.List, 0) o Right_Substring(S.List, (0 + 1)))),
                |(Left_Substring(S.List, 0) o Right_Substring(S.List, (0 + 1)))|)
         o <Element_At(0, S.List)>)
         o Right_Substring(
                Reverse((Left_Substring(S.List, 0) o Right_Substring(S.List, (0 + 1)))),
                |(Left_Substring(S.List, 0) o Right_Substring(S.List, (0 + 1)))|))|)

Given:
(((((((min_int <= 0) and
(0 < max_int)) and
(Last_Char_Num > 0)) and
(((S.Insertion_Point <= |S.List|) and
(0 <= S.Insertion_Point)) and
S.Insertion_Point = 0)) and
P_val = (|S.List| - 0)) and
((|S.List| - 0) >= 0)) and
Entry.is_initial(Temp'))
```

This VC expresses a tautology, but requires seventeen steps to prove. This raised serious concerns that we would not be able to use a simple term-rewrite prover as we had hoped—with hundreds or thousands of theorems in play and the inherent combinatorial complexity, it seems impractical to search a proof space that size, even with some guidance about which transformations

to prioritize.

Our insight here was that a mathematician, upon seeing this VC, spots a number of simplifications that should "obviously" be applied immediately. We subjectively speculate that part of what qualifies a proof step as "obvious" is the intuition that there's an extremely low likelihood we will wish to backtrack over that step in the future. For example, clearly `(0 + 1)` adds nothing and should be simplified to `1`. Similarly, because of the semantics of `Left_Substring`, $\forall S : \text{SStr}, \texttt{Left\_Substring}(S, 0) = \texttt{Empty\_String}$. And $\forall S : \text{SStr}, \texttt{Empty\_String} o S = S$.

In fact, if we continue to apply such "obvious" simplifications we can quickly reduce the same VC to how it appears in Listing 4.3.

Listing 4.3: Simplified VC for the Requires Clause of Advance

```
Goal:
|(Right_Substring(S.List, 1))| <
        |((Reverse((Right_Substring(S.List, 1)))
         o <Element_At(0, S.List)>))|

Given:
(((((((min_int <= 0) and
(0 < max_int)) and
(Last_Char_Num > 0)) and
(((S.Insertion_Point <= |S.List|) and
(0 <= S.Insertion_Point)) and
S.Insertion_Point = 0)) and
P_val = (|S.List| - 0)) and
((|S.List| - 0) >= 0)) and
Entry.is_initial(Temp'))
```

From here, a four-step proof suffices:

```
|(Right_Substring(S.List, 1))| <
        |((Reverse((Right_Substring(S.List, 1)))
         o <Element_At(0, S.List)>))|

|(Right_Substring(S.List, 1))| <                    by (|S o T| = |S| + |T|)
        |Reverse((Right_Substring(S.List, 1)))|
         + |<Element_At(0, S.List)>|

|(Right_Substring(S.List, 1))| <                    by (|Reverse(S)| = |S|)
```

```
        |Right_Substring(S.List, 1)|
         + |<Element_At(0, S.List)>|

|(Right_Substring(S.List, 1))| <                    by (|<E>| = 1)
        |Right_Substring(S.List, 1)|
         + 1

true                                                 by (i < i + 1)
```

We qualified such "obvious" steps as those that maintain the tautological property (i.e., could not make a tautologically true VC into one that is not tautologically true) and that *strictly reduce* the number of function applications. We hypothesized that if we could add such obvious simplifications to the proof search algorithm as a preprocessing step, we would see improvement on many VCs. Unfortunately, the proof search algorithm of the first prover was such that to do so would require more hard-coding. Intead, we conceived of a new prover, capable of being "driven" by an object that described its behavior. Every action it took would be part of the main proof loop, allowing for the consistent collection of metrics and the easy addition of new kinds of proof steps.

This new prover enabled us to implement the described pre-processing step, which we termed "Minimization"[2] using the same machinery as the main proof search, while still allowing us to dictate, e.g., that minimzation should not be backtracked over, while proof steps in the main search might be.

In addition to needing new pre-processing phases, new kinds of proof steps were required for the main proof-space exploration phase. Motivated by a number of examples that required extensive reasoning about integer bounds—including our queue averaging example and operations on bounded data structures like stacks—we found it was important to be able to strengthen the consequents of a VC by applying implication theorems as well as introduce and instantiate existentially-quantified variables. The design of the second prover permitted such new kinds of proof rules to be put into place.

XXX Need an example! XXX

[2]We avoided terms loaded with existing mathematical meaning, such as "Simplification", because minimization merely provides a best-effort heuristic for reducing function application count. Changing the available theorems or even the order of those theorems may affect the outcome of minimization.

4.2.1 Implementation

Our existing AST data-structure was poorly suited for formal reasoning and required constant, defensive deep-copying to ensure that changes made in one area would not propogate to another due to some shared reference. As a result, routine tasks became and inefficient. Exarcerbating this issue, the design of the AST left much to be desired and copying required many scattered, error-prone operations that often failed to copy important data attached to an AST node such as its type.

In order to address these shortcomings, we created a separate data structure to hold expressions that were involved in the proving process—both those related to the VC itself, and those that were part of theorems. This structure was immutable and thus permitted references to be passed and subtrees to be shared without fear of accidental modifications of the sort usually associated with shared aliases. Transforming the existing ASTs into this new immutable data-structure also provided a convenient point at which to sanitize and perform defensive checks, ensuring that those checks only needed to happen once.

Additionally, the design of the initial prover was not sufficiently flexible to permit steps other than equality substitution to be included in the proof-search algorithm without hard coding them. This required all pre-processing and simplification to take place outside the main prover loop, which made the collection of metrics such as run time or proof length difficult to keep, since they could not be collected uniformly.

By replacing the hard-coded proof search with a general mechanism that deferred to proving tactic objects, our second design permitted a more uniform mechanism by which steps were applied and rolled back. The basic implementation remained a recursive one, with backtracking falling out naturally from unwinding the recursive loop. Tactic objects implemented the *Strategy* pattern, which permitted significantly more flexibility on what exactly constituted a "step". Together, these design decisions permitted more consistent metric gathering as well as increased robustness.

4.3 Version 3 Prover

While the second prover was much more flexible and brought many new VCs into the realm of provability, we found that it did not meet our needs with regard to a number of non-functional attributres unrelated to its strict mathematical power. This motivated us to develop the current

version of the prover to address these issues.

4.3.1 Issues

4.3.1.1 Performance

The second prover eliminated the need to make deep copies at each step by ensuring that VCs were immutable and could thus be safely shared with deeper parts of the recursion. However, it did this at the cost of eliminating the possibility of making changes to expressions in place—each change had to spawn a completely new structure. While any unchanged subexpressions could be recycled, this generated a great deal of dynamic memory allocations that slowed down the tight prover loop. Additionally, while it shared this weakness with the first prover, the second prover's reliance on the old type system prevented proof states from being hashed efficiently and thus precluded a number of optimizations such as efficiently detecting proof cycles.

4.3.1.2 Debugging and Understandability

The capabilities and requirements of the prover are rapidly changing in a research environment and a number of design decisions of the second prover made implementation of new features or the improvement of existing ones very difficult.

Because of the tight recursive loop that applied steps and handled backtracking, adding or modifying cross-cutting concerns such as the collection of new metrics, visualizations of proof state, or prover interactions such as timeouts or cancel buttons became difficult. This had a corresponding effect on prover improvements, since introducing tools to examine proof state was correspondingly more difficult, and even setting sane breakpoints became challenging.

Additionally, the object that served to "direct" the proof was unweildy and difficult to understand or update. Many layers of inductive, embedded decisions were represented in a functional way that often required straightforward conceptual decisions (e.g., "apply this strategy until it cannot be applied anymore, then move on to this strategy", "only apply this strategy if this other one is not applicable") to be encoded in counterintuitive ways.

4.3.1.3 Metric Collection

While the second prover was an improvement, it still fell short of our metric-collection needs. Most steps were now governed by the consistent machinery of the main proof loop, but not all. For example, variable expansion was still the purview of a hard-coded pre-processing step. Additionally, in order to enable the sorts of complexities we required to be described by the strategy object, what should have been considered entire "phases" of the proof were occasionally embedded into single steps that made many different, unrelated modifications to the proof state.

An additional issue was that, with antecedent development now under the purview of the main prover loop, these steps contributed to proof-length-based metrics. However, since antecedent developments are purely speculative, in most cases, the vast majority of those steps did not contribute in any useful way to the eventual final proof. A mechanism was required to trace which steps actually contributed to the eventual result.

4.3.1.4 Educational Suitability

While the primary thrust of the RSRG's research is the development of a toolchain to broaden the scope of verified software, another important aspect of our research is the use of formal reasoning as an educational tool for teaching software engineering. Because the design of the first two provers had been single-mindedly on mathematical power and cross-cutting concerns were difficult to satisfy under that design, creating visualizations of the proof state, permitting the user to slow down and "watch" the automated prover work, or effectively allowing the user to control the course of the proof was very difficult. Both of the first two versions of the prover had a debug mode in which a window would appear at each step and permit the user to select the next theorem, but it was extremely rudimentary by necessity (see Figure 4.1).

4.3.2 Design and Implementation

For the third prover, visualization and control of the proof process, along with features to aid debugging and the writing of unit tests have been made first-class design concerns. To this end, it eschews the functional, recursive design of the first two provers in favor of a model/view/controller decomposition.

The third prover maintains the immutable expression data structures of the second version

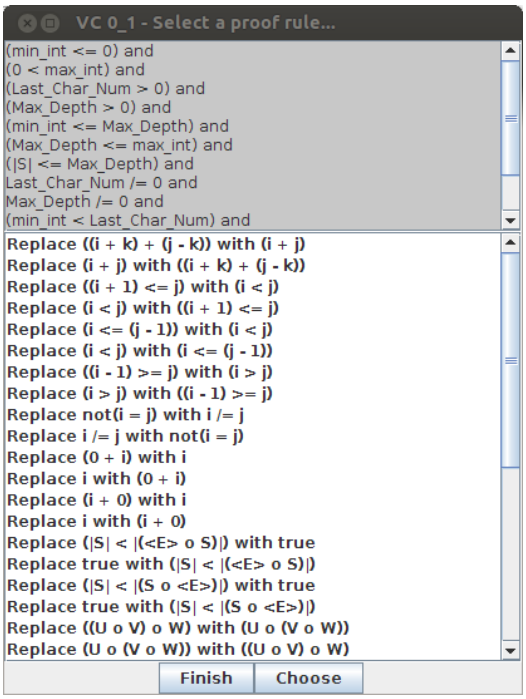


Figure 4.1: V1 and V2 Prover Visualization

and replaces the legacy math type system with the new one implemented for Chapter 5, which also uses immutable structures for the representation of types.

While the expressions are immutable, this version of the prover uses a single, mutable proof state object to hold information about proof state, implementing backtracking via explicit undoing of steps rather than recursive unrolling with copies at each level. This is significantly more efficient since often a proof step impacts only one small part of the proof state.

Each antecedent in the proof state is wrapped in an object that permits meta-information to be attached about where it came from. This information, in turn, permits the prover to trace backwards on a successful proof and determine exactly which antecedents (and thus exactly which proof steps) contributed to the final result.

Rather than a single strategy object that was built up by composing sub-strategies, the third prover's artificial intelligence is represented by a *goal stack*. The general proof loop repeatedly queries the top of this stack, giving the active goal an opportunity to take one of a small set of well-defined actions such as taking a single proof step or modifying the goal stack. This leads to a much more understandable controller and makes the construction of new goals straightforward.

Much consideration was given to a user interface that would make debugging easy and

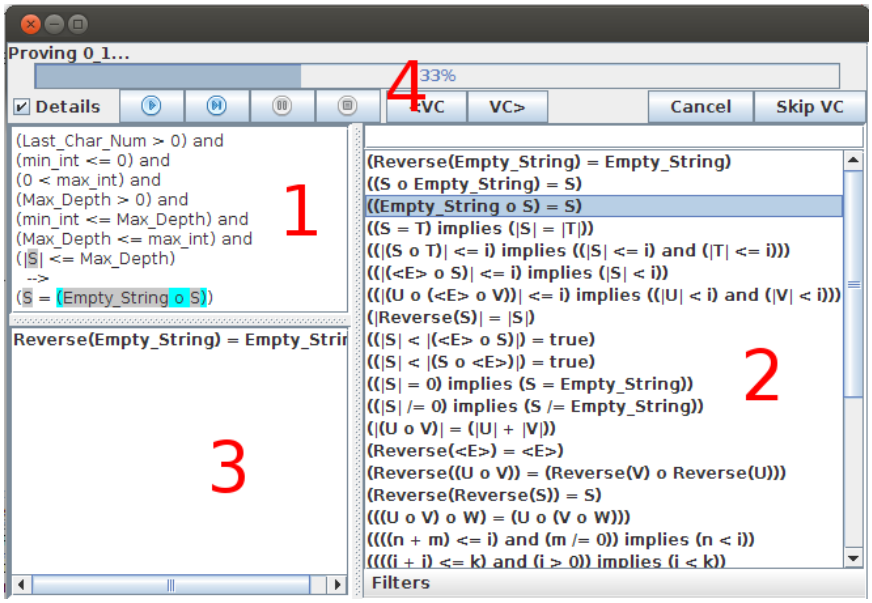


Figure 4.2: V3 Prover Interface

provide an excellent educational interface. An image of this interface is provided in Figure 4.2.

The current state of the VC is displayed in (1), while a list of available theorems is displayed in (2). The textbox at the top of the theorem list allows the user to search theorems based on the symbols it contains. (3) shows any proof steps taken so far, and clicking on one permits the user to undo it. Finally, play/stop/pause controls at (4) permit the user to seamlessly transition between interactive and automated proving mode.

When a theorem is selected in interactive mode, any possible applications of that theorem become highlighted in (1) in grey. Hovering the mouse over a possible application causes it to highlight, and selecting it applies the theorem. If no theorem has been selected from (2), each antecedent of the VC also becomes highlighted and may be selected and applied as an ordinary theorem.

When in automated proving mode, (1) updates periodically with the current state of the proof, allowing the user to visualize the work the prover is doing. Step functionality is also provided on pause to allow the user to watch the automated prover work step by step.

With fully immutable expressions, a number of optimizations become possible. Proof state is efficiently hashed using a polynomial hash that can be rapidly updated as expressions are added or removed. This permits us to rapidly detect proof cycles and prune those parts of the tree from the

search. Memoization is used when comparing types to ensure that once we establish a particular type relationship (which can be quite costly under the new analysis), we are not required to calculate it again. Finally, both the singleton and flyweight patterns are used along with object pools to minimize dynamic object creation.

Finally, for the collection of metrics, the current version of the prover outputs significantly more data in a more readable form than either earlier version. These proof files include all productive steps of the proof, with a snapshot of the VC at that time, as well as metrics about the proving process, such as length and elapsed time.

XXX Insert a figure of this output when it's working XXX

4.3.3 Domain-Specific Optimizations

An important hypothesis of this research is that a cutting-edge automated prover should not be required to prove the sorts of VCs that arise from well-engineered code. To this end a significant thrust of our experimentation with the prover has been in developing heuristics that assist it in dispatching not general mathematical statements, but rather the sorts of obligations that arise from code.

Each heuristic technique is presented at a high level here, and data on its effectiveness is presented in Chapter 6.

4.3.3.1 Intelligent Antecedent Development

Antecedent development refers to the process of establishing new known facts based on existing ones. For example, given `A ---> B` as a VC and a theorem stating `A ---> C`, we may develop the VC's antecedent into `A and C ---> B`. Because such developments always hold true and may often be efficiently searched, it is generally useful to spend some time expanding our list of known facts before embarking on our proof search. In some sense, this increases the size of the "target" we're trying to hit–since transforming the consequent into a match for *any single* antecedent allows us to establish the VC.

However, given the comparatively-large number of knowns at any step of a program (almost always more than is strictly neccessary to establish the VC), combined with the large number of available theorems and their frequently closed nature, blindly adding antecedents results in a

combinatorial explosion of required space that yields a low number of useful facts compared to irrelevant ones. We therefore use a number of heuristics to attempt to improve this situation.

Qualify and Avoid Useless Transformations. Ultimately, for automated verification to scale up, we must insulate the end users—whether the programmer or the mathematician—from the details of the prover. To this end we have routinely rejected designs and methodologies that require the theory developer to "tag" theorems or otherwise provide hints to the prover inside a theory or a program, a technique that is frequently employed by other mathematical reasoning systems (see, e.g., [20]).

First, note that while we often use language such as "applying a theorem", there is not a one-to-one correspondence between a theorem, which is a general statements of mathematical truth, and the ways that it can be applied. For example, a theorem that states `A = B` could be applied to reduce the expression `A = B` to `true`, or to replace an `A` with a `B`, or in the other direction to replace a `B` with an `A`. We term these individual ways of applying a theorem *transformations*.

Despite our lack of human-provided hints about theorems, it is often possible to qualify different kinds of resulting transformations and thus treat them differently based on their unique usefulness in a given situation. We use this technique to limit useless antecedent development by "spotting" identity functions and refusing to apply them to complexify a known fact. For example, consider this theorem from `String_Theory`:

