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[**?**], but based on research into program correctness stretching back to the 1960s[**?**]. 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.

As a powerful demonstration of the weakness of traditional testing and informal reasoning, we consider the case of binary search. Binary search is a simple, well-understood, and widely-implemented algorithm. Yet, Joshua Bloch of Google Research wrote a blog post[**?**] in 2006 about a subtle bug in the standard Java library's implementation of binary search—an implementation that had been in place for nine years 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*[**?**]. 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 revealed not by a nit-picking graduate student but by a client in the wild whose code broke when the component failed to meet its specification (specifically causing an `ArrayIndexOutOfBoundsException`, which, in a C or C++context, would 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 shallow under formal

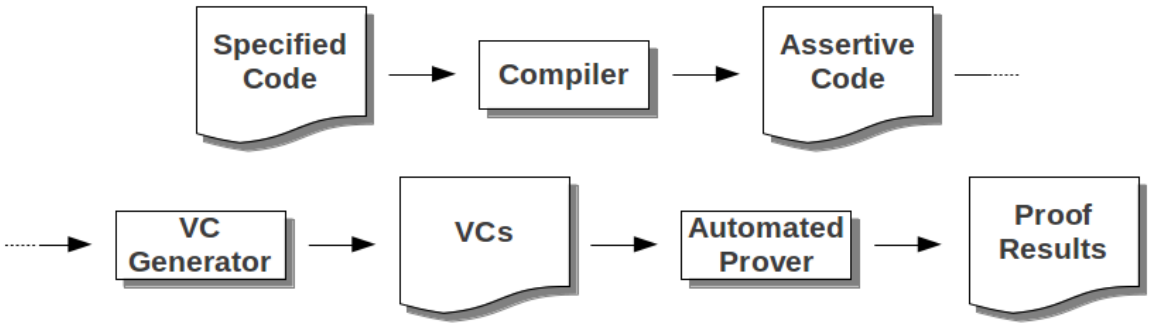

Figure 1.1: The Verification Pipeline

reasoning, where bounded, programmatic integers are well-specified in a mathematical way.

Several systems for formal verification exist, 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 express the proof obligations of the code in a purely mathematical way, before finally being sent to one or more *automated provers* which attempt to dispatch the VCs. This process is illustrated in Figure 1.1.

Despite many successes, all 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 even simple programs. This is counter-intuitive since most programs contain straight-forward logic that the programmer feels assured of without calling upon complex mathematical reasoning. The problem is compounded by a lack of integration between between modern, modular programming languages and expressive, flexible mathematics that prevents the design of reusable components, which could amortize this intense verification effort over time. We hypothesize that a well-integrated, flexible and extensible mathematical and specification subsystem would permit specifications that more closely reflect the programer's intuition and the usual patterns of reuse, resulting both in more straightforward proofs and more generic, longer-lived components. This proposal seeks to develop such a system and test this hypothesis.

1.1 Existing 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. Examples of the former include Jahob[**?**], based on Java; Spec#[**?**], based on C#; and ACL2[**?**], based on Lisp. Examples of the latter include Coq[**?**], based on the Calculus of Inductive Constructions; and Isabelle[**?**], based on a higher-order, intuitionistic logic. 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.

1.1.1 Example Practical System: Jahob

Let's consider a version of Java's `ArrayList` verified in Jahob[2]. Jahob combines code written in a subset of Java with specifications written in Isabelle. Listing 1.1 shows the `contains()` operation of an ArrayList.

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;
        */
```

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

[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`.

