Abstraction as the Key to Programming, with Issues for Software Verification in Functional Languages

Dissertation

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

By

Derek Bronish, B.S., M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Dissertation Committee:

Bruce W. Weide, Advisor

Atanas Rountev

Paul Sivilotti

Abstract

Abstraction is our most powerful tool for understanding the universe scientifically. In computer science, the term "abstraction" is overloaded, and its referents have not been enunciated with sufficient clarity. A deep understanding of what abstraction is and the different ways it can be employed would do much to strengthen both the research and the practice of software engineering.

Specifically, this work focuses on abstraction into pure mathematics as it pertains to the intellectual complexity of computer programming. A Grand Challenge for software engineering research is to develop a verifying compiler, which takes as input a computer program and a rigorous specification of its intended behavior, and which only generates an executable if the code is proven correct via automated reasoning. The hypothetical verifying compiler would radically change the face of technology by guaranteeing that code always behaves as it should. We investigate the ways in which good abstractions, maximally exploited, can make this dream a reality.

This document presents, at varying levels of technical detail, evidence for the utility of abstractions in software. We show how standard approaches to programming, even in the realm of "formal methods," lack full abstraction. We argue that data abstraction should be achieved via purely mathematical modeling, and that this approach enables modular, scalable verification of complete system behavior. We also warn that a programming language's formal semantics can dramatically impact the

feasibility of a verifying compiler for that language. One of our main results is that the class of "functional" programming languages cannot be soundly verified by the pervasive existing methods due to an insidious and subtle issue related to data abstraction and relational specifications. We build on this unsoundness proof by sketching a new solution, and fragments of a new functional programming language that incorporates it.

For myself.

Acknowledgments

I owe a debt of gratitude to the writers, thinkers, and creators who have inspired me and spurred my intellectual development: Ayn Rand, Victor Hugo, Tom Scharpling, Cormac McCarthy, Paul Thomas Anderson, Kurt Vonnegut, Wes Anderson, Bill Murray, Vincent Gallo, Henry Rollins, Thomas Pynchon, Philip K. Dick, Bob Rafelson, and many others. Most importantly, reading David Foster Wallace's *Infinite Jest* in the summer of 2011 convinced me once and for all that the mind is capable of immortal feats, and that humanity is utterly peerless in its dignity and efficacy.

My committee members and other faculty at Ohio State have been enduring sources of inspiration. I'm proud to call such people, whom I genuinely view as luminaries and role models, my colleagues. In particular, I'd like to thank my adviser, Dr. Bruce Weide, for providing much-needed reassurance whenever my resolve (so to speak) flagged, and for encouraging my non-technical flights of fancy, several of which found their way into this work. Thank you to Drs. Paul Sivilotti, Bill Ogden, Harvey Friedman, Wayne Heym, Paolo Bucci, Nasko Rountev, Tim Carlson, and Neelam Soundarajan for contributing countless ideas and suggestions to my studies. My co-workers in the Reusable Software Research Group (RSRG) have also helped me in innumerable ways—thank you Bruce Adcock, Dustin Hoffman, Jason Kirschenbaum, Ted Pavlic, Hampton Smith, Aditi Tagore, and Diego Zaccai.

My incredible girlfriend Ellen Fetterman has motivated, inspired, challenged, supported, and entertained me like no other person ever could. Thank you for making my life better daily.

Thank you to my great friends Mark Derian, Matt Gross(i), Eric Bogart, Antonio Ciaccia, and Ben Keller.

And my parents, duh. Thank you Joe and Lynn Bronish for material and emotional support. Watching "Jeopardy!" with my mom and learning guitar from my dad were, in hindsight, pivotal experiences that shaped my life and helped me to turn out okay. I didn't really understand it until I grew up and saw more of the world, but the average person is not as consistently good and kind and smart as my parents are. I'm so lucky to have had them as primary influences on my life.

Derek Bronish
Columbus, Ohio

Vita

May 15, 1983	Born – Westlake, OH
2006	B.S. Computer Science & Engineering, *cum laude*, The Ohio State University. Minors in Philosophy and Linguistics
2010	M.S. Computer Science & Engineering, The Ohio State University
2006 — Present	Graduate Research Associate/Graduate Teaching Associate, The Ohio State University

Publications

Murali Sitaraman, Bruce Adcock, Jeremy Avigad, Derek Bronish, Paolo Bucci, David Frazier, Harvey M. Friedman, Heather Harton, Wayne D. Heym, Jason Kirschenbaum, Joan Krone, Hampton Smith and Bruce W. Weide. "Building a Push-Button RESOLVE Verifier: Progress and Challenges." Formal Aspects of Computing. Volume 23, Number 5. September 2011.

Neelam Soundarajan, Derek Bronish, and Raffi Khatchadourian. "Formalizing Reusable Aspect-Oriented Concurrency Control". 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011). July 2011.

Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, et. al. "The 1st Verified Software Competition: Experience Report". 17th International Symposium on Formal Methods (FM2011). June 2011.

Derek Bronish and Hampton Smith. "Robust, Generic, Modularly Verified Map: A Software Verification Challenge Problem". 5th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV 2011). January 2011.

Derek Bronish, Jason Kirschenbaum, Aditi Tagore, and Bruce W. Weide. "A Benchmark and Competition-Based Approach to Software Engineering Research." Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010), November 2010.

Derek Bronish and Bruce W. Weide. "A Review of Verification Benchmark Solutions Using Dafny". 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10). August 2010.

Derek Bronish. "Applying the Lessons of RESOLVE to a Functional Programming Language". RESOLVE 2010 Workshop: Advances in Automated Verification. June 2010.

Jason Kirschenbaum, Bruce Adcock, Derek Bronish, and Bruce W. Weide. "Automatic Full Functional Verification of Clients of User-Defined Abstract Data Types". Technical Report OSU-CISRC-1/10-TR04, The Ohio State University. June 2010.

Scott M. Pike, Wayne D. Heym, Bruce Adcock, Derek Bronish, Jason Kirschenbaum, and Bruce W. Weide. "Traditional Assignment Considered Harmful". 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2009). October 2009.

Bruce Adcock, Jason Kirschenbaum, and Derek Bronish. "The OSU RESOLVE Verification Tool Suite". 11th International Conference on Software Reuse (ICSR 11). September 2009.

Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Hampton Smith, Heather Harton, Murali Sitaraman, and Bruce W. Weide. "Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping." 11th International Conference on Software Reuse (ICSR 11), September 2009.

Derek Bronish. "The Ramifications of Programming Language Design on Verifiability". RESOLVE 2009 Workshop: Software Verification—The Cornerstone of Reuse. September 2009.

Jason Kirschenbaum, Bruce Adcock, Derek Bronish, Paolo Bucci, and Bruce W. Weide. "Using Isabelle Theories to Help Verify Code That Uses Abstract Data Types". Workshop on Specification and Verification of Component-Based Systems (SAVCBS'08). November 2008.

Bruce W. Weide, Murali Sitaraman, Heather Harton, Bruce Adcock, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. "Incremental Benchmarks for Software Verification Tools and Techniques". 2nd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08). October 2008.

Derek Bronish, Jason Kirschenbaum, Bruce Adcock, and Bruce W. Weide. "On Soundness of Verification for Software with Functional Semantics and Abstract Data Types". Technical Report OSU-CISRC-5/08-TR26, The Ohio State University. May 2008.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Software Verification	Prof. Bruce W. Weide
Theory	Prof. Tim Carlson
Distributed Systems	Prof. Paolo A.G. Sivilotti

Table of Contents

Page

List of Tables

List of Figures

Figure **Page**

Chapter 1: Introduction: What is Abstraction?

§1.1 Overview

This work is a meditation on abstraction, conducted at various levels of technical detail (of abstraction, if you will). We begin by discussing the term's use in different intellectual disciplines, and then move on to clarifying its overloaded usage in computer science.

The subdomain of computing we're interested in is *verification*—rigorously proving that programs are correct. We advocate a particular method for versatile software abstractions known as "purely mathematical modeling," and we detail an approach to verification that exploits mathematical abstractions to prove that code written in an imperative programming language meets its formal specifications. This verification project, known as "Resolve," is discussed in technical detail, including an overview of its architecture, tool support, and notable achievements.

We also propose a new fork of the Resolve project, one that facilitates verification of a "functional" programming language in the spirit of Lisp or Haskell. The nature of functional languages introduces new issues of semantics that the current Resolve research program has not had to deal with. This work proves the unsoundness of

a widespread approach to data representation verification, and proposes a new language mechanism and proof rule that fixes the problem. Because automation is such an important aspect of the grand challenge, existing tools for automated reasoning are examined, and some results about decision procedures in mathematical logic are discussed.

§1.2 The Term "Abstraction"

> Like any organ, the brain consists of large parts (such as the hippocampus and the cortex) that are made up of small parts (such as "maps" in the visual cortex), which themselves are made up of smaller parts, until you get to neurons ... whose orchestrated firing is the stuff of thought. The neurons are made up of parts like axons and dendrites, which are made up of smaller parts like terminal buttons and receptor sites, which are made up of molecules, and so on.
>
> This hierarchical structure makes possible the research programs of psychology and neuroscience. The idea is that interesting properties of the whole (intelligence, decision-making, emotions, moral sensibility) can be understood in terms of the interaction of components that themselves lack these properties. This is how computers work; there is every reason to believe that this is how we work, too.
>
> (Paul Bloom. "First Person Plural." *The Atlantic*, Nov. 2008.)

Renowned psychologist Paul Bloom observes in his *Atlantic* essay that the ability to decompose a complicated system into its constituents is a powerful aid to human understanding. Decomposition is not a necessary task—laypeople use the concept "brain" without knowing the taxonomy this quotation details, and my parents can work their laptop without knowing about paging or threads or registers. However, as the second paragraph of Bloom's quote suggests, decomposition *is* necessary if we are to answer "why" questions about a system. Decomposition of a system into parts whose interactions can be investigated to explain the nature of the whole is sometimes referred to as a "top-down" view.

A top-down decomposition seems useful when one needs to analyze an existing system and discover how it works. When one wants to *create* a complex system, there is a dual strategy that works "bottom-up": simpler, well-understood pieces are arranged appropriately and then forgotten about as one moves to higher and higher degrees of sophistication. As an example, consider a civil engineer designing a bridge. There are certain to be many specific requirements and constraints on the problem given the environment and materials available, but reliable "black-box" components such as trusses and girders can be employed so that the engineer doesn't have to "start from scratch." As some parts are assembled, they comprise larger units that can then be thought of holistically, and so the bridge can be built without the engineer ever needing to reason about all the interactions on all the different levels of detail simultaneously. Indeed, bridges are so complicated, and science has developed such robust top-down views, that this black-boxing seems absolutely necessary, lest we drown in the details of covalent bonds as we try to cross a river.

The examples of the brain's structural architecture and the engineer's level-ascending thought process are instructive in that they are concrete: one can point to them and say: "*that* is abstraction." To state a general definition would require moving away from the specific details of any one example: ironically, one must engage in abstraction to define abstraction.

§1.2.1 "Abstraction" in Common Parlance

Abstraction is defined by the *New Oxford American Dictionary* as "the process of considering something independently of its associations, attributes, or concrete accompaniments." Not only is this process at work when one designs a bridge, it's also

necessary for anybody wishing to write a dictionary definition. To define "abstraction," the *NOAD*'s authors had to pick many examples of abstraction occurring in reality, and then consider what they all shared, despite differences in their details. What the authors must have discovered was that they themselves were embroiled in a situation that could serve as one of their own examples at that very moment. If this all seems very, well, *abstract*, then perhaps we should seek more definitions to aid our understanding.

In his book *Everything and More: A Compact History of* ∞, David Foster Wallace characterizes three different ways we use "abstract" as an adjective [75]:

1. Drawn away from particular examples
2. Arcane, abstruse
3. Compartmentalized

One could characterize these definitions as metaphysical, normative, and structural, respectively. In the metaphysical sense, arithmetic is an abstract model of the practical task of counting collections of individual entities. In the normative sense, Foucault was an abstract writer. In the structural sense, to crib an example from Wallace, one's knowledge that our genes drive us to love our children is abstract relative to the tangible emotions one feels about one's children. The normative definition is not useful for our discussion here, although it should become ironic after we elaborate all the ways abstraction *increases* understanding. We will also set aside definition three; it is an interesting and less common usage, but in the technical vocabulary we will soon advocate, definition three relates more to the term "information hiding" than "abstraction." Let us follow the suggestion of definition one, and consider "abstraction"

to be the process of moving away from concrete particulars and thinking holistically relative to some detailed view. We will continue to refine this notion by analyzing the term "abstraction" as it is used in the world of science.

§1.2.2 "Abstraction" in Science

As was alluded to earlier, there's an interesting contrast between Bloom's top-down modularized view of the brain and an engineer's bottom-up design of a bridge. The decomposition of the brain's structure into smaller and smaller units that explain larger-scale phenomenon is an after-the-fact (*a posteriori*) abstraction (this characterization is due to [77]). Neurosurgeons did not invent axons and dendrites in an effort to cause electrical signals to be transmitted across the brain, but a civil engineer (or, to foreshadow, a computer programmer) *does* invent new assemblages of units that can be thought of more simply as black-boxed wholes. This is *a priori* abstraction.

The natural sciences are mostly concerned with *a posteriori* abstractions. Physicists, chemists, and biologists, for example, seek the best explanations for observed phenomenon, where simplicity and predictive power are the main criteria of comparison. Building a hierarchy of such abstractions is useful for facilitating explanation and prediction at multiple levels of detail. Any formal relationships between levels are valued for their ability to establish causality, or at least a model for how to think about causal connections in the system [77].

Applied sciences, like linguistics, nuclear technology, or civil engineering, are more focused on *a priori* abstractions. Practitioners in these fields build new systems to solve new problems, and they manage the intellectual complexity of their projects by omitting details whenever it is safe to do so. Of course these abstractions are still best

structured in a hierarchy for purposes of flexibility and robust analysis. Relationships that can be established between levels in the hierarchy serve as justification of the system's suitability. Going back to the bridge example, its designer may do a statics calculation to argue that some assemblage of trusses can be thought of as a black-box component capable of bearing some specific load.

Our observable reality lives somewhere in the middle of this milieu: we perform *a posteriori* abstractions to understand how the elements of the physical world interact and affect us, and we use that knowledge to construct *a priori* abstractions that serve our purposes. It should be clear that in either direction, abstraction is an indispensable aid to human understanding.

In this survey of abstraction's role in science, mathematics is an interesting outlier. Throughout the history of mathematics, new abstractions have often been heavy-weight theoretical contributions that *complicate* rather than simplify matters. When one wishes to generalize a theorem that is obvious in some small domain, the theorem may become false, or at a bare minimum its proof substantially more difficult. Non-Euclidian geometries provide a persuasive example of the difficulty of abstraction in math: it took over two thousand years for geometers to draw away from the particulars of the parallel postulate and create a more general theory that black-boxed the intersection properties of parallel lines.

Computer science is in essence a fusion of these two worlds—it is a discipline in which pragmatic issues of real-world technology and theoretical insights similar to those of math and logic synergize in a tangled mutual feedback loop. Thus it may not be surprising that the meaning of "abstraction" in this field has become muddled. Indeed, some computer scientists view abstraction as a complication, only to

be countenanced out of a duty to robustness and generality. We disagree, and seek to demonstrate that abstraction is *the key* to computer science, not just aesthetically or theoretically, but in the real study of solving problems with computers, *i.e.*, software engineering. More specifically, we'll consider abstraction's role in software *verification*—the practice of proving code correct.

§1.2.3 "Abstraction" in Computing

In writing a computer program, the laws of reality don't constrain the possible approaches to a problem, at least not with the same severity that they do in bridge building, circuit design, or hydraulic systems. This freedom makes programming an arguably more creative task, but also one fraught with more opportunities for mistakes at all degrees of insidiousness. Programming has spawned a wide variety of research into best practices, design patterns, workflow processes, and ergonomics. As one should anticipate by this point, these conceptual aids make use of abstraction. In fact, abstraction is so pivotal to the tractability of computer science, that research in computing has helped to forge the beginnings of a true ontology for the concept of abstraction itself.

This ontology begins with the idea that the information processed by a computer can be abstracted over along different dimensions. For example, one prevalent coding strategy is to organize programs into units called "procedures," each of which performs one specific task. With this design, tasks that need to be performed multiple times can be coded once but executed repeatedly via function calls. Unsurprisingly, this is known as "procedural abstraction." The advantage, aside from having a single point

of control over changes in any particular task, is that once the procedures are written their algorithmic details can be forgotten about: they just do what they do, somehow.

Given its obvious utility, both for managing intellectual complexity and simplifying code maintenance, procedural abstraction is now ubiquitous and taken for granted; a different dimension of information organization has become the focus of study. "Data abstraction" is the process of uniting primitive pieces of data such as integers and characters into more intricately structured wholes. The real power of data abstraction lies in the fact that the person performing the abstraction decides how the new amalgamation of data can be manipulated, and can choose to restrict end-users of the abstraction to only those operations that make sense for the whole.

One simple example is a "rational number" data type represented by two integers, which might helpfully be named `num` and `denom`. The user of a rational number might need the ability to increment its value by one, but simply adding one to either (or both) of its so-called "data members" will not get the job done. `num` and `denom` should only be changed in ways that make sense relative to the conceptual meaning we're ascribing to them—to effect a real increment, the denominator should be added to the numerator. Here's a procedure that will do the trick, written in a programming language called "Resolve," which will be discussed later:

```
1 procedure Increment (updates r: Rational)
2     Add(r.num, r.denom)
3 end Increment
```

Formalizing the connection between the data that "actually exist"[1] and our thoughts about what the data "really mean" is a key topic in the theory of programming languages with data abstraction—a topic we will return to later in this work.

We've seen now the essence of what's come to be known as "Object-Oriented Programming," wherein a program is organized primarily by identifying the structured data that interact in a proposed solution to a problem. Object-Orientation says, roughly, that each "noun" in the description of a system (for example, a rational number) should be the target of a data abstraction. These data abstractions define families of objects called "datatypes," and objects of conceptually related datatypes can beneficially interact and collaborate through a language construct known as "inheritance."[2]

One programming role in an OO/component-based system is the identification, design, and implementation of useful data abstractions and their inheritance relationships. A separate role is the utilization of this infrastructure to achieve the desired end result. These roles are known, respectively, as "implementer" and "client." This will all be unpacked in more detail, but the important points for now are that component-based software is the most popular programming methodology in current use, that its fundamental task consists of performing data abstraction, and that good data abstractions enable clearly delineated and simplified programming roles. Thus, a

[1]Remember: no data *actually* exist if we consider the *a priori* abstractions that computer engineers make when they build a machine, *e.g.*, integers are made up of bytes which are made up of bits, which are realized in hardware and modeled by electrical or magnetic phenomena, etc.

[2]Object-Oriented Programming is a wide and deep area of study. Many differing definitions of its essence have been proffered, and experts would surely dispute the details of any particular description, including the one given here. We wish only to establish some connection with this commonly used term, and for the remainder of this work will adopt the less loaded and more descriptive term "component-based software." Inessential complexities of object-orientation, such as the issue of reference types vs. value types, are not the focus of this work.

deep understanding of data abstraction lies at the foundations of software engineering research and practice.

In common data abstraction terminology, "encapsulation" means the ability to protect data from outside manipulation, and is often provided by built-in language mechanisms, e.g., the `private` keyword of C++ and Java. The disciplined use of encapsulation to completely shield data members from the view of clients is known as "information hiding," and is usually considered necessary to industrial-strength component-based programming. Encapsulation and information hiding are sometimes posited as synonyms of abstraction, but really they are specialized technical terms for certain practical characteristics of some data abstractions. "Abstraction," even if we restrict ourselves to the domain of computing, is a broader concept—more abstract, one could say.

Regardless of the different dimensions along which abstraction can be performed in computing, and regardless of the finer-grained characterizations of abstractions in component-based software, we conclude by distinguishing two "flavors" of data abstraction that computing employs extensively: generalization and reconceptualization.

§1.2.3.1 Generalization

Consider again the quote that began this chapter. Bloom's decomposition of the brain, and his assertion that "this is how computers work" might lead one to concoct a paragraph like this:

> Like any operating system, the Linux kernel consists of large parts (such as the scheduler and the memory manager) that are made up of small parts (such as "map" data structures), which themselves are made up of smaller parts, until you get to machine instructions . . . whose orchestrated

> firing is the stuff of computation. The machine instructions are made up of parts like instruction codes and addresses, which are made up of smaller parts like bits, which are made up of voltage differences, and so on.

Actually, there's a more *abstract* paragraph we could write:

> Like any $[x_1]$, the $[x_2]$ consists of large parts (such as the $[x_3]$ and the $[x_4]$) that are made up of small parts (such as $[x_5]$), which themselves are made up of smaller parts, until you get to $[x_6]$ ... whose orchestrated firing is the stuff of $[x_7]$. The $[x_6]$ are made up of parts like $[x_8]$ and $[x_9]$, which are made up of smaller parts like $[x_{10}]$, which are made up of $[x_{11}]$, and so on.

Let's call this last form a "paragraph template." In order to create a paragraph that describes the top-down view of some system, we can simply replace our x variables with the appropriate terms. Some choices will of course yield nonsense—let x_1 be "prime number," let x_2 be "sauerkraut," let x_3 be "vacuum of space," and so on. We could imagine stating some restrictions on the terms we choose for the 'x'es: that they should all be noun phrases seems obvious, but moreover, they should exhibit certain relationships among each other, which cause the resulting paragraph to have identifiable meaning. As an example, one requirement is that 'x_{11}'s should be in some sense small relative to 'x_6'es. There are certainly several other such conditions.

Bloom has given us, implicitly, a schematic for generating a rather elegant English paragraph describing the decomposition of any system that has seven levels of abstraction (I counted seven). All we had to do to obtain this schematic is "draw away from the particular examples" of the brain. It's clear, then, that our template is "abstract" in the metaphysical sense of §1.2.1. We will consider situations such as the foregoing, in which concrete details can be replaced by "stand-ins" to create a reusable template, to be the first flavor of abstraction. We'll describe these abstractions with the term "generalization."

Generalization is employed extensively throughout the practice of software engineering. A language mechanism known as "templating" in C++ and "generic programming" in Java is the most direct example. A template/generic is a schematic for generating data abstractions, wherein some details are left open for clients to fix as they see fit. A client's declaration of a template, in which they decide how to "fill in the blanks" that the template has left open, is known as an "instantiation."

In the most common case, templates are used for "container classes"—datatypes that are meant to hold other pieces of data and process them in some particular manner. Templating is used for containers so that they can be written once and then instantiated by clients to hold different varieties of data. For instance, a queue is a type of container that processes its elements in a "first in, first out" manner, and is usually written as a template for maximum generality and thus reusability. Below, we show the client-view ("contract") of a queue in Resolve. The details will be discussed in the next chapter, but for now the reader should note that in the first line the contract is parameterized by a type, which is given the name "Item" so that it can be referred to throughout the rest of the contract. This parameterization is the essence of generalization in templating/generic programming: it means that the queue will behave the same way for any type which is provided as this "Item" parameter.

```
1 contract QueueTemplate (type Item)
2
3     uses UnboundedIntegerFacility
4
5     math subtype QUEUE_MODEL is string of Item
6
7     type Queue is modeled by QUEUE_MODEL
8         exemplar q
9         initialization ensures
```

```
10              q = empty_string
11
12      procedure Enqueue (updates q: Queue, clears x: Item)
13          ensures
14              q = #q * <#x>
15
16      procedure Dequeue (updates q: Queue, replaces x: Item)
17          requires
18              q /= empty_string
19          ensures
20              #q = <x> * q
21
22      function Length (restores q: Queue): Integer
23          ensures
24              Length = |q|
25
26      function IsEmpty (restores q: Queue): control
27          ensures
28              IsEmpty = (q = empty_string)
29
30  end QueueTemplate
```

§1.2.3.2 Reconceptualization

The upshot of generalization in software engineering is that all the different particular instances of a generalized component are thought of the same way; the actual values ascribed to the parameters do not impact the way that the generic behaves. When one thinks about how to compute with a Queue, the question of what the "Item" actually is doesn't matter. In other words: if we had *not* generalized, and had written a Queue that was only allowed to contain, say, integers, we would reason about it exactly the same way: as a mathematical string. Lines 5 and 7 above make this explicit. As a preview of things to come, we remark here that in fact *all* contracts

in Resolve provide a mathematical model for how to reason about a component without revealing its implementation details. Resolve contracts are abstractions; most interesting contracts involve both data- and procedural abstraction.

Naturally, we are left to consider the implementer's view of such data abstractions. As was discussed in section §1.2.2, a relationship between the actual data members that make the Queue work and the client's model of how to reason about Queue behavior is necessary if we are to convince ourselves that an implementation does what it should.

Below we see a "realization" (the Resolve term denoting a component implementer's view) of the Queue template. It uses another template called "List" to get its job done, so of course a client's view of that contract would be necessary to really understand how this realization works.[3]

```
1  realization ListRealization implements QueueTemplate
2
3      uses ListTemplate
4      uses IsPositive for UnboundedIntegerFacility
5
6      facility ListFacility is ListTemplate (Item)
7
8      type representation for Queue is (
9              itemlist: List
10         )
11         exemplar q
12         convention
13             q.itemlist.left = empty_string
14         correspondence function
15             q.itemlist.right
16     end Queue
17
```

[3]This is an important point that will be emphasized later. Note for now that it's only the *client's view* of List (and of "UnboundedIntegerFacility" with its extension called "IsPositive") that is necessary here.

```
18      procedure Enqueue (updates q: Queue, clears x: Item)
19          AdvanceToEnd (q.itemlist)
20          Insert (q.itemlist, x)
21          Reset (q.itemlist)
22      end Enqueue
23
24      procedure Dequeue (updates q: Queue, replaces x: Item)
25          Remove (q.itemlist, x)
26      end Dequeue
27
28      function Length (restores q: Queue): Integer
29          Length := RightLength (q.itemlist)
30      end Length
31
32      function IsEmpty (restores q: Queue): control
33          variable len: Integer
34          len := RightLength (q.itemlist)
35          IsEmpty := not IsPositive (len)
36      end IsEmpty
37
38  end ListRealization
```

The important part of this realization for our present discussion is the *correspondence function*. Also known as an "abstraction function," a correspondence function is a rigorous mathematical statement of how the data members used to realize a contract can be reconceptualized as a value of the component's mathematical model.[4]

Most languages do not feature special syntactic slots for correspondence functions, and yet software engineers still must engage in the mental process of reconceptualization. Even when working in languages that are cavalier about the rigor with which such abstractions are expressed, reconceptualizations are necessary in order to work effectively in most large systems. Lisp, for example, is a revered programming language

[4]In fact, allowing abstraction *relations* is known to be useful [68]. We discuss this in detail in Chapter 6.

that allows values of its fundamental data structure, the s-expression, to be reconceptualized as sets by building in operations such as `union` and `intersection` [29]. There is no formal correspondence function—indeed sets and s-expressions really have the same type in Lisp—but the language still facilitates some degree of reconceptualization to simplify the programmer's thinking.

§1.3 Software Verification

Verification is perhaps the loftiest goal of software engineering. Its most complete hypothetical form, the so-called "verifying compiler" has been promoted as a "grand challenge" for computer science, similar in scope to putting a man on the moon or mapping the human genome [37].

What "verification" really means for software is the ability to prove mathematically that a program (or some other unit of code) does exactly what it's supposed to do. This of course requires a precise mathematical statement of the code's intent—known as the *formal specification*—and a rigorous mathematical view of exactly what any given piece of code means. Some programming language definitions provide a *formal semantics*, which defines the mathematical meaning of any legal piece of code in the language. These languages, of which Resolve is one, are obviously our best hope for achieving verification.

We can think of the specification of some desired behavior and the semantics of code purporting to implement it each as a relation—a set of input/output pairs. The specification states which outputs are legal for which inputs, and the semantics states which outputs the code may actually give for any given input. Verification means being able to prove, ideally automatically, whether or not the semantics really is a

subset of the specification. In full generality, verification is known to be impossible due to the undecidability of the mathematics normally used in the specifications. In practice, however, verification of real programs appears tractable, as it is almost always more a matter of detailed bookkeeping and disciplined annotation than of proving deep mathematical theorems [42].

The aforementioned verifying compiler is a piece of software that can turn source code written in a high-level programming language into an executable file, but only after first verifying the source code relative to its specification. Such a tool would mean the end of testing, debugging, and patching in the software engineering process, at least with respect to problems inside the code (verification cannot ensure that somebody won't unplug the computer, for example). The verifying compiler is a grand challenge because, among other reasons, its scope is so broad. It must be able to prove (or refute) the claimed correctness of *any* program in its accepted language, and ideally to do so with no human guidance or interaction. Many existing approaches to verification can prove certain limited properties (*e.g.*, that no arrays are accessed outside of their bounds), or can prove programs relative to some safety assumptions (*e.g.*, that the code doesn't create any aliases), or make conservative approximations of what the code will do when run and thus leave open the possibility of false negatives, but a full verifying compiler remains a lofty research goal.

§1.4 Organization

Armed with a clearer view of what "abstraction" means technically, and how related terms like "information hiding" and "generalization" differ subtly, we are prepared to explore abstraction's role in software engineering, specifically verification.

This work is focused on the ways in which abstraction, specifically reconceptualization, is necessary in all plausible attempts at a verifying compiler.

In Chapter 2, we discuss the role of mathematics in quality software engineering abstractions. We formulate an argument for "strict abstraction," and contrast this approach with related work. We present new examples in a specific language for strict abstraction, demonstrating its generality and reusability. We also discuss the role that interactive proof assistants can play in the development of mathematical theories to enable strict abstraction. We contribute a new development in the Coq proof assistant, defining and proving theorems for a large portion of mathematical string theory.