```
Theorem Concatenate_Empty_String_Identity_Right:
    For all S : SStr, S o Empty_String = S;
```

Certainly, to apply this theorem from left to right would often be useful, but to apply it in the other direction adds little to the coverage of our antecedent. The antecedent development stage therefore ignores such identity-maintaining transformations when they increase the number of function applications.

Develop Antecedents Only About Relevant Terms. It is often the case that a VC contains many irrelevant antecedents providing information that is not useful to the final proof. In general it is not possibile to sort relevant from irrelevant without first arriving at a proof. As an extreme example of this, along dead code paths contradictory antecedents with no other bearing on the consequent may render otherwise unprovable VCs true. However, we recall our hypothesis that all

VCs ought to be straightforward and thus apply a simple heuristic: a new antecedent should only be considered useful if it tells us something about a term that appears in the consequent[3].

While in a general proof system, this would limit our useful developments, since proofs might be quite deep, based on our hypothesis that reasoning should be straightforward we expect that any required information should already be available roughly in the vocabulary of the consequent.

Maximize Diversity. Any transformation that may be applied to develop the antecedent has a dual that could be instead applied to the consequent. We've therefore tried to explore which kinds of steps are most advantageous during the antecedent development phase and which are better left for the consequent exploration phase. Intuitively, we note that the antecedent development phase, which occurs once as a preprocessing step and requires no back-tracking, is an ideal place to apply more computationally expensive steps; by contrast, the consequent exploration phase, which does not accumulate related facts and is therefore not subject to a combinatorial explosion of space, is an ideal place to explore many small variations[4]. We therefore hypothesized that antecedent-development time would be best spent establishing varied facts that put antecedents in new terms. To qualify this, we added the requirement that each antecedent introduced must both eliminate some term and introduce one.

Take, for example, this partial VC:

```
|S| > 0
  ---->
S /= Empty_String
```

We would permit the development `|S| /= 0`, since it eliminates the greater-than symbol and introduces the not-equal symbol, but not `|S| > 0 + 1`, since, while it introduces two symbols, it does not eliminate any.

Maintain Simplicity. We have discussed our minimization algorithm in Section 4.2. After each round of antecedent development, we minimize the resultant antecedents. This provides two advantages: first, it supports the diversity maximization from above by exposing antecedents that essentially establish the same fact in the same terms (which are then removed as redundant); sec-

[3]We take note of any equalities such as `A = B`, where `A` and `B` are terms, to permit developments about terms that are equivalent to terms that appear in the consequent.

[4]This informal analysis is confirmed empirically in Chapter **??**, but we note that it relies on a number of assumptions about the nature of the proving task and the specific proving algorithm.

ond, it increases the liklihood of transforming the consequent into an antecedent during the proof exploration phase, since both antecedents and consequent orbit the same "normal form".

4.3.3.2 Intelligent Consequent Exploration

Consequent exploration refers to the final phase of the proof search during which the consequent is repeatedly transformed using available theorems until it reduces to `true` or the search times out.

However, given the large number of available theorems and their frequently closed nature, the consequent space to be explored is extremely large, suffering from a combinatorial explosion of time. It therefore becomes important to search this space in a reasonable manner, since in general an exhaustive search will be computationally infeasible.

Minimize Complications. Before beginning consequent exploration in earnest, we apply the minimization algorithm described in Section 4.2. This eliminates many complications that to a human user would consitute "obvious" steps.

Qualify and Avoid Useless Transformations. As during antecedent development, we identify transformations that simply introduce identity functions and ignore them during our proof search.

Detect Cycles. While the VC state itself is mutable, all components of the state (the expressions and their types) are immutable, which lends itself to efficient calculation of expression hashes. Using a polynomial hash, the overall hash of the VC state can be updated in constant time each time an expression is added, removed, or modified. In this way, we can efficiently detect cycles and thus not bother to inductively explore beyond them.

This ability also supports our desire to divorce theory creation from reasoning about the prover in the following way: mathematical systems like ACL2[20] and Issabelle[46] require theorems to be annotated with the direction in which they should be applied in order to prevent cycles that arise from the same theorem being applied repeatedly backward and forward. Our prover is able to detect such a situation and avoid it.

Tether Search. The main consequent exploration algorithm is a depth-first search. While cycle-detection eliminates one class of infinite, unproductive path, others exist. For example: the prover

might repeatedly add one to both sides of an equation. Consistent with our hypothesis that VCs should be straightforward to prove, we tether the search to a short length, after which no further exploration will be applied. While this depth is parameterizable, we've found that the default of four is generally sufficient.

Step Intelligently. Rather than step blindly, we apply a greedy best-first algorithm in our search. In order to quantify what we mean by "best", we must establish a useful fitness function for each transformation. Empirically, we have found that re-calculating fitness at each step is far too costly in terms of time, but we are able to find a sensible ordering of available theorems on a per-VC basis.

We identified three criteria on which to determine the fitness of a transformation: 1) what effect does applying the transformation have on the unique symbols present? Does it introduce new unique symbols? Does it eliminate existing ones? 2) what effect does applying the transformation have on the number of function applications? 3) how many symbols does the transformation share with the VC?

After experimentation, we have found that the third criteria is not useful, since a transformation attempting to match symbols not contained in the VC can never be applied, and thus any such symbols that might be *produced* by the transformation, must be newly unique, which is subsumed by the second criteria.

By deprioritizing transformations that will introduce new symbols, we prevent the prover from "entering new territory" before it's finished exploring all the ways it can transform the current symbols. Similarly, by deprioritizing transformations that increase the number of function applications, we encourage proof states toward simplicity and parsimony that are easier to explore. In a *general* proof system, one might assert that these tactics are just as likely to lead us *away* from a correct proof as to bring us closer, but in the specific domain of verifying well-engineered software, we hypothesize that the programmer is not taking leaps of logic and thus the reasoning should be simple.

Chapter 5

Mathematical Flexibility

Introducing formal methods into a software engineering project causes a sharp increase in the up-front effort required to develop the software. Appropriate mathematical theories must be developed, components must be formally specified by a software engineer trained in formal techniques, and verification conditions must be dispatched by a mathematician in a formal environment that eliminates the possibility of human error, possibly with the assistance of an automated proving tool.

While we argue that this cost is justified in critical or long-running deployments by a corresponding reduction in effort during the maintenance phase of the software lifecycle, it is undeniable that this remains a barrier for entry to many projects that could otherwise benefit from formal methods of quality assurance. We hypothesized that a number of steps could be taken to reduce the real or percieved additional effort, as well as increase the corresponding long-term benefits.

5.1 Preliminaries

For the sake of clarity and brevity, we will explain a few important concepts and notations for this section upfront.

5.1.1 Objects *vs.* Expressions

We will occasionally need to distinguish between a *mathematical value and its corresponding type* and a *RESOLVE expression and its corresponding type.* The former is a mathematical object representable in the universe of RESOLVE. The latter is a meta-concept pertaining to syntax.

When we wish to indicate a mathematical value, we will use the standard mathematical syntax. E.g., $\forall i : \mathbb{Z}, ...$ quantifies over all mathematical integers. When we wish to indicate a RESOLVE expression of some type, we will use the meta-type **Exp**, subscripted with the type of the expression. E.g., $\forall i : \mathbf{Exp}_{\mathbb{Z}}, ...$ quantifies over all RESOLVE expressions that, when evaluated, would yield a mathematical integer. We will also use the meta-function *Eval*, which takes an **Exp** and returns the results of evaluating the expression.

5.1.2 Operating on Expressions

When normal mathematical functions (e.g., $\cup$, $\times$, etc.) are shown operating on **Exp**s, this is a shorthand for constructing a new RESOLVE expression resembling the given expression, with any **Exp**-valued variables macro-expanded into their full expression value. For example, $\forall R, T : \mathbf{Exp}_{\mathcal{P}(\mathbb{Z})}, \text{Eval}(R \cup T) \supseteq \text{Eval}(R)$ does not contain an attempt to somehow "union" two expressions ($\cup$ operates on types, not expressions), but rather is constructing a new union expression with R and T as sub-expressions.

5.1.3 Type Equality

Equality of types can be a complex topic. In this text, when we say, for some types T and R, that $T = R$ (or use the word "equals") we mean that T and R are *alpha equivalent*. Two types are alpha equivalent if they are identical except for the names of any quantified variables, which may be different so long as they do not change the internal relationships between the variables (i.e., the names can not be changed to make two previously differently-named variables the same.)

5.2 General Design Goals

After considering the problem, we arrived at three overarching strategies to address verificaton complexity, then brainstormed a number of features that would support these objectives. This section covers the overarching strategies, while the next lists specific features and explains how they support these strategies.

5.2.1 Usable Theories

Certainly the most straightforward way of addressing this issue is simply to reduce the effort associated with formal verification. These complexities may generally be understood to arise from two situations: writing theories, along with utilizing them in specifications (that is, *producing* mathematical content) and dispatching verification conditions, either by hand or via an automated prover (that is, *consuming* mathematical content).

The first situation may be addressed by making it as easy as possible to work with the mathematical subsystem of the verification tool. *Familiarity* is the most obvious kind of ease and encompasses both syntax and environment. Because the target users of the RESOLVE mathematical subsystem are mathematicians, we strive to make the RESOLVE theory language look and behave as much like traditional mathematics as possible. Another kind of ease is *tool support*, wherein the compiler assists the user in spotting and correcting errors.

The second situation may be addressed by making VCs easier to dispatch. While the subject of increasing the power of the automated prover to encompass more VCs is the topic of Chapter 4 of this dissertation, we can affect this effort in the theory development domain by creating theories that result in *more straightforward* VCs.

5.2.2 Modular Theories

Borrowing from the software development world, we note that when complexity is unavoidable, it can be mitigated by strong modularity in the form of clear demarcations between atomic conceptual units. Such units serve both to organize the thoughts of the human user and to provide a finer granularity with which to import such concepts, decreasing the clutter in namespaces and ensuring that only the resources that are truly needed are loaded. In the presence of an automated prover, we hypothesized that such a strategy would also assist the ease with which VCs can be dispatched by ensuring that the prover only considers those theorems relevant to the current domain.

Note that it's not our desire to place on the user the burden of deciding which theorems might be relevant, but rather to ensure that appropriate theorems are bundled with the mathematical objects upon which they operate. A user might work with binary relations a great deal, but only when she wishes to use the `Is_Total_Preordering` predicate would she import `Preordering_Theory`.

By importing what is necessary to gain access to such a predicate, she also gains a body of theorems for reasoning about such relations.

5.2.3 Reusable Theories

Finally, we note that a theory or component that enjoys continuous reuse amortizes over time the up-front effort required to create and verify it. We therefore sought to create and add those features that would permit the development of reusable theories, types, theorems, and components. Such a strategy both decreases the effort required to create new theories and increases the liklihood that an existing theory is already available that can meet the user's needs.

In seeking out such features, we drew from the body of existing programming languages techniques, including encapsulation, polymorphism, and inheritance, and sought to adapt them for specifications and theories.

5.3 Concrete Features

5.3.1 Higher-order Logic

The availability of first-class functions in theories and specifications both increases mathematically *familiarity* (supporting usability) and encourage patterns of reuse. Many reuse patterns found in modern programming languages are difficult or impossible to specify or verify using the first-order logic dictated by most practical verification systems and automated provers. For example, the Strategy patten, functional constructs, and ??? all become impossible to specify.

Higher-order logic is presumably omitted from such systems because of the added complexity introduced by reasoning over quantified functions. However, as discussed further in Chapter 4, our minimalist prover leaves functions and definitions *uninterpretted*. A function or definition is uninterpretted when we do not expand it to consider its definition—i.e., the mathematical subsystem looks at a function or definition variable as a black box, treating it no differently from an ordinary variable. As a result, quantifying over functions introduces no additional complexity. The tradeoffs inherent in such a design decision are discussed more completely in [42].

In addition to the familiarity gained by permitting mathematicians to treat functions as first-class citizens, their presence introduces a number of dimensions of usability and reuse:

5.3.1.1 Higher-order Theorems

Consider, for example the $foldr$ function ubiquitous in functional languages. $foldr$ takes as its parameters a starting value of type γ, a function of type $(\gamma * \delta) \rightarrow \gamma$, and a list of elements of type δ. Starting with the starting value and the first element of the list, the function is applied to yield a new value of type γ before repeating the procedure with the resultant value and the next element of the list. The result of the final function application is returned. A summing function for lists of integers could thus be defined as:

$sum(zs) = foldr(0, +, zs)$

The broad applicability of such a function for specification should be obvious. However, even simple theorems describing the mathematical properties of this function run afoul of the first-order restriction that functions may not be quantified over. For example, Theorem 1 states that $foldr$ applied with an initial value to an empty list simply returns the initial value:

Theorem 1. $\forall f : (\gamma * \delta) \rightarrow \gamma, \forall ds : List(\delta), (|ds| = 0) \Rightarrow (foldr(i : \gamma, f, ds) = i)$

RESOLVE permits such a theorem, enabling the development of a *theory* of $foldr$, that in turn permits the automated prover to reason about expressions using such an expression at a high level of abstraction.

5.3.1.2 Generic Theories of Functions

Quantifying over functions also provides a straightforward mechanism for developing bodies of theorems that may be quickly applied to a new function. This permits a number of useful properties to be proved once in the general case, and then be reused over multiple different instantiations. Consider this snippet from `Preordering_Theory`:

```
Precis Preordering_Theory;

    Definition Is_Total_Preordering(f : (v1 : (D : MType) * v2 : D) -> B) : B;

    Theorem Total_Preorder_Total:
            For all D : MType,
            For all f : D * D -> B,
            For all x, y : D,
                    Is_Total_Preordering(f) implies
                            f(x, y) or f(y, x);
```

```
Theorem Preorder_Reflexive:
        For all D : MType,
        For all f : D * D -> B,
        For all x : D,
                Is_Total_Preordering(f) implies
                        f(x, x);

Theorem Preorder_Transitive:
        For all D : MType,
        For all f : D * D -> B,
        For all x, y, z : D,
                Is_Total_Preordering(f) implies
                        Is_Transitive(f);

end;
```

Now imagine we are defining a new operator, `Compare_Zero_Count`, which takes two `Str`s of Z and compares the number of occurances of zero:

```
Definition Compare_Zero_Count(S1 : Str(Z), S2 : Str(Z)) : B;
```

Clearly, such a function represents a total preording, and those are properties we may wish to rely on. Rather than state the properties of a total-preordering again, specifically for this function, we may instead simply add:

```
Theorem Compare_Zero_Count_Is_Total_Preordering:
        Is_Total_Preordering(Compare_Zero_Count);
```

This theorem must be supported with a proof, of course, which requires that we establish that the function in question have the properties of transitivity and totality. Following this, the full body of theorems available about total preorderings applies to `Compare_Zero_Count`.

It may at first seem that this is not much of a help—in order to use some of the theorems, we must first establish them, saving us no work. However, note that the `Preorder_Reflexive` theorem is not a defining property of a total preordering, but rather follows from `Total_Preorder_Total`. Such theorems are now available for free with `Compare_Zero_Count`, because we are able to establish, once, in this module, that any function meeting the two properties of total preorder also meet these other theorems.

Note also that since such a statement serves to associate a symbol in one module with those in another, and that only those theorems in imported modules will be available—a decision which may be delayed until a third module *incorporates* `Compare_Zero_Count`, this also strengthens the modularity of our theorems.

5.3.1.3 Extensible Specification

Inheritence is a powerful tool, but the cause of much problematic reasoning[43]. While RESOLVE does not support direct inheritence, a specification may provide points for extension by permitting function parameters that modify its behavior. This, in turn, can permit a client to simplify their own reasoning while still using an off-the-shelf component.

As an example, consider the `Predicate_Stack`, which ensures a predicate holds on each of its elements:

```
Concept Predicate_Stack(Type Entry, Definition Predicate : Entry -> B);

    ...