```
        return res;
    }
```

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. `init` is a predicate defined elsewhere in the class. Jahob encourages the inclusion of inline "hints" targeted at the backend provers[**?**] (of which it supports many) and we see several of those here interspersed in the Java code.

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[**?**]. 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[**?**], 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[**?**] and Z3[**?**]. 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, a number of the realities of this system (and others like it) are counter-intuitive and seem geared toward short-term verification of small Java programs rather than long-term, scalable verification of complex, modular applications.

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].

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

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[**?**]. These complexities must be accounted for in its VCs. 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 for 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 such a 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 definitions (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 [**?**]. The trouble of aliased keys is dealt with 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

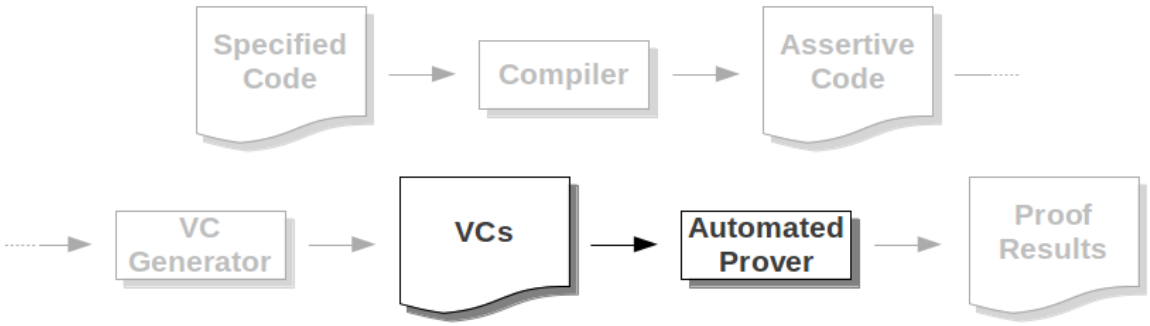

Figure 1.2: Shifting the Burden to the Prover

with respect to `equals()`. The correctness of the `Map` implementation requires changes to external components, rather than being part of its self-contained specification. We discuss this problem in more detail in [**?**].

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. It 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.

1.1.2 Example Pure System: Coq

Let's 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 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 :=
```

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

```
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 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, that for all inputs, if the inputs meet the precondition, then the behavioral relation 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).
```

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. Obviously, however, 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 this have been used to excellent effect verifying complex programs. Coq has been used to verify a C compiler[**?**], while Isabelle has been used to verify an operating system kernel[**?**].

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 repeatedly decomposed into smaller and 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's 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 small enough that the auto-

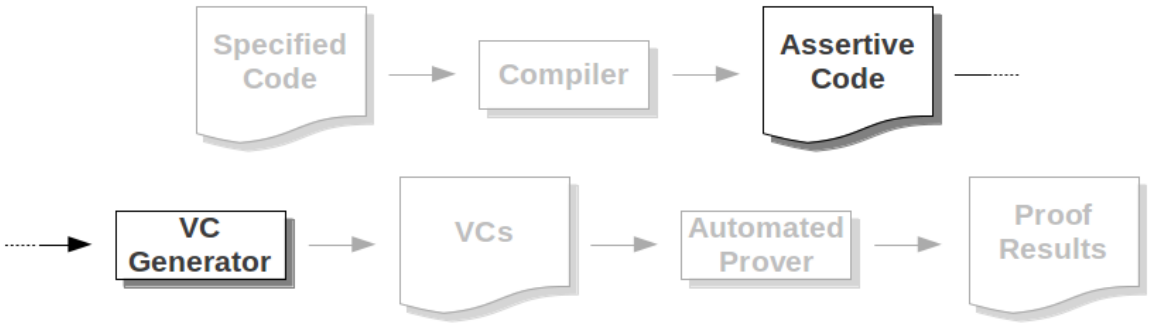

Figure 1.3: Shifting the Burden to the VC Generator

mated prover can take it 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, in order to exploit the flexibility of the mathematical system, the automated prover must be more general, unable to take advantage of the numerous domain-specific tricks that cutting-edge narrowly-applicable theorem provers exploit. 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.