Chapter 3 presents a verification strategy for client code written in a programming language that employs strict abstraction. We define two different views of programming language semantics, and contrast them in terms of client reasoning and verifiability. We also discuss our participation in a competition among different software verification research groups, comparing our approaches to others and identifying issues involving insufficient abstraction.

In Chapter 4, we move from the client view to the implementer's view, and explain how strict abstraction with some new annotation constructs can facilitate proofs of correctness for data representation. After discussing the method in the framework of procedural programming, we shift to a functional language, and present one of our key theoretical results: an unsoundness for referential transparency in expressions involving abstract data types verified by a widely accepted proof rule. We propose a new rule and a new workflow in the verified software process to solve the problem.

Chapter 5 discusses tool support for software verification, specifically the Resolve verification tool chain at OSU. After identifying another abstraction issue involving the degree to which push-button proof automation can properly be expected in software verification, we discuss our work on SplitDecision, a custom-built prover that implements a new decision procedure for a fragment of mathematical string theory. Our approach is contrasted with important related work, particularly ACL2 and Coq/Ynot.

Finally, in Chapter 6 we argue that our emphasis on abstraction opens new doors for future verification technologies. We explore the feasibility of automating proofs of data representations that use abstraction relations, discuss the potential of incorporating specifications into memoizing interpreters for functional languages, and also suggest some syntactic constructs for iteration that can ease the annotation burden and increase the reusability of iterative code.

Chapter 2: Achieving Strict Abstraction with Mathematical Modeling

§2.1 The Resolve Approach

Now that we know what abstraction is, we should consider the technical details of how it is utilized in software engineering. How do we express abstractions in computer programs? If expressions in a programming language are the concrete particulars that we want to draw away from, in what language do we talk about them? Once we've decided on a "target language" for conveying our abstractions, can it be used to facilitate tasks other than verification?

Answering these questions in effect establishes a foundation for the study and practice of software engineering. The particular answers that we will commit to are those of an approach known as "Resolve" (sometimes "RESOLVE"). In this chapter, we will present a "specification language," which Resolve uses for expressing data abstractions and intended program behavior in terms of pure mathematics. We'll explain why this choice of language is advantageous, providing a historical perspective on specification techniques and contrasting Resolve to other popular methodologies. Finally, we will demonstrate the broader utility of Resolve's specification language in

other domains of computing, and discuss how existing tools can support the development of new mathematical abstractions in the specification language.

§2.1.1 Choosing a Specification Language

Recall the definition of software verification given in the previous chapter: proving mathematically that a code unit does exactly what it's supposed to do. This definition precipitates an important insight: verification requires a rigorous statement of the code's intended behavior. Such statements are commonly called "formal specifications." We will focus on formal statements of the code's input/output behavior, not orthogonal characteristics such as average-case running time or memory consumption. In other words, we're interested in proving program correctness in the sense of *functionality*, not non-functional properties like performance. Henceforth we will abbreviate "formal behavioral specification" as just "specification" or "spec."

§2.1.1.1 A Selected History of Specification Methodologies

The idea of software verification dates back to the 1960s [25, 55, 35], but there is a marked lack of theoretical cohesion in the field, largely due to the fact that the approaches to specification employed have varied wildly. Bertrand Meyer is widely considered the progenitor of "programming by contract"—separating code from specifications—in his object-oriented language Eiffel [56]. Eiffel specifications are executable, however, so they can be contrasted with the purely mathematical approaches inspired by Hoare, e.g., the observation that (higher-order) predicate calculus suffices to specify even complicated constructs such as first-class procedures [22].

On the whole, formalisms for program specification can be seen as lying on a continuum. At one end there are fully programmatic approaches to verification, wherein

specs simply are code. ACL2 [40], which will be discussed in some detail shortly, is a popular example of this approach, as is Microsoft's Spec# [6]. At the other end of the spectrum are purely mathematical specification languages such as Resolve [20]. Of course there are many approaches that combine both viewpoints—JML specifications [62] are written in an augmented subset of the programming language that is executable under certain conditions, and Dafny specifications [49] are programming expressions in an intermediate-level language that are restricted to a small domain of quasi-mathematical programming types such as sets and sequences.

§2.1.1.2 A Case Study in Three Modern Specification Languages

For the sake of parsimony if nothing else, one might initially decide that specifications should be programmatic. As was mentioned above, some serious verification projects do employ this strategy. A key design decision of the Resolve project places it at the other end of the spectrum: Resolve practices "strict abstraction," *i.e.*, reconceptualization in terms of purely mathematical modeling.[5] Strict abstraction consists of treating all pieces of data as mathematical values such as real numbers or finite sets (rather than, say, as contiguous chunks of bits in memory, or as references to said chunks), and using mathematical logic to express, in a declarative fashion, intended code behavior in terms of these values. On this view, the specification of a piece of code is just a mathematical relation, perhaps not even a computable one, between incoming and outgoing values.

One advantage of mathematical modeling is that it results in highly reusable specifications. A Resolve-style spec describes behavior in a programming-language

[5] "Reconceptualization" is in some sense a misnomer. To a client, the mathematical model of a component is the *only* conceptualization possible.

agnostic fashion, thus easing the dreaded "annotation burden" that commonly hinders software verification in practice. The key idea here is that specifying intended program behavior is itself a serious intellectual task, and so one would like to be able to specify behavior "once and for all" with no regard to the concrete details of any particular programming language. It should be clear from the previous chapter that a reusable spec language is itself an abstraction. As a case-study in specification reusability, we chose a common programming (and verification) task—sorting a collection of numbers—and found real examples of how different specification languages formally express this behavior.

Below we see a `sort` specification (from [18]) that uses JML [62], an augmented subset of Java often used in verification projects relating to Java such as ESC4 [38] or KeY [7]. Note that `numbers` is an array of integers that is in scope for this specification, and `length` is an accessible attribute of it.

```
1  /*@ public normal_behavior
2    @ requires numbers != null && numbers.length > 0;
3    @ ensures \old(numbers.length) == numbers.length;
4    @ ensures (\forall int i; 0 <= i &&
5                              i < \old(numbers.length);
6                (\exists int j; 0 <= j &&
7                                j < numbers.length;
8                  numbers[j] == \old(numbers[i])));
9    @ ensures (\forall int i; 0 <= i &&
10                             i < numbers.length-1;
11               numbers[i]<=numbers[i+1]);
12   @*/
```

Firstly, this specification demonstrates how it can sometimes be difficult to properly formalize correctness properties that might seem obvious and/or intuitive in natural language. The JML above actually fails to properly express sorting—the

desired property that the sorted array be a permutation of the incoming argument is incorrectly specified. To see this, consider that if `\old(numbers)` is $[2, 1, 2]$ and `numbers` is $[1, 1, 2]$, the spec is satisfied and yet the latter is not the sorted version of the former; `numbers` is not even a permutation of `\old(numbers)` in this case.

Aside from the presumably repairable correctness issues, we point out that this JML spec is inexorably wedded to programming language particulars. Note that the first meaningful token of the spec is `public`—a Java keyword concerning access control that has no meaning in many other programming languages. Note also the mention of `null`, another keyword that concerns a detail of Java's programming model, namely the fact that there is a special value shared by all Java reference types. Furthermore we emphasize that although operators such as `&&` and `==` may look mathematical, they actually denote executable functions built into the programming language.[6] We grant that to a JML partisan, these are all advantageous facts because they mean that this specification can be executed as a run-time check,[7] but for our purposes here they are a hinderance to abstraction/reusability: this spec does not express the notion of sorting in any language that differs substantively from Java.

The canonical specification of sorting in ACL2, a verification suite for functional programming, is also not reusable. Its formulation, taken directly from the current (as of March 2012) version of the ACL2 repository [39], is shown below.

```
1 (defthm orderedp-isort
2         (orderedp (isort x)))
```

[6]Indeed, their meanings do not even match those of the corresponding mathematical operations: `&&` is short-circuiting, which is consistent with but not identical to logical conjunction, and worse, `==` tests equality of reference, not of value, in some cases.

[7]This depends upon all the quantifications being bounded, by the way, and also the ability to make deep copies of any parameters whose `\old` values are mentioned.

```
3
4 (defthm true-listp-isort
5   (true-listp (isort x)))
6
7 (defthm how-many-isort
8   (equal (how-many e (isort x))
9          (how-many e x)))
```

Essentially, the specification states that the `isort` function preserves the "listness" of its argument (ACL2 is an untyped programming language) and returns a sorted permutation of that argument. This seems like the right idea, but again it is too concrete-bound. If nothing else, the heavily parenthesized prefix notion used in ACL2 suggests that this spec will not be well-formed in less exotic languages. Furthermore, just as in the JML example, this specification depends upon other programmatic functions, e.g., `how-many` and `true-listp`, and has no well-defined meaning in their absence. Indeed, the ACL2 verification philosophy only allows code to be proven equivalent to *other* code, rather than correct relative to some qualitatively different description of intended behavior.

Of course we should not be surprised to find that specifications of interesting behavior depend upon other definitions, but the key insight here is that reusability demands these dependencies *not* involve programming operations. Below, we show a Resolve specification for sorting that is fully reusable. This specification has been successfully employed in a Resolve verification case study [44].

```
1 procedure Sort (updates q: Queue)
2    ensures
3       IS_PERMUTATION (q, #q) and
4       IS_NONDECREASING (q)
```

`IS_PERMUTATION` and `IS_NONDECREASING` are defined below. Crucially, these are *mathematical* definitions. They concern strings of integers—a mathematical type—and thus are not committed to any one programming language, nor even any one programming type. The definitions are well-formed and meaningful for any programming type mathematically modeled as a string of integers (likely candidates include Queues, Stacks, Sequences, and Lists). This dimension of abstraction is absent in the JML and ACL2 examples, which were specific to arrays and lists of integers, respectively. Of course, the reusability of Resolve's `Sort` spec could be further increased by using templates to eliminate the commitment to strings of *integers*; this additional layer of abstraction was omitted only for simplicity of presentation.

```
1  definition OCCURS_COUNT (
2        s: string of integer,
3        i: integer
4     ) : integer satisfies
5     if s = empty_string
6     then OCCURS_COUNT (s, i) = 0
7     else
8        there exists x: integer, r: string of integer
9           ((s = <x> * r)  and
10           (if x = i
11            then OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i) + 1
12            else OCCURS_COUNT (s, i) = OCCURS_COUNT (r, i)))
13
14 definition IS_PERMUTATION (
15       s1: string of integer,
16       s2: string of integer
17    ) : boolean is
18    for all i: integer
19       (OCCURS_COUNT (s1, i) = OCCURS_COUNT (s2, i))
20
21 definition IS_PRECEDING (
22       s1: string of integer,
23       s2: string of integer
```

```
24      ) : boolean is
25      for all i, j: integer
26          where (OCCURS_COUNT (s1, i) > 0 and
27                 OCCURS_COUNT (s2, j) > 0)
28          (i <= j)
29
30  definition IS_NONDECREASING (
31          s: string of integer
32      ) : boolean is
33      for all a, b: string of integer
34          where (s = a * b)
35          (IS_PRECEDING (a, b))
```

§2.1.2 The Resolve Specification Language

A detailed description of the Resolve specification language is laid out in [34], but a brief overview is in order here. We will use the `Sort` specification and its related math definitions given above as driving examples, because they illustrate most of the important issues for our purposes in this work.

The process of specifying intended code behavior is best thought of in two phases:

(1) establish a context of relevant purely mathematical definitions, and then

(2) use special syntactic slots in the language to link these mathematical notions to the relevant aspects of the code.

This division of labor is itself an advantageous abstraction: a mathematician can do productive work in the first phase without needing to consider the task of computer programming at all.

§2.1.2.1 Math Types and Subtypes

To facilitate mathematical modeling, a small, highly expressive selection of mathematical theories are considered "built-in" to the spec language, and serve as the

rudiments for modeling programmatic data. New "math subtypes" can be created by combining and/or placing constraints on existing ones.

The important built-in theories in the foregoing examples are `string` and `integer`. `string`s are ordered[8] arrangements of values of some particular mathematical type (`integer` in this case). They are defined inductively—a `string` is either empty or is the extension of some `string` by an additional element—and they can be "recursively instantiated" (e.g., we can have a `string of string of integer`). Other important built-in theories not used in the `Sort` example include tuples (ordered fixed-length collections with named fields), booleans, sets (unordered collections with no duplicates), multisets (unordered collections that allow duplicates), binary trees, functions, real numbers, etc. The collection of math theories is extensible, but we predict extensions will rarely be required because combining these rudiments in smart ways and/or using constraints usually suffices. For example, the natural numbers in Resolve are a subtype defined by constraining the values of the `integer` math type to be non-negative.

§2.1.2.2 Math Definitions

In addition to constructing math subtypes to be used for modeling purposes, phase one of the specification workflow involves defining relations between mathematical values. These are labeled with the keyword `definition`; four were shown earlier in connection with `Sort`. Each definition has a name, a list of parameters, a return type, and a body. A definition's body can be an expression that explicitly

[8]The word "ordered" is somewhat overloaded because of the ACL2 example in the previous §. Here, "ordered" means that the order of entries in a string is part of that string's identity. For example, the string inequality $\langle 1, 2\rangle \neq \langle 2, 1\rangle$ is true, whereas it would be false if these had been sets instead, *i.e.*, $\{1, 2\} = \{2, 1\}$. In the ACL2 example, `orderedp` was an executable function that tested whether the elements in a list were arranged in nondecreasing *value* order.

defines the definition's value (usually in terms of the parameters), or it can be a predicate that is true for any element of the relation. The former ("explicit definitions") use the keyword `is`, whereas the latter ("implicit definitions") are distinguished with the keyword `satisfies`. Note that in the body of an implicit definition, the return value is denoted by a parameterized invocation of the definition's name, e.g., `OCCURS_COUNT(s, i)` in the body of `OCCURS_COUNT`. As we see in the examples, the bodies of definitions are just statements of predicate calculus—all first-order in this example, although higher-order formulations are also allowed.

§2.1.2.3 Contracts

For phase two—tying the pure math to selected programmatic entities—the spec language uses special syntactic slots. In other words, a well-formed Resolve program has specific locations at which mathematical formalizations of behavior must be placed for verification to be possible. The most common of these are the `requires` and `ensures` annotations that are used to specify operations.[9] The `requires` clause defines what must be true of the incoming parameter values for any legal call to the operation, and the `ensures` clause formally expresses the results of the call, often by relating incoming and outgoing parameter values. The special token `#` is prefixed to a parameter name in an `ensures` clause to denote that parameter's incoming value. It should be obvious that `#` is never needed to express `requires` clauses.[10] We refer to an operation's requires and ensures clauses collectively as its "contract."

[9] A note on Resolve terminology: "procedure" refers to a programming operation that does not return a value but instead can change some or all of its parameters. These are sometimes called "void functions" in other languages. "Function" refers to a programming operation that leaves its parameters unchanged but returns a value. An "operation" is any procedure or function. No Resolve operations return a value *and* alter a parameter.

[10] We could say that *every* mention of a parameter in a `requires` clause has an implicit `#` that we omit just for brevity's sake.

§2.1.2.4 Parameter Modes

The header of a programming operation has other syntactic slots for specifications in addition to its contract: each parameter is annotated with a *mode*. Each Resolve parameter mode is denoted with a keyword that attempts to capture its meaning in English: `restores`, `clears`, `replaces`, and `updates`. The first two of these are just abbreviations of conjuncts that are often desired in `ensures` clauses—if a parameter `x` is in `restores` mode, this means the same as conjoining `x = #x` to the operation's `ensures` clause, and `clears` means the same as conjoining `is_initial(x)`, where `is_initial(x)` is just a reserved way of denoting the `initialization ensures` predicate. We saw an `initialization ensures` annotation in the contract for `Queue` in Chapter 1; it states a condition which constrains the initial value that objects of a programmatic type (like `Queue`) take on when they are declared (or cleared, as we've just seen). The `updates` and `replaces` modes indicate that the outgoing value of the parameter can be different from the incoming value—in `updates` mode the new value may depend on the old value, and in `replaces` it may not. `clears` of course now appears to be a special case of `replaces`, but it is more specific and is used often enough to merit special treatment.

§2.1.2.5 Other Slots

Resolve has another syntactic slot, annotated as "`is modeled by`", for declaring that some specific programming type is modeled by some mathematical one. For example, `Queue is modeled by string of Item`, where `Item` is the mathematical model of the `Queue`'s template parameter. There are more syntactic slots—the careful reader will remember at least one of them from our discussion of `Queue`'s `List`

realization—but we'll reserve their explanations until they're needed. At this point, the Resolve specifications encountered thus far in this chapter should be intelligible.

§2.2 The Wider Applicability of Our Specification Language

We remark for emphasis that the two-phase nature of the Resolve specification process achieves *strict abstraction*, in the sense that the specs are statements of ordinary predicate calculus that are programming language independent. Specs are only married to programmatic constructs by a small number of well-defined syntactic slots, such as operation contracts. To demonstrate the utility of strict abstraction, we will now present two examples of how our specification language can be a useful formalism in domains not related to verification, nor even to Resolve programs.

§2.2.1 Modeling Multithreaded Computing

In [72], we brought Resolve-style specification to bear on the problem of modeling multithreaded computation. Specifically, we wished to define a purely mathematical model of basic Java-style concurrency [48], which features reference semantics, and the ability to lock access to objects via synchronized methods. For generality, we also modeled locks that aren't specific to an object, e.g., to provide mutual exclusion on a critical section. Below we list the definitions we created to formalize this model.

```
1 math subtype HEAP_STATE is
2     partial function of (objid: OBJECT_ID, val: OBJECT_VALUE)
3
4 math subtype CONTROL_FLOW_STATE is
5     string of (m: METHOD_SIGNATURE, objid: OBJECT_ID)
6
7 math subtype THREAD is (thrid: THREAD_ID,
8                          active: boolean,
```

```
9                                   cfs: CONTROL_FLOW_STATE)
10
11 math subtype SYNCH_LOCK is (
12        objid: OBJECT_ID, locked: boolean,
13        holder: THREAD, waitersForSynch: finite set of THREAD,
14        waitersForUnsynch: finite set of THREAD)
15    exemplar l
16    constraint
17        if l.locked then
18           l.holder.active and l.holder.cfs /= empty_string and
19           IS_SYNCHNRONIZED(head(l.holder.cfs).m) and
20           for all w: THREAD
21              where (w is in l.waitersForSynch)
22              (not w.active and
23               IS_SYNCHRONIZED(head(w.cfs).m) and
24               head(w.cfs).objid = head(l.holder.cfs).objid)
25
26 math subtype OTHER_LOCK is (locked: boolean, holder: THREAD,
27                              waiters: finite set of THREAD)
28    exemplar l
29    constraint
30        if l.locked then
31           for all w: THREAD where (w is in l.waiters)
32              (not w.active)
33
34 math subtype THREADED_SYSTEM is (
35        heap: HEAP_STATE, threads: finite set of THREAD,
36        synchlocks: finite set of SYNCH_LOCK,
37        otherlocks: finite set of OTHER_LOCK)
```

This model is not a complete characterization of typical Java concurrency—for example, it does not state that *every* thread whose control flow is at a synchronized method on some already-locked object must be inactive and in the lock's waiting set, nor does it characterize `waitersForUnsynch` at all—but it was sufficient for the purpose of [72], which was to provide a well-defined domain in which we could discuss the definition and implementation of a new lock written in the style of aspect-oriented

programming. This idea of underspecification will become important in a different setting in Chapter 4.

§2.2.2 Specifying Functional Constructs

A programming language is considered "functional" if it deemphasizes the idea of a "state"—a mapping of variable names to values that persists and mutates as the program executes. A functional program is not an arrangement of commands that execute one after another, but rather a single expression that is evaluated by evaluating subexpressions and combining results. In such a language, all procedural abstractions are "functions" in the sense defined in footnote 9. This may seem like a straightforward restriction of standard "imperative" programming, but the history of computing is rife with contributions spawned from this more minimal, unadorned approach.

Functional programming's role as an engine of innovation is partly due to seemingly inconsequential syntactic characteristics. Most notably, the quintessential functional language, Lisp [29], uses a parenthesized prefix notation, which allows code itself to be treated as a simply structured piece of programmatic data.[11] Thus, powerful advanced programming techniques like macros and reflection are natural in Lisp. We glimpsed Lisp syntax earlier, in the ACL2 specification of `sort`—ACL2 is a dialect of Lisp that makes a good candidate for verification because it is "purely functional." In fact, "ACL" stands for "Applicative Common Lisp," and "applicative" is a synonym for the usage of "pure" that we are about to elaborate.

[11]The name of the primitive data type that Lisp code is a member of is "list," and the etymology of the name "Lisp" is "List Processing Language." We will formalize the meaning of "list" shortly.

Most industrial-strength functional programming languages are not "pure," meaning that some of their constructs alter an argument or have some other effect which causes a "state" to persist across the execution of multiple programming commands. The most basic example of impurity is assignment to a variable. Purely functional languages like ACL2 disallow such complications, making the translation from code to mathematics much more direct. In some sense this is a boon for verification, and indeed ACL2 has many achievements to boast of. However, the deceptively mathematical nature of pure functional programming obscures the importance of mathematical modeling and leads to reusability issues as discussed in §2.1.1. We seek to remedy this situation by showing that the Resolve specification language is just as useful for formalizing important constructs of functional programming as it has been in imperative programming.

Before demonstrating Resolve's utility for modeling these constructs, we must emphasize that such a demonstration runs quite contrary to functional programming orthodoxy. The oft-claimed advantage of functional programming, particularly the pure form, is that "the code is the spec," meaning that purely functional code is so clean and rigorously understood that it itself serves as a formal explanation of its behavior. This is why, for example, ACL2 users sometimes refer to the programs they write as "models," and why strict abstraction is not (to this author's knowledge) present in any existing effort to verify functional programs.

However, the "code *is* specification" viewpoint implies that attempts to prove functional programs correct do not actually comprise "verification" in the sense that we've defined—in such a setting, there is no qualitatively different description of

intended behavior relative to which verification can be performed. A primary motivation for the work we present in this document is the observation that *all known efforts to verify functional programs just prove code to be functionally equivalent to other code.* If that other code is broken, then code that is proven correct will still fail in practice. We should be concerned about this, because the literature of computer science abounds with discoveries of subtle, detrimental flaws in even straightforward computational settings—see the repeated argument issues of Cook [17] or Bloch's discovery of a fatal flaw in the most common binary search and mergesort algorithms [9].

§2.2.2.1 S-Expressions and Proper Lists

In response to these concerns about unsoundness, we wish to imbue the verification of functional languages with strict abstraction. There are serious theoretical complications here relating to the semantics usually ascribed to purely functional programs. These issues will be addressed in Chapter 4, but for now let's begin by presenting a Resolve definition of a new mathematical type, one isomorphic to the fundamental data structure of all the most popular functional languages: the S-Expression. The aforementioned "lists" of languages like Lisp are in fact S-Expressions, but for reasons that will become clear soon it is useful to henceforth avoid the overloaded term "list."

Structurally, an S-Expression is a binary tree. However, it differs from binary trees in that its non-leaf nodes are unlabeled. Due to this difference in kind, and also just for the sake of a workable example, we choose to add S-Expressions as a *math module*, *i.e.*, an extension of the small collection of Resolve mathematical theories [34]. The core of this theory is shown below.

```
1  mathematics S_EXPRESSION_THEORY (math type LABEL)
2
3     math type S_EXPRESSION of LABEL
4
5     math operation NIL: S_EXPRESSION of LABEL
6
7     math operation ATOM (l: LABEL): S_EXPRESSION of LABEL
8
9     math operation CONS (
10          car: S_EXPRESSION of LABEL
11          cdr: S_EXPRESSION of LABEL
12       ): S_EXPRESSION of LABEL
13
14    axiom atom_neq_nil is
15       for all l: LABEL (ATOM(l) /= NIL)
16
17    axiom atom_neq_cons is
18       for all car, cdr: S_EXPRESSION of LABEL, l: LABEL
19          (CONS(car, cdr) /= ATOM(l))
20
21    axiom cons_neq_nil is
22       for all car, cdr: S_EXPRESSION of LABEL
23          (CONS(car, cdr) /= NIL)
24
25    axiom atom_injective is
26       for all l, m: LABEL (if ATOM(l) = ATOM(m) then l = m)
27
28    axiom cons_injective is
29       for all car1, car2, cdr1, cdr2: S_EXPRESSION of LABEL
30          (if CONS(car1, cdr1) = CONS(car2, cdr2 )
31           then car1 = car2 and cdr1 = cdr2)
32
33    axiom induction is
34       for all s: set of S_EXPRESSION, l: LABEL
35                car, cdr :S_EXPRESSION
36          (if (NIL is in s and ATOM(l) is in s and
37               (if (car is in s and cdr is in s)
38                then CONS(car, cdr) is in s))
39           then s = universal_set)
40
41 end S_EXPRESSION_THEORY
```

This definition of S-Expressions is just a minimalistic kernel. For this theory to be useful in practice, it should be more expressive, with more definitions and theorems about the relationships among S-Expressions. One particularly useful subset of S-Expressions is that of "proper lists," which have a strictly right-branching structure—*i.e.*, all left branches are `LABEL`s, and which terminate with a `NIL`. A math subtype defining proper lists is shown next.

```
1  definition IS_PROPER_LIST (
2          a: S_EXPRESSION of T
3      ): boolean is
4      s = NIL or
5      (there exists l: T, cdr: S_EXPRESSION of T
6          (s = CONS(ATOM(l), cdr) and IS_PROPER_LIST(cdr)))
7
8  math subtype PROPER_LIST of T is S_EXPRESSION of T
9       exemplar a
10      constraint
11          IS_PROPER_LIST(a)
```

Languages like Lisp often provide functions to compute the length of a proper list—its height when considered as an S-Expression, minus one for the terminal NIL—and to append one proper list onto another. Resolve definitions describing this functionality are given below, and they make use of the fact that a proper list can be thought of abstractly by just considering the string consisting of all the `LABEL`s in its left branches, in top-to-bottom order. Contrast this to the "code *is* spec" approach, wherein the recursive programmatic definitions of these functions would be considered a sufficient description of their behavior. In the Resolve approach, we see that there are now qualitatively different descriptions (in terms of mathematical string theory) of the length and append functions relative to which any of their possible

implementations, recursive or otherwise, could be verified.

```
1  definition STRING_OF (
2        s: PROPER_LIST of T
3     ): string of T satisfies
4     if s = NIL
5     then STRING_OF(s) = empty_string
6     else there exists l: T, cdr: PROPER_LIST of T
7        (s = CONS(ATOM(l), cdr) and
8         STRING_OF(s) = <l> * STRING_OF(cdr)
9
10 definition LEN (
11       s: PROPER_LIST of T
12    ): integer is
13    |STRING_OF(s)|
14
15 definition APPEND (
16       s: PROPER_LIST of T,
17       t: PROPER_LIST of T
18    ): PROPER_LIST of T satisfies
19    STRING_OF(APPEND(s, t)) = STRING_OF(s) * STRING_OF(t)
```

§2.2.2.2 Map, Filter, and Fold

In terms of theoretical innovation, the `Map` function is probably the most important fundamental construct that can be credited to functional programming. `Map` applies a function to every element of a proper list. The reason why `Map` is closely tied to functional programming is that it requires a function to be passed as a parameter, and as we mentioned above, functional languages make code (read: function implementations) easy to treat as data. No such treatment of code as data occurs in our definition of `MAP` below, however—remember that the Resolve snippets shown in this chapter depict specifications, not code, and that this strict separation is advantageous. `MAP` (likewise `f`) is not executable, but in practice would be used in the specification of

something that is. For example, a programming function called `Map` involving programming objects modeled as `PROPER_LIST`s could have "`ensures STRING_OF(Map) = MAP(STRING_OF(s), f)`" as its contract. Note that the typographical convention of using all caps for math definition names helps avoid ambiguity in Resolve specifications, since the name of a programming function may appear in its contract's `ensures` clause to denote the function's return value.

```
1  definition MAP (
2        s: string of T1,
3        f: function of (T1, T2)
4     ): string of T2 satisfies
5     if (s = empty_string)
6     then MAP(s, f) = empty_string
7     else
8        there exists i: T1, t: string of T1
9           (s = <i> * t and
10           MAP(s, f) = <f(i)> * MAP(t, f))
```

In addition to its aesthetic elegance, `MAP` is powerful because it lends itself to parallelization—Google's algorithms are examples of demanding computational tasks that leverage `MAP` extensively [19]. Two closely related constructs with similar advantages are `FILTER` and `FOLD`. `FILTER` eliminates the elements of a list that fail some provided test, and `FOLD` computes a result by applying an operator successively to each element of a list starting with some seed value. For example, `FOLD`ing addition over the list of the first three nonzero natural numbers and starting with a seed of zero yields 6. Resolve definitions for `FILTER` and `FOLD` are given below.

```
1  definition FILTER (
2        s: string of T,
3        p: function of (T, boolean)
4     ): string of T satisfies
```

```
5     if (s = empty_string)
6     then FILTER(s) = empty_string
7     else
8        there exists l: T, t: string of T
9           (s = <l> * t and
10           if (p(l))
11           then FILTER(s, p) = <l> * FILTER(t, p)
12           else FILTER(s, p) = FILTER(t, p))
13
14 definition FOLD (
15        s: string of T2,
16        v: T1
17        f: function of ((T1, T2), T1)
18    ): string of T1 satisfies
19    if (s = empty_string)
20    then FOLD(s, v, f) = v
21    else
22       there exists l: T1, t: string of T1
23          (s = t * <l> and
24           FOLD(s, v, f) = f(FOLD(t, v, f), l))
```

Note that the latter definition expresses left-to-right folding. Some functional languages provide both directions of folding as built-in functions; the right-to-left direction is of course easy to specify in a manner entirely analogous to this, although its recursive implementation would be markedly different than in the left-to-right case. Again we see utility in the expressiveness of strict abstraction.