    Operation Push(alters E : Entry, updates S : Predicate_Stack);
        requires Predicate(#E);
        ensures S = #S o <#E>;

    Operation Pop(replaces E : Entry, updates S : Predicate_Stack);
        requires |S| > 0;
        ensures #S = S o <E> and Predicate(E);

end;
```

Just as a type-parameterized stack component prevents the user from needing to reassure the type system of the types of elements as they are removed from the stack, a predicate-parameterized stack component prevents the user from engaging in complex gymnastics to assure the verification system of properties that hold on its elements each time they are removed—those properties may simply be assumed.

5.3.1.4 Strategy Pattern

The *Strategy Pattern* permits an operation to be encapsulated in an object that can then be programmatically manipulated, e.g., by being passed as a parameter. This pattern is an important one for resuse, since it allows a client to inject an algorithmic decision into a larger component. By utilizing first-class functions in specifications, we are able to formally specify the strategy pattern, which to our knowledge is unique among practical programming systems.

Consider `Lazy_Filtering_Bag`, a component which permits elements to be added, the retrieved in no particular order. At instantiation time, the client provides a *filtering strategy*, which is applied to each element as it is removed:

```
Concept Lazy_Filtering_Bag_Template(Type Entry, Definition Filter : Entry -> Entry);

        Family Lazy_Filtering_Bag : MultiSet(Entry);
                exemplar B;
                initialization ensures B = Empty_Multi_Set;

        Operation Add(alters E : Entry, updates B : Lazy_Filtering_Bag);
                ensures B = #B + {#E};

        Operation Retrieve(replaces E : Entry, updates B : Lazy_Filtering_Bag);
                requires |B| > 0;
                ensures there exists F : Entry,
                        #B = B + {F} and E = Filter(F);
end;
```

We are then able to create a realization that provides parameter to implement the mathematical concept of `Filter` with a procedure.

```
Realization Stack_LFBag_Realiz(
                Operation DoFilter(updates E : Entry);
                        ensures E = Filter(#E);)
                for Lazy_Filtering_Bag_Template;

        ...

        Procedure Retrieve(replaces E : Entry, updates B : Lazy_Filtering_Bag);
                Pop(E, B);
                DoFilter(E);
```

```
        end;
end;
```

And finally we may instantiate our realization, providing a filter and reasoning about the results of manipulating our `Lazy_Filtering_Bag`.

```
Definition Integer_Half(i : Z) : Z = Div(i, 2);

Operation Half(updates I : Integer);
        ensures I = Integer_Half(I);
Procedure;
        I := I / 2;
end;

Facility Bag_Fac is Lazy_Filtering_Bag_Template(Integer, Integer_Half)
        realized by Stack_LFBag_Realiz(Half);

Operation Main;
Procedure;
        Var I : Integer;
        Var B : Lazy_Filtering_Bag;

        I := 5;
        Add(I, B);
        Retrieve(I, B);

        Assert I = 2;
end;
```

5.3.2 First-class Types

First-class types are a feature of several mathematical systems and a handful of experimental programming languages, but are rarely found in practical verification systems. When types are treated as a special case they are difficult and inconsistent to manipulate, limiting the facility of mathematical extension.

RESOLVE now incorporates first-class mathematical types that are treated as normal mathematical values. They can be manipulated, passed as parameters, returned as the result of a relation,

and quantified over. This provides both a great deal of flexibilty, as well as a straightforward mechanic for specifying certain generic programming paradigms (most obviously: parameterized type variables).

Because there are now type values, this implies those values are, in some sense, *of type Type*. We call this type **MType** as an abbreviation for *Math Type*[1].

For example, the following line would introduce a particular integer called **1**:

```
Definition 1 : Z = succ(0);
```

In exactly the same way, a new type called **N** could be defined:

```
Definition N : Power(Z) = {n : Z | n >= 0};
```

The symbol table maintains information about the kinds of elements that make up any existing class and can thus infer when the symbol introduced by a definition can safely be used as a type. This corresponds to the static surety that *the declared type of the symbol is a class known to contain only **MTypes**.* For brevity we will call this predicate τ. That is, $\tau(T)$ is true if T is known to contain only **MType**s. The question of whether or not such a class is inhabited is important, and our current design calls for any assertion that an element is in a set raise a proof obligation that the set is inhabited. We note, however, that this may be cumbersome and alternatives exist, including insisting that such a type be demonstrated to be inhabited when it is defined, and requiring the user to explicitly state that the set is inhabited (and proving that statement), before permitting the set to be used as a type. Module this complexity, the full list of judgements for statically determining if a symbol's type meets this property is given in Figure 5.1. (Judgement syntax is read as follows: "in order to demonstrate what is below the line, it is sufficient to demonstrate what is above the lone".)

Thus this is a valid sequence of definitions:

```
Definition N : Power(Z) = {n : Z | n >= 0};
Definition NAcceptor(m : N) : B = true;
```

But this one is not:

```
Definition 1 : Z = succ(0);
Definition OneAcceptor(o : 1) : B = true;
```

[1] `MType` is comparable to * in Haskell, where * is the kind of a type.

Figure 5.1: Static judgements for determining if a symbol may be used as a type

$$\frac{\top}{\tau(\mathbf{MType})} \quad \text{(a)} \qquad \frac{\top}{\tau(\mathbf{Set})} \quad \text{(b)} \qquad \frac{\top}{\forall T : \mathbf{Exp_{MType}}, \mathrm{Power}(T)} \quad \text{(c)}$$

$$\frac{\tau(T)}{\forall T : \mathbf{Exp_{MType}}, \forall p : \mathbf{Exp}_{\mathbb{B}}, \{t : T | p\}} \quad \text{(d)} \qquad \frac{\tau(T_1) \wedge \tau(T_2) \wedge ...\tau(T_n)}{\forall T_1, T_2, ...T_n : \mathbf{Exp_{MType}}, T_1 \cup T_2 \cup ...T_n} \quad \text{(e)}$$

$$\frac{\tau(T_1) \vee \tau(T_2) \vee ...\tau(T_n)}{\forall T_1, T_2, ...T_n : \mathbf{Exp_{MType}}, T_1 \cap T_2 \cap ...T_n} \quad \text{(f)} \qquad \frac{\bot}{\forall T, R : \mathbf{Exp_{MType}}, T \times R} \quad \text{(g)}$$

$$\frac{\bot}{\forall e : \mathbf{Exp_{Entity}}, \forall f : \mathbf{Entity} \to \mathbf{MType}, f(e)} \quad \text{(h)}$$

Type schemas and dependent types, which define generalized types parameterized by arbitrary values take advantage of the first-class nature of types and can be defined as normal relations that return a type, rather than using a special syntax.

In addition increasing usability by reducing special cases, this flexibility permits increased modularity and reuse in a number of ways:

5.3.2.1 Generic Types

Generics in the mathematical space fall out of first-class types easily. In fact, in the examples above, the "type" `Power(Z)` is not a type at all, but rather the application of a function from a type to a type. We say such a function defines a *type schema*. As another example, we may consider this snippet of `String_Theory`, where we first introduce the set of strings of heterogenous type, `SStr`, before introducing a function to yield restricted strings of homogenous type, `Str`:

```
Definition SStr : MType = ...;
Definition Str : MType -> Power(SStr) = ...;
```

We are thereafter able to declare variables of type, e.g., `Str(Z)` without issue. Happily for reuse, any theorems defined to operate on `SStr`s are equally applicable to `Str(...)`s.

5.3.2.2 Generic Theories of Types

Just as we are able to create generic theories of functions, we may do the same for types—after all, both functions and types are merely mathematical values. This means that theories of kinds of types may be developed to provide utility functions and useful theorems.

Consider the class of types with inductive structure such as lambda expressions, mathematical strings, and trees. Such a theory might establish important properties of well-foundedness, the finite nature of the objects, as well as provide utility functions for traversing them.

5.3.2.3 Higher-order Types

With first class types it becomes possible to quantify over them, and thus to support type schemas with theorems. For example:

```
Theorem All_Equivalent_and_Not_Singleton_Means_Empty:
        For all T : MType,
        For all t1, t2 : T,
                t1 = t2 and not Is_Singleton_Type(T) implies
                        Is_Empty_Type(T);
```

5.3.3 Inferred Generic Types

While we get generic types in the mathematical realm for free, it is often useful to *infer* such types based on what is provided. Consider how cumbersome the set membership function would be without this feature:

```
Definition Is_In_Set(T : MType, S : Set(T), E : T) : B;
```

We therefore have added specific syntactic sugar to permit, for example, the type of elements of the set to be inferred, thus increasing the usability of the theory:

```
Definition Is_In_Set(S : Set(T : MType), E : T), B;
```

Such inferred type parameters may be nested arbitrarily deep and are extremely flexible. As with explicit type parameters, they are evaluated left to right and a captured type may be used later in the same signature.

In order to keep the type system simple (from the user's perspective) each actual parameter may take advantage of only one of the type theorem mechanism or the inferred generic type mecha-

nism. The type system will not search for equivalent types that might have a form suitable to bind an inferred generic type.

5.3.4 Rich Type System

When expressing types in a mathematical system, it is natural to look to the Set abstraction. However, in doing so one must be careful not to inherit any of the many paradoxes and inconsistencies that have dogged the development of theories of sets. Many schemes exist to correct the deficiencies in naive set theory, with most pure systems using theories based on inductive structures and intuitionistic logics, as these are often well suited to computation. However, these are quite disjoint from a modern mathematician's conception of the universe. We choose instead Morse-Kelley Set Theory to be our basis, augmented with higher-order functions.

First, note that RESOLVE values come in two flavors—mathematical values like the empty set, ϕ, and programming values like the empty array. We may trivially show that there are a finite collection of progamming values (after all, there are only a finite number of machine states,) and thus, without loss of generality, we may confine our thinking to the mathematical values, trusting that we can, if nothing else, provide an explicit mapping between the two later. We thus dispense with distinguishing between them for the moment, using "value" to mean "mathematical value".

Morse-Kelley Set Theory (MK) defeats Russel's Paradox in the same way as, for example, von Neumann-Bernays-Gödel Set Theory, by imagining that sets are each members of a larger meta-set called the *classes* and admitting the existence of *proper classes*—i.e., set-like objects that are not sets. We then restrict sets to containing only non-set values, while proper classes may contain sets (but nothing may contain a proper class.) Under this light, we may rephrase the classic example of Russel's Paradox into "the class of sets that don't contain themselves", and view its contradiction not as an inconsistency, but rather as a proof that the class in question is proper.

MK permits us all the familiar and natural set constructors, restricted only by the necessity to reason carefully about what classes might be proper. This is ideal, since mathematicians need not be limited by glaring restrictions to class construction that exist only to eliminate corner-case inconsistencies. While, in general, only a formal proof can establish a given class as a set, in most cases we can infer it easily as most constructors are closed under the sets—e.g., the union of two sets is always a set.

We will imagine the universe of MK classes to be our universe of types—that is, `MType`

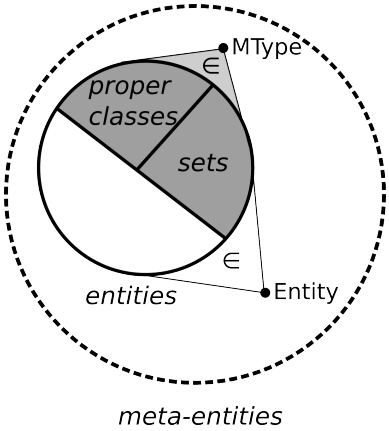


Figure 5.2: A High-level View of the RESOLVE Mathematical Universe

from Section 5.3.2. Because describing the class that contains all classes would once again introduce Russel's Paradox, we imagine `MType` is a *meta-class* that exists "above" the classes, just as the classes exist "above" the sets.

We will imagine that the union of `MType` with all those things that can be elements of some class to be the universe of RESOLVE values. We will call this meta-class `Entity`. Note that while we may permit `MType` to be used as part of a type signature, we must be careful not to let it be passed as a parameter—it is not a value. Though we might permit it to be used informally by imagining that when used as a value it indicates some broad subset of types.

With all this in mind, the RESOLVE mathematical universe can be visualized as in Figure 5.2.

5.3.5 Mechanisms for Static Reasoning

While the flexibilities in the previous sections result in a very expressive language that allows mathematical theories to be created in straightforward, obvious ways, this section is devoted to those features that permit tool support in the form of static reasoning despite the complexities introduced by this expressiveness. At a high level, we use the technique of allowing the mathematician to explicitly state important relationships that would otherwise by undecidable to reason able. These relationships must be proved, at which point they are incorporated into the static checker.

5.3.5.1 Type Theorems

Design A practical problem with first-class types is the ability to specify types and type-matching situations that are undecidable. This is an issue common to all systems that permit dependent types—which a system of truly first-class types must necessarily do.

For example we could imagine two types defined by arbitrary predicates:

```
Definition T1 : MType = {E : MType | P(E)};
Definition T2 : MType = {E : MType | Q(E)};
```

Determining whether or not an object modeled by `T1` can be passed where one modeled by `T2` is expected is equivalent to asking if $\forall e$:`MType`$, Q(e) \rightarrow P(e)$, which is, of course, undecidable.

Some verification systems (e.g., Coq) take advantage of this very property as an aid to verification, reducing all programs to reasoning about complex types and casting verification as a type-matching problems. However, because our goal is to increase the usability of our system by providing the user with strong tool support, we would ideally like to provide the usual static type-safety expected by object-oriented programmers.

In order to provide such static type checking in a variety of useful (but potentially undecideable) situations, we rely on the mathematician to provide and prove explicit type mappings in cases where the reasoning is complicated.

In some cases, relationships can be easily inferred. For example, the case of simple sub-types:

```
Definition Z : MType = ...;
Definition N : Power(Z) = ...;
```

Here we can easily note in the symbol table that an `N` would be acceptable wherever a `Z` is required and permit such a shift in conception-of-type in certain well-defined cases.

However, hard-coding such relationships does not provide a general mechanism for reasoning about type relationships and with first-class types, we may quickly arrive in situations where we'd expect the ability to reason about complex type relationships without requiring explicit assertions from the programmer or mathematician. Consider this application of `Str`s:

```
Definition Average(S : Str(Z)) : Z = ...;
Definition SomeNs : Str(N) = <1, 10, 3, 19>;
Definition AverageOfSomeNs : Z = Average(SomeNs);
```

In an ideal world we would expect this to type-check. But it is not the case that for two arbitrary type schemas, if the parameters of the one are subtypes of the parameters for the other,

then the results are themselves subtypes. Consider this (somewhat contrived) complement string type:

```
Definition ComplementStr(T : MType) : MType =
        {S : UntypedStr | For all i, where 0 <= i < |S|,
                Element_At(S, i) not_in T);
```

This is the type of all strings containing possibly-heterogenously-typed elements where no element is of type `T`. Clearly, *this* set of definitions should *not* typecheck:

```
Definition Average(S : ComplementStr(Z)) : Z = ...;
Definition NotSomeNs : ComplementStr(N) = <-1, -10, -3, -19>;
Definition AverageOfNotSomeNs : Z = Average(SomeNs);
```

However, the same thing that got us into this mess—first-class types—provides a road out. Because types are normal mathematical values and we already have a mechanism for asserting theorems about mathematical values, we can use that existing mechanism to provide information about type relationships.

Because such theorems must take a specific form if they're to be understood by the static type-checker, we add some special syntax, calling them `Type Theorem`s instead of ordinary `Theorem`s. This flags these theorems for the type checker and ensures that we can raise an error if they do not have the proper form.

This design splits the difference between a rich, expressive type system and the straightforward static typing programmers have come to expect. Simple cases can be covered without thought on the part of the programmer, while complex, undecidable cases are permitted by explicitely deferring to the prover.

Such theorems have a number of uses:

Simple Subtype Here is a type theorem stating that, among other things, `Str(N)`s should be acceptable where `Str(Z)`s are required:

```
Type Theorem Str_Subtype_Theorem:
        For all T1 : MType, For all T2 : Power(T1),
        For all S : Str(T2),
                S : Str(T1);
```

As with any other theorem in the RESOLVE system, this one would require a proof to establish its correctness and maintain soundness. We assume the presence of a proof-checking

subsystem and leave such proofs outside the scope of this research.

Overlapping Types Sometimes, more complex relationships are required. For example, in some circumstances, providing a `Z` where an `N` is expected should be acceptable. We can use the same mechanism to provide for this case:

```
Type Theorem Z_Superset_of_N:
        For all m : Z, (m >= 0) implies m : N;
```

This permits us, under a limited set of circumstances, to provisionally accept a "sane" type reassignment, while raising an appropriate proof obligation that the value in question is non-negative. This is similar to Java, where sane typecasts (i.e., from a `List` to an `ArrayList`) are permitted, but cause a run-time check to be compiled into the code. Here, however, we pay the penalty only once—during verification-time—rather than with each run of the program.

Complex Expression Type We can also use this flexibility to assure the static type-checker that expressions with certain forms have particular types. For example, we may indicate that, while multiplication generally yields `Z`s, when one of the factors is from the set of even numbers, `E`, the result will be as well:

```
Type Theorem Multiplication_By_Even_Also_Even_Right:
        For all i : Z, For all e : E,
                i * e : E;
```

Such theorems need no be quantified. And so, for example, we can use one to resolve a tricky design problem: given that we'd usually like to work with restricted sets that contain only elements of some homogenous type, do we define a separate `Empty_Set` for each? Clearly this is non-intuitive and introduces a number of strange situations. But type theorems resolves this issue easily. Given RESOLVE's built-in "class of all sets", `SSet`:

```
Definition Set : MType -> Power(SSet);

Definition Empty_Set : Set;

Type Theorem Empty_Set_In_All_Sets:
        For all T : MType,
        For all S : Set(T),
                Empty_Set : S;
```

We now have a single term, `Empty_Set`, that is defined to be a member of the hetergenous `SSet`, but then stated to be in any restricted `Set`.

Modular Relationships In addition to the obvious usability this flexibility adds to RESOLVE's mathematical theories, it also support modularity by permitting types to relate to each other only when they are loaded. A common arrangement in virtually all mathematical systems is to have a "numerical tower", in which `N` is defined in terms of `Z`, which is in term defined in terms of `R`, and so forth.

To begin with, this creates complexity for the end user if the numeric tower is not rooted sufficiently deep for their needs. The system's entire conception of numbers must change as a higher theory is added. But more importantly for our design, this violates modularity—in order to use `N`s, I must also import `Z`s and `R`s and complex numbers and the rest of the tower. The user, and more importantly the automated prover, has no idea which body of theorems are practically relevant.

By using type theorems, relationships can be established as needed. For example, natural number theory need not advertise a relationship to `Z` (though practically, integers will likely form part of its internal definition details):

```
Definition N : MType;
```

However, when `Integer_Theory` is loaded, it may then establish its relationship with `N`:

```
Definition Z: MType,

Type Theorem N_Subset_of_Z:
        For all n : N,
                 n : Z;
```

Implementation Most of the features listed in this chapter exist in other systems and are novel primarily for being gathered in one place alongside a practical, imperative programming lanaguage. Type theorems, however, are to our knowledge entirely novel, and we therefore will spend a short amount of time discussing their implementation.

Each type theorem introduces two, possibly three pieces of information: an *expression pattern*, which we seek to bind against the actual provided expression, an *asserted type*, which the pattern is said to inhabit, and, optionally, a *condition*, which the actual value must meet in order for the relationship to hold. As a practical example, consider:

```
Type Theorem Z_Superset_of_N:
    For all m : Z, (m >= 0) implies m : N;
```

Here, the expression pattern is `m`, the asserted type is `N`, and the condition is `(m >= 0)`. If the condition is omitted, we default it to `true`, and so without loss of generality we will imagine that all type theorems contain a condition.

Each time a new type theorem is accepted, we take the type of the expression pattern and the asserted type and determine their *canonical bucket*, before finding the associated buckets in the type graph and added an edge.

Canonicalization transforms a type expression that possibly contains free type variables into a Big Union type containing every possible valuation of those variables (and possibly more). This is accomplished by locating any free type variables in the type expression, giving them unique names, then binding those unique variable names at the top level in a Big Union expression, each ranging over **MType**.

So, if a type expression started life as $T \cup W \cup T$, where T ranges over `MType` and W over `Power(T)`, it would be canonicalized into $\left(\bigcup_{T_1:MType, T_2:MType, T_3:MType} T_1 \cup T_2 \cup T_3 \right)$. Clearly, we have lost a fair bit of information here, but that will be attached to the newly added edge in the form of *static requirements*.

Static requirements encode the relationships between the original free type variables. Two types of relationships are permitted: 1) *equality*—two type variables were originally the same and 2) *membership*—a variable was originally declared as a member of a certain type.

So, in our above example, the static requirements would be: $T_1 : MType, T_2 : Power(T_1), T_3 = T_1$, recapturing the information we lost during canonicalization.

In addition to these static requirements, the expression pattern and condition are added to the edge.

Consider that we want to add the following three type theorems:

```
Type Theorem N_Subset_of_Z:
    For all n : N,
            n : Z;

Type Theorem Positive_Zs_Are_Ns:
    For all i : Z,
            (i >= 0) implies i : N;
```

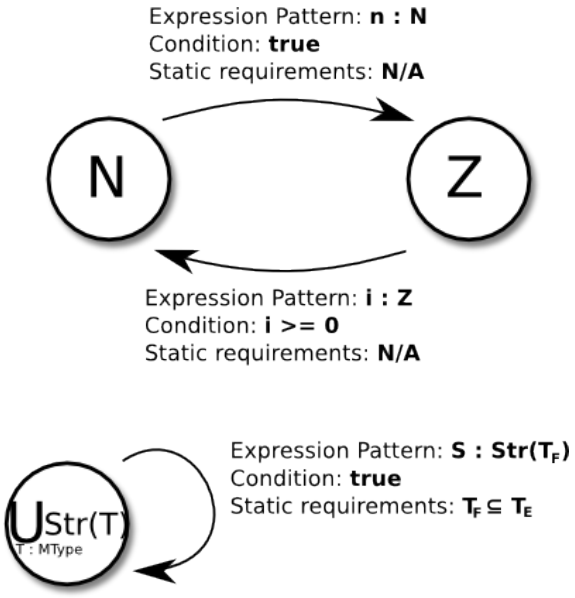


Figure 5.3: Example Type Graph

```
Type Theorem String_Generics_Related:
        For all T : MType,
        For all R : Power(T),
        For all S : Str(R),
                S : Str(T);
```

The resulting type graph would appear as in Figure 5.3. Note that in that figure, T_E and T_F denote "the actual value of T from the expected type" and "the actual value of T from the found type" respectively.

When we seek to determine if a given expression is acceptable as a particular type, we first canonicalize the type of the expression, then seek out any canonical buckets that are *syntactic supertypes* of that type, which is a simpler kind of type reasoning based on a short list of hard-coded rules. For brevity, we establish the predicate $\sigma(T, R)$, which is true if T is a *syntactic subtype* of R. The full list of judgements for making this determination is provided in Figure 5.4. We do the same for the expected type. We then determine if there are edges from any of the provided value's buckets to any of the expected type's buckets. For each such edge, we iterate, attempting to bind the value to the pattern expression, then use the resulting bindings to satisfy the static requirements, before finally instantiating the condition and adding it to a list.

Figure 5.4: Syntactic judgements for determining if one type expression is a subtype of another

$$\frac{\top}{\forall T : \mathbf{MType}, \sigma(T, \mathbf{MType})} \quad \text{(a)} \qquad \frac{\top}{\forall T : \mathbf{MType}, \sigma(T, \mathbf{Element})} \quad \text{(b)}$$

$$\frac{\top}{\forall T : \mathbf{MType}, \sigma(\phi, T)} \quad \text{(c)} \qquad \frac{T = R}{\forall T, R : \mathbf{MType}, \sigma(T, R)} \quad \text{(d)}$$

$$\frac{\sigma\left(\bigcup_{t_1, t_2, \ldots t_n : \mathbf{MType}} T, R\right), \text{where } t_1, t_2, \ldots t_n \text{ do not appear in } T}{\forall T, R : \mathbf{MType}, \sigma(T, R)} \quad \text{(e)}$$

$$\frac{\sigma\left(\bigcup_{t_1, t_2, \ldots t_n : \mathbf{MType}} T, R\right), \text{where } t_1, t_2, \ldots t_n \text{ do not appear in } T}{\forall T, R : \mathbf{MType}, \sigma(T, R)} \quad \text{(f)}$$

We continue searching until we find an edge whose condition is simply `true`, or we run out of edges. If any edge's condition is `true`, we simply return true. If no edge matched, we return `false`. In any other case, we return the disjunction of the condition of each matched edge.

5.3.5.2 Subsumption Theorems

With such a complex type system in play, binding a function invocation to its intended function definition can become tricky in the presence of overloading, which RESOLVE permits[2]. Certainly, if the parameters types match exactly, it's obvious which definition to match to. Ideally, we might choose the "tightest" definition, but with multiple parameters we quickly run into the mathematical equivalent of the double dispatch problem.

Our solution was to keep the binding algorithm simple: for an unqualified function name, if there is exactly one function that matches parameters types exactly, we bind to it; if there is more than one such function, we give an ambiguous symbol error; if there are zero such functions, we then look for functions whose parameters are *inexact* matches; again, if there is one such function

[2]While the same symbol may not be defined more than once in the same module, multiple modules may define the same symbol, which may simultaneously become available via imports. This ensures that every symbol has a unique fully-qualified name.

we bind to that; if there is more than one, we give an "ambiguous symbol" error; and if there are zero we give a "no such symbol" error.

This works reasonably well, but causes problems, e.g., when more than two levels of the numeric tower have been imported. Consider this example:

```
Definition Problematic(i : Z, n : N) : R = i + n;
```

Clearly, `Natural_Number_Theory`, `Integer_Theory`, and `Real_Number_Theory` have all been imported to make this definition possible, along with the various type theorems that relate them. With that in mind, which `+` does the right-hand-side of the definition bind to? It can't bind to the one from `Natural_Number_Theory`, because its first parameter is a $\mathbb{Z}$. Both the one from `Integer_Theory` and `Real_Number_Theory` match inexactly. As a result, our algorithm gives an "ambiguous symbol" error.

While, as with type relationships, there's no general solution to this problem, it would be useful if we could flag for the type system that in this case the integer `+` is subsumed by the real-number `+` (i.e., all theorems that hold form integer `+` also hold for real-number `+`) and, thus, if the two are in competition, the integer `+` ought be chosen. This would be equivalent to stating (and proving) that the integer `+` really is the "tighter" function.

Our system permits this in the form of a *subsumption theorem*, which in this case would look like this:

```
Subsumption Theorem R_Plus_Subsumes_Z_Plus:
        R.+ subsumes Z.+;
```

As with all other theorems, this one must be backed with a proof, but once accepted it assures the type-system that nothing is lost by resolving ambiguity in favor of the tighter function.

5.3.6 Categorical Definitions

Categorical definitions are a new construct in the language which address the issue of mutually-dependent symbols. They permit one or more symbols to be introduced together, then defined by a simple predicate in terms of each other. As an example, consider this definition of the set of natural numbers:

```
Categorical Definition introduces
        N : MType, 0 : N, Succ : N -> N
related by
```

```
For all e : Entity, (
        (Exists i : Z, Repeatedly_Apply(N, 0, i) = e)
                implies Is_In(N, e));
```

The definition introduces `N`, `0`, and `Succ` simultaneously. They are defined only by the predicate that relates them to each other. Because we lack the base case of an inductive definition or the witness of a direct definition, categorical definitions raise a proof obligation that they are inhabitted. After all, nothing prevents the user from including `related by false`.

Chapter 6

Evaluation of Minimalist Prover

The evaluation of our minimalist prover orbits three fundamental questions: first, does our prover succeed in verifying programs; second, are our heuristics effective at increasing the usefulness of the prover; and third, does the data gathered by our prover expose any interesting properties of the VCs that arise from well-engineered programs.

The first and second question are explored by considering a series of verification benchmarks. In Section 6.2 we present each benchmark, then explore the effectiveness of the automated prover when applied to that benchmark. Following that, in Section 6.3, we use those same benchmarks to explore how our various heuristics impact the effectiveness of the prover. Finally, the third question is addressed in Section 6.4, where we provide some observations and conclusions based on the collected data.

This chapter focuses on the specifications and realizations of the benchmarks, with discussion of relevant mathematics forming the basis of Chapter 7. We will generally present only enough of each specification and realization to motivate our discussion, but the full specifications of each benchmark and any associated datastructures can be found in Appendix **??** and full realizations in Appendix **??**. Full listings of the proofs generated as solutions to the benchmarks in this chapter are available in Appendix **??**.

While the usability of our prover and its application in educational settings is not a core part of this thesis, we take this opportunity to note that our minimalistic prover has provided the backbone of our educational web-interface for three years now, and point the interested reader at [**?**], [**?**], and [**?**] for examples of research where our prover is applied in educational settings.

6.1 Description of Metrics

For this chapter we focus on four primary metrics for exploring the effectiveness of the prover and the difficulty of the VCs. These are:

- **VCs proved.** The number of VCs that were successfully dispatched by the prover. Since the proof space is quite large, the prover often runs with a set timeout, after which it gives up, so in reality this metric refers to the number of VCs that were proved within the timeout. Throughout this chapter, the timeout was set at 20 seconds.
- **Real time.** The amount of time required to dispatch the VC, generally measured in milliseconds. Because this metric can vary depending on a number of variables, we will generally use the median of five repeated trials.
- **Proof steps.** The number of relevant steps taken by the prover. One of the features of the prover is to prune steps that did not impact the final result.
- **Search steps.** A subset of the overall proof steps including only those steps taken during the "consequent exploration" phase described in Section 4.3.3.2, and not including those consequent steps that are "obvious", namely: replacing a consequent that matches a given with `true`, replacing a consequent that is a symmetric equality with `true`, and eliminating a conjunct that is `true`. Intuitively this metric refers to the number of steps that were blind exploration of the proof space and which may have to be backtracked over, rather than deterministic steps over which we do not backtrack.

6.2 Benchmark Solutions

6.2.1 VSTTE Benchmarks

In 2010, members of the RSRG at Clemson and The Ohio State University published a set of incremental verification benchmarks at the Verified Software: Tools, Theories, and Experiments workshopVSTTE[45]. These benchmarks were intended to provide a basis for experimentation and discussion between verification efforts. In [15], the author presents a selection of these benchmarks implemented in RESOLVE in order to demonstrate the VC-generation capabilities of the RESOLVE

compiler. We mirror that methodology in this section, presenting a selection of the VSTTE benchmarks, including several that are borrowed or adapted from [15], before applying our minimalist prover to them and presenting resulting data.

6.2.1.1 Benchmark 1: Adding and Multiplying Integers

Problem Statement Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.[1]

Solution Discussion In Listing 6.1 we present specifications of integer addition and multiplication operations in RESOLVE. Each of these operations is specified to work "in place", transforming the first parameter (which is also used as one of the inputs) into the final solution.

Listing 6.1: The specification of `Adding_Capabilitiy` and `Multiplying_Capability`