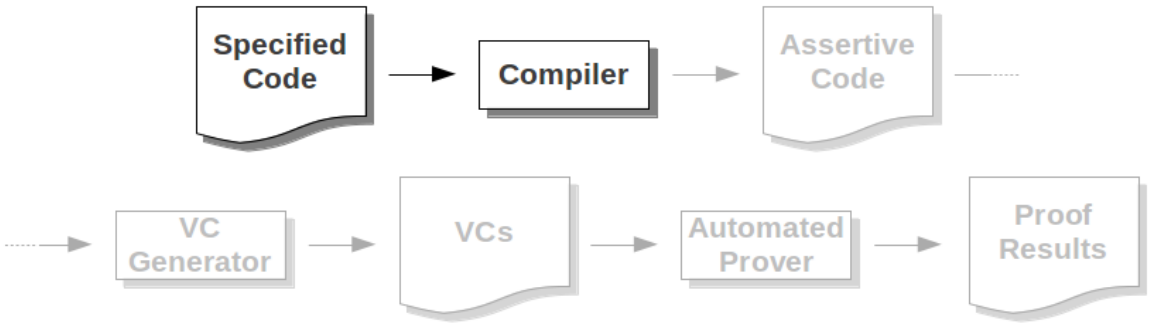

Figure 1.4: Shifting the Burden to Specification

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, unlike proofs of correctness.

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 in pure system like Coq in making better up-front choices to pay long-term dividends.

1.3 Problem Statement

This proposal seeks to determine if the mechanical verifiability of software components can be improved by better-engineered mathematics and specifications and, if so, explore which techniques might best support these goals. In the process it will address the following open problems in the area of formal methods:

- Design of an extensible, flexible mathematical framework and a well-integrated associated specification framework, supporting verifiability by allowing specifications based on the *best* mathematical model, rather than simply the most convenient, and supporting scalability through mathematical reuse.
- Architecture of and experimentation with a minimalist rewrite prover to support reasoning in the above framework and determine those prover capabilities practically necessary to mechanically verify well-engineered, modular components.
- Creation of a diverse library of components of the sort found in standard programming libraries, specified and modeled using a broad set of techniques to enable experimentation on mathematical and specification best-practices for practical verification.

1.4 Research Approach

Building on previous work on specification language design, VC generation, and automated prover development, this research seeks to augment the existing RESOLVE[**?**] system with a flexible mathematical subsystem and modular specification subsystem in order to experiment with the specification and mathematics best-practices in support of modularity.

Unlike 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 proposed here will be tightly integrated and based on an extensible Morse-Kelley Set Theory, extended to include higher-order definitions, and will permit 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 proposed here will cooperate 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 will test its flexibility by designing, specifying, and implementing a library of programming components ranging from simple stacks to more complicated tree structures, along with the algorithms for manipulating these structures. These structures and algorithms will be modeled and specified using a variety of techniques in order to gain insight into how these techniques impact verifiability and scalability. We will analyze resulting VCs to determine which prover capabilities are strictly necessary, designing and building a minimalist rewrite-based prover based on a plug-in architecture to support only those capabilites necessary for practical VCs.

The evaluation of the system as a whole will include classification of VCs by the minimalist prover based on a number of proof metrics including number of required theorems, number of required proof steps, and proof time; as well as subjective, qualitative metrics derived from field tests with programming and mathematics students and professionals.

1.4.1 Contributions

Such a system would 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[**?**], 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[**?**]. 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, here 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 and what it means to be a "well-engineered specification" are 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 theorem provers.

1.6 Proposal Organization

The remainder of this proposal is organized as follows: Section 2 gives an overview of the state-of-the-art in each of the problem areas, with information on work related to this research, Section 3 provides background on the RESOLVE verification system that is the platform for this research, Section **??** presents the proposed research including work already completed, Section **??** explains the criteria against which the work will be evaluated, and Section **??** will offer some concluding thoughts.

Chapter 2

Verification Background and Related Work

In order to justify the proposed research as well as place it in context, it is important to consider the full breadth of available mathematical systems, provers, and verification libraries. In the following three sections, we break related work and necessary background down along those lines.

2.1 Specification and Mathematical Systems

Early systems for program specification included Z[**?**] and Larch[**?**]. Both are first-order predicate logics not grounded in any particular programming system, though each has been adapted for many different 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, the first-order restriction remains ubiquitous, severely restricting the genericity (and thus the reusability) of mathematical theories and specifications, as

discussed in some detail in Section **??**.

An extremely 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 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". In addition, if the language in question does not have a formal semantic, then the specification is only as clear as our understanding of the meaning of the underlying language. A number of systems use this method as their basis (though in some cases they offer non-executable extensions).