Having seen that formalizations of the common data structures and algorithmic constructs of functional programming are feasible in Resolve, we'll conclude this chapter with a discussion of how automated tools for mathematical proofs can aid in the engineering of specifications.

§2.3 Tool Support for Mathematical Theory Development

Consider the alternative method for defining the length of a proper list shown below. The recursive nature of this definition is similar in spirit to how one would actually code a length function in a functional programming language.

```
1  definition LEN2 (
2        s: PROPER_LIST of T
3     ): integer satisfies
4     if (s = NIL) then LEN2(s) = 0
5     else
6        there exists l: T, cdr: PROPER_LIST of T
7           (s = CONS(ATOM(l), cdr) and
8            LEN2(s) = 1 + LEN2(cdr))
```

We may be curious to know whether or not this definition is equivalent to our original, string-based version. For example, if the body of `LEN2` was the semantics a functional programming language ascribed to a recursive implementation of length (a likely scenario), and if our original `LEN` was used to specify that function, proving the two definitions equivalent would essentially be a verification of the code. We can conduct this proof by hand:

Claim. *Let* `T` *be a mathematical type.* $\forall s$: `PROPER_LIST of T`, `LEN`$(s) =$ `LEN2`(s).

Proof. Let s be an arbitrary `PROPER_LIST of T`. We proceed by induction on s. In the base case, $s =$ `NIL`. Since `STRING_OF`(`NIL`) = `empty_string`, we have `LEN`$(s) = 0$. The definition of `LEN2` gives `LEN2`(`NIL`) $= 0$ as well.

Our inductive hypothesis states that when $s =$ `CONS`(`ATOM`$(l), c)$ for some `T` l and `PROPER_LIST of T` c, `LEN`$(c) =$ `LEN2`(c). Now for our induction step, assume that $s =$ `CONS`(`ATOM`$(l), c)$ for some `T` l and `PROPER_LIST of T` c. By the definition of

STRING_OF we have LEN$(s) = |\langle l \rangle * \texttt{STRING_OF}(c)|$, which by string theory is equal to $1 + |\texttt{STRING\_OF}(c)|$. The second term is just LEN(c), so by inductive hypothesis and the definition of LEN2 we have LEN$(s) = 1 + \texttt{LEN}(c) = 1 + \texttt{LEN2}(c) = \texttt{LEN2}(s)$, thus completing the induction step. □

Notice that, despite the fact that S-Expression theory and its attendant definitions are all meant to serve as a model of programmatic data, proofs of properties of S-Expressions are conducted in ordinary mathematics. This again is an upshot of strict abstraction. It should be clear now that there are multiple separable roles in the Resolve-style software verification workflow, *e.g.*, specification writer, math theory developer, programmer, etc. We will elaborate these roles into a full vision of the Resolve verified software process in Chapter 5.

The property we've proven here is quite simple, and yet its proof is non-trivial and provides a glimpse of the "bookkeeping overhead" that proofs relating to verification often entail [42]. Formally proving mathematical properties with full rigor—a practice that most math texts do not even attempt—carries a heavy burden of intellectual complexity, much of it due to "clerical" matters like properly formulating inductive hypotheses rather than developing deep mathematical insights. It seems natural, then, to look for automated assistance. We conclude our survey of math's role in verification by discussing tool support for developing mathematical theories.

§2.3.1 Isabelle

Isabelle [59] is an interactive proof assistant that is commonly employed for a wide range of mathematical and computational tasks. Researchers in the verification community use Isabelle to help manage the bookkeeping of difficult proofs, and also

use scripts called "tactics" to run Isabelle in an automatic, "push-button" manner. Isabelle is also used for heavyweight mathematical proofs and formalization efforts, e.g., a formally checked proof of the prime number theorem [4].

An investigation of Isabelle's suitability as a push-button prover for Resolve has been conducted [43]. The most pertinent aspect of this work for our purposes here is that Resolve mathematical theories can be incorporated into Isabelle's automated reasoning process by axiomatizing them using a syntax similar in spirit to that of our S-Expression theory, and by providing a witness to the existence of a satisfying theory via built-in Isabelle types. The full details are contained in a Resolve researcher's Ph.D dissertation [41], and we augment this work by investigating a different tool that is often cited as an alternative to Isabelle: Coq.

§2.3.2 Coq

The Coq Proof Assistant [8] is an interactive implementation of the Calculus of Inductive Constructions, an intuitionistic logic with a theory of types that models the inductive data types typical of strongly-typed functional programming languages such as ML and Haskell. The Coq type system is rich enough to facilitate verification based on the Curry-Howard isomorphism [30], meaning that the types of objects in Coq can be arbitrary logical formulae, and that a well-typed program can be considered a constructive proof of the formula that program's type encodes. This makes Coq similar in a sense to ACL2—it is simultaneously a programming language and a logic for proving properties of programs. In contrast to ACL2, though, Coq is statically typed, and these types are what serve as specifications. In other words, ACL2 proves programs equivalent to other programs, with these other programs comprising

the descriptions of behavior relative to which verification is performed, whereas Coq proves programs type-correct, with type signatures serving as these descriptions.

The reader may already be getting a sense that the theoretical trappings of Coq are extremely technical and language-specific. However, continuing in the vein of strict abstraction, this author has chosen to use Coq solely as a programming language-independent tool for developing mathematical theories. This approach simplifies matters tremendously, and no other work is known to use Coq in such a manner.

The intuitionistic nature of Coq, specifically the Calculus of Inductive Constructions, restricts the kinds of mathematical theories that can be expressed. Coq data types—which for our simplified purposes should be thought of as Resolve math theories—can only be defined by a collection of (perhaps inductive) constructor functions, and these are automatically assumed to have certain properties, most notably injectivity and disjointness of images. Fortunately, many Resolve math theories fit this mold, and Coq eases their development by providing a relatively simple framework for expressing theories and proving theorems about them with less clerical overhead. The listing below shows a definition of proper lists in Coq syntax, along with the reply that Coq gives when the definition is entered. The `Implicit Arguments` directives are not important for conceptual purposes here, they just allow future references to `plist` to omit the `Type` parameter.

```
1 Coq < Inductive plist (X:Type): Type :=
2 Coq <   | nil: plist X
3 Coq <   | cons: X -> plist X -> plist X.
4 plist is defined
5 plist_rect is defined
6 plist_ind is defined
7 plist_rec is defined
```

```
8
9  Coq < Implicit Arguments nil [[X]].
10
11 Coq < Implicit Arguments cons [[X]].
```

This definition is different than the one we formulated in Resolve, because in Coq we did not first define S-Expressions and then subsequently define proper lists as a subset of them. Such "math subtyping" is indeed possible in Coq, but it involves precisely those type system complications that we wish to avoid here. It should be clear nonetheless that the definition of `plist` creates a type that is structurally equivalent to our previous description of proper lists—`plist`s' derivation trees are isomorphic to `PROPER_LIST`s.

The response from the Coq environment indicates that four new facts are automatically deduced from the proper list data type declaration. The first, `plist` is just the definition of the datatype itself. `plist_rect` and `plist_rec` are of no practical interest here, but the other, `plist_ind`, is in a sense equivalent to S-Expression theory's `induction` axiom: it states that any property which holds of `nil` and also of all `cons`es whose argument `plist` satisfies the property is true of all proper lists. The body of this definition is shown in Coq syntax below (for our purposes, `Prop` can be thought of as the type of boolean mathematical values in Coq).

```
1  Coq < Check plist_ind.
2  plist_ind
3       : forall (X : Type) (P : plist X -> Prop),
4         P nil ->
5         (forall (x : X) (p : plist X), P p -> P (cons x p)) ->
6         forall p : plist X, P p
```

As a final demonstration of Coq's efficacy as a tool for theory development, we will prove our previous claim regarding `LEN` and `LEN2` inside the Coq environment. Since `LEN` depends on string theory, which is not built in to Coq, a definition of `string`, as well as some of its fundamental operators—`length` (| |), `concat` ($*$), and `stringleton` ($\langle\ \rangle$)—are prerequisite to this exercise. The author has conducted a lengthy development of a large portion of Resolve string theory, which includes these definitions, inside of Coq. This appears as Appendix A as the end of this document.

In addition to those fundamental string definitions, we will of course need to define `STRING_OF`, `LEN`, and `LEN2` inside of Coq. It will also be useful to define a lemma relating stating that a the length of a stringleton concatenated with another string is one plus the length of that string. All four of these definitions are shown in the following listing, which is lengthy due to the verbosity of the Coq proof interaction. We will discuss the Coq proof process next.

```
1  Coq < Fixpoint string_of (X:Type) (p: plist X) : string X :=
2  Coq <   match p with
3  Coq <   | nil => empty_string
4  Coq <   | cons a q => concat (stringleton a) (string_of X q)
5  Coq <   end.
6  string_of is recursively defined (decreasing on 2nd argument)
7
8  Coq < Implicit Arguments string_of [[X]].
9
10 Coq < Definition len1 (X:Type) (p: plist X): nat :=
11 Coq <   length (string_of p).
12 len1 is defined
13
14 Coq < Implicit Arguments len1 [[X]].
15
16 Coq < Fixpoint len2 (X:Type) (p: plist X): nat :=
17 Coq <   match p with
18 Coq <   | nil => 0
```

```
Coq <    | cons a q => S (len2 X q)
Coq <  end.
len2 is recursively defined (decreasing on 2nd argument)

Coq < Implicit Arguments len2 [[X]].

Coq < Lemma len_stringleton_concat:
Coq <   forall (X:Type) (x: X) (s: string X),
Coq <     length (concat (stringleton x) s) = S (length s).
1 subgoal

  ============================
   forall (X : Type) (x : X) (s : string X),
   length (concat (stringleton x) s) = S (length s)

len_stringleton_concat < induction s.
2 subgoals

  X : Type
  x : X
  ============================
   length (concat (stringleton x) empty_string) =
   S (length empty_string)

subgoal 2 is:
 length (concat (stringleton x) (ext s x0)) =
 S (length (ext s x0))

len_stringleton_concat < simpl.
2 subgoals

  X : Type
  x : X
  ============================
   1 = 1

subgoal 2 is:
 length (concat (stringleton x) (ext s x0)) =
 S (length (ext s x0))

len_stringleton_concat < reflexivity.
1 subgoal

```

```
  X : Type
  x : X
  s : string X
  x0 : X
  IHs : length (concat (stringleton x) s) = S (length s)
  ============================
   length (concat (stringleton x) (ext s x0)) =
   S (length (ext s x0))

len_stringleton_concat < simpl.
1 subgoal

  X : Type
  x : X
  s : string X
  x0 : X
  IHs : length (concat (stringleton x) s) = S (length s)
  ============================
   S (length (concat (stringleton x) s)) = S (S (length s))

len_stringleton_concat < rewrite -> IHs.
1 subgoal

  X : Type
  x : X
  s : string X
  x0 : X
  IHs : length (concat (stringleton x) s) = S (length s)
  ============================
   S (S (length s)) = S (S (length s))

len_stringleton_concat < reflexivity.
Proof completed.
```

`Fixpoint` is the Coq keyword for recursive function definitions, `nat` is a built-in type for nonnegative integers, and `S` is its successor function. When a `Lemma` (or a `Theorem`, as we'll see shortly) is introduced, the Coq top-level enters a new mode of interaction wherein the user must supply a proof. Each prompt for user input ends with a `<` symbol. Coq proofs are usually constructed in a backwards-reasoning style

via commands called "tactics"—subroutines for symbol manipulation that have been established to be sound relative to the Calculus of Inductive Constructions. A rich arsenal of tactics are available in Coq, but only a few are necessary to prove our lemma:

induction Breaks into cases on a given variable, using the variable's type definition—its constructors—to determine the form of each case. In the case of an inductive constructor (such as `ext`, which is `string`'s extension function), then the appropriate inductive hypothesis is automatically generated for that case.

simpl Performs simple rewriting of terms in the goal using the bodies of any definitions mentioned.

reflexivity Discharges a goal of the form $x = x$.

rewrite Changes the goal by rewriting terms directionally in accordance with the assumption mentioned. In our proof of `len_stringleton_concat`, the goal was changed so that any of its subformulas matching the left-hand side of the inductive hypothesis were rewritten in the form indicated by the right-hand side.

With these definitions in scope, we can now state and prove the claim of `len1`'s and `len2`'s equivalence. This proof is entirely isomorphic to the one we conducted by hand earlier, but in Coq the proof process becomes a simple matter of knowing which tactic to apply at each step; the bookkeeping of rewriting terms, generating inductive hypotheses, and managing cases is all automatic. Before listing the proof, we need to describe one more built-in tactic:

unfold Rewrites a defined name with the body of its definition. This is often necessary to facilitate further simplifications via `simpl`.

```
1  Coq < Theorem lens_equal:
2  Coq <   forall (X:Type) (p: plist X), len1 p = len2 p.
3  1 subgoal
4
5    ============================
6      forall (X : Type) (p : plist X), len1 p = len2 p
7
8  lens_equal < induction p.
9  2 subgoals
10
11   X : Type
12   ============================
13     len1 nil = len2 nil
14
15 subgoal 2 is:
16  len1 (cons X x p) = len2 (cons X x p)
17
18 lens_equal < unfold len1.
19 2 subgoals
20
21   X : Type
22   ============================
23     length (string_of nil) = len2 nil
24
25 subgoal 2 is:
26  len1 (cons X x p) = len2 (cons X x p)
27
28 lens_equal < simpl.
29 2 subgoals
30
31   X : Type
32   ============================
33     0 = 0
34
35 subgoal 2 is:
36  len1 (cons X x p) = len2 (cons X x p)
37
38 lens_equal < reflexivity.
```

```
1 subgoal

  X : Type
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   len1 (cons X x p) = len2 (cons X x p)

lens_equal < simpl.
1 subgoal

  X : Type
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   len1 (cons X x p) = S (len2 p)

lens_equal < rewrite <- IHp.
1 subgoal

  X : Type
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   len1 (cons X x p) = S (len1 p)

lens_equal < unfold len1.
1 subgoal

  X : Type
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   length (string_of (cons X x p)) = S (length (string_of p))

lens_equal < simpl.
1 subgoal

  X : Type
```

```
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   length (concat (stringleton x) (string_of p)) =
   S (length (string_of p))

lens_equal < rewrite -> len_stringleton_concat.
1 subgoal

  X : Type
  x : X
  p : plist X
  IHp : len1 p = len2 p
  ============================
   S (length (string_of p)) = S (length (string_of p))

lens_equal < reflexivity.
Proof completed.
```

The fragment of the S-Expression theory in Coq glimpsed thus far has been a very small example, meant only to convey the plausibility of Coq as a useful tool to aid the formulation of mathematical theories for Resolve. As was mentioned before, the author has conducted a much larger case study, which defines and proves a substantial portion of mathematical string theory, shown in Appendix A.

It is our hope that the reader is now pondering the wider applicability of tools like Coq for software verification. Can automated tools synthesize code and specifications to create "verification conditions," *i.e.*, the logical formulae whose validity corresponds to the correctness of the code? Can proof assistants like Coq be used to decide the truth or falsehood of these verification conditions? Do such proofs always require detailed interaction via tactics? We will answer all of these questions in good time, but for now it remains to be seen how a *human*, let alone a computer program,

could make meaningful formal connections between code and specifications in order to determine whether or not a program is correct.

Chapter 3: Modular Program Verification, Take One: Client Programming

§3.1 The Resolve Imperative Programming Language

Having seen the key features of the Resolve specification language, we now move on to an investigation of the imperative programming language that the current tools developed at OSU verify. What does Resolve code look like, and by what process can one prove that a Resolve program satisfies its formal behavioral specification? Again, we present this material by example, and trust the reader to abstract away from it.

The listing below shows Resolve code that satisfies the `Sort` specification discussed in the previous chapter.[12]

```
1 realization SelectionSort (
2         function AreInOrder (restores i: Item,
3                              restores j: Item): control
4             ensures
5                 AreInOrder = ARE_IN_ORDER (i, j)
6     ) implements Sort for QueueTemplate
7
```

[12]No compiler for the Resolve programming language dialect described here currently exists; although this code can be verified as correct, it cannot be compiled and executed. No technical impediment stands in the way of a Resolve compiler—it is considered a straightforward enough task that it has never been a priority of our research program. However, there is a compiler for another dialect of Resolve [69].

```
8       local procedure RemoveMin (updates q: Queue,
9                                  replaces min: Item)
10          requires
11              q /= empty_string
12          ensures
13              IS_PERMUTATION (q * <min>, #q)  and
14              IS_PRECEDING (<min>, q)
15          ...
16      end RemoveMin
17
18      procedure Sort (updates q: Queue)
19          variable sorted: Queue
20          loop
21              maintains
22                  IS_PERMUTATION (q*sorted, #q*#sorted)  and
23                  IS_NONDECREASING (sorted)  and
24                  IS_PRECEDING (sorted, q)
25              decreases |q|
26          while not IsEmpty (q) do
27              variable min: Integer
28              RemoveMin (q, min)
29              Enqueue (sorted, min)
30          end loop
31          q :=: sorted
32      end Sort
33  end SelectionSort
```

At a coarse view, our `Sort` algorithm only does three things: it declares a variable called `sorted` for storing the sorted Queue it builds, it iteratively transfers the smallest thing in `q` to the end of `sorted` over and over until `q` is empty, and then it exchanges `q` with `sorted`[13], thus updating `q` in accordance with the contract for `Sort` from the previous chapter. This is the classic "selection sort" algorithm, presented in the Resolve programming language. We have omitted the body of `RemoveMin` in the

[13]Those familiar with languages like Java and C++ may have expected an assignment operator (`:=` or `=`) instead, but in Resolve, swapping (`:=:`) is the primary operator for moving data. This choice has many practical advantages, which we discuss in [61].

above code both to simplify the presentation, and also to emphasize that its specification is a sound conceptualization of its behavior—callers of `RemoveMin` need not know how it is implemented, only what it does. Notice also that this realization is parameterized by an `AreInOrder` function, thus liberating the implementation from a commitment to any one particular order by which to sort the data.

§3.1.1 Loop Invariants and Partial vs. Total Correctness

We see in the `Sort` realization a new syntactic slot for formal specification: a `maintains` clause, also called a *loop invariant*. Loop invariants are a common annotation construct that predate Resolve and trace their history to some of the earliest efforts at program verification [35]. An invariant allows the behavior of a loop to be reasoned about statically, *i.e.*, without having to consider the individual iterations that will occur when the program is executed. In fact, an invariant can be used to reason about the results of a loop's execution even in the absence of confidence that the loop will ever terminate. This is known as the *partial correctness* approach—partial correctness verification consists of proving that a program is correct relative to its specification provided that the program actually terminates. Resolve pursues a *total correctness* strategy by adding a `decreases` clause to the loop invariant. The `decreases` clause contains an expression over some well-founded set, usually natural numbers. To prove that the loop will actually terminate when executed, the body of the loop must be shown to decrease this well-founded measure unconditionally.

The `maintains` clause relates values at the beginning of a loop iteration to those at the end of an iteration, and so by transitivity it expresses how the values of the

program variables immediately prior to the loop relate to those after the loop has terminated. In addition, we know that the negation of the loop condition (the *guard*) is also true immediately after the `end loop` statement: either we never entered the loop or else it terminated by failing its guard. The conjunction of these two mathematical assertions—the `maintains` clause and the negated guard—gives a characterization of the program state immediately after the loop, without having to consider the loop iteration-by-iteration.

Loop invariants are in essence a formal statement of the thought process that the programmer must undergo in order to write a correct loop in the first place; Resolve (like many other verification systems) simply requires that programmers make this reasoning explicit via formal annotation.

Occasionally, other mathematical properties relevant to the correctness of an algorithm are helpful to express inside its implementation, either for documentation purposes, or to provide "hints" to the automated reasoning techniques we will soon discuss. Resolve offers a syntactic slot labeled `confirm` for stating such properties. As a trivial example, we could have written `confirm sorted = empty_string` between lines 19 and 20 in our `Sort` realization.

We will say more about the role of programmer annotations in a framework for push-button verification when we discuss tool support in Chapter 5. For now, note once more that each annotation slot in Resolve is filled by with an assertion in the specification language, never mentioning any programming operations or operators.[14] This, roughly, is the syntactic characterization of strict abstraction. However, in

[14]Except for the aforementioned special case of mentioning the name of a function in its `ensures` clause to denote its return value

Chapter 6 we will consider a scenario in which a programming operation name may be mentioned in an assertion in a manner that preserves strict abstraction.

Having now seen examples of both Resolve specifications and code, along with the small well-defined interface between these two worlds, we will now examine the Resolve verification strategy. What we require are rules for combining the mathematical meaning of code (its *semantics*), with the specifications/annotations presented heretofore, into *verification conditions*: sentences of predicate calculus whose truth corresponds to the correctness of the code relative to its spec.

§3.2 Tabular Verification in Resolve

The linchpin of the Resolve verification methodology we will describe is a diagram known as a *tracing table* [67]. A tracing table for the selection sort code seen in §3.1 is shown below. The general idea of a tracing table is to depict the state of the program—the values of all in-scope programming variables—and in particular to show how each line of code will affect the state of the program when executed. Every state is given a numeric index, and each boxed row of the table shows what is known at that state and what is required in order for the next piece of code to be legal. Subscripts are affixed to all programming variables mentioned in tracing tables, so the variables' values at different states can be differentiated. Note that some straightforward abbreviations, such as $\langle\rangle$ for the empty string, have been adopted for simplicity's sake in this tracing table:

State	Path Conds	Facts	Obligations
0			

```
variable sorted:  Queue
```

1		$sorted_1 = \langle\rangle$ $q_1 = q_0$	IS_PERM($q_1 * sorted_1, q_1 * sorted_1$) IS_NONDEC($sorted_1$) IS_PREC($sorted_1, q_1$) $q_1 \neq \langle\rangle \Rightarrow 0 < \lvert q_1 \rvert$

```
loop
    maintains
        IS_PERMUTATION (q * sorted, #q * #sorted) and
        IS_NONDECREASING (sorted) and IS_PRECEDING (sorted, q)
    decreases |q|
```

2	$q_2 \neq \langle\rangle$	IS_PERM($q_2 * sorted_2, q_1 * sorted_1$) IS_NONDEC($sorted_2$) IS_PREC($sorted_2, q_2$)	

```
        variable min:  Integer
```

3	$q_2 \neq \langle\rangle$	$q_3 = q_2$ $sorted_3 = sorted_2$ $min_3 = 0$	$q_3 \neq \langle\rangle$

```
        RemoveMin(q, min)
```

4	$q_2 \neq \langle\rangle$	$sorted_4 = sorted_3$ IS_PERM($q_4 * \langle min_4 \rangle, q_3$) IS_PREC($\langle min_4 \rangle, q_4$)	

```
        Enqueue(sorted, min)
```

5	$q_2 \neq \langle\rangle$	$q_5 = q_4$ $sorted_5 = sorted_4 * \langle min_4 \rangle$ $min_5 = 0$	IS_PERM($q_5 * sorted_5, q_2 * sorted_2$) IS_NONDEC($sorted_5$) IS_PREC($sorted_5, q_5$) $\lvert q_5 \rvert < \lvert q_2 \rvert$

```
end loop
```

6		IS_PERM($q_6 * sorted_6, q_1 * sorted_1$) IS_NONDEC($sorted_6$) IS_PREC($sorted_6, q_6$) $q_6 = \langle\rangle$	

```
q :=:  sorted
```

7		$q_7 = sorted_6$ $sorted_7 = q_6$	IS_PERM(q_7, q_0) IS_NONDEC(q_7)

A state's "Facts" consist of the postcondition of the previous line of code (if it was a procedure call), statements of equality between subscripted variables (*e.g.*, due to a swap, or due to the "frame property"—the fact that variables not mentioned in a line of code remain unchanged in the next state), or the loop invariant (if the current state is the first inside a loop or the first immediately following a loop). Also, the 0^{th} state of a procedure that has a precondition will contain a manifestation of

that `requires` clause (with the appropriate substitutions of indexed actual parameter names for formal parameters) as a fact. Contrariwise, the "Obligations" at a state consist of the `requires` clause of the procedure we are about to call (if the next line of code contains a call), the `ensures` clause of the current operation (if this is the final state in a procedure body), or the loop invariant (if the current state is the one immediately preceding the start of a loop, or is the final state in a loop's body). Of course, `confirm` statements also cause an obligation to be introduced. "Path Conditions" are additional facts that are assumed to be true inside of control structures (`if`s and `loop`s) due to their guards.

It is interesting to observe the manifestations of the loop invariant at various states in the `Sort` tracing table. We see in state 1 that the invariant must hold prior to the loop, but that here there is no difference between variables with a `#` and those without. At such points, loop invariants formulated carefully can often be proven by simple properties such as the reflexivity of a definition like `IS_PERMUTATION`. Inside of the loop but prior to the first line of its body, the invariant is available as facts, and it appears as an obligation immediately after the body's final line of code. Finally, the invariant is available as a fact immediately after the loop. Our tracing table thus demonstrates exactly how loop invariants are used to reason about a loop's behavior without considering multiple iterations of its body. The reader should be able to convince herself at this point that generating tracing tables is a purely mechanical process, albeit somewhat complicated by issues such as choosing correct state indices to correspond to the meaning of `#` in invariants and postconditions.

§3.2.1 VCs, Soundness and Relative Completeness

Let P_i denote the conjunction of all path conditions at state i. Likewise, let A_i be the conjunction of that state's facts (what we "assume") and let C_i be the conjunction of that state's obligations (what we must "confirm"). Then verifying a piece of code by the tabular approach consists of proving, for each state index n, the following formula:

$$\bigwedge_{0 \leq i \leq n} (P_i \Rightarrow A_i) \Rightarrow (P_n \Rightarrow C_n)$$

For a given table, each implication that arises by applying this formula to a particular state is known as a *verification condition* ("VC"). The formula demonstrates that "facts" are only available for the proof of an obligation if the path conditions those facts are true under actually obtain. Furthermore, the formula shows that any obligation guarded by a false path condition leads to a VC that is vacuously true due to a false antecedent—such VCs correspond to unreachable or "dead" code, which the tabular proof process can discover by determining P_n to be unsatisfiable. This is one way in which the tabular proof strategy can provide useful feedback when an attempted verification fails [3].

The soundness and relative completeness of the tabular approach have been proven elsewhere [46, 33]. We could now proceed to prove this selection sort code correct, although carrying the proof through in full detail would be quite tedious. Again we see that automated proof assistants have an important role to play in program verification. For the sake of example, we show a simplified form of one of `Sort`'s verification conditions in Fig. 3.1.

$$
\begin{aligned}
&(s_1 = \langle\rangle \\
&\wedge\ q_1 = q_0 \\
&\wedge\ \texttt{IS\_PERM}(q_1 * s_1, q_1 * s_1) \\
&\wedge\ \texttt{IS\_NONDEC}(s_1) \\
&\wedge\ \texttt{IS\_PREC}(s_1, q_1) \\
&\wedge\ \texttt{IS\_PERM}(q_6 * s_6, q_1 * s_1) \\
&\wedge\ \texttt{IS\_NONDEC}(s_6) \\
&\wedge\ \texttt{IS\_PREC}(s_6, q_6) \\
&\wedge\ q_6 = \langle\rangle \\
&\wedge\ q_7 = s_6 \\
&\wedge\ s_7 = q_6) \\
&\quad \Longrightarrow \\
&\texttt{IS\_PERM}(q_7, q_0)
\end{aligned}
$$

Figure 3.1: A `Sort` verification condition. Some straightforward abbreviations, *e.g.*, s for `sorted`, have been adopted for simplicity's sake. Some available facts that are not useful for the proof of this VC (but not all of them) have been omitted, *i.e.*, those arising from the loop body.

In [33], Heym observes that the definition of "completeness" for a verification technique must not only account for the incompleteness of formal systems due to Gödel's theorems [58], but also the possibility that a program may be correct yet insufficiently annotated. Not only are programs whose meanings are Gödel sentences unverifiable, but so might be programs whose loop invariants are not sufficiently strong. A verification system is called "relatively complete" if all valid programs whose mathematical interpretations are provable in mathematical logic can be proven correct, perhaps by strengthening annotations such as loop invariants.

§3.2.2 Two Views of Programming Language Semantics

If `RemoveMin` were incorrect, *i.e.*, if the code that actually executed when `Sort` called `RemoveMin` did not meet its contract, then `Sort` would be incorrect as well, regardless of whether or not the obligations in `Sort`'s tracing table could all be proven. The reason for this vulnerability is that tabular verification in Resolve is a completely *modular* task—verifying a piece of code involves only the specifications (never the code) of the components and operations that code depends on.

As an exercise, we could force the tabular methodology to process `Sort` in a less modular way. Namely, we could inline the body of `RemoveMin` (not yet seen), rather than making it a procedural abstraction. A tracing table for this new approach to `Sort` is given below. The code for `RemoveMin` appears between states 3 and 11. `AreInOrder` is a comparison function for `Integer`s, so this code still has external dependencies, even with `RemoveMin` inlined, however, the previous version's vulnerability to an incorrect `RemoveMin` has been obviated.