```
Enhancement Adding_Capability for Integer_Template;
        uses Integer_Theory, Std_Integer_Fac;

    Operation Add_to(updates i : Integer; evaluates j : Integer);
        requires min_int <= i + j and i + j <= max_int and j >= 0;
        ensures i= #i + j;

end Adding_Capability;

Enhancement Multiplying_Capability for Integer_Template;

    Operation Multiply_into(updates i : Integer; evaluates j : Integer);
        requires min_int <= i * j and i * j <= max_int and j >= 0;
        ensures i = #i * j;

end;
```

Each specifies that its second parameter must be positive—dealing correctly with the presence of negative numbers adds a surprising amount of implementation complexity for a language like RESOLVE that strives to correctly reason about bounded integers, since the maximum and minimum integer are generally not symmetrical. Note, however, that we lose no generality—a more general addition or multiplication function could be specified and wrapped around the ones presented in this section, transforming any addition or multiplication task into a finite subset of tasks that could computed correctly by these implementations.

We experimented both with an iterative and recursive implementation of `Adding_Capability`.

[1] This and all other problem statements quote directly from [45].

We present the rescursive implementation in Listing 6.2. The iterative implementation is available in Appendix **??**.

Listing 6.2: A recursive implementation of `Adding_Capabilitiy`

```
Realization Recursive_Add_to_Realiz for Adding_Capability of Integer_Template;

        Recursive Procedure Add_to(updates i : Integer; evaluates j : Integer);
                decreasing j;

                If (not Is_Zero(j)) then
                        Increment (i);
                        Decrement (j);
                        Add_to (i, j);
                end;
        end;
end;
```

Our implementation of `Multiplying_Capabilitiy` is iterative and presented in Listing 6.3.

Listing 6.3: An iterative implementation of `Multiplying_Capabilitiy`

```
Realization Iterative_Multiply_into_Realiz for Multiplying_Capability
                of Integer_Template;

        Procedure Multiply_into(updates i : Integer; evaluates j : Integer);
                Var result : Integer;

                While (j /= 0)
                        changing result, j;
                        maintaining result = i * (#j - j) and j >= 0;
                        decreasing j;
                do
                        result := result + i;
                        j := j - 1;
                end;

                i :=: result;
        end;
end;
```

Integers are partially built-in to RESOLVE, which makes using one enhancement to `Integer_Template` from another fairly unweildy, which is why we do not use `Adding_Capability` to implement `Multiplying_Capability` as requested by the benchmark, but rather the normal integer `+`. We note, however, that `+` is drawn from `Integer_Template` as a normal specification and thus presents the same verification challenges (that is: reasoning about it is not built in).

Results Each of our three implementations is fully verifiable. The results of those verifications, with associated metrics is presented in Figure 6.1.

	Time (ms)	σ	Steps	Search
VC 0_1	5560	182	5	0
VC 0_2	3456	280	5	0
VC 0_3	565	78	10	0
VC 0_4	834	197	9	0
VC 0_5	3873	219	6	0
VC 0_6	613	101	8	0
VC 0_7	591	106	5	0
VC 1_1	5020	286	9	0

(a) Iterative `Adding_Capabilitiy` results

	Time (ms)	σ	Steps	Search
VC 0_1	2549	124	9	0
VC 0_2	1244	221	9	0
VC 0_3	977	177	5	0
VC 0_4	1379	222	6	0
VC 0_5	1118	159	6	0
VC 0_6	757	82	8	0
VC 0_7	1113	178	5	0
VC 1_1	379	76	8	0

(b) Recursive `Adding_Capabilitiy` results

	Time (ms)	σ	Steps	Search
VC 0_1	6264	195	7	0
VC 0_2	3484	365	5	0
VC 0_3	3495	182	7	2
VC 0_4	393	149	7	0
VC 0_5	333	53	5	0
VC 1_1	5917	124	8	0

(c) Iterative `Multiplying_Capabilitiy` results

Figure 6.1: Results from verification of Benchmark 1 solutions

6.2.1.2 Benchmark 2: Benchmark 2: Binary Search an Array

Problem Statement Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

Solution Discussion In Listing 6.4 we present a specification of a searching operation on an array. The operation takes as its input an entry and an array in sorted order, then uses a parameterizable comparison function, `LEQ`, to search the array.

The *requires* clause of the operation uses higher-order definitions to establish that the array is in order—i.e., that it is *conformal* with the given comparator. The *ensures* clause states that the operation will return `true` if and only if the given entry exists between the lower and upper bound of the array. The details of these higher-order definitions are explored more fully in Chapter 7.

An implementation for this operation is provided in Listing 6.5.

The realization takes a function, `Are_Ordered()` which provides a programmatic way of establishing `LEQ`. From there, the implementation is the usual straightforward binary search implementation. Note that when calculating the new mid, we first take the difference, then divide,

Listing 6.4: The specification of `Searching_Capabilitiy`

```
Enhancement Search_Capability(definition LEQ(x,y: Entry): B)
                    for Static_Array_Template;
               uses Std_Boolean_Fac, Total_Preordering_Theory,
                    Binary_Relation_Properties_Theory,
                    Binary_Iterator_Theory;
               requires Is_Total_Preordering(LEQ) and Is_Symmetric(LEQ);

        Operation Is_Present(restores key: Entry;
                    restores A: Static_Array) : Boolean;
               requires Is_Conformal_With(LEQ, Concatenate(
                    Shift(A, (Lower_Bound - 1) * -1), Upper_Bound -
                        Lower_Bound));
               ensures Is_Present =
                    Exists_Between(key, Concatenate(
                                Shift(A, (Lower_Bound - 1) * -1),
                                Upper_Bound - Lower_Bound),
                            Lower_Bound, Upper_Bound);
end;
```

avoiding the overflow problem exposed in [4].

While we provide syntactic sugar for array operations, they are supported by an ordinary component that provides specifications for all array operations, a snippet of which is provided in Listing 6.6.

Results Of the 45 VCs generated by this example, we are able to mechanically verify 39. These result are summarized in Figure 6.2.

The six VCs that are not mechanically verified represent a failure not of the prover—but rather of RESOLVE's underlying syntax. Because at this time RESOLVE does not permit the application of an arbitrary expression (as opposed to a named function), we are unable to formulate the required theorems to prove these VCs.

We now present a manual proof of a representative one of these VCs by hand and argue that once this syntactic problem is resolved, our minimalist prover will be easily able to dispatch the VC.

Consider the VC in Listing 6.7 corresponding to the inductive case of the while loop's maintaining clause. Irrelevant givens have been ommited for brevity.

Let us now proceed as the mechanical prover would. We first replace instances of `A''` with `A`, resulting in these givens:

```
A' = lambda ( j : Z ).(

{{midVal'' if j = (low'  + ((high'  - low')  / 2))
```

	Time (ms)	σ	Steps	Search
VC 0_1	1411	174	6	1
VC 0_2	819	161	5	0
VC 0_3	573	107	5	0
VC 1_1	2523	249	8	0
VC 1_2	1607	148	5	0
VC 1_3	1318	111	5	0
VC 1_4	792	155	5	0
VC 1_5	6438	261	10	3
VC 1_6	2401	302	9	1
VC 1_7	6838	178	10	3
VC 1_8	2430	188	9	1
VC 1_9	Not Proved			
VC 1_10	2108	191	7	1
VC 1_11	2097	305	5	0
VC 1_12	Not Proved			
VC 1_13	3259	309	9	3
VC 2_1	1637	116	8	0
VC 2_2	1346	98	5	0
VC 2_3	1361	99	5	0
VC 2_4	595	35	5	0
VC 2_5	5816	247	10	3
VC 2_6	2414	121	9	1
VC 2_7	6465	338	10	3

	Time (ms)	σ	Steps	Search
VC 2_8	2813	187	9	1
VC 2_9	Not Proved			
VC 2_10	2923	393	11	4
VC 2_11	2369	260	5	0
VC 2_12	Not Proved			
VC 2_13	11229	204	8	3
VC 3_1	1715	124	8	0
VC 3_2	1322	139	5	0
VC 3_3	1242	140	5	0
VC 3_4	630	64	5	0
VC 3_5	6095	151	10	3
VC 3_6	2540	306	9	1
VC 3_7	6388	335	10	3
VC 3_8	2730	230	9	1
VC 3_9	Not Proved			
VC 3_10	2386	228	5	0
VC 3_11	2785	300	10	2
VC 3_12	Not Proved			
VC 3_13	18032	273	9	2
VC 4_1	2474	70	9	3
VC 4_2	787	129	5	0
VC 4_3	813	71	5	0

Figure 6.2: Binary search `Searching_Capabilitiy` results

```
  A(j) otherwise}}) and
A((low'  + ((high'  - low')  / 2))) = key)
```

We now replace `A'` with its full expansion in the goal, resulting in this new goal:

```
lambda ( j : Z ).(
{{key if j = (low'  + ((high'  - low')  / 2))
  lambda ( j : Z ).(
{{midVal'' if j = (low'  + ((high'  - low')  / 2))
   A(j) otherwise}}(j) otherwise}})
= A
```

Finally, we replace instances of `key` with its expansion, resulting in this new goal (we change the internal lambda's variable to `k` for clarity, though the prover does not have to deal with this complication):

```
lambda ( j : Z ).(
{{A((low'  + ((high'  - low')  / 2)))
if j = (low'  + ((high'  - low')  / 2))
  lambda ( k : Z ).(
{{midVal'' if k = (low'  + ((high'  - low')  / 2))
   A(k) otherwise}})(j) otherwise}})
= A
```

At this point we are able to apply the theorem given in Listing 6.8 (note that it applies a function-valued expression when it evaluates a lambda expression at `x` and therefore cannot be represented in our current syntax).

We are then left with simply `A = A`, at which point the remainder of the proof is trivial. Each of these six VCs has a similar straightforward proof.

Listing 6.5: An implementation of `Searching_Capabilitiy`

```
Realization Bin_Search_Realiz(
                Operation Are_Ordered(restores x,y: Entry): Boolean;
                        ensures Are_Ordered = (LEQ(x,y));)
        for Search_Capability of Static_Array_Template;
        uses Std_Boolean_Fac;

        ...

        Procedure Is_Present(restores key: Entry; restores A: Static_Array): Boolean;
                Var low, mid, high, one, two : Integer;
                Var midVal : Entry;
                Var result : Boolean;

                one := One();
                two := Two();

                result := False();
                low := Lower_Bound;
                high := Upper_Bound;

                While (low <= high)
                        changing low, mid, high, A, midVal, result;
                        maintaining ...;
                        decreasing (high - low);
                do
                        Divide(Difference(high, low), two, mid);
                        mid := Sum(low, mid);
                        Swap_Entry(A, midVal, mid);
                        if (Are_Equal(midVal, key)) then
                                result := True();
                                low := Sum(high, one);
                        else
                                if (Are_Ordered(midVal, key)) then
                                        low := Sum(mid, one);
                                else
                                        high := Difference(mid, one);
                                end;
                        end;
                        Swap_Entry(A, midVal, mid);
                end;

                Is_Present := result;
        end;
end;
```

Listing 6.6: A snippet of the specification of arrays

```
Concept Static_Array_Template(type Entry; evaluates Lower_Bound, Upper_Bound: Integer);
        uses Std_Integer_Fac, Integer_Theory, Conditional_Function_Theory;
        requires (Lower_Bound <= Upper_Bound);

    Type Family Static_Array is modeled by (Z -> Entry);
        exemplar A;
        constraint true;
        initialization ensures
            for all i: Z, Entry.Is_Initial(A(i));

    Operation Swap_Entry(updates A: Static_Array; updates E: Entry; evaluates i: Integer);
        requires Lower_Bound <= i  and i <= Upper_Bound;
        ensures E = #A(i) and A = lambda (j : Z).(
            {{#E if j = i;
              #A(j) otherwise;}});

    ...
end;
```

Listing 6.7: A problematic VC

```
VC: 1_12:
Inductive Case of Invariant of While Statement in Procedure Is_Present ,
If "if" condition at Bin_Search_Realiz.rb(45) is true: Bin_Search_Realiz.rb(39)

Goal:
lambda ( j : Z ).(
    {{key if j = (low' + ((high' - low') / 2))
      A'(j) otherwise}})
= A

Given:
A'' = A and
A' = lambda ( j : Z ).(
    {{midVal'' if j = (low' + ((high' - low') / 2))
      A''(j) otherwise}}) and
A''((low' + ((high' - low') / 2))) = key)
```

Listing 6.8: A useful theorem about lambda expressions

```
Theorem Shadowed_Function_Piece:
    For all T, R : MType,
    For all t : T,
    For all r : R,
    For all f : T -> R,
        f = lambda (x : T).(
            {{f(x) if x = t;
              lambda (y : Z).(
                  {{r if y = t;
                    f(y) otherwise}})(x) otherwise}});
```

6.2.1.3 Benchmark 3: Sorting a Queue

Problem Statement Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

Solution Discussion In Listing 6.9 we present a specification of a queue sorting enhancement. As with Benchmark 2, it takes as a parameter a client-provided definition to define the sorted order, `LEQV`. It uses two higher-order definitions `Is_Coformal_With()` and `Is_Permutation()` to ensure that the final queue is ordered according to `LEQV` and contains the same elements as those original provided, respectively.

Listing 6.9: The specification of `Sorting_Capability`

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
                        Queue_Template;
                uses String_Theory, Total_Preordering_Theory;
                requires Is_Total_Preordering(LEQV);

        Operation Sort(updates Q : Queue);
                ensures Is_Conformal_With(LEQV, Q) and Is_Permutation(#Q, Q);

end;
```

A straightforward selection sort realization is provided in Listing 6.10.

Again, as in Benchmark 2, we take a client-provided operation, `Compare()`, that implements `LEQV`, and use it to implement our `Remove_Min()` sub-operation. The `One()` and `Two()` procedures simply provide a way of getting the programmatic values `1` and `2` using a normal specification.

Results This implementation is fully verifiable. The results of that verifications, with associated metrics, is presented in Figure 6.3. Based on our literature review, we believe that RESOLVE is the only verification effort with a generic, automatically-verifiable selection sort[2]

[2]The RESOLVE group at The Ohio State University has their own version of this automatically-verifiable selection sort. We note, however, that their system takes advantage of a specialized decision procedure for reasoning about strings, whereas our solution uses only our generalized minimalist prover.