One prominent example of this style is the Java Modeling Language (JML)[**?**]. 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, their behavioral specification is 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[1].

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 development 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 are both arbitrary-feeling (e.g., no multiple subtyping) and lead to a decrease in the allowable genericity of mathematical theories, curtailing their reuse.

Another system of specification along these lines is Why[**?**]. Why is ML-like and focussed on reasoning about built-in structures like arrays rather than general data abstractions. By using

[1]We effectively are able to include an instance of the *Strategy* design pattern in our specification.

a variety of translation front-ends, Why can be an intermediate specification and programming representation for different programming languages including Java, C, and ML.

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

Another popular style of specification in practical systems is separation logic[**?**]. Because it models the heap directly, this style provides mechanisms for reasoning about aliasing, pointer arithmetic, et al. 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[**?**], a system for verifying Java and C programs that uses separation logic expressed in a custom specification language and targets the Z3 prover.

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 already discussed 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. RESOLVE strives to provide a sorted, higher-order mathematical language, though the complete design and implementation of this language is part of the topic of this proposal. 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, reasoning, instead, about a series of built-in sorts.

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

Amongst the pure systems, the style of specification and mathematical representation be-

comes more uniform, at least in their almost-universal inclusion the features we feel are crucial for a component specification platform (higher-order definitions; first-class types; and a rich, extensible theory of sorts.) By far the most popular 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[**?**].

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)[2], 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 an extremely specific domain to arrive at conclusions quickly. Examples include linear arithmetic solvers of the kind found in Z3[**?**] and OSU's SplitDecision[**?**] system for reasoning about 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 simply 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

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

to drastically reduce the problem space for practical formulae. Almost universally these 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 [**?**].

A number of SAT-solvers exist. Z3 and Yices[**?**] 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, [**?**] and [**?**].) SAT-solvers that extend themselves 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 being would proceed with a proof—by matching theorems against a set of known facts, transforming them until we derive the result we want. This general pattern-matching 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 without backtracking. This differs from the approach of our minimalist prover—where we employ backtracking and 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.

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

2.3 Benchmarks, Libraries, 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 to how they appear in the Java standard library as possible. In a recent paper [**?**], they published about 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. This library is geared towards linked data structures and the expressiveness not to have to compromise a component design from what we see in normal industrial systems.

Additionally of interest from the Jahob team is a recent paper[**?**] that deals directly with specification engineering, noting that quantifiers consistently complicate verification and engineering the specifications of a set of data structures to do without, enabling them to be translated down to a first-order logic, then passed to an efficient first-order prover. This result, however, focussed on the translation process rather than the novelty of the importance of specification style. To our knowledge, the team has not embarked on a systematic exploration of this facet of verification, nor do they have any plans to create a library focussed on this dimension.

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.

The nature of these challenges have been all over the map, 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 our purpose here: 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, instead re-implementing a linked list, presumably because they required different properties to achieve verification.

In 2008, researchers here at Clemson and at the Ohio State University compiled and published[**?**] 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[**?**], which revealed a number of interesting properties of that system. Our hope is that the library developed for this research can have a similar effect, eliciting discussion and comparison of different verification systems by encouraging other verification efforts to explore the effect of differing specification on properties of the verification.

Chapter 3

RESOLVE Background

The RESOLVE[?] 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, which attempts scalability by ensuring that each component is verified in isolation and need not be re-verified regardless of deployment. This means that encapsulation must be strictly assured and thus 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 operate over any other RESOLVE type, and `Max_Depth` is an integer indicating the maximum number of elements that can ever be on the 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[**?**]. The *uses* line includes one such file—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. `exemplar` introduces a name to be used for a hypothetical 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.

We then define some operations on `Stacks`—`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 will be changed to 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; a `replaces` parameter will be passed an arbitrary value and be changed to a meaningful value by the end of the call.

`requires` and `ensures` clauses are mathematical assertions and their variables refer to the mathematical models of the parameters. So, while `Push` operates on physical `Stack`s, `S` in its `ensures` clause refers to the mathematical representation 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 there is no conception of the the values of parameters at the end of the call and so pounds are not used and 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;
        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;
```

[1] In addition to this array-based implementation, we are experimenting with a linked-structure mathematical component in order to provide, in this case, a linked-list based implementation. The details of this exploration can be found in [**?**].