State	Path Conds	Facts	Obligations
0			

```
variable sorted:  Queue
```

1		$sorted_1 = \langle\rangle$ $q_1 = q_0$	IS_PERM$(q_1 * sorted_1, q_1 * sorted_1)$ IS_NONDEC$(sorted_1)$ IS_PREC$(sorted_1, q_1)$ $q_1 \neq \langle\rangle \Rightarrow 0 < \lvert q_1 \rvert$

```
loop
    maintains
        IS_PERMUTATION (q * sorted, #q * #sorted) and
        IS_NONDECREASING (sorted) and IS_PRECEDING (sorted, q)
    decreases |q|
```

2	$q_2 \neq \langle\rangle$	IS_PERM$(q_2 * sorted_2, q_1 * sorted_1)$ IS_NONDEC$(sorted_2)$ IS_PREC$(sorted_2, q_2)$	

```
        variable min:  Integer
```

3	$q_2 \neq \langle\rangle$	$q_3 = q_2$ $sorted_3 = sorted_2$ $min_3 = 0$	

```
        variable tmp:  Queue
```

4	$q_2 \neq \langle\rangle$	$min_4 = min_3$ $q_4 = q_3$ $sorted_4 = sorted_3$ $tmp_4 = \langle\rangle$	$q_4 \neq \langle\rangle$

```
        Dequeue (q, min)
```

5	$q_2 \neq \langle\rangle$	$q_4 = \langle min_5 \rangle * q_5$ $tmp_5 = tmp_4$ $sorted_5 = sorted_4$	IS_PERM$(tmp_5 * q_5 * \langle min_5 \rangle,$ $tmp_5 * q_5 * \langle min_5 \rangle)$ IS_PREC$(\langle min_5 \rangle, tmp_5)$

```
        loop
            maintains IS_PERMUTATION(tmp * q * <min>, #tmp * #q * <#min>) and
                      IS_PRECEDING(<min>, tmp)
            decreases |q|
        while not IsEmpty (q) do
```

6	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	IS_PERM$(tmp_6 * q_6 * \langle min_6 \rangle,$ $tmp_5 * q_5 * \langle min_5 \rangle)$ IS_PREC$(\langle min_6 \rangle, tmp_6)$	

```
            variable x:  Integer
```

7	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	$min_7 = min_6$ $q_7 = q_6$ $sorted_7 = sorted_6$ $tmp_7 = tmp_6$ $x_7 = 0$	$q_7 \neq \langle\rangle$

```
            Dequeue (q, x)
```

8	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	$min_8 = min_7$ $sorted_8 = sorted_7$ $tmp_8 = tmp_7$ $q_7 = \langle x_8 \rangle * q_8$	

```
            if not AreInOrder (min, x) then
```

9	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	$min_9 = min_8$ $sorted_9 = sorted_8$	

	$\neg min_8 \leq x_8$	$tmp_9 = tmp_8$ $q_9 = q_8$ $x_9 = x_8$	

`min :=: x`

10	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$ $\neg min_8 \leq x_8$	$min_{10} = x_9$ $x_{10} = min_9$ $sorted_{10} = sorted_9$ $tmp_{10} = tmp_9$ $q_{10} = q_9$	

`end if`

11	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	$\neg min_8 \leq x_8 \Rightarrow min_{11} = min_{10}$ $\neg min_8 \leq x_8 \Rightarrow q_{11} = q_{10}$ $\neg min_8 \leq x_8 \Rightarrow sorted_{11} = sorted_{10}$ $\neg min_8 \leq x_8 \Rightarrow tmp_{11} = tmp_{10}$ $\neg min_8 \leq x_8 \Rightarrow x_{11} = x_{10}$ $min_8 \leq x_8 \Rightarrow min_{11} = min_8$ $min_8 \leq x_8 \Rightarrow q_{11} = q_8$ $min_8 \leq x_8 \Rightarrow sorted_{11} = sorted_8$ $min_8 \leq x_8 \Rightarrow tmp_{11} = tmp_8$ $min_8 \leq x_8 \Rightarrow x_{11} = x_8$	

`Enqueue (tmp, x)`

12	$q_2 \neq \langle\rangle$ $q_6 \neq \langle\rangle$	$min_{12} = min_{11}$ $sorted_{12} = sorted_{11}$ $tmp_{12} = tmp_{11} * \langle x_{11} \rangle$ $q_{12} = q_{11}$ $x_{12} = 0$	IS_PERM($tmp_{12} * q_{12} * \langle min_{12},$ $tmp_5 * q_5 * \langle min_5 \rangle\rangle$) IS_PREC($\langle min_{12} \rangle, tmp_{12}$) $\lvert q_{12} \rvert \leq \lvert q_6 \rvert$

`end loop`

13	$q_2 \neq \langle\rangle$	$q_{13} = \langle\rangle$ IS_PERM($tmp_{13} * q_{13} * \langle min_{13} \rangle,$ $tmp_5 * q_5 * \langle min_5 \rangle$) IS_PREC($\langle min_{13} \rangle, tmp_{13}$)	

`q :=: tmp`

14	$q_2 \neq \langle\rangle$	$min_{14} = min_{13}$ $sorted_{14} = sorted_{13}$ $tmp_{14} = q_{13}$ $q_{14} = tmp_{13}$	

`Enqueue (sorted, min)`

15	$q_2 \neq \langle\rangle$	$min = 0$ $sorted_{15} = sorted_{14} * \langle min_{14} \rangle$ $tmp_{15} = tmp_{14}$ $q_{15} = q_{14}$	IS_PERM($q_{15} * sorted_{15}, q_1 * sorted_1$) IS_NONDEC($sorted_{15}$) IS_PREC($sorted_{15}, q_{15}$) $\lvert q_{15} \rvert = \lvert q_2 \rvert$

`end loop`

16		$q_{16} = \langle\rangle$ IS_PERM($q_{16} * sorted_{16}, q_1 * sorted_1$) IS_NONDEC($sorted_{16}$) IS_PREC($sorted_{16}, q_{16}$)	

`q :=: sorted`

17		$q_{17} = sorted_{16}$ $sorted_{17} = q_{16}$	IS_PERM(q_{17}, q_0) IS_NONDEC(q_{17})

The inlined body of `RemoveMin` makes clear how decision structures are processed in tabular verification: the `then` clause (and the `else` clause if there is one) carries additional path conditions corresponding to the truth value of the `if` guard for that clause, and at the end of the structure facts are added relating the values of the variables to the final values they took on in each clause. These are phrased as implications, with the guard (or its negation, in the case of fall-through or an `else` clause) as the antecedent.

The contrast between inlining and procedural abstraction concretized in our two tracing tables is emblematic of a larger issue, namely, how to think of the *meaning* of a procedure call when formulating code's semantics. One could argue that the meaning of an operation call is just its body, or take the opposing view that the call's meaning is just its contract.[15] For our discussion, let us call the former contention *synthesized semantics*, and the latter *decoupled semantics*. The term "synthesized" invokes the idea of a bottom-up reasoning process, whereas "decoupled" emphasizes the philosophy that code and specs are separate entities whose relation is only established by verification.

In a synthesized semantics approach, the "real" tracing table, even for the original realization of `Sort` shown in §3.1, is the long table that inlines `RemoveMin`. With decoupled semantics, our first, briefer table is a legitimate representation of our selection sort implementation's meaning. Krone [46] justifies a decoupled view of semantics by giving a sound and relatively complete method for reasoning about operation calls in terms of the operations' contracts. Heym [33] builds on this work by relating Krone's proof method to the tabular method evinced here via a "bridge rule" that preserves

[15] In either case, appropriate actual-for-formal parameter substitution is of course necessary.

soundness and relative completeness. Crucial to both of these works is Resolve's *relational semantics*—the principle that a piece of code's mathematical meaning may be a relation and not a function. In the next chapter we will see that when we consider data representations, a discrepancy between decoupled and synthesized semantics can lead to unsoundness when verifying programs in languages where every operation is a function.

Decoupled semantics are pragmatically advantageous to the reasoning, and thus the verification, process. For any operation whose body contains more than one line of code, inlining results in a table with more states, and also eliminates the use of that operation's contract from the table altogether. Hence, although inlining reduces annotation burden—*e.g.*, no contract for `RemoveMin` need be formulated to verify `Sort`—it *increases bookkeeping overhead* and hinders one's ability to recognize "easy" VCs by adding more facts and obligations. The situation is even worse when we consider recursive operations. An attempt to inline these yields an infinite tracing table, and so their synthesized semantics must be derived by other, more intellectually complex means such as fixed-point combinators. [26] gives an elegant presentation of the Y-combinator, which facilitates reasoning about recursion in the style of synthesized semantics—*i.e.*, without specifications, and without the need for infinite inlining—but such theoretical encumbrances are unnecessary and unwieldy by the lights of decoupled semantics.

Aside from the problems of recursion and VC complexity, we can argue against synthesized semantics by remembering Chapter 1's argument that abstraction, specifically *reconceptualization*, eases intellectual complexity by allowing entities such as

programming operations to be treated as black boxes. *How* `RemoveMin` algorithmically attains the behavior its contract specifies is conceptually immaterial to the correctness of `Sort`—from the perspective of modularly establishing `Sort`'s correctness, it is only the *what* and not the *how* of `RemoveMin` that is important.

The modular view of software systems enabled by decoupled semantics also promotes *interchangeability* of component realizations, which is crucial for non-functional requirements such as efficiency. Consider, for example, that there are many useful realizations of `Sort` that have markedly different performance characteristics, and client programmers should have the flexibility to experiment with different choices of sorting algorithm without having to rewrite, nor reverify, their client code. With decoupled semantics, we can prove a program that uses `Sort` correct once and for all, and if the `Sort` realization is later changed to a different (verified) algorithm (say, Quicksort), the proof of the client program still stands: the proof never depended on that code anyway.

In some sense, the principle of *reuse* is a dual of interchangeability, and like interchangeability, it is well facilitated by decoupled semantics. Typically, we think of reusability as a property of code: an abstraction such as `RemoveMin` (or `Queue`, for that matter) might be useful in a wide variety of applications and algorithms, and it would be silly to copy and paste its realization into many different programs. For one thing, this would mean surrendering a single point of control over change, thus making the code less maintainable.

As was hinted at in the discussion of interchangeability, the concept of reuse also applies to proofs of programs: once `Sort` has been verified, any program can reliably invoke it as a black box, and once a client program is verified in relation to some black

box, the realization of that contract can be changed to any other verified one without impacting to proof's validity. This avoidance of re-conducting proofs is crucial to the scalability of software verification. Notice that strict abstraction is what makes proof reuse possible: if we didn't have a purely mathematical view of `Sort`'s behavior, we couldn't fully ignore its implementation details.

From the perspectives of bookkeeping overhead, leveraging reconceptualization, interchangeability, and reuse, tabular verification with decoupled semantics has been seen to be a wise choice for practical, scalable software verification. Furthermore, we have seen the pivotal role that abstraction plays in this view. We can now synthesize these insights into a description of a sound, relatively complete, realistic verification environment. The core principles of such a system are:

(1) Use strict abstraction in all specifications/annotations.

(2) Formulate VCs in accordance with the tabular method and decoupled semantics.

(3) Flag components that have been changed since the last time they were verified.

(4) When verifying a piece of code, only the flagged components that it depends on need to be re-verified. Once verified, these components' flags should be removed so proof reuse is leveraged in later verifications.

The last two items indicate how interchangeability and reuse can be enabled in a verification environment while maintaining soundness by noticing when proofs actually need to be re-conducted.

§3.3 Resolve in the Verification Research Community

The Resolve tool suite, currently being developed in parallel at Clemson University and Ohio State, adheres to the principles enumerated above. A broad overview is available as a journal article [66]. A deeper analysis of the specific tools in use at Ohio State appears in a recent Ph.D dissertation [3], and in Chapter 5 of this work, wherein we discuss proof automation. Here, we elaborate some notable achievements of the Resolve research effort at OSU, and relate them to the state of the art in the broader verification community.[16]

The examples we present here serve not only to demonstrate Resolve's suitability to the task of software verification, but also to establish that the community at large has not sufficiently embraced abstraction. Violations of abstraction will be revealed not only in competing verification technologies, but also in the statements of verification challenge problems themselves.

§3.3.1 The First Verified Software Competition

At the 2010 conference on Verified Software: Theories, Tools, and Experiments, members of the Verified Software Initiative organized a competition to pit different verification tools against each other. After the competition, the participating research teams remained in dialogue and collaborated on an award-winning experience report detailing their approaches to the competition's five verification challenges [45]. This author was not present at the competition itself, but was the sole representative of

[16] Of course, verification is a wide-ranging discipline encompassing many approaches, some of which have little to do with the specific methods discussed thus far. For example, in embedded systems, control systems, and other relatively small domains of computing, a powerful technique called *model checking* is employed extensively, as opposed to the "theorem proving"-based methods of Resolve and other attempts at meeting Hoare's Grand Challenge [37].

the Resolve research group in the aforementioned experience report. We formulated solutions to all five of the competition's verification challenges in Resolve—only six out of 11 groups were able to do so. Some interesting details are presented below, and our full solutions are available in Appendix B.

§3.3.1.1 Modeling Arrays in Resolve

The first two problems of the Verified Software Competition involved programming with arrays. We quote all problem statements verbatim from [45]:

> **Problem One**: Given an `N`-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array. Prove the postcondition that `sum <= N * max`.
>
> **Problem Two**: Invert an injective (and thus surjective) array `A` of `N` elements in the subrange from `0` to `N-1`. Prove that the output array `B` is injective and that `B[A[i]] = i` for `0 <= i < N`.

The statement of the first problem is somewhat mystifying. Consider the following purported implementation:

```
1 procedure FindMaxAndSum (restores a: Array,
2                          replaces max: Integer,
3                          replaces sum: Integer)
4     Clear(max)
5     Clear(sum)
6 end FindMaxAndSum
```

This code doesn't even inspect `a`, and yet it satisfies the only postcondition that the problem demands, since clearing an `Integer` in Resolve makes its value zero. `sum <= N * max` is *not* a pertinent postcondition for `FindMaxAndSum`, but rather a trivial corollary of `FindMaxAndSum`'s correctness. Although this implementation is clearly contrary to the spirit of the problem, one is left to wonder whether some other

implementation, which seems to compute the max and the sum but fails due to a subtle error, might also "accidentally" satisfy the `sum <= N * max` postcondition. The failure to distinguish genuine specifications of code correctness from mathematical corollaries will be analyzed further in Chapter 5, when we investigate the question of what automated tools should and should not be able to prove.

Setting aside the underspecification of Problem One, we now discuss Resolve's solutions to these two problems. From the perspective of strict mathematical abstraction, arrays do not comprise an interesting new theory. Arrays are of primary importance in many programming languages partly for historical reasons and partly for their performance characteristics (*i.e.*, constant-time random access), but the functionality they provide is easy to express mathematically with existing theories. Interestingly, while the first problem is easy to express in a string-based model, the specification of problem two is arguably more elegant when modeling arrays as sets of (index, value) pairs.

This is a form of modularity not yet encountered, but also well-accommodated by Resolve: mathematical models of data ("concepts") are interchangeable just like their implementations ("realizations"). Of course there are dependencies involved: when, for example, a loop invariant in client code needs to talk about the value of an array, it must commit to one (and only one) of these models. There is thus not a single point of control over model change in client code,[17] but the different dimensions of expressiveness alternative models offer make them useful in some settings. Model changes should rarely if ever be necessary; for these two problems we chose to perform a model change just as a demonstration of this facility in Resolve. Appendix B depicts

[17]Unfortunately, nor is there a single point of control over model change in code that implements a concept (*i.e.*, the "data representation"). Data representations will be discussed in the next chapter.

the two different models of arrays in full detail, as well as client code that solves the challenge problems themselves.

In both models, integers are used to represent the upper- and lower-bounds of the array. This is a degree of abstraction not offered by the arrays built in to most mainstream languages (*e.g.*, C/C++): it is useful to choose a lower bound of 1, for example, when using arrays as a constant-time random access representation of binary trees, so that for any node with index i, indices $2i$ and $2i+1$ can be reliably computed to obtain the location of the left and right children.

In addition to the subtleties of array modeling, these two challenge problems bring to light another important aspect of Resolve verification: the use of universal algebraic lemmas about math definitions to help prove VCs without needing to reason about quantifiers. Consider for example the VC shown in Fig. 3.2, which arises from state 17 of `FindMaxAndSum`, wherein the loop invariant must be proven to be true at the end of the loop body.

Fig. 3.2 involves a primitive of string theory not yet encountered: the "substring" function. Its name is written in lower-case to distinguish it from specifier-supplied definitions such as `SUM` and `IS_MAX_OF`. substring is parameterized by a string, a starting position, and an ending position. The substring it identifies includes the entry at the starting position, but not the one at the ending position, *e.g.*, $\text{substring}(\langle \alpha, \beta, \gamma, \delta \rangle, 1, 3) = \langle \beta, \gamma \rangle$. It is a total relation (all mathematical definitions in Resolve are)—its value is an empty string in ill-defined cases, for example:

$$
\begin{aligned}
&(a_0.lb \leq a_0.ub + 1 \\
&\wedge\ |a_0.s| = (a_o.ub - a_0.lb) + 1 \\
&\wedge\ a_0.s \neq \langle\rangle \\
&\wedge\ \text{substring}(a_0.s, a_0.lb - a_0.lb, (a_0.lb - a_0.lb) + 1) = \langle max_7 \rangle \\
&\wedge\ \text{substring}(a_0.s, a_0.lb - a_0.lb, (a_0.lb - a_0.lb) + 1) = \langle sum_8 \rangle \\
&\wedge\ count_{10} \leq a_0.ub \\
&\wedge\ a_0.lb \leq count_{10} \\
&\wedge\ count_{10} \leq a_0.ub + 1 \\
&\wedge\ \texttt{IS\_MAX\_OF}(\text{substring}(a_0.s, 0, count_{10} - a_0.lb), max_{10}) \\
&\wedge\ 0 \leq (a_0.ub - count_{10}) + 1 \\
&\wedge\ \text{substring}(a_0.s, count_{10} - a_0.lb, (count_{10} - a_0.lb) + 1) = \langle value_{12} \rangle \\
&\wedge\ max_{10} < value_{12}) \\
&\quad \Longrightarrow \\
&\texttt{SUM}(\text{substring}(a_0.s, 0, count_{10} - a_0.lb)) + value_{12} = \\
&\texttt{SUM}(\text{substring}(a_0.s, 0, (count_{10} + 1) - a_0.lb))
\end{aligned}
$$

Figure 3.2: A `FindMaxAndSum` verification condition.

$$\begin{aligned}
&\text{substring}(\langle\alpha,\beta,\gamma,\delta\rangle,1,1)\\
=&\text{substring}(\langle\alpha,\beta,\gamma,\delta\rangle,1,99)\\
=&\text{substring}(\langle\alpha,\beta,\gamma,\delta\rangle,-3,1)\\
=&\text{substring}(\langle\alpha,\beta,\gamma,\delta\rangle,3,1)=\langle\rangle.
\end{aligned}$$

Now in order to prove the VC shown in Fig. 3.2, we should find the antecedent clauses ("givens") that are actually relevant to the truth of the consequent, and then use definitions of and/or properties about the things mentioned in order to show that the consequent actually holds. The penultimate given is the linchpin of the proof: the fact that the entry in $a_0.s$ at position $(count_{10} - a_0.lb)$ is $value_{12}$, combined with our understanding of what `SUM` means, demonstrates that the two `SUM`s of the consequent really are equal. Discharging this VC with an achingly rigorous automated proof assistant such as Coq or Isabelle, however, would involve much pedantic complexity. The implicit definition of `SUM` would certainly have to be involved (thus introducing an existential quantification—a notoriously difficult structure to reason about automatically), as would the meaning of substring, expressed either as another implicit definition, or by axioms and theorems in string theory itself.

These kinds of issues—automatic reasoning involving quantifiers, selection of well-behaved math-theoretic primitives, etc.—are precisely what many research teams at the Verified Software Competition investigate most actively. Resolve takes a more abstract approach: we simply attempt to avoid as much proof complexity as possible by reasoning in terms of universal algebraic lemmas, which allow VCs involving math

definitions to be proven without expanding the definitions themselves and introducing quantifiers.[18]

Universal algebraic lemmas are another crystallization of the principle of proof reuse: if some property is useful to the proofs of multiple VCs, then reusability dictates that property should be proven once, and then applied without further proof wherever necessary. The idea of separate roles in the verified software process again seems wise: a math expert can work on developing and proving a rich set of lemmas, perhaps with bookkeeping assistance from a tool like Coq or Isabelle, completely independently of programming experts who assemble components and write algorithms to solve computational problems.

For the completeness of our discussion, we present in Fig. 3.3 two lemmas that enable a proof of Fig. 3.2's VC without expanding the definitions of `SUM` and substring. These particular lemmas are straightforward to prove, especially if an assistant like Coq is employed, but in general the development of useful lemmas for a theory is itself a critical aspect of abstraction-embracing software verification.

It should be clear how the lemmas in Fig. 3.3 could be used to show that Fig. 3.2's pertinent antecedents—particularly the second-to-last one, and a few others establishing that the integer arguments to substring are valid—establish its consequent.

A recent report by Resolve practitioners at Ohio State [73] empirically investigates the ramifications of lemma-based rewriting as opposed to expanding mathematical definitions, and concludes that lemmas offer advantages meriting their further investigation. This work is ongoing.

[18] We will conduct a more substantive discussion of Resolve's proof automation techniques in §5. We provide a glimpse here simply to contrast Resolve's approach with the others at the Verified Software Competition.

$$0 \leq start \wedge start \leq end \wedge end < |s| \Longrightarrow$$
$$\quad \text{substring}(s, start, end + 1) =$$
$$\quad\quad \text{substring}(s, start, end) * \text{substring}(s, end, end + 1)$$

$$\texttt{SUM}(s * \langle i \rangle) = \texttt{SUM}(s) + i$$

Figure 3.3: Two rewrite rules useful in the proof of `FindMaxAndSum`.

§3.3.1.2 Broken Abstraction in Problem Statements

> **Problem Three**: Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number `i` equal to the length of the list if there is no such element. Otherwise, the element at index `i` must be equal to zero, and all the preceding elements must be non-zero.

A recurring theme in the software verification literature, indeed a primary motivation for this dissertation, is a lack of proper emphasis on abstraction. Problem statements such as the one above are examples of this phenomenon. Why fix the representation of the list to be a linked-list pointer structure? With a proper abstraction and a smart choice of data representation, realizations of the list datatype not involving pointers can offer amortized performance that is equally efficient.[19] Why fix the type of data in the list to be integers? Isn't any type with an equality test reasonable for this task? Why fix the value the code searches for to be zero? Why fix the exceptional behavior (*i.e.*, what happens when zero isn't in the list)? In fact, if we read Problem Three legalistically, it doesn't even require a procedural abstraction at all—a main body that implements list searching apparently would have been an

[19]Indeed, the fifth problem of this very competition involved one such data representation.

acceptable solution, despite the fact that it would arguably demonstrate a disregard for basic principles of modern software engineering.

A more abstract reformulation of the problem, allowing solutions with a proper degree of genericity and modularity, might read as follows:

> **Problem Three, Emended**: Write a procedure to find the first occurrence of some caller-specified value in a list. Prove that it is correct.

Or, if we are specifically interested in the ability to prove code about indirection, *e.g.*, via pointers, we should write something like:

> **Problem Three, Emended and Amended**: Write a procedure to find the first occurrence of some caller-specified value in a list. Prove that it is correct. In addition, verify a linked-list realization of the list datatype.

The key observation of this last formulation is that linked lists have nothing to do with the task of searching. Ordered arrangements of data—of which lists are one example—are easily modeled mathematically, and searching is expressible in terms of this model. So, following the principles argued for thus far, strict abstraction should be adopted: list searching should be expressed as a contract written in terms of a mathematical model, an algorithm implementing searching can be verified relative to that contract, and as an *entirely separate* verification task, realizations of the datatype, such as linked lists, can be proven correct.[20] Appendix B shows a list template concept (*i.e.*, the entries are not fixed to be integers), an enhancement adding the ability to search for an arbitrary value (*i.e.*, the target is not fixed to

[20] An argument could be made for adding searching to the core "kernel" of functionality that the list datatype provides *if* having access to a particular representation such as a linked-list allowed the searching to be implemented more efficiently. This is not the case here: the list concept as specified in Resolve can be implemented as a linked-list (or in other ways), and searching can be programmed on a client-view without any negative impact of the asymptotic running time of the algorithm. This is evidence of the quality of Resolve's list concept.

be zero), and a verified implementation of that contract. The issues of specification and implementation of pointers in Resolve constitute ongoing and active research, which has generated some interest within the research community represented at the Verified Software Competition itself [47].

§3.3.2 The Resolve Verification Benchmarks

In [12], we argued that benchmark problems and formal competitions are effective and under-utilized methods for advancing the state of the art in the field of software engineering. Verification is a prime example of a discipline that stands to gain from these practices, as the Verified Software Competition demonstrates. Indeed, the Resolve research group has published a collection of benchmark problems [78] intended to spur tool development and encourage scalable design along the lines enumerated at the beginning of this section.

The benchmarks proposed by the Resolve group are intended to serve as early, incremental guideposts for verification systems that can remain practical as they are scaled up to larger problems. The first benchmark problem is to verify algorithms for simple arithmetic—adding and multiplying numbers—but later problems involve data representations, I/O streams, and finally an end-user application that integrates solutions to the previous problems as an exercise in modularity and reuse.

These benchmark problems were not cherry-picked for the purposes of Resolve proselytizing—indeed some of them remain unsolved in the current version of Resolve at Ohio State. Encouragingly, there has been some interest in the verification community in sharing solutions to these benchmarks, and also in proposing new ones.

Notably, the Dafny [49] group, affiliated with Microsoft Research, has published proposed solutions to all eight benchmark problems [50], although their solutions often made dramatically simplifying assumptions that arguably obscured the true spirit of the benchmarks [14]. These investigators have also formulated an interesting set of new problems focusing on data structure invariants [51], and several other groups have responded in turn to these challenges. Thus, Resolve is currently playing an active role in the ongoing discussion and comparison of qualitatively different verification approaches in the research community. We now elaborate on two such approaches for the sake of contrast.

§3.3.3 Dafny

In [14], we reviewed the benchmark solutions proposed by the Dafny group at Microsoft Research, and discussed modularity and abstraction issues of the sort emphasized in this work.

Dafny is a programming language whose semantics is defined by translation into an intermediate language called Boogie [5]. Boogie verification conditions can be discharged by Microsoft's Z3 proof tool [57]. The Dafny language features Java-like reference semantics and a unique selection of primitives, including generic `set` and `seq` collection types that serve both as programmatic entities and as quasi-mathematical models of other container types. `set` and `seq` are interesting conflations of domains that Resolve keeps strictly separate, and there are some serious ramifications to this entanglement.

Consider as an example the Dafny `Array` component shown below.

```
1  class Array {
2    var contents: seq<int>;
3
4    method Init(n: int);
5      requires 0 <= n;
6      modifies this;
7      ensures |contents| == n;
8
9    function Length(): int
10     reads this;
11   { |contents| }
12
13   function Get(i: int): int
14     requires 0 <= i && i < |contents|;
15     reads this;
16   { contents[i] }
17
18   method Set(i: int, x: int);
19     requires 0 <= i && i < |contents|;
20     modifies this;
21     ensures |contents| == |old(contents)|;
22     ensures contents[..i] == old(contents[..i]);
23     ensures contents[i] == x;
24     ensures contents[i+1..] == old(contents[i+1..]);
25 }
```

Despite the field declaration and apparently programmatic mentions of the `seq` called `contents`, the Dafny methodology considers components such as `Array` to be interfaces. However, a `seq` is not a purely abstract mathematical entity used only to describe the functionality of `Array` operations; instead it is a real programmatic object that can only be mentioned in specifications if it is an actual field of a particular `Array` representation. In other words, there is no `is modeled by` slot in Dafny, and so fields that comprise a datatype's "abstract" value are intermixed with real representation fields. Dafny's workaround is to allow certain variables to be "ghosted," meaning that they can be declared and mutated, but they are not actually allocated at runtime,

and no actual program behavior should depend on them. Ghost variables exist in Resolve as well, where they are known as "adjunct" variables, but are used in very limited circumstances—their use in Dafny is pervasive, and displaces a more modular approach to data representation, which will be discussed in the next chapter.

Notice that the `Array` operations `Length` and `Get` are specified not with contracts, but rather with expressions involving `contents`. These are called "pure methods," because they are side-effect free and can be expressed in a manner analogous to Resolve's explicit definitions. However, unlike Resolve definitions, they are also intended to be executable; implementers don't override them. Thus quasi-mathematical types such as `seq`s must be programmatically realized in any implementation of Dafny, which of course provokes questions of efficiency, and precludes interchangeability of different implementations, thus hindering abstraction.

In addition to our review of Dafny's benchmark solutions in [14], we also investigated the relationship between Dafny's infrastructure and issues of abstraction in [13]. One glaring problem enunciated in that work is the fact that Dafny specifications seem to use `==` to denote equality of values, but for user-defined types `==` computes only *reference* equality. For example, `a1 == a2` would not be a correct assertion of the equality of two `Array`s in Dafny, because `a1 == a2` is true iff `a1` and `a2` are references to the exact same portion of memory. A client cannot consistently think of variable values as mathematical entities in such a setting; a severe abstraction problem.

§3.3.4 Jahob

Jahob [79] is a verification system that targets the Java programming language, attempting to handle the maximum possible amount of functionality that industrial-strength Java offers. This approach differs markedly from the limited linguistic domains of Resolve and Dafny, for example. In [79], the authors elaborate their efforts in verifying linked data structures, including a linked-list implementation of Java's `HashMap` interface. We identified substantial modularity problems in the Jahob solution in [13].

As published in [79], Jahob's verification of `HashMap` is actually unsound. The issue has to do with the fact that, because of Java's reference semantics, clients can define a `(key, value)` mapping and retain an alias to `key`. Then, `key` can be mutated in a way that changes its `hashCode`, thus breaking the representation invariant stating that all `(key, value)` pairs reside in the bucket corresponding to their `hashCode`. The Jahob authors revealed in personal communication with our research group that the interface of `Object`, the most fundamental superclass of Java, was changed to guarantee hash code immutability—a profound deviation from Jahob's goal of verifying "real" Java code, and also a serious abstraction issue, since proper understanding of `HashMap`'s specification now requires inspecting a redefinition of `Object`.