	Time (ms)	σ	Steps	Search
VC 0_1	1344	242	6	0
VC 0_2	479	73	5	0
VC 0_3	912	56	5	0
VC 0_4	1305	198	6	1
VC 0_5	2762	300	8	0
VC 0_6	3128	372	9	3
VC 0_7	1513	271	8	0
VC 0_8	1710	136	10	1
VC 0_9	2219	166	9	3
VC 1_1	501	100	5	0
VC 1_2	1012	165	10	1
VC 2_1	806	136	5	0
VC 2_2	1918	237	8	0
VC 2_3	1781	197	5	0
VC 2_4	1620	284	6	1
VC 2_5	3021	387	14	0

	Time (ms)	σ	Steps	Search
VC 2_6	4441	278	12	3
VC 2_7	1919	108	10	2
VC 2_8	1843	139	7	0
VC 3_1	819	141	5	0
VC 3_2	1896	213	8	0
VC 3_3	1642	124	5	0
VC 3_4	1493	197	6	1
VC 3_5	2965	239	14	0
VC 3_6	4202	200	10	1
VC 3_7	1897	194	10	2
VC 3_8	1863	165	7	0
VC 4_1	838	139	5	0
VC 4_2	2939	150	12	0
VC 4_3	1132	189	5	0
VC 4_4	3056	262	18	3

Figure 6.3: Selection sort `Sorting_Capabilitiy` results

Listing 6.10: A selection sort realization of `Sorting_Capabilitiy`

```
Realization Selection_Sort_Realization(
                        Operation Compare(restores E1, E2 : Entry)
                                : Boolean;
                                ensures Compare = LEQV(E1, E2);)
                for Sorting_Capability of Queue_Template;
        uses String_Theory;

        Procedure Sort(updates Q : Queue);
                Var Sorted_Queue : Queue;
                Var Lowest_Remaining : Entry;

                While (Length(Q) > 0)
                        changing Q, Sorted_Queue, Lowest_Remaining;
                        maintaining ...;
                        decreasing |Q|;
                do
                        Remove_Min(Q, Lowest_Remaining);
                        Enqueue(Lowest_Remaining, Sorted_Queue);
                end;
                Q :=: Sorted_Queue;
        end;

        Operation Remove_Min(updates Q : Queue; replaces Min : Entry);
                requires |Q| /= 0;
                ensures ...;
        Procedure
                Var Considered_Entry : Entry;
                Var New_Queue : Queue;
                Dequeue(Min, Q);
                While (Length(Q) > 0)
                        changing Q, New_Queue, Min, Considered_Entry;
                        maintaining ...;
                        decreasing |Q|;
                do
                        Dequeue(Considered_Entry, Q);
                        if (Compare(Considered_Entry, Min)) then
                                Min :=: Considered_Entry;
                        end;
                        Enqueue(Considered_Entry, New_Queue);
                end;
                New_Queue :=: Q;
        end;
end;
```

6.2.2 Other Benchmarks

6.2.2.1 Benchmark 4: Reversing a Queue

Problem Statement Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that reverses the entries in a given queue.

Solution Discussion In Listing 6.11 we present a specification of a queue reversing enhancement called `Flipping_Capability`. A recursive solution follows in Listing 6.12.

Listing 6.11: The specification of `Flipping_Capabilitiy`

```
Enhancement Flipping_Capability for Queue_Template;

        Operation Flip (updates Q: Queue);
                ensures Q = Reverse(#Q);

end;
```

Listing 6.12: A recursive realization of `Flipping_Capabilitiy`

```
Realization Recursive_Flipping_Realiz for Flipping_Capability of Queue_Template;

        Recursive Procedure Flip(updates Q: Queue);
                        decreasing |Q|;

                Var E: Entry;
                If (Length(Q) /= 0) then
                        Dequeue(E, Q);
                        Flip(Q);
                        Enqueue(E, Q);
                end;
        end;
end;
```

Results This implementation is fully verifiable. The results of that verifications, with associated metrics, is presented in Figure 6.4.

	Time (ms)	σ	Steps	Search
VC 0_1	1520	141	5	0
VC 0_2	3118	295	7	0
VC 0_3	2741	222	8	0
VC 0_4	2174	170	9	2
VC 1_1	366	83	10	0

Figure 6.4: Recursive `Flipping_Capabilitiy` results

6.2.2.2 Benchmark 5: Array Realization of Stack

Problem Statement Specify a user-defined LIFO stack ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an implementation of that array.

Solution Discussion Listing 6.13 gives RESOLVE's specification for a stack in the context of a `Stack_Template`, which is generic and bounded.

Listing 6.13: The specification of `Stack`

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
        uses Std_Integer_Fac, String_Theory, Integer_Theory;
        requires Max_Depth > 0;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = Empty_String;

    Operation Push(alters E: Entry; updates S: Stack);
        requires |S| < Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires |S| /= 0;
        ensures #S = <R> o S;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (|S|);

    Operation Rem_Capacity(restores S: Stack): Integer;
        ensures Rem_Capacity = (Max_Depth - |S|);

    Operation Clear(clears S: Stack);

end;
```

We implement this specification on top of an array, which, as in Benchmark 2, is a first-class component with its own template and operation specifications. This array-based implementation is provided in Listing 6.14.

Results This implementation is fully verifiable. The results of that verifications, with associated metrics, is presented in Figure 6.5.

Listing 6.14: An array-based implementation of `Stack`

```
Realization Array_Realiz for Stack_Template;
                uses Binary_Iterator_Theory;

        Type Stack is represented by Record
                Contents: Array 1..Max_Depth of Entry;
                Top: Integer;
        end;
        convention
                0 <= S.Top <= Max_Depth;
        correspondence
                Conc.S = Reverse(Concatenate(S.Contents, S.Top));

        Procedure Push(alters E: Entry; updates S: Stack);
                S.Top := S.Top + 1;
                E :=: S.Contents[S.Top];
        end;

        Procedure Pop(replaces R: Entry; updates S: Stack);
                R :=: S.Contents[S.Top];
                S.Top := S.Top - 1;
        end;

        Procedure Depth(preserves S: Stack): Integer;
                Depth := S.Top;
        end;

        Procedure Rem_Capacity(preserves S: Stack): Integer;
                Rem_Capacity := Max_Depth - S.Top;
        end;

        Procedure Clear(clears S: Stack);
                S.Top := 0;
        end;
end;
```

	Time (ms)	σ	Steps	Search
VC 0_1	493	58	5	0
VC 1_1	4116	552	7	0
VC 2_1	240	132	5	0
VC 2_2	4324	268	7	1
VC 2_3	200	42	6	0
VC 3_1	18583	490	7	2
VC 3_2	2331	516	8	0
VC 3_3	1709	178	6	1
VC 3_4	2038	83	8	0
VC 3_5	2243	475	9	2
VC 4_1	6592	314	11	3
VC 4_2	1187	234	5	0
VC 4_3	891	216	9	0
VC 4_4	1124	142	6	1

	Time (ms)	σ	Steps	Search
VC 4_5	956	291	7	0
VC 5_1	1558	408	5	0
VC 5_2	2070	300	5	0
VC 5_3	1705	299	7	0
VC 5_4	1769	428	5	0
VC 6_1	1668	465	5	0
VC 6_2	2344	300	5	0
VC 6_3	2217	386	7	0
VC 6_4	1729	392	5	0
VC 7_1	625	146	5	0
VC 7_2	1390	342	7	0
VC 7_3	598	169	4	0
VC 7_4	683	217	6	0

Figure 6.5: Array-based `Stack` implementation results

6.3 Heuristic Evaluation

In order to evaluate the effectiveness of our heuristics from Section 4.3.3, the prover was instrumented so that each of six different heuristics could be disabled individually. The heuristics we targetted were: ignoring useless transformations, developing the antecedent only about relevant terms, focus on diversity in antecedent development, minimization of both consequent and antecedent, cycle detection, and transformation prioritization.

We then re-ran the verification process on each of our benchmarks and collected data about the changes to our various metrics. This data is summarized in Figure 6.6.

As expected, we see that our heuristics are at least partially responsible for a number of VCs being proved. Ignoring useless transformations, limiting development to relevant terms, diversifying antecedents, minimizing, and prioritizing transformations all make the difference between some VCs proving or not proving. Of those, it is interesting to note that diversifying antecedents poses a trade off—it requires more time (2 minutes, 33 seconds on these 130 VCs), but it enabled five more VCs to prove. This may indicate that it would benefit from being targetted for further optimization to reduce the cost of applying it. On the other hand, ignoring useless transformations, limiting development to relevant terms, minimization, and prioritization are nothing but a net gain. They are collectively responsible for speeding up the proving process by over four and a half minutes and permitting 42 additional VCs to prove (though, for both time and the provability of VCs we imagine that there is overlap in the time saved and VCs proved and thus this is not a strictly additive relationship).

An important metric is the change in search steps, since from an algorithmic perspective these steps are the most expensive to take: they contribute to the overall combinatorial explosion of time as we must inductively search deeper and deeper for a solution. We note that three of our heuristics improve this metric, while only one has adverse effects (and a small one at that). All told, the may summarize the net effect of all our heuristics as saving 48 search steps.

It may at first be unclear how minimization can eliminate thiry-five search steps but only three steps—after all, search steps are a subset of minimization—but consider that minimization is, at its core, a heuristic for taking *extra steps* that are likely to be useful. The discrepancy of twenty-three steps represent steps that were *not* taken in the preprocessing phase where they contributed no combinatorial complexity, and were instead deferred to the consequent exploration phase, where

	$\sum \Delta$Proved	$\overline{\Delta t/\sigma}$	$\sum \Delta t$	$\overline{\Delta\text{steps}}$	$\sum \Delta$steps	$\overline{\Delta\text{search}}$	$\sum \Delta$search
With useless transformations	-12	4.04	80591	0	0	0	0
Developing about irrelevant terms	-1	8.92	152165	0	0	0	0
Not checking for diversity of givens	-5	-5.51	-139787	-0.03	-4	-0.01	-1
No minimization	-10	2.41	33794	0.03	3	0.31	35
No cycle detection	0	0.62	9917	0.07	9	0.07	9
No prioritization of transformations	-19	2.51	14386	0.06	6	0.05	5

Figure 6.6: Summary of heuristic evaluation results. From left to right the columns are: total change in the number of proved VCs (negative means fewer were proved), average standard deviations change in time to prove, total change in time to prove, average change to the number of steps required, total change in number of steps required, average change in number of search steps required, and total change in number of search steps required. Average and total changes only take into account VCs that were proved.

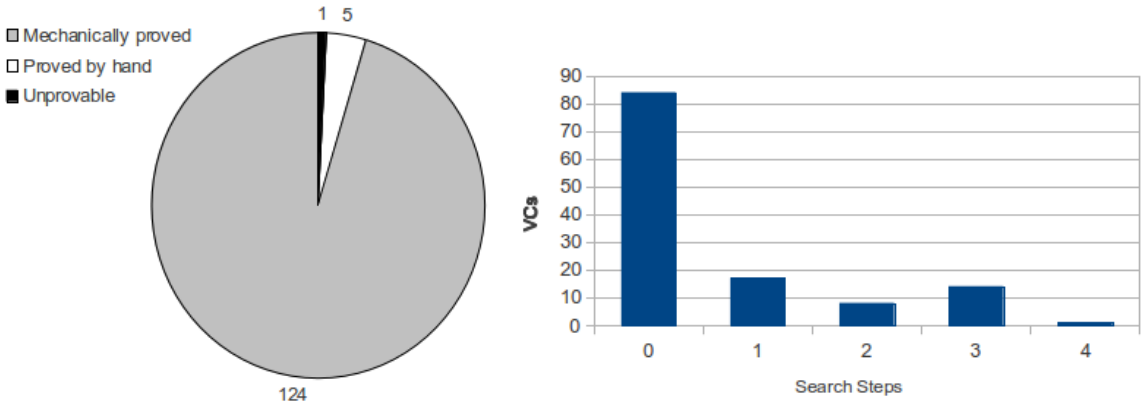
they were much more expensive.

6.4 Observations and Conclusions

These five examples were chosen to be representative of the sorts of problems to which our prover can be applied, but do not constitute the total body of components we can automatically verify. We have a library of many other verified operations operating on stacks, queues, lists, and integers.

These benchmarks together involve 130 VCs ranging over the mathematical domains of functions, strings, integers, and booleans. Of those, we are able to mechanically verify 124, with an additional 5 provable by hand, and a single VC that appears to be unprovable. This is visualized in Figure 6.7a. Of the mechanically verifiable VCs, the median time to prove was 1696 milliseconds, and the mean time was 2487 milliseconds. The median number of proof steps was seven and the median number of search steps was zero.

As we have previously mentioned, we are particularly interested in the search step metric since it represents the only non-deterministic portion of the proof search. That the median number of such steps was zero is extremely heartening and we are further encouraged that no VC required



(a) Number of Search Steps Required by VC (b) Number of Search Steps Required by Proofs

Figure 6.7: Summary of Proof Evaluation Results

more than four. Figure 6.7b shows a histogram of the number of VCs requiring different numbers of search steps.

The vast majority (84, or 68%) of VCs required no search steps once all heuristics were applied. 39 (31%) required three or fewer steps, while only a single VC required four search steps. This data supports our hypothesis that VCs arising from well-engineered software should require only simple analysis to dispatch.

Chapter 7

Evaluation of Mathematical System

The evaluation of our mathematical system is fundamentally based on the ease with which it permits us to express the necessary and diverse mathematical ideas that arise when specifying programs formally. In Chapter 6, we present a number of solutions to selected verification benchmarks and discuss the effectiveness of our automated prover at discharging the VCs that arise from those implementations. In this chapter, we will consider the mathematical developments that arise when specifying those benchmarks and provide commentary on the effectiveness of our mathematical system.

7.1 VSTTE Benchmarks

7.1.1 Benchmark 1: Adding and Multiplying Integers

Problem Statement:[1] Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition, using the first operation to do the addition. Make one algorithm iterative, the other recursive.

[1] This and all other problem statements quote directly from [45].

7.1.1.1 Supporting Mathematics

The primary mathematical support required for verifying this benchmark is simply the availability of the integers, along with operators for manipulating them and theorems for reasoning about them. The set `Z` and its fundamental operations are built-in for RESOLVE (since they must back up programmatic integers), making this benchmark somewhat trivial. However, we can still use it as a launching-point for exploring our mathematical flexibility by engaging in the exercise of introducing an identically-behaving mathematical set, `OtherZ`.

Listing 7.1 gives a snippet from `Integer_Theory` in which we introduce this set.

Listing 7.1: The definition of `OtherZ`

```
Theory Integer_Theory;
                uses Monogenerator_Theory;

    ...

        Categorical Definition introduces
                OtherZ : MType, Other0 : OtherZ, Bounce : OtherZ -> OtherZ
        related by
                Is_Monogenerator_For(OtherZ, Other0, Bounce);

    ...
end;
```

Categorical definitions were described in Section 5.3.6 and permit multiple mutually-dependent symbols to be introduced simultaneously and related by a predicate. Here we use a higher-order definition, `Is_Monogenerator_For()` to relate these symbols. This is drawn from `Monogenerator_Theory` and describes sets that can be described by some fixed starting point (here, `Other0`) and iterated over by repeated applications of a closed function. We imagine that function, `Bounce`, returns a sequence like `0, 1, -1, 2, -2, 3, -3 ...`, though this is an arbitrary semantic notion that would be codified by a later definition of negativity. The definition of `Is_Monogenerator_For()` from `Monogenerator_Theory` is given in Listing 7.2.

This definition takes advantage of first class types, permitting us to pass in a type as the first parameter which is then used to define the types of the second two parameters. Note that the definition of `Is_Monogenerator_For()` itself takes advantage of a higher-order definition, `Is_Injective()`, which is drawn from `Basic_Function_Properties_Theory`, shown in Listing 7.3.

`Is_Injective()` uses the implicit type parameter feature described in Section 5.3.3 to establish the domain (`D`) and range (`R`) of the function `F`.

Listing 7.2: The definition of `Is_Monogenerator_For()`

```
Theory Monogenerator_Theory;
                uses Basic_Function_Properties_Theory, ...;

    ...

        Implicit Definition Is_Monogenerator_For(
                        T : MType, Start : T, F : T -> T) : B is
                (For all t : T, F(t) /= t) and
                (Is_Injective(F)) and
                (For all T2 : Powerset(T),
                        Instance_Of(T2, Start) and
                        (for all t : T,
                                        Instance_Of(T2, t)
                                                implies Instance_Of(T2, F(t)))
                                implies T2 = T);

    ...
end;
```

Listing 7.3: The definition of `Is_Injective()`

```
Theory Basic_Function_Properties_Theory;
        uses ...;

    ...