```
    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` represents an *invariant* that must hold after initialization, before finalization, and before and after each method. In this case, the top must be at a valid index (save index 0, where it is permitted to reside if the Stack is empty.)

A `correspondence` provides a mapping between the physical `Stack` and its mathematical conceptualization, in this case using a mathematical definition `Concatenation`, which is to string concatenation what big Σ is to integer addition, to build a string of all its elements.

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 [**?**]. 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[**?**] and is discussed more thoroughly in Section **??**.

Concepts may further define *extension operations*, which are secondary operations describing useful operations that not all realizations will be able to implement practically or efficiently. As an example, `Stack_Template` has an extension operation called `Flip`:

```
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 flipping procedure establishes a new `Stack`, `S_Flipped`, then iteratively pops each element off the original stack and into the new one. Having done so, the `:=:` operator swaps the value of `S_Flipped` with `S`. Using swapping as its basic data movement operator enables RESOLVE to minimize undesirable aliasing[**?**].

Notice that the while loop requires a number of 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. Finally, the *decreasing* clause establishes a termination metric to allow us to guarantee 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[**?**]. We note, however, that some work has been done on automatically discovering, e.g., loop invariants on the programmer's behalf[**?**]. 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 Section **??**.

Chapter 4

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.

4.1 Preliminaries

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

4.1.1 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.

4.1.2 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.

4.1.3 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.)

4.2 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.

4.2.1 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 5 of this dissertation, we can affect this effort in the theory development domain by creating theories that result in *more straightforward* VCs.

4.2.2 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.

4.2.3 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.

4.3 Concrete Features

4.3.1 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 5, 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 [**?**].

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:

4.3.1.1 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.

4.3.1.2 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.

4.3.1.3 Extensible Specification

Inheritence is a powerful tool, but the cause of much problematic reasoning[**?**]. 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.

4.3.1.4 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;
```

4.3.2 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 **MType**s.* 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 4.1. (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 4.1: 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:

4.3.2.1 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.

4.3.2.2 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.

4.3.2.3 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);
```

4.3.3 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.

4.3.4 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`

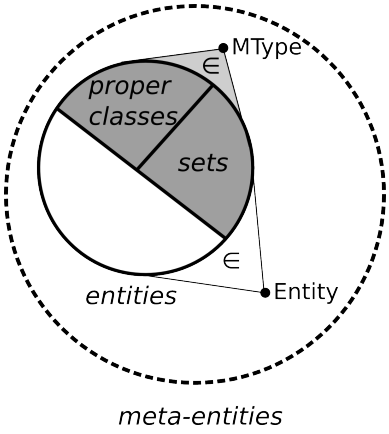

Figure 4.2: A High-level View of the RESOLVE Mathematical Universe

from Section 4.3.2. 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 4.2.

4.3.5 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.

4.3.5.1 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.

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 4.3. 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

Figure 4.3: Syntactic judgements for determining if one type expression is a subtype of another

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

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

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

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

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.

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.

4.3.5.2 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

[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.

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 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.

Chapter 5

Minimalist Automated Prover

At the core of any mechanical verification system is an automated theorem prover responsible for discharging VCs. By definition, it is the last word on whether or not a particular technique is yielding more or less easily-proved VCs. As a result of this, most practical systems have focussed 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 we are happy to support the latest and greatest suite of provers, we hypothesize that in many cases *flexibility* may trump raw performance with respect to mechanically verifying well-engineered software by encouraging good specification and mathematical engineering that captures the programmer's intuition rather than compromising to work within the framework of a target prover.

In order to experiment with this hypothesis and identify those prover capabilities and performance tunings required to verify well-engineered software, we set out to create a *minimalist automated prover*, starting with only the bare essential capabilities and expanding only when a significant number of VCs appeared that could not be addressed with the prover as it stood. 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 within the group and we intend to use it as the yardstick against which to measure our success using our new mathematical system to verify components.