§3.4 Conclusion

Having now presented the Resolve programming language, the tabular verification method based upon decoupled semantics, and comparisons and contrasts among Resolve and other state of the art verification systems, we should pause to consider what technical issues remain to be discussed. The discussions of Dafny and Jahob

clearly indicate that verifying data representations, as opposed to client code that treats datatypes as black-box concepts, is a subtle issue that can easily introduce abstraction problems or all-out unsoundness. In the next chapter, we will discuss Resolve's approach to proofs of correctness for data representations, and see that this approach is not universally applicable. Specifically, we will return to the issue of functional programming languages, first glimpsed in Chapter 2, and discover that a functional variant of Resolve would be unsound if the approach to data representation verification were left unchanged. We will propose a solution to this serious theoretical problem.

Chapter 4: Modular Program Verification, Take Two: Realization Programming

§4.1 Verifying Realizations in Resolve

Having completed our presentation of "client view" programming (and verification) in Resolve—that is, the style of programming that uses data abstractions but does not provide code to make them work—we are now ready to investigate the "implementer's view," in which new data abstractions are designed and realized programmatically.

Chapter 1's contract for `Queue` was our first example of a data abstraction. Being a contract component, it describes queues at the level of abstraction appropriate to clients: it defines what a queue can do without showing how that behavior is accomplished algorithmically. That contract describes the queue *kernel*, which is a term connoting the core functionality of a data abstraction, rather than its *enhancements*, *e.g.*, an operation for sorting. A kernel contract conveys three essential pieces of information:

(1) The mathematical model of the component in question (`Queue` is modeled by a string of items, where "item" refers to the mathematical model of the template parameter).

(2) A predicate that is true for initial values of the type (The value of an initial `Queue` is the empty string).

(3) Signatures and contracts for the kernel operations (The kernel operations for `Queue` are `Enqueue`, `Dequeue`, `Length`, and `IsEmpty`).

Also in Chapter 1, we saw one of many possible *realizations* (also known as *implementations*; we will use the terms interchangeably) of `QueueTemplate`. It used a `List` object to simulate the behavior of `Queue`, and contained a *correspondence function*, which states the mathematical relationship between the kernel realization's `List` and the abstract `Queue` value it purports to represent. The correspondence function is never executed; it is a *mathematical* function. There is never a time when the `List` is computationally converted into a `Queue`.

We wish to describe the process of verification for such data representations. Although it would be possible to conduct this discussion with respect to the `List` representation of `Queue`,[21] we will instead focus on a different, more interesting representation—one that uses two `Stack`s, one of which holds its data in reverse order, to represent a `Queue` efficiently. This component comprised Resolve's solution to the fifth VSComp challenge problem [45]. We will describe the realization and its verification in detail, and then proceed to show that this same data representation verification process has a serious problem if we apply it to functional programming languages of the type glimpsed in Chapter 2.

Before we show the implementation of `Queue` that uses two `Stack`s, we must explain what a `Stack` is. Again, modularity demands that a client's view of `Stack` is

[21] Indeed, the current Resolve tools at OSU verify the `List` realization of `Queue` fully automatically.

sufficient here: `Stack`'s implementation is free to vary independently of the behavior clients observe because there is a strict abstraction boundary between client and implementer, *i.e.*, between the mathematical and the programmatic view of `Stack`'s functionality. In other words, verification of kernel realizations uses decoupled semantics.

The listing below shows the contract for `Stack`. It is similar to `Queue`, but its operations for adding and removing data—`Push` and `Pop`, respectively—behave in a "last in, first out" manner, unlike the "first in, first out" behavior of `Enqueue` and `Dequeue`.

```
1  contract StackTemplate (type Item)
2
3      uses UnboundedIntegerFacility
4
5      math subtype STACK_MODEL is string of Item
6
7      type Stack is modeled by STACK_MODEL
8          exemplar s
9          initialization ensures
10             s = empty_string
11
12     procedure Push (updates s: Stack, clears x: Item)
13         ensures
14             s = <#x> * #s
15
16     procedure Pop (updates s: Stack, replaces x: Item)
17         requires
18             s /= empty_string
19         ensures
20             #s = <x> * s
21
22     function IsEmpty (restores s: Stack): control
23         ensures
24             IsEmpty = (s = empty_string)
25
```

```
26      function Length (restores s: Stack): Integer
27          ensures
28              Length = |s|
29  end StackTemplate
```

The `Queue` realization we will use to explain proofs for correctness of data representations in shown below. Some contracts that the implementation depends on, *i.e.*, `Reverse` and `Concatenate` for `Stack`, are not shown, but are straightforward and informally explained by their names. The body of `FixThings` is omitted, and is unnecessary for verification of this kernel implementation according to decoupled semantics.

```
1   realization TwoStacks implements QueueTemplate
2
3       uses StackTemplate
4       uses Reverse for StackTemplate
5       uses Concatenate for StackTemplate
6       facility StackFacility is StackTemplate (Item)
7
8       type representation for Queue is (
9               front: Stack, back: Stack, length: Integer
10          )
11          exemplar q
12          convention
13              |q.back| <= |q.front| and
14              q.length = |q.back| + |q.front|
15          correspondence function
16              q.front * reverse (q.back)
17      end Queue
18
19      local procedure FixThings (updates s1: Stack,
20                                 updates s2: Stack)
21          ensures
22              |s2| <= |s1| and
23              s1 * reverse (s2) = #s1 * reverse (#s2)
24          ...
25      end FixThings
```

```
26
27      procedure Enqueue (updates q: Queue, clears x: Item)
28          Push (q.back, x)
29          FixThings (q.front, q.back)
30          Increment (q.length)
31      end Enqueue
32
33      procedure Dequeue (updates q: Queue, replaces x: Item)
34          Pop (q.front, x)
35          FixThings (q.front, q.back)
36          Decrement (q.length)
37      end Dequeue
38
39      function Length (restores q: Queue): Integer
40          Length := Replica (q.length)
41      end Length
42
43      function IsEmpty (restores q: Queue): control
44          variable zero: Integer
45          IsEmpty := AreEqual (q.length, zero)
46      end IsEmpty
47
48  end TwoStacks
```

§4.1.1 Convention and Correspondence

A kernel implementation in Resolve defines a tuple of concrete values, a *representation*, that will be manipulated programmatically to simulate the behavior the kernel contract describes mathematically. The syntactic slot for this tuple definition is marked by the keywords `type representation`. There are two additional crucial annotations unique to implementer's view, the aforementioned correspondence function, which links concrete states to the abstract values they represent,[22] and the

[22]In general, relations are known to be necessary here [68], but currently the OSU Resolve tools only support functions, which constitute the more common case. The proof strategy we will present for data representations is known to accommodate abstraction relations [32, 21]. We discuss some issues for the automation of proofs of data representations that use abstraction relations in Chapter 6.

convention, which is a formal restriction of the legal concrete states. The convention acts as an additional requires and ensures clause for each kernel operation: it is assumed to be true as a precondition to any kernel operation implementation, and in order to be correct, each kernel operation implementation must satisfy the convention as a postcondition. Conventions are usually used to assist the kernel implementer in devising simpler code by ruling out problematic cases. In Chapter 1's `List` representation of `Queue`, the convention stated that the left side of the `List` was empty, which simplified the implementation of (amongst other things) `Length` by eliminating the need to sum the lengths of both the `List`'s sides. Notice that conventions are essentially a concrete state analog of the `constraint`s we saw in Chapter 2 for restricting the legal values of a mathematical subtype.

For the `TwoStacks` representation, the convention states that the `back Stack` must be at most as long as the `front Stack`. This assumption actually doesn't help simplify the kernel operation implementations, but rather helps (potentially) to improve the performance of the `Queue` relative to some restrictive assumptions that the VSComp organizers imposed. Specifically, this challenge problem concerned implementing a `Queue` with good amortized performance using two singly-linked lists. Although Resolve doesn't preclude the ability to use pointers [47], a more direct solution in our framework was to use `Stack`s, which have the same functionality as the challenge problem assumed for linked lists, namely the abilities to prepend a new element onto a stack, concatenate two stacks, and reverse the order of a stack's entries.

To see how the solution requested in VSComp only *potentially* offers a performance advantage, we should discuss some non-functional behavior of the data representation. The first key assumption to a performance analysis of this implementation is that the `Stack` kernel operations `Push` and `Pop` run in constant time. The straightforward singly-linked list implementation of `Stack`—omitted harmlessly, thanks to strict abstraction and decoupled semantics—does indeed offer this performance profile. Furthermore, observe that representing a queue as a singly-linked list would *not* allow the `Queue` kernel operations to all have constant running times: only one end of the list is directly accessible. So the problem concerns leveraging `Stack` to somehow speed up `Queue`. This is where the convention and correspondence come into play. The correspondence states that we can represent a `Queue` by storing its entries in two `Stack`s, the second of which stores its entries in reverse order (relative to their order in our model of the `Queue`: a string). This is advantageous because now an `Enqueue` can be simulated by `Push`ing the argument onto `back`—a constant-time task. `Dequeue` is likewise just a `Pop` off of `front`.

There are subtle dangers involved in this approach, which the convention addresses. If there were no convention constraining the legal representation values, `front` could be empty on some calls to `Dequeue`, and just `Pop`ping off of `back` instead would be incorrect, since its entries are in reverse order. The convention protects against this problematic case by requiring that the length of `front` always be at least the length of `back`. Thus, `front` is only empty if there are no entries at all in the `Queue`—precisely the case that `Dequeue`'s precondition disallows.

It now seems that `Enqueue` and `Dequeue` can both have constant-time performance, save for the fact that sometimes entries will have to be moved from the front to the

back (or vice versa), which is what the local helper operation `FixThings` is for: if its first argument is too short, it concatenates the reverse of its second argument onto the first one, and clears the second argument. So when a kernel operation calls `FixThings (q.front, q.back)`, the lengths of the `Stack`s are altered so that they satisfy the convention, and the order of their entries, interpreted as a `Queue` by the correspondence, is preserved. The final non-functional assumption that caused the VSComp organizers to refer to this as a good "amortized complexity" implementation of `Queue` is that `FixThings` is the only place where a linear-time algorithm might be necessary: the tasks of reversing and concatenating singly-linked lists—`Stack`s, in our solution—require linear running time. The idea is that over the course of a `Queue`'s usage, things will not have to be fixed too dramatically, particularly in the common use case that interleaves many of the calls to `Enqueue` and `Dequeue`.

The fifth VSComp challenge problem is yet more evidence for a lack of proper emphasis on abstraction in the larger verification community. It is Resolve's approach, not the statement of the problem itself, that evinces a proper separation between the roles of client and implementer, and thus between functional and nonfunctional requirements. Due to our verification methodology's use of decoupled semantics, we can omit the code for `FixThings` entirely and never consider what its running time might be. When clients *do* care about performance, it can be expressed without revealing concrete details. A new kind of component might be proposed to address this issue—one which expresses the running time of an algorithm in big-oh notation without showing its code, for example. Nothing of the sort was done here, but Resolve's emphasis on modularity suggests such a component-based solution.

§4.1.2 The Classical Proof Approach

The tabular approach of the previous chapter can be employed for verification of kernel implementations, but it must be modified to reflect the fact that the correspondence function intercedes between the programmatic values the code actually manipulates and the abstract values of the kernel type, in terms of which the contract is written. The correctness criterion was first formulated in [36], and then later revised in [32] to account for an incompleteness issue. The authors of [21] build on this work by incorporating preconditions, conventions, and constraints, and establishing the soundness and relative completeness of their approach in a setting very similar to that of Resolve. [68] exemplifies the approach by verifying a complex data abstraction involving an abstraction relation in Resolve. We will thus characterize the Resolve methodology for proofs of correctness of data representation as "classical" due to its rich history in the literature.[23]

To facilitate an explanation of the "classical" proof rule, we first define a straightforward notation for composition of relations:

$$\text{Given } P \subseteq S \times T, Q \subseteq T \times U, \text{ and } R \subseteq T,$$

$$P \circ Q \stackrel{\text{def}}{\equiv} \{(x, z) \mid \exists y \in T,\ (x, y) \in P \wedge (y, z) \in Q\}$$

$$R \circ Q \stackrel{\text{def}}{\equiv} \{(y) \mid \exists x \in T,\ (x) \in R \wedge (x, y) \in Q\}$$

Now, at an intuitive level, a proof of correctness for a data representation consists of showing that for each kernel operation implementation $\texttt{op}$, $\boldsymbol{Sem}(\texttt{op}) \circ AR \subseteq AR \circ \boldsymbol{Spec}(\texttt{op})$, where $\boldsymbol{Sem}(\texttt{op})$ is the decoupled semantics of the kernel operation

[23]It should be noted that none of these cited works make use of a tabular reasoning method as has been seen in this work so far. However, Heym's aforementioned bridge rule [33] links tabular reasoning to the more standard backwards reasoning approach in a manner that is proven to preserve soundness and relative completeness.

implementation, ***Spec***(`op`) is the operation's specification, and AR is the abstraction relation. We hasten to remark that this formula is inaccurate in a subtle, technical sense that we will discuss, but it is still useful for the purposes of high-level explication: it states that any abstract value corresponding (according to the abstraction relation) to a concrete state that the semantics of the operation (*i.e.*, the code implementing it) results in must be a value that the specification allows as a result, relative to any abstract value our starting concrete state corresponded to. This rather nuanced property is perhaps best clarified by a *commutative diagram*, shown in Fig. 4.1.

The commutative diagram depicts visually the core criterion for valid kernel operation implementations: that they remain inside the envelope of behavior that the operation specification allows. The concrete state space—the implementer's view of the kernel—before and after the operation is executed is shown on the bottom-left and bottom-right, respectively. The abstract state space—the client's view—is shown above each of these. It must be the case that, for any legitimate "concrete pre state" (the shaded region on the bottom left), the operation implementation results in a concrete result that corresponds to an abstract result allowed by the spec. The term "legitimate" in the previous sentence has a rigorous meaning according to [68]. Specifically, we only need to consider concrete pre states that satisfy the convention and correspond to abstract values that satisfy the operation's precondition. Moreover, the commutative diagram as shown fails to properly express the role of the convention; as we've said, the convention is assumed to be true in the concrete pre state, and the operation implementation must only yield concrete post states that satisfy the convention. The lack of accounting for preconditions, conventions, and constraints is

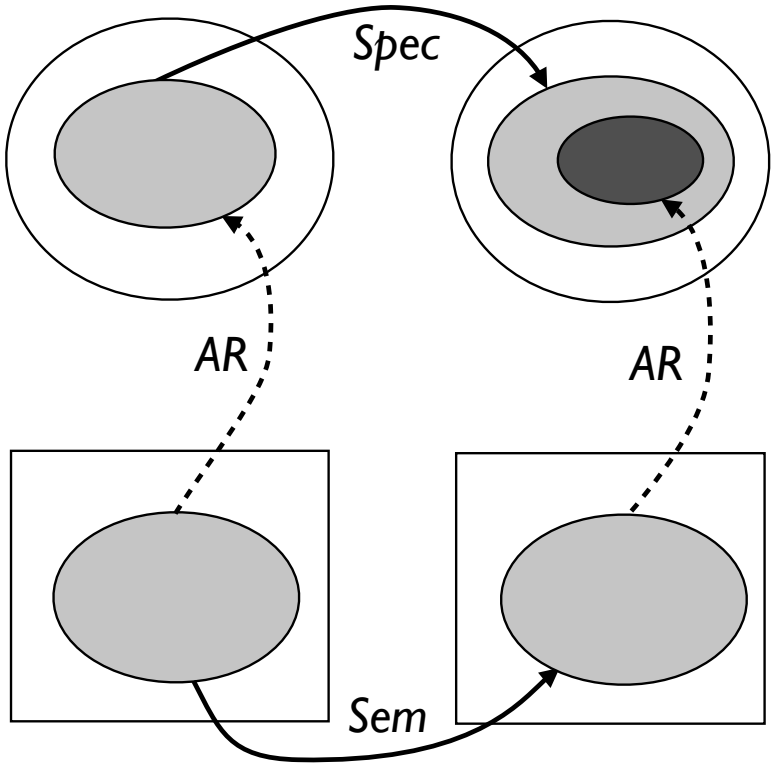

Figure 4.1: A commutative diagram illustrating the proof rule for correctness of data representations.

the sense in which the "classical" rule $\boldsymbol{Sem}(\texttt{op}) \circ AR \subseteq AR \circ \boldsymbol{Spec}(\texttt{op})$ is inaccurate for Resolve.

If the abstraction relation is a function, we can verify operation implementations in a kernel by simply applying the normal tabular approach, with a few modifications to account for the special syntactic slots of kernel implementations, and their meanings:

(1) In the table's first state, include the convention as a fact.

(2) In the table's last state, include the convention as an obligation.

(3) In the table's first state, for each operation parameter of the kernel's type, add as a fact an equality between that parameter and a representation tuple.

(4) In the table's last state, for each operation parameter of the kernel's type, add as a fact an equality between that parameter and a representation tuple

The first two modifications cause the convention to serve as a *representation invariant* consistently throughout the kernel, the last two are direct applications of the correspondence precisely in the states where the table makes mention of the operation's contract, and thus needs a connection between the representation(s) and the abstract value(s) of the kernel-type parameter(s). The abstract value of kernel-type parameters (`Queues`, in our example), will not be mentioned in intermediate states of the table, only fields of the representation. Notice that there are no distinguished parameters in Resolve, so it is possible for more than one parameter of the kernel's type to appear in a kernel operation's signature; our discussion of kernel implementation correctness can be generalized easily to such cases.

A kernel implementation is correct if all of its operations can be verified with this modified tabular approach and any of its helper operations (like `FixThings`) can be

verified with the original tabular approach of Chapter 3.[24] To complete our view of kernel verification in Resolve, we show the tracing table for one of `Queue`'s kernel operations, `Dequeue`:

State	Path Conds	Facts	Obligations
0		$q_0 \neq \langle\rangle$ $q_0 = q.front_0 * \text{reverse}(q.back_0)$ $\lvert q.back_0\rvert \leq \lvert q.front_0\rvert$ $q.length_0 = \lvert q.back_0\rvert + \lvert q.front_0\rvert$	$q.front_0 \neq \langle\rangle$
`Pop (q.front, x)`			
1		$q.front_0 = \langle x_1\rangle * q.front_1$ $q.length_1 = q.length_0$ $q.back_1 = q.back_0$	
`FixThings (q.front, q.back)`			
2		$\lvert q.back_2\rvert < \lvert q.front_2\rvert$ $q.front_2 * \text{reverse}(q.back_2) =$ $\quad q.front_1 * \text{reverse}(q.back_1)$ $q.length_2 = q.length_1$ $x_2 = x_1$	
`Decrement (q.length)`			
3		$q.length_3 = q.length_2 - 1$ $q.front_3 = q.front_2$ $q.back_3 = q.back_2$ $x_3 = x_2$ $q_3 = q.front_3 * \text{reverse}(q.back_3)$	$q_0 = \langle x_3\rangle * q_3$ $\lvert q.back_3\rvert \leq \lvert q.front_3\rvert$ $q.length_3 = \lvert q.back_3\rvert + \lvert q.front_3\rvert$

Notice that the convention appears as the third and fourth facts in state 0, and the correspondence appears as the second fact. In state 3, the final state, the correspondence appears as the fifth fact and the convention appears as the second and third obligations. Everything else in the table is derived by the usual tabular approach as described in Chapter 3.

[24] [21] also gives two proof obligations that are not specific to any particular kernel operation implementation: every concrete state satisfying the convention must correspond to some abstract value, and every abstract value in the image of the abstraction relation should satisfy any constraints that were made on the abstract state space. These obligations are trivially satisfied in most reasonable kernel implementations, and are not crucial to our discussion here. We address these issues in more detail in Chapter 6.

§4.2 The Soundness Problem

Although the tabular method of verification modified for kernel implementations is sound and relatively complete in Resolve, there are nontrivial semantic issues in play which make this so. To facilitate an analysis of these subtleties, we will now demonstrate that the approach seen in the previous section actually yields *unsoundness* in a slightly different but entirely reasonable computational setting.[25]

In addition to clarifying crucial terms and proving the unsoundness of a seemingly reasonable verification approach, we will also propose a new solution to the problem.

§4.2.1 Functional Semantics and Referential Transparency

The slightly different setting for verification we will consider is that of a *functional* programming language. Recall from Chapter 2 that in a functional programming language, all operations behave as functions. That is, they leave their arguments unchanged and return a value. More specifically, we will consider *pure* functional languages, which lack the notion of state altogether. In a pure functional language, every operation, when executed, behaves as a true mathematical function—its return value is determined entirely by its arguments' values, and all calls to the same operation with the same arguments yield the same result. We will refer to this property as *functional semantics*.

A key advantage to pure functional programming is the property of *referential transparency*, first enunciated by Quine [63] and defined by him as follows: "I call a mode of containment ϕ referentially transparent if, whenever an occurrence of a singular term t is purely referential in a term or sentence $\psi(t)$, it is purely referential

[25]Throughout this section, we will often quote verbatim and at length from the latest revision [11] of our technical report [10], currently in review.

also in the containing term or sentence $\phi(\psi(t))$". In formalizing this idea for programming languages, Søndergaard and Sestoft [71] conclude by saying informally, "[A]n operator is referentially transparent if it preserves applicability of Leibniz's law, or substitutivity of identity: the principle that any subexpression can be replaced by any other equal in value." Reasoning on the basis of referential transparency is a major practical assumption of functional languages. For example, it is used in refactoring: if `not` is a negation operation for some user-defined type that behaves like a boolean, then `(not (not b))` should be able to be rewritten as just `b` without changing the meaning of the program. It might seem that referential transparency is an immediate and unequivocal consequence of functional semantics, yet this is not quite so. We will show that referential transparency is unsound in any language that combines three common and useful features: (1) functional semantics, (2) the ability to perform data abstractions of the sort seen previously in this work (these are sometimes called *abstract data types*, or just ADTs), and (3) the ability to write specifications that are relations and not functions (operation contracts in Resolve, for example, grant this ability).

Referential transparency seems to lie at the heart of reasoning about (and thus verifying) functional programs. Reasoning about software behavior by applying referential transparency *when it is not valid to do so* entails unsoundness: one can predict that two values are equal, whereas in fact they are not when the program is executed. On the other hand, reasoning about software behavior by failing to apply referential transparency *when it is valid to do so* entails a troublesome incompleteness: one cannot predict that two values are equal,[26] whereas in fact they are when the program

[26]This is not the same as predicting that the two values are unequal.

is executed. In this sense our analysis raises a dilemma for programming language and verification system designers: soundness *vs.* relative completeness. It might seem at first that the answer to this dilemma is relatively easy: of course one should give up relative completeness rather than soundness, lest verification yield false positives. But giving up relative completeness here has serious practical consequences. It entails not being able to rely on referential transparency—even though in some cases it might be valid to rely on it—and hence relinquishing various means of simplified reasoning it provides. The solution we will formulate seeks to restore the soundness of referential transparency by establishing the criteria for its valid usage, instead of just concluding that referential transparency must be abandoned outright in the face of our unsoundness example, which conclusion would yield relative incompleteness.

§4.2.2 A Small, Finite Example

Our first example shows the unsoundness problem in a very concise, albeit slightly contrived, form. This serves to isolate the essential roots of the problem (functional semantics, relational specifications, and data abstraction) from inessential complexities such as operation preconditions, non-functional abstraction relations, infinite value domains, recursion, etc.

Consider first an ADT called `Z4`, which is modeled by natural numbers modulo 4 (we write this set as $\mathbb{Z}_4$), and which implements a small amount of basic arithmetic for them. Since the domain of this ADT contains only four values, we may conveniently express the specifications of these operations as explicit input/output relations rather than with `requires` and `ensures` clauses. For an operation with one parameter (leading to a binary input/output relation), we use the two-tuple (x, y) to

denote that, on input x, y is an allowable return value; similarly, for an operation with no parameters (a unary relation), we use the one-tuple (y) to denote that y is an allowable return value. The `Z4` operation specifications are:

$$\boldsymbol{Spec}(\texttt{Zero}) \stackrel{\text{def}}{\equiv} \{(0)\}$$

$$\boldsymbol{Spec}(\texttt{One}) \stackrel{\text{def}}{\equiv} \{(1)\}$$

$$\boldsymbol{Spec}(\texttt{Half}) \stackrel{\text{def}}{\equiv} \{(0,0),(1,0),(2,1),(3,1)\}$$

$$\boldsymbol{Spec}(\texttt{Double}) \stackrel{\text{def}}{\equiv} \{(0,0),(1,2),(2,0),(3,2)\}^{27}$$

Now, suppose we wish to implement a new ADT called `Coin`, which has two possible values, corresponding to the two faces of a coin. A boolean obviously suffices as the mathematical model of `Coin`. As with the `Z4` ADT above, we give set-based specifications of the `Coin` operations:

$$\boldsymbol{Spec}(\texttt{Heads}) \stackrel{\text{def}}{\equiv} \{(\mathit{true})\}$$

$$\boldsymbol{Spec}(\texttt{Tails}) \stackrel{\text{def}}{\equiv} \{(\mathit{false})\}$$

$$\boldsymbol{Spec}(\texttt{MakeHeads}) \stackrel{\text{def}}{\equiv} \{(\mathit{true},\mathit{true}),(\mathit{false},\mathit{true})\}$$

$$\boldsymbol{Spec}(\texttt{Toss}) \stackrel{\text{def}}{\equiv} \{(\mathit{true},\mathit{true}),(\mathit{true},\mathit{false}),$$
$$(\mathit{false},\mathit{true}),(\mathit{false},\mathit{false})\}$$

[27] The observant reader will notice that the value 3 is not reachable by this interface. Another 0-ary operation called `three` could be added without changing the forthcoming example of unsoundness (a consequence of modularity)—it is omitted only for simplicity's sake.

Note that ***Spec***(`Toss`) is a relation and not a function, and is the only such specification in our example. So, for instance, `Toss (Heads ())` is permitted by the specification to return either *true* or *false*. However, if reasoning about a client program is to soundly leverage referential transparency, `Toss` must always return the *same* value whenever it is called with equal arguments: `toss`ing a *true* `Coin` must always yield the same result, and likewise for a *false* `Coin`.

Suppose now an implementer of `Coin` chooses to represent a `Coin` value using a `Z4` value that is conceptually interpreted modulo 2, where 0 represents *true* and 1 represents *false*. In other words, the abstract `Coin` value is realized concretely by a `Z4`, with a simple abstraction function establishing their correspondence. A realization for `Coin` can then be written as shown below.

```
1  realization Z4Realization implements CoinFacility
2
3      uses Z4Facility
4
5      type representation for Coin is (
6              value: Z4
7          )
8          exemplar c
9          correspondence function
10             ((c.value) mod 2) = 0
11     end Coin
12
13     function Heads () : Coin
14         Heads.value := Zero()
15     end Heads
16
17     function Tails () : Coin
18         Tails.value := One()
19     end Tails
20
21     function MakeHeads (restores c: Coin) : Coin
22         MakeHeads.value := Double(c.value)
```

```
23      end MakeHeads
24
25      function Toss (restores c: Coin): Coin
26          Toss.value := Half(c.value)
27      end Toss
28
29  end Z4Realization
```

Now, to show that this code correctly implements `Coin`, we should adopt the proof strategy introduced in this chapter. Because of the small, finite nature of the example, and the lack of any convention, constraint or operation precondition to consider, we can omit the tracing tables for each operation, and instead just demonstrate that the semantics of each kernel operation, interpreted through the correspondence, remains within the envelope of behavior allowed by the operation's specification. In other words, our proof obligations are precisely those demanded by the classical rule $\boldsymbol{Sem}(\texttt{op}) \circ AR \subseteq AR \circ \boldsymbol{Spec}(\texttt{op})$. In the following table, we expand the quantifier structure of this formula and explain its meaning informally. C and A denote the concrete and abstract state spaces, respectively. For our example, $C = \{0, 1, 2, 3\}$ and $A = \{true, false\}$.

$\forall c_1 \in C,$	For any concrete pre state,
$\quad \forall a_2 \in A,$	for any abstract value,
$\quad\quad (\exists c_2 \in \boldsymbol{Sem}(c_1),$	if the operation computes a result
$\quad\quad\ (c_2, a_2) \in AR) \Rightarrow$	corresponding to the abstract value,
$\quad\quad (\exists a_1 \in AR(c_1),$	then the concrete pre state must correspond to a
$\quad\quad (a_1, a_2) \in \boldsymbol{Spec}$	value that the spec maps to our abstract result.

Table 4.2: An expanded explanation of the classical proof rule.

The rule is obviously different in the case of 0-ary operations, for which there is no "pre-state," but this is a straightforward adjustment: just omit AR from the

right-hand side of the classical rule. Applying [32]'s proof rule to our `Coin` example, we see that the following four formulas are proof obligations, which, if true, establish the correctness of the purported `Z4`-based implementation:

(a) $\boldsymbol{Sem}(\texttt{Heads}) \circ AR \subseteq \boldsymbol{Spec}(\texttt{Heads})$

(b) $\boldsymbol{Sem}(\texttt{Tails}) \circ AR \subseteq \boldsymbol{Spec}(\texttt{Tails})$

(c) $\boldsymbol{Sem}(\texttt{MakeHeads}) \circ AR \subseteq AR \circ \boldsymbol{Spec}(\texttt{MakeHeads})$

(d) $\boldsymbol{Sem}(\texttt{Toss}) \circ AR \subseteq AR \circ \boldsymbol{Spec}(\texttt{Toss})$

Since each kernel operation body is a "one-liner" simply invoking a `Z4` operation, and because of our modular, decoupled semantics philosophy, ***Sem*** for each operation is just the specification of whatever `Z4` operation it invokes. For example, $\boldsymbol{Sem}(\texttt{Toss}) = \boldsymbol{Spec}(\texttt{Half})$. It is easy to establish all four proof obligations; the necessary intermediate calculations are given in Table 4.3:

Proof Obligation	Left-hand side	Right-hand side
(a)	$\{(true)\}$	$\{(true)\}$
(b)	$\{(false)\}$	$\{(false)\}$
(c)	$\{(0, true), (1, true),$ $(2, true), (3, true)\}$	$\{(0, true), (1, true),$ $(2, true), (3, true)\}$
(d)	$\{(0, true), (1, true)$ $(2, false), (3, false)\}$	$\{(0, true), (0, false), (1, true),$ $(1, false), (2, true), (2, false)$ $(3, true), (3, false)\}$

Table 4.3: Calculations establishing the correctness of our `Coin` implementation.