        Implicit Definition Is_Injective(F : (D : MType) -> (R : MType)) : B is
                For all d1, d2 : D,
                        F(d1) = F(d2) implies d1 = d2;

    ...
end;
```

7.1.1.2 When Things Go Wrong

Of course, a mathematical system could easily support all of these features if it simply accepted all inputs. An important component of any static system is raising appropriate errors on invalid input. We will present an example here of how the input could be changed to be in error and the output of the compiler in that case.

When using `Is_Monogenerator_For()`, the type of the second and third parameters depend on the first-class type passed for the first parameter. Suppose we were to change the type of the third parameter to no longer be consistent. An example of this is given in Listing 7.4, where `Bounce` is changed from type `OtherZ -> OtherZ` to type `B -> OtherZ`, which is invalid. Given this input, the corresponding RESOLVE output is shown in Figure 7.5.

Listing 7.4: Note that `Bounce` no longer has an acceptable type

```
Categorical Definition introduces
                OtherZ : MType, Other0 : OtherZ, Bounce : B -> OtherZ
        related by
                Is_Monogenerator_For(OtherZ, Other0, Bounce);
```

Listing 7.5: `Bounce` does not have the appropriate type

```
Error: Integer_Theory.mt(50):
No function applicable for domain: ((MType * OtherZ) * (B -> OtherZ))

Candidates:
        Is_Monogenerator_For : ((((T : MType) * (Start : 'T')) *
                (F : ('T' -> 'T'))) -> B)

        Is_Monogenerator_For(OtherZ, Other0, Bounce);
        ^
```

7.1.2 Benchmark 2: Binary Search an Array

Problem Statement: Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

7.1.2.1 Supporting Mathematics

An important design goal of RESOLVE is that as much reasoning as possible happens through the general component machinery. As a result, RESOLVE does not elevate arrays (or pointers, as we will see later) to special status, unlike nearly all other practical verification languages ([**?**], [**?**], or [**?**], e.g.).

While some syntactic sugar exists to ease working with arrays, this is all translated to normal object operations as a pre-processing step, after which reasoning about arrays based on an ordinary concept with operations and specifications. Listing 7.6 shows the first part of that specification, including the definition of the array type and a common operation, `Swap()`.

For our model of arrays, we choose functions, which naturally describe an object mapping an integer to an entry. Note that the normal parameterization mechanism is used. `Is_Initial()` is a meta-field of all programmatic types of type `T -> B`, where `T` is the type from which the meta-field is derived. Meta-fields are handled gracefully by our mathematical system using a plugin architecture that allows the type-checker to defer when it encounters a field-access that is not strictly correct (as here, where the type of `Entry` is not a tuple.)

The double-curly-bracket notation in the *ensures* clause of the operation represents a piece-

Listing 7.6: A snippet of `Static_Array_Template`

```
Concept Static_Array_Template(type Entry; evaluates Lower_Bound, Upper_Bound: Integer);
            uses Std_Integer_Fac, Integer_Theory, Conditional_Function_Theory;
            requires (Lower_Bound <= Upper_Bound);

        Family Static_Array is modeled by (Z -> Entry);
            exemplar A;
            initialization ensures
                for all i: Z, Entry.Is_Initial(A(i));

        Operation Swap_Entry(updates A: Static_Array; updates E: Entry;
                evaluates i: Integer);
            requires Lower_Bound <= i and i <= Upper_Bound;
            ensures E = #A(i) and A = lambda (j : Z).(
                {{#E if j = i;
                  #A(j) otherwise;}});

    ...
end;
```

wise function, whose type is implicitly determined as the type of the first possible return value. The return type of the lambda expression is similarly implicitly determined by the type of its body.

Function composition is used extensively in `Static_Array_Template` to represent the changing value of the array. In `Swap_Entry`, the lambda function that represents the final value of the array simply wraps the existing array—defering to that array at all indecis other than `i`, and providing the value of `#E` otherwise.

XXX A little more info once we have some VCs for binary search XXX

7.1.3 Benchmark 3: Reversing a Queue

Problem Statement: Specify a user-defined FIFO queue ADT. Verify two implementations (one iterative, one recursive) for an operation that reverses a queue. (Note: the iterative version may need to use another component, e.g., a LIFO stack ADT, in which case that also needs to be specified, of course.)

7.1.3.1 Supporting Mathematics

In order to model a queue, we will need access to a suitable mathematical object for representing it. We choose mathematical *strings*, which are finite sequences. A portion of `String_Theory` is given in Listing 7.7.

First, note that this module is labeled as a *precìs* rather than a *theory*. As we have published

Listing 7.7: A snippet of `String_Theory`

```
Precis String_Theory;

        --The type of all strings of heterogenous type
        Definition SStr : MType;

        --A function that restricts SStr to the type of all strings of some homogenous
        --type
        Definition Str : MType -> MType;

        Type Theorem All_Strs_In_SStr:
                For all T : MType,
                For all S : Str(T),
                        S : SStr;

        --If R is a subset of T, then Str(R) is a subset of Str(T)
        Type Theorem Str_Subsets:
                For all T : MType,
                For all R : Powerset(T),
                For all s : Str(R),
                        s : Str(T);

        Definition Empty_String : SStr;

        Type Theorem Empty_String_In_All_Strs:
                For all T : MType,
                        Empty_String : Str(T);

        --String length
        Definition |(s : SStr)| : N;

    ...
end;
```

in [41], RESOLVE permits us to separate the public signatures of definitions from their full details in a way similar to how C++ separates header files from class files. This increases readability for the consumer of the theory and here conveniently allows us to focus on interesting features rather than getting tangled in fine details.

We begin by introducing the type `SStr`, which represents the type of all strings containing elements of hetergenous type. We then introduce a function from types to types (which thus takes advantage of first-class types) called `Str()`, which conceptually restricts `SStr` to only those strings containing elements of the provided type. I.e., `Str(Z)` is the type of all `SStr`s containing only integers, `Str(SStr)` is the type of all `SStr`s containing only other `SStr`s, and so forth.

Following this, two type theorems (described in Section 5.3.5.1) establish important relationships between and within these types. The first states that any parameterization of `Str` is a subset of `SStr`, while the second states that one parameterization of `Str` is a subset of another if the type-parameter of the former is a subset of the type-parameter of the latter.

A tricky nuance of permitting such type-parameterized strings is how to type the `Empty_String`[2]. One option is to have a single object of type `SStr` that lies at the intersection of every possible parameterization. This causes problems for the static type checker when we want to provide `Empty_String` where a more specific type of string is required—after all, not every `SStr` is a `Str(Z)`, for example. The other option is to define a separate `Empty_String` for each parameterization. However, this only changes the problem from one of typing (which becomes straightforward) to one of identity: can `Empty_String_Of_Z` and `Empty_String_Of_B` coinhabit a set? If not, what happens if one takes the intersection with the set containing `Empty_String_Of_R`?

The flexibility of type theorems permits us a graceful solution to this problem:

We define `Empty_String` to be of type `SStr`. Because type theorems are not restricted to relating types, but may also relate *expressions*, we then define a type theorem that states that `Empty_String` is a member of any parameterization of `Str`.

Having now seen three type theorems, we can see some at work in Listing 7.8, where we present a small snippet from `Queue_Template`.

Listing 7.8: Type theorems at work in `Queue_Template`

```
Concept Queue_Template(type Entry; evaluates Max_Length: Integer);
                uses String_Theory;
                requires Max_Length > 0;

         Family Queue is modeled by Str(Entry);
                exemplar Q;
                constraint |Q| <= Max_Length;
                initialization ensures Q = Empty_String;

   ...
end;
```

Note that string length, denoted by vertical pipes as in `|Q|`, is defined to take a `SStr`, yet is offering `Q`, which is of type `Str(Entry)`. This is acceptable because we have established that all parameterizations of `Str` are subtypes of `SStr`. Similarly, the equals operator is defined to take two parameters of the same type, resolved from left to right, but `Empty_String` is acceptable where a `Str(Entry)` is required (even though it is certainly not the case that all `SStr`s are in `Str(Entry)`), because a type theorem establishes that `Empty_String` is in all parameterizations of `Str`.

Also of interest in our development of string theory is the definition of concatenate, shown in Listing 7.9.

[2]This same nuance shows up with sets and other similar "container" mathematical objects.

Listing 7.9: String concatenation and an associated type theorem

```
Precis String_Theory;
    ...

        --String concatenation
        Definition (s : SStr) o (t : SStr) : SStr;

        Type Theorem Concatenation_Preserves_Generic_Type:
                For all T : MType,
                For all U, V : Str(T),
                        U o V : Str(T);

    ...
end;
```

We define the concatenation operator (`o`) to operate on two `SStr`s instead of taking advantage of implicit type parameters. This design decision permits flexibility when constructing intermediate strings that might have heterogenous type, and also prevents each theorem about concatenation (of which there are many) from having to individually quantify over all types, leading to a more succinct theory. That the result when the two parameters happen to be `Str`s with the same parameterized type is closed on that parmeterization is established by the type theorem, which permits us to provide the results of such a concatenation to a function that requires the more specific type. This technique is used with many other string manipulation operations.

7.1.3.2 When Things Go Wrong

As an example of what would happen without one of the necessary type theorems, we can remove the type theorem that establishes that all parameterized strings are also in `SStr`. If we then attempt to compile `Queue_Template`, the RESOLVE compiler gives the output given in Listing 7.10.

Listing 7.10: The vertical pipe operator is associated with input of type `SStr` or `Z`, neither of which is matched by `Str('Entry')` without an appropriate type theorem.

```
Error: Queue_Template.co(7):
No function applicable for domain: Str('Entry')

Candidates:
        |_| : (SStr -> N)
        |_| : (Z -> Z)

    constraint |Q| <= Max_Length;
               ^
```

7.1.4 Benchmark 4: Sorting a Queue

Problem Statement: Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

7.1.4.1 Supporting Mathematics

The previous benchmarks had straightforward specifications and most of the interesting things were happening in the supporting mathematics. Starting with Benchmark 4, we begin to see specifications that make use of interesting features of our mathematical system directly.

In order to implement sorting on a queue, we rely on all the mathematics already discussed in Benchmark 3. We then introduce the `Sorting_Capability` enhancement shown in Listing 7.11, which extends a queue with the ability to be sorted.

Listing 7.11: `Sorting_Capability` provides the `Sort` Operation

```
Enhancement Sorting_Capability(Definition LEQV(x, y : Entry) : B) for
                Queue_Template;
        uses String_Theory, Total_Preordering_Theory;
        requires Is_Total_Preordering(LEQV);

        Operation Sort(updates Q : Queue);
                ensures Is_Conformal_With(LEQV, Q) and Is_Permutation(#Q, Q);

end Sorting_Capability;
```

The sorting enhancement is parameterized by a first-class definition, `LEQV`, which provides a client-definable mathematical conception of the ordering of the elements. We use the higher-order predicate `Is_Total_Preordering()` drawn from `Total_Preordering_Theory` to establish the needed properties—namely, that `LEQV` is reflexive, transitive, and total. A related definition, `Is_Conformal_With()` is used in the *ensures* clause of `Sort` to indicate that the final queue is in sorted order: `Is_Conformal_With()` takes a comparison function and a string and ensures that all pairs of elements in the string appear in order with respect to the function.

The definitions of both `Is_Total_Preordering()` and `Is_Conformal_With()` are shown in Listing 7.12.

The use of these high-level, semantic predicates over complex quantified expressions is a key part of our methodology, and our flexible type system makes defining and working with such definitions straightforward. `Total_Preordering_Theory` contains a number of theorems for manipulating

Listing 7.12: Definition of `Is_Total_Preordering()` and `Is_Conformal_With()`

```
Theory Total_Preordering_Theory;
        uses String_Theory;

    Definition Is_Total_Preordering(f : ((D : MType) * D) -> B) : B =
        (For all d1 : D, f(d1, d1)) and
        (For all d1, d2, d3 : D, f(d1, d2) and f(d2, d3)
            implies f(d1, d3)) and
        (For all d1, d2 : D, f(d1, d2) or f(d2, d1));

    Definition Is_Conformal_With(f : (Entity * Entity) -> B, S : SStr) : B =
        For all i, j : Z,
            1 <= i and i <= j and j <= |S| implies
                f(Element_At(i, S), Element_At(j, S));

    ...
end;
```

expressions involving `Is_Total_Preordering()` and `Is_Conformal_With()`, eliminating the need for the prover to reason about quantified expressions at all.

As an example of how mathematical flexibility supports proving, consider the VC presented in Listing 7.13, which arises from a selection sort implementation of `Sorting_Capability`.

Listing 7.13: A VC arising from a selection sort implementation

```
Goal:
Is_Universally_Related(<Considered_Entry'>, (New_Queue'  o <Min'>), LEQV)

Given:
(((((((((Last_Char_Num  > 0) and
(((min_int  <= 0) and
(0  < max_int)) and
((Max_Length  > 0) and
((min_int  <= Max_Length) and
(Max_Length  <= max_int))))) and
Is_Total_Preordering(LEQV)) and
(Entry.is_initial(Min) and
((|Q|  <= Max_Length) and
|Q|  /= 0))) and
Q = (<Min''>  o Q'')) and
(Is_Permutation(((New_Queue'  o Q'')  o <Min'>), Q) and
Is_Universally_Related(<Min'>, New_Queue', LEQV))) and
(|Q''|  > 0)) and
Q'' = (<Considered_Entry'>  o Q')) and
LEQV(Considered_Entry', Min'))
```

The predicate `Is_Universally_Related()` states that every element in one string is related to every element in another by a function. The proof of this VC is involved, but requires a theorem like the one in Listing 7.14.

The requirement that `f` be a total preordering is critical, since only for a transitive function

Listing 7.14: A Useful `Is_Universally_Related` theorem

```
For all f : (Entity * Entity) -> B,
For all E1, E2 : Entity,
For all S : String,
        Is_Total_Preordering(f) and
        f(E1, E2) and
        Is_Universally_Related(<E2>, S, f)
                implies Is_Unversally_Related(E1, S);
```

does this statement hold[3].

Note, however, that the VC provides `Is_Total_Preordering(LEQV)` as a given (derived from the module-level requires clause of `Sorting_Capability`. We may therefore make use of this theorem without needing to reason deeper about what it means for a function to be a total preordering.

In order to implement `Sort`, we require the client to pass in an operation for comparing two elements, `Compare()`, which provides a programmatic way of determining if two objects are related via `LEQV`. We see this in the header of `Selection_Sort_Realization` shown in Listing 7.15.

Listing 7.15: `Selection_Sort_Realization` taking an operation that implements `LEQV`

```
Realization Selection_Sort_Realization(
                        Operation Compare(restores E1, E2 : Entry)
                                : Boolean;
                                ensures Compare = LEQV(E1, E2);)
                for Sorting_Capability of Queue_Template;
        uses String_Theory;
```

This use of higher-order definitions provides the possibility of a generic sorting capability.

7.2 Other Benchmarks

This section contains other examples that we feel highlight the capabilities of the mathematical type system.

7.2.1 Cartesian Product Subtypes

Cartesian products are a built-in notion in RESOLVE, primarly so that they can support multi-parameter functions. However, the design of our type theorems is flexible enough that it was not necessary to build in important type-relationships between cartesian product types.

[3]It would, of course, be sufficient to simply know `Is_Transitive(f)`, but since none of our benchmarks or examples make use of this weaker statement, we see no reason to multiply predicates unnecessarily.

7.2.1.1 Supporting Mathematics

As an example, consider the snippet of theory in Listing 7.16.

Listing 7.16: Passing a Cartesian product subtype

```
Definition TakesZs(zs : (Z * Z * Z)) : B;
Definition SomeNs : (N * N * N);
Definition AFact : B = TakesZs(SomeNs);
```

Certainly, this should be acceptible—a value in `(N * N * N)` is also in `(Z * Z * Z)`. This general subtype relationship between Cartesian products can be expressed as a type theorem, as shown in Listing 7.17.

Listing 7.17: A type theorem expressing Cartesian subtypes

```
Type Theorem Cart_Prod_Thingy:
        For all T1, T2 : MType,
        For all R1 : Powerset(T1),
        For all R2 : Powerset(T2),
        For all r1 : R1,
        For all r2 : R2,
                (r1, r2) : (T1 * T2);
```

Because of the inductive structure of our representation of Cartesian products (namely, `(N * N * N)` is merely shorthand for `((N * N) * N)`), this single definition covers arbitrarily large and nested products.

7.2.1.2 When Things Go Wrong

Removing the type theorem and running the same input, the RESOLVE compiler gives the error given in Listing 7.18.

Listing 7.18: Results without type theorem