With these sets fully elaborated, we see by inspection that the proof obligations, each of which states the left-hand side is a subset of the right-hand side, are satisfied. The proof of correctness for this data representation is complete.

We are now in a position to see the unsoundness problem as an issue with reasoning based on referential transparency. Consider a client of `Coin`, working in a language with functional semantics. We present two different programming expressions, and show the decoupled semantics reasoning that our client could perform about them:

$$\texttt{Toss (MakeHeads (Heads ()))} \tag{4.1}$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\textbf{\textit{Spec}}(\texttt{MakeHeads})(\textbf{\textit{Spec}}(\texttt{Heads}))$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\textbf{\textit{Spec}}(\texttt{MakeHeads})(\mathit{true}))$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\mathit{true})$$
$$\texttt{Toss (MakeHeads (Tails ()))} \tag{4.2}$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\textbf{\textit{Spec}}(\texttt{MakeHeads})(\textbf{\textit{Spec}}(\texttt{Tails}))))$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\textbf{\textit{Spec}}(\texttt{MakeHeads})(\mathit{false}))$$
$$= \textbf{\textit{Spec}}(\texttt{Toss})(\mathit{true})$$

Having seen that both expressions are equal, referential transparency now dictates that either can be replaced with the other in any piece of program text without altering its meaning. This seems to be a natural consequence of functional semantics: the specifications allow `(toss` *true*`)` to return either one of two different values, but functional semantics demand that this value be *the same* each time the expression is evaluated. If the two expressions are not actually equal when the code is executed, such replacement is not allowed and thus referential transparency is unsound.

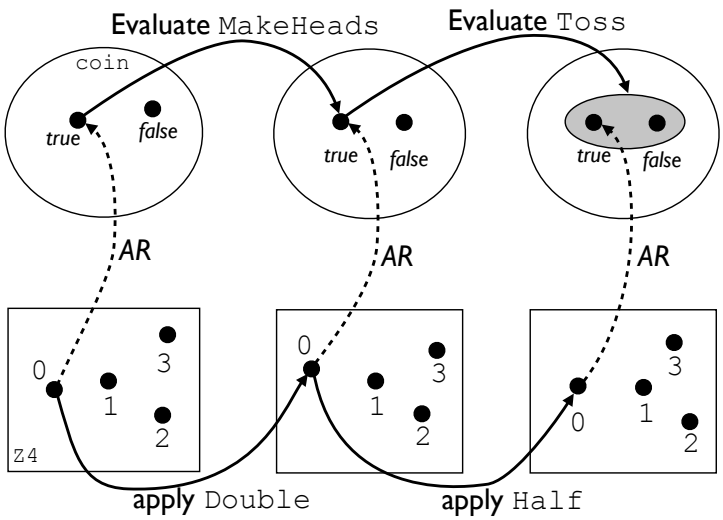

Figure 4.2: A step-by-step illustration of the evaluation of expression 4.1, showing the values in both the abstract and concrete state spaces. The shaded cloud in the final abstract state space indicates allowed relational behavior.

We now explain diagrammatically the unexpected runtime behavior of expressions 4.1 and 4.2. Execution of the bodies of `Toss`, `MakeHeads`, and `Heads` in expression 4.1 proceeds as illustrated in Fig. 4.2; for expression 4.2, see Fig. 4.3. These figures track not only the boolean `Coin` values step-by-step through the abstract state space, but also the `Z4` values that represent those `Coin`s through the concrete state space. The step-by-step evaluation of expression 4.1 leads to a return value of *true*, whereas the step-by-step evaluation of expression 4.2 leads to a return value of *false*. And so we see the confluence of functional semantics, relational specifications, and abstract data types resulting in the unsoundness of reasoning based on referential transparency: 4.1 and 4.2 are not substitutable for each other, even though client-view reasoning demonstrates that they are equal.

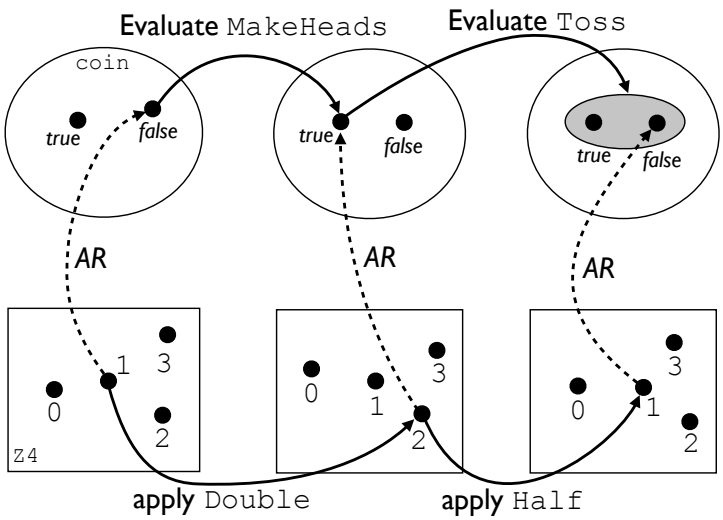

Figure 4.3: A step-by-step illustration of the evaluation of expression 4.2, showing the values in both the abstract and concrete state spaces.

The problem is that there has been a failure to distinguish between (1) the semantics of each program function being a mathematical function *on the concrete (representation) domain*, and (2) the client's understanding of each program function as a mathematical function *on the abstract domain of the ADT*. Our example shows that `Toss` is not a function on the abstract domain: `Toss` (*true*) has two different values depending on the concrete representation of the argument. This is illustrated in the "Evaluate `Toss`" and "Apply `Half`" transitions of Figures 4.2 and 4.3.

In an intuitive sense, the implementation of `Toss` seems somehow biased towards the "history" of the coin, *i.e.*, the face that coin initially showed can impact the results of a toss, even if the coin has been "reset" to heads prior to the toss. The specification of `Toss` combined with functional semantics dictates that history should *not* matter: `Toss` was not parameterized by a "coin history" parameter to bring this

attribute into the scope of client-view reasoning, and so history should be irrelevant to the function's result. This is the essence of referential transparency.

To salvage referential transparency, it might be recommended that relational specifications be disallowed on the grounds that they seem in some sense discordant with the code's presumed functional behavior. When we consider common ADTs such as collection components, however, this position appears untenable. In general, any unordered collection requires a way for clients to iterate through its contents. If the type of the container's contents is left completely unconstrained—as it should, for reusability's sake—then there is no straightforward way of specifying this behavior functionally. Such a specification would place onerous requirements on the instantiation, *e.g.*, a total ordering for the contents type so that a particular value could be singled out as the result of each iteration. On the other hand, omitting such an operation renders collections cumbersome in practice, as [31] demonstrates. In that work, the author specifies a `Set` ADT in a purely functional manner, but must immediately resort to ungainly workarounds to compensate for its lack of relational behavior. For example, in a proposed solution to the set-covering problem, the design forces the collection of subsets to be passed in as a list of sets—because if it were passed in as a set of sets (which would make logical sense), then there would be no way to actually examine or process the subsets as the solution requires.

Another attempt to repudiate problematic cases such as `Coin` might be to demand that we choose data representations that correspond to abstract values in a one-to-one fashion, thus collapsing functions on abstract domains and functions on concrete domains into a single notion. However, this amounts to a relinquishment of true

abstraction. The ability to represent a single abstract value in multiple ways is essential, not only for elegant design, but also for efficiency.

§4.2.3 A "Real" Example: Sets

Consider a new data abstraction for finite sets, specified in a "functional style," as shown in `FunctionalSetTemplate` below. The math definition `MEMBER` is an infelicity due only to current restrictions on the Resolve tools at OSU: they require that the `ensures` clause of all functions be written explicitly, *i.e.*, by giving a mathematical expression identifying the return value. We work around this syntactic limitation by defining `MEMBER` relationally (it can return any member of its argument set), and then using it in the explicit definition of `MemberFrom`. Currently, the verification condition generator does not accommodate functions which have preconditions, so all of the specifications in `FunctionalSetTemplate` are total, even though this involves some slight inelegances, such as defining the behavior of `Remove` on an empty set: `Remove` will simply return an initial `Item` in this case. There is no theoretical hinderance requiring these limitations on Resolve VC generation; the necessary refinements simply haven't been implemented yet.

```
1  contract FunctionalSetTemplate (type Item)
2
3      definition MEMBER (s: finite set of Item): Item
4          satisfies
5              if s = empty_set then
6                  there exists x: Item
7                      (is_initial(x) and MEMBER(s) = x)
8              else
9                  (MEMBER(s) is in s)
10
11     math subtype FSET_MODEL is finite set of Item
12
```

```
13      type FSet is modeled by FSET_MODEL
14          exemplar s
15          initialization ensures
16              s = empty_set
17
18      function Add (restores s: FSet,
19                    restores x: Item): FSet
20          ensures
21              Add = s union {x}
22
23      function MemberFrom (restores s: FSet): Item
24          ensures
25              MemberFrom = MEMBER(s)
26
27      function Remove (restores s: FSet,
28                       restores x: Item): FSet
29          ensures
30              Remove = s \ {x}
31
32      function IsEmpty (restores s: FSet): control
33          ensures
34              IsEmpty = (s = empty_set)
35
36  end FunctionalSetTemplate
```

Below, we list an implementation of `FunctionalSet` that uses a `Queue` as the representation. Our `convention` states that the `Queue` contains no duplicates, by demanding that the length of the `Queue` be equal to the size of its set of entries. Our `correspondence` states that the `FSet` the `Queue` represents is simply the set of all the `Queue`'s elements. To add an element to our `FSet`, we can just do an `Enqueue`, so long as we account for the possibility that the item may have already been in the `Queue`. The `EliminateDuplicates` enhancement is thus invoked. `EliminateDuplicates` is left unimplemented—as always, thanks to decoupled semantics no implementation is necessary. `Remove` uses a `Queue` enhancement which, given a `Queue` and an `Item`,

moves that `Item` to the front of the `Queue` if it is one of the `Queue`'s entries.

```
realization QueueRealization (
        function Replica (restores i: Item): Item
            ensures
                Replica = i,
        function AreEqual (restores i: Item,
                           restores j: Item): control
            ensures
                AreEqual = (i = j)
    ) implements FunctionalSetTemplate

    uses UnboundedIntegerFacility
    uses QueueTemplate
    uses EliminateDuplicates for QueueTemplate
    uses MoveToFront for QueueTemplate
    uses Replica for QueueTemplate

    facility QueueFacility is QueueTemplate(Item)

    type representation for FSet is (
            queue: Queue
        )
        exemplar s
        convention
            |s.queue| = |elements(s.queue)|
        correspondence function
            elements(s.queue)
    end FSet

    function Add (restores s: FSet, restores x: Item): FSet
        variable y: Item
        y := Replica (x)
        Add.queue := Replica (s.queue)
        Enqueue (Add.queue, y)
        EliminateDuplicates (Add.queue)
    end Add

    function MemberFrom (restores s: FSet): Item
        Clear (MemberFrom)
        if not IsEmpty (s.queue) then
            variable x: Item
```

```
41              Dequeue (s.queue, MemberFrom)
42              x := Replica (MemberFrom)
43              Enqueue (s.queue, x)
44          end if
45      end MemberFrom
46
47      function Remove (restores s: FSet, restores x: Item): FSet
48          Remove.queue := Replica (s.queue)
49          if not IsEmpty (s.queue) then
50              variable y: Item
51              MoveToFront(Remove.queue, x)
52              Dequeue(Remove.queue, y)
53              if not AreEqual(x, y) then
54                  Enqueue(Remove.queue, y)
55              end if
56          end if
57      end Remove
58
59      function IsEmpty (restores s: FSet): control
60          IsEmpty := IsEmpty (s.queue)
61      end IsEmpty
62
63  end QueueRealization
```

Notice the use of `Replica` functions in this realization. Resolve has *value semantics*, meaning that all variables and function invocations denote actual values, not references. Assignment in Resolve is a matter destructively moving the value indicated by the right-hand side to the variable appearing on the left, so a `Replica` function is necessary in situations where destruction is not desired, *e.g.*, to adhere to `restores` mode for `x` on line 31. Not all types are guaranteed to have `Replica` available, so we parameterize our realization by the necessary `Replica` function: clients will only be able to instantiate the `QueueRealization` of `FunctionalSetTemplate` if they provide a `Replica` function for the `Item` type. Likewise, not all `Items` have

equality functions available, but our implementation of `Remove` requires one, so we parameterize the implementation by this as well.

Again, this implementation can be proven correct by the classical proof rule [10], but it leads to unsoundness when all the kernel operations are considered referentially transparent. Consider two different programming expressions involving `FSet`s of integers: `Add (Add (s, 0), 1)` and `Add (Add (s, 1), 0)`. A client can use the specification of `Add` to determine that they both denote the same set: $\{0, 1\}$, but one cannot be substituted for the other transparently: if we assume that `s` is the empty set, then `MemberFrom (Add (Add (s, 0), 1))` will return `0` whereas `MemberFrom (Add (Add (s, 1), 0))` will return `1`. This is due to the different order in which the `Add` implementation's `Enqueue`s will be performed.

§4.3 A Proposed Solution

We emphasize that the unsoundness of referential transparency constitutes a serious abstraction problem. Some functional programmers would respond to our examples by claiming that they would never consider expressions such as `Add (Add (s, 0), 1)` and `Add (Add (s, 1), 0)` equal because they realize that they yield unequal representations, but this means that they are reasoning about the code in a non-modular, and thus non-scalable manner. It is simply untenable to think down to the realization level about every data abstraction in a real program. Moreover, in some cases, the code implementing a component may be inaccessible due to permissions settings, or volatile due to dynamic class reloading. We must think about data abstractions according to their interfaces only, and referential transparency is unsound when we do.

And so devising a satisfactory solution to the unsoundness hinges on a key question: *should* one be allowed to replace expressions whose abstract values are equal in any program without changing its meaning? Certainly data abstraction should be preserved in all of its complexity-taming generality; it is the necessity of referential transparency that is being called into question. We propose a new proof rule for the correctness of data representations requiring that they respect referential transparency, and also discuss a workaround to allow problematic implementations such as our `Toss` and `MemberFrom`, should they be desired.

To preserve referential transparency for ADT operations, the rule for proofs of correctness of data representation must be changed in order to guarantee that functional behavior in the concrete domain always leads to functional behavior in the abstract. We propose a new proof rule to express this requirement:

$$\forall a_1 \in dom(\boldsymbol{Spec}), \exists! a_2 \in \boldsymbol{Spec}(a_1),\ \mathrm{AR}^{-1}(a_1) \times \{a_2\} \subseteq \boldsymbol{Sem} \circ \mathrm{AR}$$

$dom(\boldsymbol{Spec})$ is the domain of the relation defined by the operation's specification, *i.e.*, values that satisfy its precondition. The inverse of the abstraction relation, AR^{-1}, maps abstract values to the set of concrete values that correspond to it. Table 4.4 restates this rule in an expanded form with an informal description of its meaning.

Essentially, the rule dictates that the semantics of an operation's implementation must make the operation behave as some particular linearization of its specification, which might be a relation. The rule is easily extended to the case of 0-ary operations:

$$\exists! a, (a) \in \boldsymbol{Spec} \wedge \boldsymbol{Sem} \circ AR = \{(a)\}$$

In other words, there must be exactly one satisfactory abstract result that all concrete states the implementation computes correspond to.

$\forall a_1 \in dom(\boldsymbol{Spec})$,	For all legal calls,
$\quad \exists! a_2 \in \boldsymbol{Spec}(a_1)$	$\quad$ there's *one* legal result such that,
$\quad\quad \forall c_1 \in \mathrm{AR}^{-1}(a_1)$,	$\quad\quad$ for any concrete representation of that legal pre state
$\quad\quad\quad \forall c_2 \in \boldsymbol{Sem}(c_1)$,	$\quad\quad\quad$ for any concrete result the implementation computes,
$\quad\quad\quad\quad a_2 \in \mathrm{AR}(c_2)$	$\quad\quad\quad\quad$ that result corresponds to the one legal result.

Table 4.4: An expanded presentation of our proposed new proof rule for data representations.

Recalling the `Coin` example, our new proof rule prevents two different concrete representations of `Heads` from yielding different abstract results when supplied as arguments to `Toss`. This makes `Toss` less interesting from an intuitive perspective, since it now lacks nondeterminacy, but it restores referential transparency: the proposed implementation of `Toss` can no longer be verified, and an assumption of referential transparency for all operations dictates that it shouldn't be. One verifiable implementation of `Toss` would be:

```
Toss.value := Zero ()
```

Allowing non-determinism in such a framework is a subtle theoretical issue. Burton [15] makes an interesting suggestion along these lines: referential transparency can be preserved if all operations are additionally parameterized by a lazy infinite tree of pseudorandom boolean values. This would obviously complicate client-view reasoning, perhaps intractably so, but the idea jibes with our new proof rule since this tree would be part of the abstract state.

We note that our rule is not entirely limiting. It does not, for example, demand that all specifications be functional, nor that abstraction relations be one-to-one, nor even that the semantics must take all representations of some legal pre state to one particular concrete result.

In the absence of consensus about the role that referential transparency really should play in functional languages, the most prudent way to leverage this new rule is a two-tiered approach to proofs of data representations. For each operation in an ADT implementation, we can first attempt to prove it with our new rule. If this proof succeeds, then the operation implementation is valid and respects referential transparency. If the proof fails, we should attempt to prove the operation implementation using the original method. If that proof succeeds, then that implementation should be considered valid, but it should be flagged as referentially opaque, and client-view reasoning about code should not be allowed to apply referential transparency to subexpressions that mention that operation. If the attempted old-style proof of the operation fails altogether, then the ADT implementation is invalid.

This approach would preserve referential transparency whenever possible, and would automate the process of flagging problematic implementations like the original `Z4` implementation of `Toss`, so that clients will not unsoundly apply referential transparency.

§4.4 Conclusion

We have argued that the two-level view of program meaning, *i.e.*, the distinction between clients and implementers of a concept enabled by strict abstraction, is an ideal approach to component-based software engineering. Furthermore, we have seen

that this view can be leveraged for full functional verification of real programs: the method for verifying client code was explained in Chapter 3, and the current chapter has explained implementation verification.

In the quest for a verifying compiler, the role of proof *automation* is essential. Not only must a verifying compiler be able to generate verification conditions whose validity implies the correctness of the code relative to its specifications, it should also automatically discharge these proof obligations in a "push-button" manner. We therefore continue by discussing the extent to which the Resolve verification tools operate automatically.

Chapter 5: Tools for Modular Verification

§5.1 Introduction: How Hard is Proof Automation?

As we have argued, Resolve's emphasis on abstraction helps to manage the ambitious scope of software verification. Chapter 2 established that the choice of writing specifications in pure mathematics—as opposed to a special-purpose language that includes programming syntax—allows intended functionality to be described in a simple and reusable manner. In Chapter 3 we saw that procedural abstractions can be proven independently of client code because of operation contracts. If a synthesized semantics approach were chosen instead, the verification process would not have been modular, *i.e.*, it would have depended upon the operation implementations rather than their contracts. This was seen to be tantamount to inlining every operation body, thus substantially increasing the complexity of the tracing table. In Chapter 4 we saw that well-chosen abstractions help to manage the difficulty of data representation verification. For example, in the Resolve solution to the VSComp challenge problem involving two singly-linked lists implementing a `Queue`, we observed that pointers were not at all related to the essence of the problem, which was the "last in, first out" nature of singly-linked lists. Thus we abstracted away from pointer structures altogether and used two `Stack`s for our representation. While investigations

into the incorporation of pointers into Resolve have been promising [47], it seems clear that their use should be far less pervasive than in languages like C++, and they should be hidden behind boundaries of strict abstraction for ease of reasoning. Indeed, our `TwoStacks` realization is easy to verify.

The broader lesson of these excursions has been that abstraction is the key to writing maintainable, comprehensible, and verifiable software. When quality abstractions—specifically mathematical models for programming types, operations with contracts, algorithms annotated with loop invariants and assertions, and data representations with conventions and correspondences—are an integral part of the development process, no leaps of human ingenuity can sneak into the program undocumented. And since these specification constructs document the insights underlying abstractions in the language of mathematics, they can easily be incorporated into a mechanical reasoning process to facilitate verification, as we have seen in our tracing tables.

A verifying compiler for Resolve need not implement decision procedures for arbitrarily complex and subtle mathematical formulas, but rather just the intellectually simple (if clerically complicated) VCs that result from modular reasoning about well-designed component-based software. The VCs that arise in Resolve are almost never formulas that would interest, let alone stump, a logician. They tend instead to be simple implications that are superficially complicated by an abundance of indexed variables and extraneous facts. In [42], we analyzed 865 VCs generated from a catalog of Resolve enhancements and found that over 75% of them could be proven using at most one antecedent conjunct. The fact that this analysis did not include VCs generated from realization code should not trouble us given the discussion in the previous chapter: the only differences in realization verification as the Resolve tools at

OSU currently approach it is the inclusion of the convention as an extra fact in the first state and obligation in the last state, and an equality between any kernel-type parameters and their concrete representations via the abstraction function as a fact in the first and last states. Resolve's current treatment of realization verification does not lead to qualitatively different verification conditions, save for the fact that more mathematical theories may be involved in a realization.

Abstraction will now be seen as the key not only to programming and proofs of programs, but also to the *automation* of such proofs. This observation hinges on the importance of separate *roles* in the verified software process. Chapter 3 hinted at this distinction, and we shall now discuss it in detail. We begin with an illustrative example.

§5.1.1 Example: S and K Combinators

The Second Verified Software Competition was held in January of 2012. No Resolve group officially participated, but our investigations into the competition's challenge problems again revealed an insufficient emphasis on abstraction. Consider for example the second of the five challenge problems, quoted verbatim from [24]:

> The Turing-complete language of S and K combinators is sometimes used in compilation of functional programming languages. For this problem, you will write a simple interpreter for combinators, and prove several properties about this interpreter. The syntax of combinators is defined by:
>
> $$\textit{terms} \quad t \ ::= \ S \mid K \mid (t\ t)$$
>
> We will use a call-by-value (CBV) interpreter, based on this definition of contexts:
>
> $$\begin{aligned} \textit{CBV contexts} \quad C \ &::= \ \Box \mid (C\ t) \mid (v\ C) \\ \textit{values} \quad v \ &::= \ K \mid S \mid (K\ v) \mid (S\ v) \mid ((S\ v)\ v) \end{aligned}$$

The expression $C[t]$ denotes the term that we obtain by replacing $\Box$ with t in context C. It is recursively defined as follows:

$$\begin{aligned} \Box[t] &= t \\ (C\ t_1)[t] &= (C[t]\ t_1) \\ (v\ C)[t] &= (v\ C[t]) \end{aligned}$$

The single-step reduction relation $\longrightarrow$ can then be defined this way:

$$\begin{aligned} C[((K\ v_1)\ v_2)] &\longrightarrow C[v_1] \\ C[(((S\ v_1)\ v_2)\ v_3)] &\longrightarrow C[((v_1\ v_3)\ (v_2\ v_3))] \end{aligned}$$

The reduction relation is the reflexive transitive closure $\stackrel{*}{\longrightarrow}$ of the single-step reduction relation. We will also write $\not\longrightarrow$ if there is no t' such that $t \longrightarrow t'$. For example, $K \not\longrightarrow$.

Implementation Task

1. Define a data type for representing combinator terms and implement a function `reduction` which, when given a combinator term t as input, returns a term t' such that $t \stackrel{*}{\longrightarrow} t'$ and $t' \not\longrightarrow$, or loops if there is no such term.

Verification Tasks

1. Prove that if `reduction(t)` returns t', then $t \stackrel{*}{\longrightarrow} t'$ and $t' \not\longrightarrow$.
2. Prove that function `reduction` terminates on any term which does not contain S.
3. Consider the meta-language ks defined by

$$\begin{aligned} ks\ 0 &= K \\ ks\ (n+1) &= ((ks\ n)\ K) \end{aligned}$$

Prove that `reduction` applied to the term $(ks\ n)$ returns K when n is even, and $(K\ K)$ when n is odd.

This problem definition is obviously rather long and arcane. A better way of describing the system would simply be to say:

- The only atomic terms in the system are S and K, and new terms can be formed by applying one term to another.
- K is a function that takes two arguments, and returns the value of the second one. (*i.e.*, $K(x, y) = x$)

- S is a function that takes three arguments, and returns the value obtained by applying the first argument to the third, and then applying that result to the result of applying the second argument to the third. (*i.e.*, $S(x, y, z) = (x(z))(y(z))$)

- Both functions are curried, *e.g.*, $S(x)$ returns a (curried) binary function defined by $(S(x))(y, z) = (x(z))(y(z))$.

Now the challenge's tasks consist of implementing a data type for abstract syntax trees in the SK system, implementing a reduction function and proving some termination properties about it, and proving a special property concerning formulas only containing a flat sequence of Ks.

The most important observation to make about this problem is that almost none of it has anything to do with software verification. Granted, SK-combinators comprise an interesting formal system, but their properties and proofs are sophisticated exercises that concern a *mathematical theory*, not a piece of code. A data abstraction for SK formulae would be straightforward to implement,[28] as would be a direct translation of the reductions described above, but verification of this code would (*should*) consist only of showing that it behaves as its specifications describe, not that it has any additional special properties. The goal of verification is to prove code correct relative to its specification, not to automatically prove arbitrarily sophisticated mathematical insights. The latter is precisely the false hope concerning formalism that Gödel so profoundly dashed nearly a century ago.

[28]One idea would be to use a binary tree whose interior nodes are ignored and whose leaves are all S or K, very similar to Chapter 2's S-Expressions.

If there were a non-trivial alternative algorithm for SK reduction, and we wanted to prove that its implementation produced the same results as the straightforward mathematical description defined in the problem statement, or if there were some optimized data structure for representing SK formulae that we wanted to prove correct, then this would be an interesting (and probably challenging) software verification task. However, as it stands, the SK challenge problem doesn't actually address any functionality that a properly abstracted software verification process should be responsible for.

§5.1.2 What Should Be Automated

There is of course a fine line between the domain of the mathematician and that of the software verifier. Whereas a property such as the termination state of an SK formula consisting of an odd number of Ks has been argued to be outside the proper scope of automated verification, certainly some large amount of mathematical reasoning must be automated if the dream of "push-button" verification is to become a reality.

The first and best answer to this dilemma is to architect one's verification system so that both roles, mathematical theory development and software engineering, interact synergistically. The extensibility of Resolve's mathematical theory library is one example of this accommodation. Adcock [3] discusses ideas for translating failed verification attempts into feedback that programmers can use to debug their code (in the case of a false VC), or that mathematicians can use to strengthen their theory developments (in the case of a VC that may be true but cannot be proven automatically). Another important idea along these lines would be to give specification

writers the ability to include universal algebraic lemmas about their definitions, and then augment the proof process to incorporate these. The correctness of programs verified in this manner would hinge upon the validity of the lemmas, which mathematicians could prove offline, perhaps with the help of proof assistants like Coq or Isabelle. A more ambitious idea would be write an algorithm that generates potentially useful lemmas (according to some heuristic) automatically, and attempts to prove them valid. This would reduce the burden on humans of mathematical theory development. With these comments on system architecture in mind, we shall now describe the current state of Resolve's automation tools.

§5.2 The Resolve Tool Suite

Fig. 5.1 is a visual depiction of a software verification workflow—specifically, it is the model envisioned by the Resolve research group [66]. Executable tools are drawn as rectangles, human-generated artifacts are drawn as ovals, and data flowing between components are drawn as arrows. Arrows labeled with checkmarks or 'x's indicate "yes" or "no" answers that a tool may generate. The two shaded regions represent the distinct roles of programmer and mathematician previously discussed.

The process of generating VCs from code and specifications was described for client and implementation code in Chapters 3 and 4, respectively. The development of specifications, mathematical theories, and supplementary lemmas was discussed primarily in Chapter 2. The two 'x's emitting from the automated prover correspond to the feedback that a failed verification could generate.

The proof checker is an additional component that addresses the possibility of incorrectness in the automated prover itself. The motivating idea is that the prover

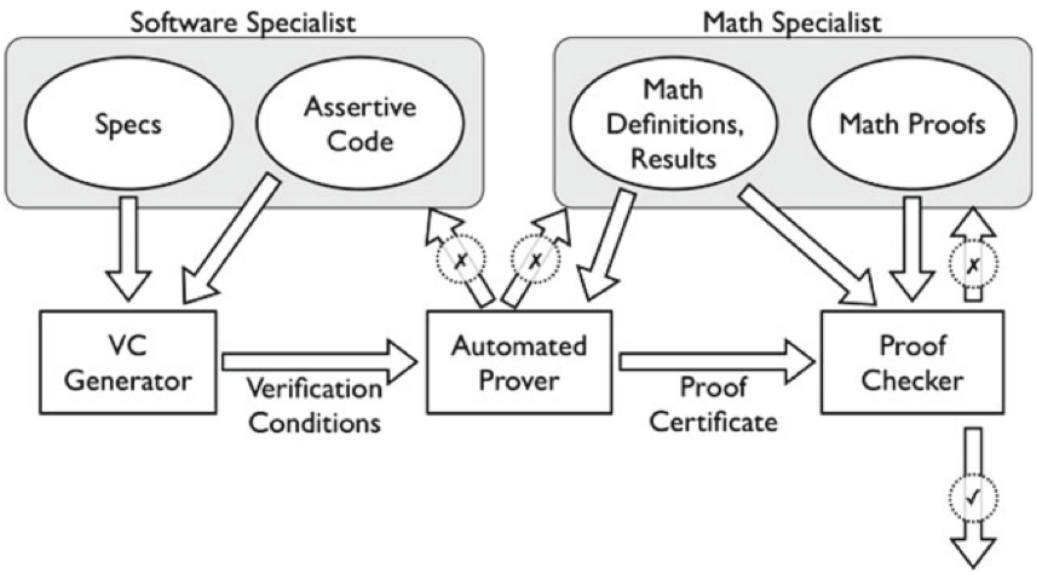

Figure 5.1: The Resolve verification workflow.

can generate a "proof certificate"—a step-by-step description of the incremental deductions that the prover makes, and their justifications—which can then be checked for soundness by a markedly simpler program that is deemed correct by usual means (testing, inspection, or even "manual" verification). Proof checking is not a focus for this work as it has little to do with abstraction. The tools at OSU do not currently employ a proof checker, but it is one of several ongoing Resolve research projects [70].

The other two tools in Figure 5.1, the VC generator and the automated prover, are both under active development at OSU. We now describe some of our work on the latter.

§5.2.1 SplitDecision

Many different tools for automated proof exist, and indeed several have been employed for Resolve specifically. One effective tool we use is a custom-built prover known as SplitDecision. An extensive analysis of the tool is given by its primary developer in [3]; here we give a high-level view and detail one specific facility of which we were the initial implementer.

Despite the availability of many powerful tools for automated proofs, the SplitDecision project arose mostly out of necessity. Proof tools intended for mathematical audiences (*e.g.*, Isabelle [59] and Mizar [28]) tend to lack automation, for precisely the reasons described earlier: the kinds of proofs that mathematicians are interested in require substantial human insight and interaction. Provers intended for software verification, on the other hand (*e.g.*, KeY [7], Z3 [57]), often restrict the available types to programmatic primitives like finite arrays and bounded integers, or deal with one programming language exclusively. As we have seen before with respect to specification languages, competing verification technologies, referential transparency, and statements of verification challenge problems, the state of the art in automated provers also reflects a sub-optimal leveraging of abstraction.

SplitDecision aims for the middle ground between mathematical generality and automatability. As its name advertises, its greatest strengths are its decision procedures and its ability to perform the case splits common to natural deduction. An ancillary advantage is that, unlike third-party tools, which must be treated as black boxes, SplitDecision can be used as a testing ground for experimental new proof strategies and theory developments, such as the decision procedure for strings we will describe shortly.

SplitDecision works by negating the logical formula it receives as input, converting this negation to disjunctive normal form, and performing rewrites to the disjuncts in an effort to find contradictions. When a rewrite converts a disjunct into a contradiction, such as $x = y \wedge x \neq y$, the entire disjunct is removed. If all disjuncts are removed, then the negated formula is false and thus the input VC is true. If not all the disjuncts are removed, then the formula that SplitDecision arrives at is a predicate describing potential counterexamples to the original verification condition.[29] Recall that variables in VCs are implicitly universally quantified, and thus the variables appearing in SplitDecision formulas after the initial negation has been performed (including any fresh variables introduced in the proof process) are all implicitly existentially quantified. A witness to such an existentially quantified formula is a counterexample establishing the invalidity of the input VC.

The key data abstraction in SplitDecision is called `Logical_Formula`, a tree of labels containing information about the structure of the sub-formula rooted at that node, such as the kind at the root (`CONSTANT`, `SYMBOL`, etc.) and the mathematical type of the sub-formula (*e.g.*, `integer`, `string of boolean`). The technical challenges in implementing SplitDecision's proof process are managing the potentially exponential increase in formula size incurred by DNF conversion and case splits, and performing the substitutions necessitated by SplitDecision's rewrite rules quickly. [3] addresses these difficulties admirably. Aside from such design issues, the most novel aspect of the SplitDecision project thus far is its implementation of a new decision procedure for a fragment of mathematical string theory.

[29]Due to incompleteness in the tool, this predicate may be too weak; it may identify cases that are not actually counterexamples.

§5.2.1.1 A Decision Procedure for Strings

A string is an ordered homogenous collection of items allowing duplications. By "homogenous," we mean that all entries in a string have the same mathematical type (they may themselves be strings of integers, for example). A Resolve development of string theory, described in [34], has been critical for modeling containers such as `Queue`s and `Stack`s with strict abstraction. An interdisciplinary research effort with the logician Harvey Friedman has reaped a substantial reward along these lines: a formally proven decision procedure for a large fragment of (finite) string theory. The decision procedure itself is presented in [27], and we describe the fragment it decides below.

The procedure is capable of deciding any formula in the propositional calculus whose atomic propositions are: equalities between two strings ($s = t$), equalities or inequalities between two objects of the string's entry type ($x = y$, $x < y$),[30] equalities or inequalities between two integers ($i = j$, $i < j$),[31] and two special predicates on strings: WINC and VAL. WINC takes a string as an argument, and its meaning is that the entries in the string occur in weakly increasing order. VAL takes three parameters, a string, an integer, and an object, and is true iff the entry in the string at the given numeric position is equal to the object. These atomic formulas can be negated, conjoined, or disjoined to create what we'll call a "string formula" decidable by the procedure.

[30]This implies that only strings of linearly-ordered types are allowed, but the procedure does accommodate strings of unordered types so long as none of the order-based operators are employed.

[31]Since the decision procedure works by converting a string formula to one of Presburger arithmetic, integer expressions involving addition, subtraction, absolute value, and division or mod by constants greater than 1 are all allowed.

In addition to the predicates mentioned above, the other operators available for strings in these formulae are $\langle\rangle$ (the empty string constant), the unary string constructor ($\langle x\rangle$ is the string of length 1 containing only x), and concatenation ($s * t$). After its original formulation, we discovered that a reverse function for strings could be added without weakening the decidability result. String variables (*e.g.*, α, β) are allowed as well, and arise often due to the tabular verification approach.

We will not reiterate all of the rewrite rules in [27]. There are 26 in total. The rules execute in two phases. In the first phase, 24 different rules are applied nondeterministically to reduce the formula into a restricted form, which is then converted by two additional rules into a formula in Presburger arithmetic, a decidable linear arithmetic. We implemented this algorithm in Resolve/C++, a dialect of C++ created at OSU that uses strict abstraction. Theoretically, Resolve/C++ is easy to translate into the pure Resolve that our tools verify; one hope of our research program is that SplitDecision itself could eventually be verified, although hoping that such a verification would automatically establish the soundness of the string decision procedure is a folly similar to hoping to automatically prove arcane results about SK combinator theory.

Representative examples of rules from phase one of the decision procedure include:

- Replace any conjuncts of the form $s = \langle\rangle$, where s contains an object variable, with *false*.

- Replace any term of the form $|s * t|$, where s and t are string expressions, by $|s| + |t|$, replacing $|\langle x\rangle|$ by 1 and $|\langle\rangle|$ by 0 as necessary.

- Replace conjuncts of the form $\text{WINC}(\alpha * \beta)$ with $\text{WINC}(\alpha) \wedge \text{WINC}(\beta) \wedge \text{VAL}(\alpha, |\alpha|, x) \wedge \text{VAL}(\beta, 1, y) \wedge x \leq y$, where x and y are fresh object variables for the formula.

- Replace any conjuncts of the form $\neg\text{WINC}(s)$ with $\text{VAL}(s, n, x) \wedge \text{VAL}(s, m, y) \wedge n < m \wedge y < x$, where n, m, x, and y are fresh variables for the formula. (VAL numbers positions of α as $1..|\alpha|$.)

- On encountering a conjunct of the form $\text{VAL}(\langle y \rangle * t, i, x)$, perform a split: add a new disjunct identical to the one containing the VAL. In one of these two identical disjuncts, replace the VAL by $i = 1 \wedge x = y$. In the other, replace the VAL by $i > 1 \wedge \text{VAL}(t, i - 1, x)$.

- Replace any conjuncts of the form $\text{VAL}(\langle y \rangle, i, x)$ with $i = 1 \wedge x = y$.

At the end of phase one, the only remaining atomic sub-formulas involving strings are WINCs and VALs whose string arguments are just string variables. In phase two, each disjunct is processed separately, and its WINCs and VALs are replaced by implications not involving strings, thus resulting in an existential formula of Presburger arithmetic:

- For every distinct pair of VALs on the same string variable in a disjunct, $\text{VAL}(\alpha, i, x)$ and $\text{VAL}(\alpha, j, y)$, such that $\text{WINC}(\alpha)$ does *not* appear in this disjunct, add $i = j \Rightarrow x = y$ as a conjunct. Once all such VAL pairs are processed, remove them all from the disjunct.

- For every distinct pair of VALs on the same string variable in a disjunct, $\text{VAL}(\alpha, i, x)$ and $\text{VAL}(\alpha, j, y)$, such that $\text{WINC}(\alpha)$ *does* appear in this disjunct,

add $i = j \Rightarrow x = y \wedge i < j \Rightarrow x \leq y \wedge x < y \Rightarrow i < j$ as a conjunct.[32] Once all such VAL pairs are processed, remove them all from this disjunct.

The string decision procedure implemented in SplitDecision lies at the heart of many of the VC proofs conducted and analyzed in our aforementioned work on bookkeeping in proof automation [42].

§5.3 Tool Support for Verifying Functional Languages

In Chapter 2, we discussed the pragmatic advantages of programming in a "functional style," but noted that the existing verification tool for the most popular functional language, *i.e.*, the ACL2 project for verifying Lisp, lacks abstraction into mathematics. This is perhaps unsurprising—historically, Lisp has been considered to be such a beautiful and math-like language that it needs no specifications. Several factors explain this mindset, among them: (1) Lisp was arguably the first modern language to be defined with true mathematical formality [53], (2) Lisp's parenthesized prefix notation feels natural, particularly to those familiar with lambda calculus, (3) Lisp's built-in integers are unbounded, so the programming operations for arithmetic (assuming a correct interpreter) really behave exactly as mathematics dictates, without overflow issues, and (4) Lisp's core semantics are small and intellectually tractable (*e.g.*, a very readable five-page definition in [54]). However, Lisp has grown since its inception, and now includes features that undermine its theoretical purity and built-in operations whose behavior is far from intuitive; specifications using strict abstraction would be a valuable addition. We shall reinforce this idea by examining shortcomings

[32] There is an error in [27] regarding this rule. It calls for the conjunct to contain $i < j \Leftrightarrow x < y$, but this is not true: in a weakly increasing string, $i < j$ and $x = y$ is possible.

in by far the most successful verification effort for Lisp, and then investigate other existing technologies for verifying functional programs.

§5.3.1 ACL2

Rather than proving that Lisp programs always behave in accordance with their specifications, ACL2 proves arbitrary theorems that are expressed in the very same syntax as the programming language. ACL2 is thus two things in one: a programming language, and an interactive theorem prover. This conflation has some benefits, but can also be dangerously confusing. For example, ACL2's documentation claims to list "the Common Lisp primitives supported by ACL2" [1], but the list includes logical functions such as `implies`, which are not defined in Common Lisp and are only intended to be used with the ACL2 theorem prover. In ACL2, one can define a function[33]:

```
ACL2 !> (defun foo (x y) (implies x y))
Since FOO is non-recursive, its admission is trivial.
```

and then prove some logical property about it:

```
ACL2 !> (defthm false-antecedent (foo nil x))
Q.E.D.
```

However, this interaction immediately leads to an error when we try to leverage our `foo` function and the property proven about it inside of a standard Common Lisp interpreter (as opposed to ACL2's):

[33]In each of our example ACL2 interactions, we show a proper substring of the response given by the top-level, omitting details not relevant to our discussion

```
[1]> (defun foo (x y) (implies x y))
FOO
[2]> (foo nil 'x)
*** - EVAL: undefined function implies
```

Thus, to the degree that ACL2 allows statements of its logic to be intermingled with the programming language it purports to verify, its prover is unsound in the sense that the code it verifies does not work reliably in a Common Lisp interpreter. ACL2 is not, as it claims to be, a "very small subset of full Common Lisp" [1].

Of course, we could mitigate this problem by performing a syntactic check for any functions not part of Common Lisp in "verified" ACL2 developments. However, the issue is exacerbated by the fact that some Common Lisp functions are overridden with *different meanings* in ACL2, *e.g.*, `round` in Common Lisp returns two values, but in ACL2 it only returns one.

ACL2 checks all function definitions for termination before admitting them. A straightforward attempt to define addition:

```
ACL2 !> (defun my-add (x y) (if (<= x 0)
                                y
                                (my-add (1- x) (1+ y))))
```

leads to an error, because the prover cannot determine that this function terminates on all inputs. This due to the lack of any restrictions on the kind of values `x` and `y` may take on—the correct definition requires a guard demanding that `x` and `y` be natural numbers:

```
ACL2 !> (defun my-add (x y) (if (and (natp x) (natp y))
```

```
(if (<= x 0)
    y
    (my-add (1- x) (1+ y)))
nil))
```

In addition to the conflation of programming language and proof logic glimpsed already, `my-add` demonstrates the second major shortcoming of ACL2 in terms of insufficient abstraction: we can only prove our function correct *relative to other code*. There is no way in ACL2 to prove the claim: "`my-add` performs true mathematical addition for all natural number arguments." Instead, we can only prove claims such as: "`my-add` computes the same thing as the `+` operator on all natural number arguments":

```
ACL2 !>(defthm my-add-is-+ (implies (and (natp x) (natp y))
                                    (equal (+ x y) (my-add x y))))
Q.E.D.
```

Given our earlier discussion about the fact that `+` in Lisp really does perform addition without the usual computational complications like overflow, ACL2's lack of mathematical modeling may not seem particularly detrimental. Then again, given our surprising discoveries about `round` and `implies`, what reason is there to think that the `+` mentioned in our theorem is the `+` that Common Lisp interpreters implement? Proving theorems about code that are expressed *as* code amounts to a verification strategy in which there is no qualitatively different description of behavior that we can rely on for conceptual understanding.

In practice, ACL2 has enjoyed considerable success despite these shortcomings, because it is used primarily for interactive verification of low level structures whose

intended behavior can be defined with recursive functions. For example, ACL2 has been used to prove that the translation of a processor's microcode for computing square roots into a collection of Lisp functions performs the computation accurately, relative to a definition of square root in terms of ACL2's implementation of rational arithmetic [65]. ACL2 researchers have published many such accomplishments. We intend our critiques not as diminution, but simply as evidence that ACL2 is not apt for verifying the correctness of functional code relative to a specification using mathematical modeling, a technique whose advantages have been observed throughout this work.

§5.3.2 Coq

Like ACL2, Coq [8] is both a programming language and a theorem prover. Unlike ACL2, however, Coq uses a rich type system to distinguish between programs and statements of mathematical logic. Coq is an implementation of the Calculus of Inductive Constructions, a typed lambda calculus with subtle theoretical tweaks that facilitate, *e.g.*, strong normalization, the option of avoiding impredicative sorts, and coinductive definitions of infinite objects. These are all advanced mathematical topics that range far beyond the scope of basic client programming that we have concerned ourselves with thus far—we will focus our discussion on an example that shows Coq being employed to verify a "real-world" algorithm of the sort that a Resolve programmer might be interested in.

Our example is a implementation of the insertion sort algorithm for lists, where the entries are natural numbers ordered by the usual $\leq$. We will define the algorithm and prove theorems that culminate in establishing the algorithm's correctness. The

skeleton of this file is an example by Andrew Appel at the Oregon Programming Languages Summer School [2]; we will make clear the portions that we authored.

After a prelude consisting of some pertinent lemmas, Appel defines his insertion sort algorithm:

```
1  Fixpoint insert (i:nat) (l: list nat) :=
2    match l with
3    | nil => i::nil
4    | h::t => if ble_nat i h then i::h::t else h :: insert i t
5    end.
6
7  Fixpoint sort (l: list nat) : list nat :=
8    match l with
9    | nil => nil
10   | h::t => insert h (sort t)
11   end.
12
13 Example sort_pi: sort [3,1,4,1,5,9,2,6,5,3,5] =
14                       [1,1,2,3,3,4,5,5,5,6,9].
15 Proof.
16   simpl. reflexivity.
17 Qed.
```

`insert` is a helper function for inserting one number into the correct position in a list, and `sort` uses `insert` to sort its argument, a list of naturals. `Example` is a small theorem demonstrating that `sort` behaves as expected on one particular input list. It is proven by the `simpl` tactic, which just applies the definition of `sort` to its argument on the left-hand side of the equation.

Next, Appel defines a useful lemma, and we supply a proof for it. We omit the fully detailed interaction with the Coq top-level, which shows the proof state at each step in the derivation, instead just listing the tactics that were employed to complete the proof. Some important tactics were explained in Chapter 2; our purpose here is

just to give the flavor of a typical Coq verification, specifically the amount of human guidance necessary to conduct even basic proofs. `Permutation` is a predicate defined in a Coq library. This library also establishes useful theorems about `Permutation`, such as `Permutation_refl`, which states that any list is a permutation of itself. As in Chapter 2, we see that Coq can be useful for mathematical theory development, so long as we trust its kernel to be consistent with standard mathematics.

```
1  Theorem insert_perm: forall i l,
2    Permutation (insert i l) (i::l).
3  Proof.
4    intros. induction l as [| head tail].
5
6    simpl. apply Permutation_refl.
7
8    Case "l is non-empty". simpl.
9      remember (ble_nat i head) as i_lt_head.
10     destruct i_lt_head.
11
12     SCase "i goes in front". apply Permutation_refl.
13
14     SCase "i goes elsewhere".
15       assert (Permutation (head :: insert i tail)
16                           (head :: i :: tail)).
17       apply perm_skip. assumption.
18
19       assert (Permutation (head :: i :: tail)
20                           (i :: head :: tail)).
21       apply perm_swap.
22
23       apply perm_trans with (l':=(head :: i :: tail)).
24       assumption. assumption.
25 Qed.
```

Now we can proceed to prove that `sort` returns a permutation of its argument. Recall from Chapter 2 that this is a crucial correctness property for sorting algorithms.

```
1  Theorem sort_perm: forall l, Permutation l (sort l).
2  Proof.
3    induction l as [| head tail].
4
5    simpl. apply perm_nil.
6
7    simpl. apply Permutation_sym.
8
9    assert (Permutation (insert head (sort tail))
10         (head :: (sort tail))).
11     apply insert_perm.
12
13   assert (Permutation (head :: sort tail)
14                       (head :: tail)).
15     apply perm_skip. apply Permutation_sym. apply IHtail.
16
17   apply perm_trans with (l':=(head :: sort tail)).
18   assumption. assumption.
19 Qed.
```

It remains to be proven that `sort` actually returns a list whose entries occur in increasing order. Unlike `Permutation`, there is no `sorted` predicate in the Coq libraries that we will rely on. Instead, Appel requests we write an "inductive predicate" of our own to capture this notion. An inductive predicate is a predicate defined by completely characterizing the situations which make it true. It is similar in structure to an inductive datatype, with named constructors for each qualitatively different case of the predicate's truth. Here is our inductive predicate `sorted`:

```
1  Inductive sorted: list nat -> Prop :=
2    | nil_sorted: sorted nil
```

```
3     | single_sorted: forall (n: nat), (sorted [n])
4     | nonnil_sorted: forall (n h: nat) (t: list nat),
5         sorted (h::t) -> (n <= h) -> (sorted (n :: h :: t))
6     | tail_sorted: forall (h: nat) (t: list nat),
7         sorted (h::t) -> sorted t
8     | with_insert_sorted: forall (i h: nat) (t: list nat),
9         sorted (h::t) -> (h <= i) -> sorted (h::insert i t).
```

`sorted` can be viewed as a family of five predicates. `nil_sorted` states that the empty list is sorted, `tail_sorted` states that the tail (or "cdr," in Chapter 2's S-Expression theory) of a sorted list is also sorted, and so on.

We can now state and prove a useful lemma about the `insert` helper function:

```
1  Theorem insert_sorted: forall l i,
2    sorted (l) -> sorted (insert i l).
3  Proof.
4    induction l as [| h t].
5      Case "l is nil". simpl. intros. apply single_sorted.
6      Case "l  is h::t". intros.
7        remember (ble_nat i h) as i_lt_h. destruct i_lt_h.
8
9        SCase "i <= h". simpl. rewrite <- Heqi_lt_h.
10         apply nonnil_sorted. assumption.
11         symmetry in Heqi_lt_h.
12         apply ble_nat_true in Heqi_lt_h. assumption.
13
14       SCase "i > h". simpl. rewrite <- Heqi_lt_h.
15
16         assert (sorted t).
17           apply tail_sorted in H. assumption.
18
19         apply with_insert_sorted. assumption.
20         symmetry in Heqi_lt_h.
21         apply false_ble_nat_e in Heqi_lt_h. omega.
22 Qed.
```

Notice that the proof uses the named constructors of `sorted`. `sorted` can be thought of, then, as defining five different categories of "evidence" that establish the

sorted-ness of a list. Useful definitions of inductive predicates are crucial to proving theorems in Coq. The proof of `insert_sorted` would have been substantially more complicated if, for example, `with_insert_sorted` were not defined in `sorted`. Finally, we prove that `sort` returns a `sorted` list:

```
1 Theorem sort_sorted: forall l, sorted (sort l).
2 Proof.
3   induction l as [| h t]. simpl. apply nil_sorted.
4     simpl. apply insert_sorted. assumption.
5 Qed.
```

Coq does not use mathematical modeling, so we see the unfortunate admixture of programming function invocations and mathematical definitions like `sorted`, but on the other hand there is no other language that Coq purports to be verifying, so this conflation cannot lead to "false positive" proofs like we saw with `implies` in ACL2. And unlike Resolve, Coq proofs are by default entirely non-automated. There are rudimentary accommodations for automating proofs in Coq, *e.g.*, a tactic called `auto`, but they are limited. `auto` only solves goals that can be proven by some combination of the `intros`, `apply`, and `reflexivity` tactics [60]. If further automation is desired, the user can program customized tactics using Coq's built-in Ltac language, but this is a daunting programming challenge having little to do with mathematical proof, again circumventing the role of the mathematician in Resolve's envisioned verified software process. Next we will briefly inspect Ynot, an alternative framework for program proof in Coq.

§5.3.3 Ynot

Ynot is a Coq library developed by a research group at Harvard University. It axiomatizes a monad [74] called `Cmd` that is parameterized by pre- and post-conditions; the type signature of a function in Ynot is a `Cmd` and thus gives a contract for the function. These contracts are phrased in terms of assertions about the heap, allowing code written in the style of functional languages like ML to be verified by separation logic [64]. We will show an example from the Ynot Tutorial [16], which gives some practical applications that make for interesting contrasts with Resolve.

The Ynot component that we will inspect is a `Stack`, whose functionality is the same as that shown for Resolve in Chapter 4. The model of the Ynot `Stack` is a Coq `list`, treated quasi-mathematically as a ghost variable, similar to what was explained for Dafny in Chapter 3. To establish this model, we define a `rep` function, which, given a `Stack` and a `list` it purports to represent, returns a predicate on heap states in which this representation holds.

```
1  Variable T: Set.
2  Record node: Set := Node {data: T;
3                            next: option ptr}.
4  Fixpoint listRep (ls: list T) (hd: option ptr)
5                   {struct ls}: hprop :=
6    match ls with
7    | nil => [hd = None]
8    | h::t => match hd with
9              | None => [False]
10             | Some hd => Exists p:@ option ptr,
11                             hd—>Node h p * listRep t p
12             end
13   end.
14 (* {struct ls} is an unnecessary annotation indicating
15    that listRep terminates by structurally reducing ls *)
16
```

```
17 Definition stack := ptr.
18 Definition rep q ls := Exists po:@ option ptr,
19                          q—>po * listRep ls po.
```

In this listing, we define a `Node` record for a singly-linked list implementation of `Stack`, and define a function `listRep`, which, given a list and a pointer, gives a predicate on heaps in which the pointer points to a head of a singly-linked list whose data fields are those of the list argument. We can thus represent a `Stack` as a pointer, where `rep` is just the existence of a pointer whose `listRep` is the list modeling our `Stack`. Needless to say, this is a substantially more complicated manner of representation definition than was seen for Resolve in Chapter 4.

The `push s x` operation can be specified by stating as a precondition that `s` represents some particular list `ls`, and as a postcondition that `s` represents `x::ls`, where `::` is the cons function in Coq.

```
1 Parameter push: forall (T: Set) (s: t T)
2                         (x: T) (ls: [list T]),
3     Cmd (ls ~~ rep s ls)
4         (fun _: unit => ls ~~ rep s (x::ls)).
```

The square brackets around `list T` indicate that `ls` is a ghost parameter, and the `~~` is a necessity of Ynot's implementation: programmers must explicitly indicate whenever a ghost variable is referred to by employing this operator. The pre- and postconditions of the `Cmd` monad are, it is safe to say, somewhat less readable than those written in the Resolve style, although for practiced functional programmers the discrepancy is not so severe.

Finally, we can show the simultaneous code and proof of `push`'s implementation. We say "simultaneous" because of the way that Ynot works: we state the `Cmd` expression for `push` as a definition, which then introduces it as a proof obligation. Ynot defines a special tactic called `refine`, inside of which we write our ML-style code. The definition of this tactic accounts for the meaning of this code in terms of separation logic, and attempts to prove that according to this semantics, the `Cmd` precondition implies the postcondition.

```
1 Definition push: forall s x ls,
2     Cmd (ls ~~ rep s ls)
3         (fun _: unit => ls ~~ rep s (x::ls)).
4   intros. refine (hd <- !s;
5                   nd <- New (Node x hd);
6                   {{s ::= Some nd}}); t.
```

The syntax of the code inside of `refine` is somewhat obscure. This is due to Ynot's need to define operators for programming in their ML-style language without overriding existing Coq operators. `<-` is assignment, `!` is pointer dereferencing, and `::=` assigns a pointer the reference of the object mentioned on its right-hand side. The double brackets around the last line of code indicate weakening of the precondition and strengthening of the postcondition in a Hoare-style program proof [35]: the precondition and the state of the heap after executing the assignment of `Some nd` to `s` implies `push`'s postcondition. `t` is a custom-defined proof tactic in the aforementioned Ltac that discharges the proof obligations remaining after `refine`. Ynot encourages the development of customized proof tactics in conjunction with authoring code. This is roughly isomorphic to Resolve's discipline of documenting programmer insights with formal annotations, but strict abstraction makes Resolve annotations

more readable, as they are phrased in mathematics rather than in an internal language for proof automation like Ltac.

Ynot is syntactically quite dissimilar from Resolve, and its targeting of a functional language with reference semantics via monads and separation logic makes a direct theoretical comparison rather difficult. We do not wish to assert that either approach is absolutely better than the other, but we feel that Resolve's value semantics and strict abstraction are likely to have more wide appeal. Moreover, we are not aware of any use of Ynot for verifying *client* code, only implementations of data abstractions. We suspect that the difficulties of modularity with respect to reference semantics glimpsed previously in Jahob and Dafny will be present in Ynot as well.

§5.4 Conclusion

This chapter has presented a workflow for verified software in Resolve, and has discussed some implementation details of this toolchain. Notably, we discussed our implementation of a decision procedure for mathematical strings. We also investigated some related proof tools for other languages.

We have now completed a broad survey of component-based software verification in Resolve, highlighting issues of abstraction in every step of the process. The choice of specification language, the decision to use strictly mathematical modeling, the use of decoupled semantics for reasoning about code, conventions and correspondences for proving data representations correct relative to a model, and the separable roles of programmer and mathematician in the verification environment are all examples of abstraction-focused design decisions that improve the modularity, and thus the

pragmatic feasibility of software verification. We have also sought, whenever possible, to contrast these decisions with cases of insufficient abstraction in related work.

Having elaborated at length abstraction's primacy to tractable software verification, we will conclude this dissertation with glimpses at future work that seek to further increase abstraction's prominence in Resolve. We do not consider these problems solved; just hinting at possible future developments is itself a research contribution, which we hope others will build upon as Resolve continues to grow.

Chapter 6: New Directions for Abstraction-Embracing Software

§6.1 Introduction

The running theme of this work has been an investigation of abstraction's role in software engineering, particularly how abstractions can be leveraged for modular, automated software verification. By no means is this work the end of that story. In fact, we wish above all else to *raise* issues concerning abstraction to the attention of the research community, not to proclaim them all solved outright. In this final chapter, we suggest some areas for future research, all of which can be considered continuations of our emphasis on abstraction.

§6.2 Abstraction Relations

In Chapter 4 we discussed verification of implementation code in Resolve. We pointed out that although our automated tools currently support only abstraction functions, the proof method they are based on is known to accommodate abstraction relations [21]. This is advantageous, because it has been discovered that abstraction relations are in fact necessary for efficiency and good design of some data abstractions [68].

Recently, the Resolve/Reusable Software Research Group (RSRG) has been exploring the possibility of automating data representation proofs in which the correspondence is a relation and not a function. Because our proof rules for correctness for data representations introduce an existential quantifier when an abstraction relation is involved, new annotations should be added to realization components so that the prover does not have to engage in the difficult task of automatically finding a suitable witness.

In Fig. 4.1, we diagrammed the primary intuition behind verifying data representations, but also noted that this methodology was incomplete relative to the rich annotation constructs used in Resolve data abstractions, *i.e.*, `convention` for restricting the concrete state space of a representation and `constraint` for restricting the abstract state space of a model. A more accurate depiction of Resolve data representation correctness is shown in Fig. 6.1.