```
Error: Demo_Theory.mt(65):
No function applicable for domain: ((N * N) * N)

Candidates:
        TakesZs : (((Z * Z) * Z) -> B)

        Definition AFact : B = TakesZs(SomeNs);
                               ^
```

7.2.2 Array Realization of Stack

For this benchmark, we used the `Static_Array_Template` discussed in Benchmark 2 to implement `Stack_Template`, which was an excellent stress-test for our array reasoning and highlights a number of interesting features.

7.2.2.1 Supporting Mathematics

Consider the representation clause from `Array_Realiz` provided in Listing 7.19.

Listing 7.19: The representation of `Stack`

```
Realization Array_Realiz for Stack_Template;
                uses Binary_Iterator_Theory;

         Type Stack is represented by Record
                        Contents: Array 1..Max_Depth of Entry;
                        Top: Integer;
                end;
                convention
                        0 <= S.Top <= Max_Depth;
                correspondence
                        Conc.S = Reverse(Concatenate(S.Contents, S.Top));

   ...
end;
```

Of particular interest here is the *correspondence* clause. The `Concatenate()` function here is the mathematical notion of "big concatenate"—i.e., the result of repeatedly concatenating elements together. It is not a special construct, but rather an ordinary function defined in `Binary_Iterator_Theory`.

That definition, along with one on which it depends, is given in Listing 7.20.

Listing 7.20: Definition of `Concatenate`

```
Theory Binary_Iterator_Theory;
                uses Integer_Theory, String_Theory;

         Definition Iterative_Apply(Step : ((R : MType) * (V : MType)) -> R,
                Start : R, Value_Function : Z -> V, Value_Count : Z) : R;

         Definition Concatenate(Value_Function : Z -> (T : MType),
                        Value_Count : Z) : Str(T) =
                Iterative_Apply(lambda (s : Str(T), t : T).(s o <t>),
                        Empty_String, Value_Function, Value_Count);

   ...
end;
```

The `Iterative_Apply()` function is a general aggregation function similar to *foldr()* in functional languages. It takes a starting aggregate value, an iteration function, an iteration count, and an aggregation function. Its semantic is to call the iteration function starting with the parameter 1 and continuing to `Value_Count` and repeatedly combining it with the working aggregate value using the aggregation function.

Thus, by supplying `Empty_String` as the starting aggregate value and a function for concatenating the next value onto the existing aggregate as the aggregation function, we achieve "big concatenate". If we were instead to supply `0` and an addition aggregator, we'd get "big sum".

Note the complexity of this type-checking task: in `Concatenate()`, the first parameter establishes the implicit type parameters of `Iterative_Apply()`, `R` and `V`. The body of the lambda expression is defined to be of type `SStr`, but a type theorem in `String_Theory` establishes that the result of concatenating two parameterized strings is another string with the same parameterization, thus allowing the lambda expression to meet the expected type of `(R * V) -> R`. The second parameter, `Empty_String` is declared of type `SStr`, but a second type theorem states that `Empty_String` is in all parameterized versions of `Str()`. For the third parameter, we match `T`, the type parameter to `Concatenate()`, with `V`, the type parameter to `Iterative_Apply()`, which has previously been established by the first parameter, despite neither type parameter having a concrete instantiation. Only the fourth parameter is straightforward.

When this realization of stack is submitted to the compiler, several of its VCs contain conbinations of these iterative aggregations and the piece-wise functions from `Static_Array_Template`. An example of such a VC is given in Listing 7.21.

While it seems complex at first glance, the goal of the VC is, in fact, a tautology that can be proven with this theorem:

```
Theorem Inductive_Reverse_1:
        For all f : Z -> Entity,
        For all i : Z,
        Reverse(Concatenate(f, i)) =
                <f(i)> o Reverse(Concatenate(f, i - 1));
```

Because of the flexibility of our type system, reasoning about these constructs requires no special machinery—functions can be defined and supporting theorems provided using standard syntax.

Listing 7.21: Complex VC arising from `Array_Realiz`

```
Goal:
Reverse(Concatenate(lambda j: Z ({{E if j = (S.Top + 1)
S.Contents(j) otherwise
}}), (S.Top + 1))) = (<E> o Reverse(Concatenate(S.Contents, S.Top)))

Given:
((((((Last_Char_Num > 0) and
(((min_int <= 0) and
(0 < max_int)) and
(((1 <= Max_Depth) and
(((min_int <= Max_Depth) and
(Max_Depth <= max_int)) and
((min_int <= 1) and
(1 <= max_int)))) and
((Max_Depth > 0) and
((min_int <= Max_Depth) and
(Max_Depth <= max_int)))))) and
((0 <= S.Top) and
(S.Top <= Max_Depth))) and
Conc.S = Reverse(Concatenate(S.Contents, S.Top))) and
(|Reverse(Concatenate(S.Contents, S.Top))| < Max_Depth)) and
E' = S.Contents((S.Top + 1)))
```

7.2.2.2 When Things Go Wrong

We will explore two things that could go wrong in the definition of `Concatenate()` offered in Listing 7.20.

First, consider Listing 7.22, where rather than pass `Value_Function` through to `Iterative_Apply()`, we instead pass a lambda expression with type `Z -> N`.

Listing 7.22: Passing an incorrectly-typed `Value_Function`

```
Definition Iterative_Apply(Step : ((R : MType) * (V : MType)) -> R,
        Start : R, Value_Function : Z -> V, Value_Count : Z) : R;

Definition Concatenate(Value_Function : Z -> (T : MType),
                Value_Count : Z) : Str(T) =
        Iterative_Apply(lambda (s : Str(T), t : T).(s o <t>),
                Empty_String, lambda(i : Z).(0), Value_Count);
```

Because `N` does not necessarily match the concrete type of `V` as established by the first parameter passed to `Iterative_Apply()` (`V` should be `T`), this should be an error, and indeed, the RESOLVE compiler gives the results shown in Listing 7.23.

Now consider that the aggregation function did not maintain the correct type, as shown in Listing 7.24, where the aggregation function returns `s o <0>`, a valid call to concatenate, but one that does not meet the qualifications for the application of the necessary type theorem.

In this case, the output from the RESOLVE compiler is shown in Listing 7.25.

Listing 7.23: The third parameter does not have correct type

```
Error: Binary_Iterator_Theory.mt(9):
No function applicable for domain: (((((Str('T') * 'T') -> SStr) * SStr)
        * (Z -> N)) * Z)

Candidates:
        Iterative_Apply : (((((Step : (('R' * 'V') -> 'R')) * (Start : 'R'))
                * (Value_Function : (Z -> 'V'))) * (Value_Count : Z)) -> 'R')

        Iterative_Apply(lambda (s : Str(T), t : T).(s o <t>),
        ^
```

Listing 7.24: The expression returned by the lambda function does not maintain the string type

```
Definition Iterative_Apply(Step : ((R : MType) * (V : MType)) -> R,
        Start : R, Value_Function : Z -> V, Value_Count : Z) : R;

Definition Concatenate(Value_Function : Z -> (T : MType),
        Value_Count : Z) : Str(T) =
        Iterative_Apply(lambda (s : Str(T), t : T).(s o <0>),
        Empty_String, Value_Function, Value_Count);
```

Listing 7.25: The first parameter does not have correct type

```
Error: Binary_Iterator_Theory.mt(9):
No function applicable for domain: (((((Str('T') * 'T') -> SStr) * SStr)
        * (Z -> 'T')) * Z)

Candidates:
        Iterative_Apply : (((((Step : (('R' * 'V') -> 'R')) * (Start : 'R'))
        * (Value_Function : (Z -> 'V'))) * (Value_Count : Z)) -> 'R')

        Iterative_Apply(lambda (s : Str(T), t : T).(s o <0>),
        ^
```

Chapter 8

Conclusion and Future Research

This research has demonstrated that a combination of better component specifications achieved with a flexible and expressive mathematical system and a domain-specific prover carefully tailored to the sorts of VCs that arise from well-engineered programs can significantly impact the verifiability of software. While faster and more advanced provers are often cited as the answer the software verification problem, this research has demonstrated that an extremely modest prover can compete with more traditional verification systems simply by exploiting properties and patterns inherent to software verification. In short, we have confirmed our hypothesis: that programmers write code they believe works, and that the necessary insights for proving resulting proof obligations should therefore be shallow.

Beyond providing evidence against the conception that reasoning formally about programs is difficult, this research has also developed a prototype verification system that exemplifies a hybrid design philosophy, taking many of the best attributes of purely mathematical reasoning systems and blending them with a practical, imperative, object-based programming language. As a result, RESOLVE now represents the only verification language to combine an imperative, object-based language with features like higher-order logic, first-class dependent types, and an extensible mathematical universe based on a traditional foundational theory. In the process, we have designed and implemented a novel compromise for type-checking a language with dependent types.

As much as we would care to close the book on the verification problem with this research, we acknowledge this represents only a small step toward the eventual goal of the true push-button verifying compiler. And, while we are pleased to have addressed a number of questions, we have

also raised many more:

Clearly, our ideas on how to qualify a likely "good" step were not supported by the data. However, the consistency with which our fitness function fails nonetheless suggests that a good function is out there—after all, we have one that describes "bad" steps. An open question is to determine if our function fails to take into account important variables, or if it merely combines the variables it has in the wrong proportions. One interesting experiment might be to use a genetic algorithm or some other general optimization technique to attempt to find a suitable function.

While we genuinely believe that our static typing system is now the most flexible of any language, practical or pure, we are already seeing some of the cost of that flexibility. Despite a straightforward algorithm, mixing the use of explicit type parameters, implicit type parameters, and type theorems often has counter-intuitive results, causing statements that are certainly mathematically sound to fail type-checking. A subtler algorithm based on a dependency tree rather that strict left-to-right evaluation might be able to statically establish such statements. Similarly, when mixing multiple types with complex relationships, RESOLVE is often unable to break ties between multiple seemingly-equally-applicable functions. A mechanism to prioritize those by giving the type-checking system insight into the relationships between *functions* in addition to types would resolve many of these issues.

The scalability of the prover is always a serious concern. After all, both the antecedent development step and the consequent exploration step suffer from combinatorial explosion—the former of space, the latter of time. While we are heartened to find that, after the application of our heuristics, very few steps are generally required in the consequent exploration phase, this does little to address the issue with the antecedent development stage. To maintain scalability, further research will need to be done to qualify useful developments and useless transformations. We hypothosize that ultimately a feedback loop between consequent exploration and antecedent development might be more appropriate than our current two-phase algorithm. This would represent a kind of simulated annealing wherein the consequent exploration would represent a random walk, and intermittent antecedent development would function as a hill-climb.

Finally, and most importantly, more research is required on the human component of verification. We are excited to have contributed to this with a more intuitive mathematical system with which—we hope—a mathematician will feel more at home than with the currently available formal mathematical languages. But the interaction of a human programmer with a programming system

is a complex and under-appreciated problem in which we often do not see the programmers of the language as true users. But if verification is to succeed, we must overcome these issues to develop a system that strikes the proper balance between the formal rigor required for successful verification and the deep insights needed to support the programmer and bring that rigor within reach.

Appendices

Appendix A Dragga and Gong's Editing Process Model

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam

voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam

et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Appendix B Manuscript Review Sign-In Datat Aheet for Students

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing

elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore

magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur

sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Appendix C Institutional Review Board (IRB) Application and Attachments

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

C.1 Importance of the Study

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

C.2 Purpose of the Study

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea

commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

C.3 Explanation of Research Design

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

C.4 Delimitations of the Study

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy

eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

C.5 Institutional Review Board (IRB) Approval Information

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed

takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

C.6 Justification for an Overview of the Methodological Decisions and Their Limits

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis.

At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, At accusam aliquyam diam diam dolore dolores duo eirmod eos erat, et nonumy sed tempor et et invidunt justo labore Stet clita ea et gubergren, kasd magna no rebum. sanctus sea sed takimata ut vero voluptua. est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat.

C.7 Summary of Methods

Consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit

amet.

Bibliography

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69149-5_16.

[2] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[3] J.L. Bentley. *Programming pearls*. ACM Press Series. Addison-Wesley, 2000.

[4] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken, June 2006.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[6] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the jahob data structure verification system. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-69738-1_5.

[7] Derek Bronish and Hampton Smith. Robust, generic, modularly-verified map: a software verification challenge problem. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 27–30, New York, NY, USA, 2011. ACM.

[8] Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78800-3_24.

[9] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[10] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27:99–123, February 2001.

[12] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[14] Douglas E. Harms and Bruce W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17:424–435, May 1991.

[15] Heather Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software.* Phd dissertation, Clemson University, School of Computing, December 2011.

[16] John Hatcliff, Gary T. Leavens, K. Rustan, M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. 2009.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[18] Tony Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50:63–69, January 2003.

[19] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Matt Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23:203–213, April 1997.

[21] James Cornelius King. *A program verifier.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970. AAI7018026.

[22] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: experience report. In *Proceedings of the 17th international conference on Formal methods*, FM'11, pages 154–168, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[24] Greg Kulczycki, Hampton Smith, Heather Harton, Murali Sitaraman, William F. Ogden, and Joseph E. Hollingsworth. The location linking concept: A basis for verification of code using pointers. In *Proceedings of VSTTE 2012 (to appear)*, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] Viktor Kuncak and Martin Rinard. An overview of the jahob analysis system: project goals and current status. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 285–285, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[27] K. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_8.

[28] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52:107–115, July 2009.

[29] LogiCal Project. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[30] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, 1988.

[31] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53:937–977, November 2006.

[32] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[33] Kalyan C Regula, Hampton Smith, Heather Harton Keown, Jason O Hallstrom, Nigamanth Sridhar, and Murali Sitaraman. A case study in verification of embedded network software. In *NASA Formal Methods*, pages 433–448. Springer, 2012.

[34] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Hossein Sheini and Karem Sakallah. A sat-based decision procedure for mixed logical/integer linear problems. In Roman Bartk and Michela Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 847–849. Springer Berlin / Heidelberg, 2005. 10.1007/11493853_24.

[36] Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey Friedman, Heather Harton, Wayne Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith, and Bruce Weide. Building a push-button resolve verifier: Progress and challenges. *Formal Aspects of Computing*, 23:607–626, 2011. 10.1007/s00165-010-0154-3.

[37] Murali Sitaraman and Bruce Weide. Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, 1994.

[38] Marulli Sitariman and Bruce Weide. Component-based software using resolve. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–22, 1994.

[39] Hampton Smith. Anatomy of a platform for prover experimentation. Technical report, Clemson University, 2010.

[40] Hampton Smith. Impact of specification abstractions on client verification. In *Proceedings of SAVCBS 2010*, SAVCBS 2010, November 2010.

[41] Hampton Smith, Kim Roche, Murali Sitaraman, Joan Krone, and William F. Ogden. Integrating math units and proof checking for specification and verification. In *Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 59–66, 2008.

[42] Aditi Tagore, Diego Zaccai, and Bruce. W. Weide. To expand or not to expand: Automatically verifying software specified with complex mathematical definitions. Technical report, The Ohio State University, September 2011.

[43] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49, 1995.

[44] Bruce W. Weide and Wayne D. Heym. Specification and verification with references. In *Proceedings of 2001 OOPSLA workshop on Specification and Verification of Component-Based Systems*, SAVCBS 2001, pages 50–59, 2001.

[45] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce M. Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 84–98, 2008.

[46] Markus M. Wenzel and Technische Universitt Mnchen. Isabelle/isar - a versatile environment for human-readable formal proof documents. In *TPHOLS*, pages 167–184, 1999.

[47] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 349–361, New York, NY, USA, 2008. ACM.

[48] Karen Zee, Viktor Kuncak, and Martin C. Rinard. An integrated proof language for imperative programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 338–351, New York, NY, USA, 2009. ACM.