As Fig. 6.1 illustrates, the image of the abstraction relation for values that satisfy the convention must be a subset of the constraint, so that no invalid abstract values may be represented in the implementation. This means that to verify a realization, we must prove:

$$\forall c \in \texttt{convention}, \forall a \in AR(c), a \in \texttt{constraint} \tag{6.1}$$

Since (6.1) is purely universal, no special witness-providing annotation is necessary. Another condition on correct implementations, however, dictates that every convention-satisfying value must correspond to something:

$$\forall c \in \texttt{convention}, \exists a \in \texttt{constraint}, a \in AR(c) \tag{6.2}$$

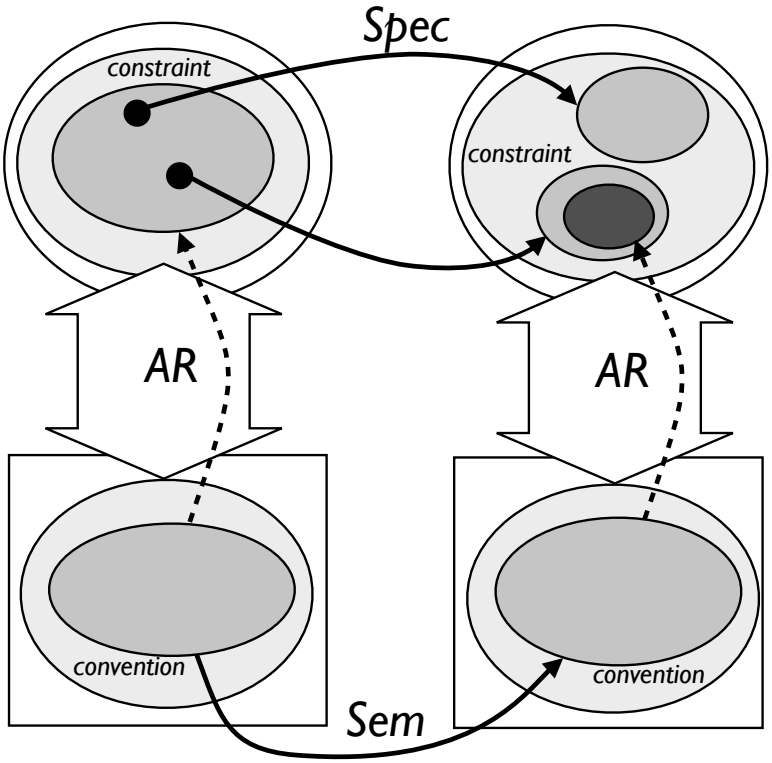

Figure 6.1: A commutative diagram for verification of implementations in Resolve.

To automate this proof, implementers should write an "instance function," which, given a concrete value, gives one particular abstract value it should correspond to. The prover can then check to see whether this is actually true.

The proof obligation that makes use of the instance function is conducted once for the entire representation; it does not concern any particular kernel operation. Additional annotations are useful to automate the proofs of existentials introduced by the rules for proving the individual kernel operations.

The conceptual idea underlying abstraction relations is that representation values correspond to equivalence classes of abstract values, and these equivalence classes consist of the abstract values that the client cannot distinguish among based on the calls that she has made thus far (these histories of calls on ADTs are sometimes called "scenarios"). When a new call is made, the contract may dictate that some of the values in our equivalence class result in a set of legal results that is disjoint from what some other values in our equivalence class would result in—*i.e.*, the call may "discriminate" between different possible incoming abstract values. Fig. 6.1 depicts this situation by showing two disjoint sets of results in the top-right. If the concrete result that the call actually computes fails to mirror this discrimination, *i.e.*, when the concrete result does not correspond to a subset of the results allowed for one particular abstract pre-state, then the implementation is incorrect. This is why an existential quantification is introduced when proving kernel operations that use abstraction relations; the proof obligation is:

$$\forall c_1 \in \texttt{convention}, AR(c_1) \subseteq (\texttt{constraint} \cap dom(\boldsymbol{Spec})) \Longrightarrow \tag{6.3}$$
$$(\exists a_1 \in (\texttt{constraint} \cap dom(\boldsymbol{Spec})), \boldsymbol{Sem}(c_1) \circ AR \subseteq \boldsymbol{Spec}(a_1))$$

The intersections with `constraint` are not necessary if the one time proof of (6.1) has been performed. In any case, (6.3) demands that, for any legal concrete pre-state, there should be one abstract value it corresponds to such that the result of executing the kernel operation corresponds to a subset of the values the specification allows as legal results for that abstract value. To provide a witness for this abstract pre-state, the implementer should provide a "precursor function," which, given a concrete value c_1 and an abstract result a_2, gives an abstract value a_1 such that $\boldsymbol{Sem}(c_1) \circ AR \subseteq \boldsymbol{Spec}(a_1)$. Notice it is important that the precursor function might be parameterized not only by a concrete value but also by an abstract result; the correct precursor may depend upon that result, as was shown in Fig. 6.1. If the abstract result were a member of the topmost dark bubble in the upper-right of that figure, then the topmost dot on the upper-left should be the precursor.

Finally, we should observe that the question of whether or not a concrete pre-state is legal at all also involves an existential quantification. This was implicit in the antecedent of formula (6.3). Some concrete values might correspond to a subset of the abstract state space that "overlaps" the subset of legal calls, with some values qualified for the call to the kernel operation and others violating the precondition (but not the constraint, if formula (6.1) has been proven). This means that the client is currently holding a value that is not guaranteed to be legal for the call to the kernel operation, and so cannot make the call and expect correct results. In the programming-by-contract discipline, this is the client's responsibility, so all we need to do for verification of the implementation is make sure that such values are discounted from the proof obligation 6.3, hence the antecedent guarding that formula. The prover needs to be able to identify when that antecedent is false, so that it doesn't subject

that concrete value to the proof of the consequent. We can expand the negation of the antecedent as follows, and see that it involves another existential quantification:

$$\exists a_1 \in AR(c_1), \neg \texttt{constraint} \vee a_1 \notin dom(\textbf{\textit{Spec}}) \tag{6.4}$$

If (6.1) has been proven we know the first disjunct of (6.4) is unnecessary; it can never be false. The disqualifier function should act as a sort of "worst-case identifier": given a concrete state, it should give a corresponding abstract value that fails to meet the kernel operation's precondition, if such a value exists. Happily, failure to provide a good disqualifier function is not a soundness issue. If the disqualifier doesn't correctly identify concrete states on which the call is illegal, then the prover will try (and fail) to establish the truth of (6.3) for those concrete states, thus rejecting the implementation as incorrect. Recall from §3.2.1 that the definition of completeness in a verification system using programmer-supplied assertions should be relative to the strength of those assertions. Just as a correct program with weak invariants might fail to be verified, so might an implementation with inapt disqualifier, precursor, or instance function annotations.

§6.3 Specification-Aware Memoization

The tabular method of verification leverages the decoupled semantics view of code to reason about the values of variables at each state of the program. Specifications are the keystone of this approach: they are what allow us to know what the results of each call will be, and they allow us to determine whether any illegal calls are made. If code is verified, then it behaves in accordance with its specifications, and if Hoare's vision of a verifying compiler is realized, then we would never attempt to execute unverified code. It thus makes sense to ask whether specifications could be used to

aid the *execution* of verified programs. We now consider such a possibility. As with many of our discussions, we focus on functional programming, wherein execution is usually an interpretive process—we execute a functional program by evaluating it like a mathematical expression, interpreting the meaning of each syntactic construct we encounter in accordance with the language's semantics.

In the interpretation of classical functional programming languages like Lisp, function evaluation proceeds in a very specific manner: first all of the function's arguments are evaluated, and then the function is applied to these resulting values. However, other purely functional languages take a "non-strict" or "lazy" approach: arguments are only evaluated if/when they are needed.[34] This is similar in spirit to the distinction between bottom-up and top-down views of abstraction we made in Chapter 1—Lisp takes a bottom-up approach, evaluating expressions by bubbling results up their abstract syntax trees, whereas Haskell works top-down, evaluating subexpressions only if they are needed and stopping as soon as it has sufficient information.

Since interpreters for lazy functional languages work by evaluating subexpressions as their values are needed for other computations, the optimization technique known as *memoization* is often advantageous. Memoization consists of saving the value computed for an expression so that it need not be recomputed if the expression arises elsewhere during evaluation. A memoizing interpreter essentially maintains a mapping from expressions to values. Whenever an expression needs to be evaluated in the process of interpretation, this map is consulted first. If the expression appears as a key in the map, its value is utilized in the evaluation. If not, its value is computed, and the appropriate new (key, value) pair is added to the mapping.

[34]Haskell is a popular example of a lazy functional languages [52].

In evaluating the expression (in Haskell-like syntax):

```
minus (plus 1 (minus (plus 1 1) 1)) (minus 2 1)
```

the subexpression `minus 2 1` will arise at least three times, although non-strict semantics implies that the order in which these three occurrences will be evaluated is not entirely fixed. We depict and annotate one potential sequence of evaluation in Table 6.1. At each step of evaluation, we underline the subexpression that will be interpreted next:

Expression	Note
`minus (plus 1 (minus (plus 1 1) 1))` `(minus 2 1)`	
`minus (plus 1 (minus` `(plus 1 1)` `1)) 1`	Memoize (`minus 2 1`, 1)
`minus (plus 1` `(minus 2 1)`) 1	Memoize (`plus 1 1`, 2)
`minus` `(plus 1 1)` 1	Using memoized result
`minus 2 1`	Using memoized result
1	Using memoized result

Table 6.1: An example of memoized lazy interpretation in a Haskell-like language.

This process seems closely linked to the property of referential transparency, discussed at length in our examples of unsoundness. Indeed, expressions involving referentially opaque operations cannot be correctly rewritten by memoization, as we have seen in Chapter 4.

Memoization exemplifies a curious lack of abstraction in common implementations of functional languages: the identities of objects are not values of some mathematical model as Chapter 2 advocated, nor are they usually references in the heap (*e.g.*, sharing common substructures is a nearly universal technique for taming the memory footprint of functional language implementations). Instead, programmers in

languages like Haskell are led to think of the identities of their objects as particular arrangements of constructors and operation invocations, because these are what determine the values that will be computed (and memoized). When presented with our unsoundness example involving `Coin`s in Chapter 4, these programmers would respond that `Toss` should not be assumed to return the same result on `MakeHeads (Heads ())` as on `MakeHeads (Tails ())`, because it's clear from inspection that these two arguments are not "equal." As we argued in that chapter, this line of reasoning constitutes a severe relinquishment of abstraction, and we reject it in favor of mathematical modeling.

To that end, we propose that a new approach to memoization could be undertaken, in which the values in the map are values of the mathematical model of the key expression's type. An expression involving `Coin`s would be memoized as a boolean, an expression involving `Queue`s would be memoized as a string of items, and so on. Of course, in the presence of relational specifications, it will not always be possible to determine the value of an expression from the specifications of the operations it mentions, but functional specifications are common enough to justify further investigations into specification-aware memoization. In a functional language that uses strict abstraction, whenever a function `Foo` is specified by giving an explicit definition of its return value in terms of its arguments, this specification can be used directly for filling in the memoization mappings for expressions involving that function.

§6.4 Model-Aware Iteration

Ideally, a loop invariant is an annotation that documents the insights that the programmer forged while designing an iterative algorithm. However, in mundane

cases, loop invariants can become tedious, and the necessity of annotating every loop with an invariant can be an obstacle rather than an aid. Some static analyses attempt to deal with this issue via heuristics for automatic inference of invariants [23]. Of course this approach is flawed in that automated invariant inference could cause code that does not behave as the programmer intended to be verified: the inferred invariant might be qualitatively different than the one the programmer would have written, and might reflect properties that she did not desire her code to exhibit. We suggest a different approach, which (in the spirit of abstraction) attempts to "reuse" invariants for simple iterative patterns by moving them into reusable components and invoking them with reader-friendly syntax.

Consider iterating over one of the data abstractions for "collections" that we have seen in this work. For example, `Queue` is an ordered collection of items that are processed in a "first in, first out" manner. `Stack` is like `Queue` but for the fact that it processes data "last in, first out." Many computational tasks involving these components are commutative: the order in which they compute with the collection's items is immaterial, *e.g.*, incrementing every entry in a `Queue` of integers. Moreover, some collections lack the notion of "order" altogether (anything modeled by a set or a multiset), or only offer a partial order, with some elements being incomparable (trees and binary trees). For components like these, the need for client code to define one particular order of computation among a collection's elements seems needlessly specific, it would be better to have the ability to just say what computation should be applied to each element.

Below we show a new Resolve contract for incrementing all of a `Queue`'s entries. We emphasize again a consequence of strict abstraction: the fact that `ADD_ONE`'s inductive definition seems to describe a left-to-right incrementing of its argument string's entries has no bearing on the order in which `AddOne` must actually compute its result.

```
1  contract AddOne enhances QueueOfIntegerFacility
2
3      definition ADD_ONE (
4              s: string of integer
5          ): string of integer
6          satisfies restriction
7              if (s = empty_string)
8              then ADD_ONE(s) = empty_string
9              else
10                 for all t: string of integer, i: integer
11                 where (s = <i> * t)
12                 (ADD_ONE(s) = <i+1> * ADD_ONE(t))
13
14     procedure AddOne (updates q: Queue)
15         ensures
16             q = ADD_ONE(#q)
17
18 end AddOne
```

Recalling the discussion of `MAP` in Chapter 2, we can reformulate this contract without introducing any new quantifications as follows:

```
1  contract AddOne enhances QueueOfIntegerFacility
2
3      definition INCREMENT (i: integer): integer
4                  is i+1
5
6      procedure AddOne (updates q: Queue)
7          ensures
8              q = MAP(INCREMENT, #q)
9
10 end AddOne
```

`MAP` in fact seems to be the perfect abstraction for problems of the sort we're considering. If we wanted to sum all of the `Queue`'s entries, `FOLD` would be apt.

One of many possible implementations of the `AddOne` contract is shown in the following listing.

```
1  realization Iterative implements AddOne
2      for QueueOfIntegerFacility
3
4      procedure AddOne (updates q: Queue)
5          variable tmp: Queue
6          loop
7              maintains
8                  tmp * MAP(INCREMENT, q) = MAP(INCREMENT,#q)
9              decreases
10                 |q|
11         while not IsEmpty (q) do
12             variable x: Integer
13             Dequeue (q, x)
14             Increment (x)
15             Enqueue (tmp, x)
16         end loop
17         tmp :=: q
18     end AddOne
19
20 end Iterative
```

However, it now becomes clear that only one line of code has anything to do with incrementing an integer, everything else is boilerplate for the iteration. Specifically, we will use the word "boilerplate" to refer to lines 4 through 16 of the `Iterative` realization of `AddOne`, where line 7 (the invariant) and line 13 (the interesting portion of the loop body) are the only portions free to change. Rather than write this boilerplate code for each such iteration we'd like to perform on a `Queue`, it would be nicer to just write programming expressions of the form `Map (q, Increment)`,

where the specification of `Map` is a straightforward application of our math definition `MAP`. The first problem with this approach is that `Increment` is a procedure, not a function. The math definition for `MAP` that we gave in Chapter 2 was parameterized by a function. We could define a new `MAP` function that is parameterized by a unary relation (the mathematical model of procedures with a single parameter), as follows:

```
1  definition MAP (
2        s: string of T1,
3        r: set of (T1, T2)
4     ): string of T2 satisfies
5     if (s = empty_string)
6     then MAP(s, r) = empty_string
7     else
8        there exists i: T1, t: string of T1
9           (s = <i> * t and
10           (i, j) is in r
11           MAP(s, r) = <j> * MAP(t, r))
```

This new definition of `MAP` accommodates unary programming procedures or functions,[35] but it is not satisfiable if `r` is not total. In addition to this complication, there are implementation issues involved with reifying operations as data so that they can be passed to `Map`. Instead, we could imagine implementing `AddOne` via a new syntactic construct as follows:

```
1  for each x in q
2     Increment (x)
3  end for
```

[35]If functions are viewed, as is common, as sets of pairs

`for each...in...end for` can be implemented as a macro that expands into the boilerplate evinced in the `Iterative` implementation of `Queue`, where the "interesting" portion of the loop is replaced by the body of the `for each`, and the invariant is changed so that the relation mapped to the strings `q` and `#q` is the composition of the specs of operations the `for each` body invokes on x. We discussed relational composition in Chapter 4, where we showed that the decoupled semantics of a sequence of procedure calls is just the appropriate relational composition of their specifications. To make `for each` available for a component, an extension should be written that provides the boilerplate appropriate to that component—`Set`s will not be iterated upon in the same manner as we have shown for `Queue`s, for example. Boilerplate for `Set` is shown below.

```
1  variable tmp: Set
2  loop
3     maintains
4        there exists s_str, old_s_str: string of Item
5           (elements(s_str) = s and
6            elements(old_s_str) = #s and
7            tmp union elements(MAP(___, s_str)) =
8              elements(MAP(___, old_s_str)))
9     decreases
10       |s|
11 while not IsEmpty(s) do
12    variable x: Item
13    RemoveAny (s, x)
14    ___
15    Add (tmp, s)
16 end loop
17 s :=: tmp
```

In general, `for each` could be considered just one instance of a broader family of iteration constructs that the programming language could offer for collections. Often

in `Set`s, for example, one might like to perform some computation on all distinct pairs of entries:

```
for each distinct (x, y) in s
   ---
end for
```

This of course would require different boilerplate to be provided, but the idea is that this syntax is readable, and natural to properly abstracted client reasoning, because it is phrased in terms of a mathematical model. Traditional approaches to the issue of iteration, *i.e.*, iterators and higher-order functions, lack this degree of abstraction. Iterators at best require new mathematical models [76], or else involve modularity problems such as reasoning about iterator return values that are references to the contents of data structures, as in Java. Higher-order functions are a common solution in settings that already reify operations as data, but they lack mathematical modeling; the meaning of a `map` expression in Haskell requires knowing the implementation of the function it is provided as an argument. Our proposed approach eliminates this problem, allowing client-view reasoning about all operations invoked in a `for each`.

§6.5 Conclusions

In this final chapter, we have demonstrated that a view of programming that emphasizes abstraction opens new avenues of research. We have seen that proof automation, language implementation, and language design all offer opportunities for embracing abstraction that the state of the art does not currently account for. We

hope that this work will continue, and that the power of abstraction will be ever more appreciated as it tames larger problems and engenders more reusable solutions.

More broadly, we have completed our discussion of abstraction as the key to programming. We have seen that generalization and reconceptualization ameliorate issues of specification and implementation, and that the power of a programmer to reason about her code is due to the use of good abstractions for hiding irrelevant details and providing a sound model of software component behavior. We began by defining our terms and surveying abstraction's role in science, and then focused on the term's multifarious usage in the field of software engineering. We discussed verification in particular, showing a specification language and a program proof technique that leverages strict abstraction into pure mathematics. We discussed data abstractions, saw that the issue of functional semantics could introduce unsoundness into modular client reasoning based on decoupled semantics, and presented a solution to the problem. Finally we discussed proof automation and future directions, thus concretizing our abstract discussion with real applications.

We conclude in the hope that this work lies along a critical path to solving Hoare's challange, and that the dream of a verifying compiler will someday be achieved, with abstraction in the starring role.

Bibliography

[1] ACL2 walking tour: Common lisp. `http://www.cs.utexas.edu/~moore/acl2/current/Common_Lisp.html`.

[2] 10th annual Oregon programming languages summer school curriculum. `http://www.cs.uoregon.edu/Research/summerschool/summer11/curriculum.html`, June 2011.

[3] B. M. Adcock. *Working Towards The Verifed Software Process*. PhD thesis, Department of Computer Science and Engineering, The Ohio State University, Columbus, Ohio USA, 2011.

[4] J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9, December 2007.

[5] M. Barnett, B.-Y. E. Chang, R. Deline, B. Jacobs, and K. R. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] J. Bloch. Extra, extra, read all about it: Nearly all binary searches and mergesorts are broken. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`, June 2006.

[10] D. Bronish, J. Kirschenbaum, B. Adcock, and B. W. Weide. On soundness of verification for software with functional semantics and abstract data types. Technical Report TR26, Department of Computer Science and Engineering, The Ohio State University, 2008.

[11] D. Bronish, J. Kirschenbaum, B. Adcock, and B. W. Weide. Unsoundness of referential transparency in languages with functional semantics and abstract data types: A proof and a solution. Submitted to the 2012 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), April 2012.

[12] D. Bronish, J. Kirschenbaum, and A. Tagore. A benchmark- and competition-based approach to software engineering research. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 43–46, New York, NY, USA, 2010. ACM.

[13] D. Bronish and H. Smith. Robust, generic, modularly-verified map: a software verification challenge problem. In *Proceedings of the 5th ACM workshop on Programming Languages meets Program Verification*, pages 27–30, New York, NY, USA, 2011. ACM.

[14] D. Bronish and B. W. Weide. A review of verification benchmark solutions using dafny. In *Proceedings of the 2010 Workshop on Verified Software: Theories, Tools, and Experiments*, 2010.

[15] F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *Comput. J.*, 31(3):243–247, 1988.

[16] A. Chlipala. The Ynot tutorial. `http://ynot.cs.harvard.edu/Tutorial.pdf`, July 2009.

[17] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[18] M. A. D. Darab. Towards a GUI for program verification with KeY. Master's thesis, Chalmers University of Technology, University of Gothenburg, January 2010.

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, 2004.

[20] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part II: specifying components in RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[21] G. W. Ernst, R. J. Hookway, and W. F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Trans. Softw. Eng.*, 20(4):288–307, Apr. 1994.

[22] G. W. Ernst, J. K. Navlakha, and W. F. Ogden. Verification of programs with procedure-type parameters. *Acta Informatica*, 18:149–169, 1982. 10.1007/BF00264436.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[24] J.-C. Filliatre, A. Paskevich, and A. Stump. VSTTE 2012 software verification competition problems. `https://sites.google.com/site/vstte2012/compet/problems`, November 2011.

[25] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[26] D. P. Friedman and M. Felleisen. *The Little Schemer (4th ed.)*. MIT Press, Cambridge, MA, USA, 1996.

[27] H. Friedman. Some algorithms for strings with applications to program verification. Technical Report OSU-CISRC-8/09-TR42, Department of Computer Science, Ohio State University, Aug. 2009.

[28] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[29] P. Graham. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[30] P. D. Groote, editor. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de logique*. Academia, Louvain-la-Neuve (Belgium), 1995.

[31] R. Harrison. *Abstract Data Types in Standard ML*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[32] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *ESOP '86: Proceedings of the European Symposium on Programming*, pages 187–196, London, UK, 1986. Springer-Verlag.

[33] W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH USA, 1995.

[34] W. D. Heym, T. J. Long, W. F. Ogden, and B. W. Weide. Mathematical foundations and notation of RESOLVE. Technical Report OSU-CISRC-8/94-TR45, The Ohio State University, August 1994.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[36] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[37] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[38] P. R. James and P. Chalin. Extended static checking in JML4: benefits of multiple-prover support. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 609–614, New York, NY, USA, 2009. ACM.

[39] M. Kaufmann. The ACL2 "isort" book. `http://code.google.com/p/acl2-books/source/browse/trunk/sorting/isort.lisp`.

[40] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[41] J. Kirschenbaum. *Investigations in Automating Software Verification*. PhD thesis, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH USA, 2011.

[42] J. Kirschenbaum, B. Adcock, D. Bronish, H. Smith, H. Harton, M. Sitaraman, and B. W. Weide. Verifying component-based software: Deep mathematics or simple bookkeeping? In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering (ICSR '09)*, pages 31–40, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] J. Kirschenbaum, B. M. Adcock, D. Bronish, P. Bucci, and B. W. Weide. Adapting isabelle theories to help verify code that uses abstract data types. In *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 67–74, November 2008.

[44] J. Kirschenbaum, H. Harton, and M. Sitaraman. A case study in automated, modular, and full functional verification. In *Proceedings of the Third Workshop on Automated Formal Methods (AFM '08)*, pages 53–58. ACM Press, July 2008.

[45] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st verified software competition: experience report. In *Proceedings of the 17th International Conference on Formal Methods (FM 2011)*, FM2011, pages 154–168, Berlin, Heidelberg, 2011. Springer-Verlag.

[46] J. E. Krone. *The Role of Verification in Software Reusability*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH USA, 1988.

[47] G. Kulczycki, H. Smith, H. Harton, M. Sitaraman, W. F. Ogden, and J. E. Hollingsworth. The location linking concept: A basis for verification of code using pointers. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2012)*, volume 7152 of *Lecture Notes in Computer Science*, pages 34–49. Springer-Verlag, 2012.

[48] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[49] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[50] K. R. M. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In *Proceedings of the 2010 Workshop on Verified Software: Theories, Tools, and Experiments*, pages 112–126. Springer-Verlag, 2010.

[51] K. R. M. Leino and M. Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of the 2010 Workshop on Verified Software: Theories, Tools, and Experiments*, 2010.

[52] S. Marlow, editor. *Haskell 2010 Language Report*. `http://www.haskell.org/onlinereport/haskell2010`, 2010.

[53] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[54] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[55] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science 1*, volume 19 of *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[56] B. Meyer. *Eiffel: the Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[57] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[58] E. Nagel and J. R. Newman. *Gödel's Proof*. New York University Press, revised edition, 2001.

[59] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[60] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjoberg, and B. Yorgey. *Software Foundations*. Distributed electronically, `http://www.cis.upenn.edu/~bcpierce/sf`, 2011.

[61] S. M. Pike, W. D. Heym, B. Adcock, D. Bronish, J. Kirschenbaum, and B. W. Weide. Traditional assignment considered harmful. In *The 2009 International Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 909–916, New York, 2009. ACM.

[62] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T. Leavens. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, pages 342–363. Springer, 2006.

[63] W. V. O. Quine. *Word and Object*. MIT Press, 1960.

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

[65] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Form. Methods Syst. Des.*, 14(1):75–125, Jan. 1999.

[66] M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. Weide. Building a push-button resolve verifier: Progress and challenges. *Formal Aspects of Computing*, 23(5):607–626, 2011.

[67] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.

[68] M. Sitaraman, B. W. Weide, and W. F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Trans. Softw. Eng.*, 23(3):157–170, 1997.

[69] H. Smith, H. Harton, D. Frazier, R. Mohan, and M. Sitaraman. Generating verified Java components through RESOLVE. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, ICSR '09, pages 11–20, Berlin, Heidelberg, 2009. Springer-Verlag.

[70] H. Smith, K. Roche, M. Sitaraman, J. Krone, and W. F. Ogden. Integrating math units and proof checking for specification and verification. In *Proceedings of the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 59–67, November 2008.

[71] H. Søndergaard and P. Sestoft. Referntial transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[72] N. Soundarajan, D. Bronish, and R. Khatchadourian. Formalizing reusable aspect-oriented concurrency control. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011)*, pages 111–114, July 2011.

[73] A. Tagore, D. Zaccai, and B. Weide. Automatically proving thousands of verification conditions using an SMT solver: An empirical study. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 195–209. Springer Berlin, 2012.

[74] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques (Tutorial Text)*, pages 24–52, London, UK, 1995. Springer-Verlag.

[75] D. F. Wallace. *Everything and More: A Compact History of* ∞. W. W. Norton & Company, Inc., 2003.

[76] B. W. Weide. SAVCBS 2006 challenge: Specification of iterators. In *Proceedings of FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 75–77, Portland, OR, November 2006.

[77] B. W. Weide. Software verification with towers of abstraction. *Foundational Adventures: Conference in Honor of the 60th Birthday of Harvey M. Friedman*, 2009.

[78] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *Proceedings of the 2008 Workshop on Verified Software: Theories, Tools, and Experiments*, pages 84–98. Springer-Verlag, 2008.

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

Appendix A: Resolve String Theory in Coq

```
Require Import BoolEq.
Require Import Arith.
Require Import Omega.

Inductive string (X:Type): Type :=
  | empty_string: string X
  | ext: string X -> X -> string X.

Implicit Arguments empty_string [[X]].
Implicit Arguments ext [[X]].

Theorem ext_not_empty: forall (X:Type) (a: X) (s: string X),
  (ext s a <> empty_string).
Proof.
  congruence.
Qed.

Theorem ext_inj_1: forall (X:Type) (a b: X) (s t: string X),
  (ext s a = ext t b) -> a = b.
Proof.
  congruence.
Qed.

Theorem ext_inj_2: forall (X:Type) (a b: X) (s t: string X),
  (ext s a = ext t b) -> s = t.
Proof.
  congruence.
Qed.

Theorem equal_extensions:
  forall (X:Type) (s t: string X) (x: X),
  (s = t) -> (ext s x) = (ext t x).
Proof.
  intros. rewrite -> H. reflexivity.
Qed.

Fixpoint concat (X:Type) (s t: string X): string X :=
  match t with
  | empty_string => s
```

```
  | ext t' x => ext (concat X s t') x
  end.

Implicit Arguments concat [[X]].

Notation "x ** y" :=
  (concat x y)  (at level 50, left associativity) : nat_scope.

Theorem concat_left_id:
  forall (X:Type) (s: string X), empty_string ** s = s.
Proof.
  intros. induction s. simpl. reflexivity.
    simpl. rewrite -> IHs. reflexivity.
Qed.

Theorem concat_right_id:
  forall (X:Type) (s: string X), s ** empty_string = s.
Proof.
  intros. simpl. reflexivity.
Qed.

Theorem concat_right_cancel:
  forall (X:Type) (a b c: string X), b ** a = c ** a -> b = c.
Proof.
  intros. induction a. simpl in H. assumption.
  simpl in H. apply ext_inj_2 in H. apply IHa. assumption.
Qed.

Theorem concat_assoc: forall (X:Type) (a b c: string X),
  (a**b)**c = a**(b**c).
Proof.
  induction c as [| c' x]. simpl. reflexivity.
  simpl. rewrite -> x. reflexivity.
Qed.

Fixpoint length (X:Type) (s: string X): nat :=
  match s with
  | empty_string => 0
  | ext s' x => S (length X s')
  end.

Implicit Arguments length [[X]].

Notation "| x |" := (length x) (at level 50): nat_scope.

Theorem len_0_implies_empty:
  forall (X:Type) (s: string X), |s| = 0 -> s = empty_string.
Proof.
  destruct s. intros. reflexivity.
  intros. simpl in H. inversion H.
Qed.
```

```
Theorem concat_sums_lengths:
  forall (X:Type) (s t: string X), |s**t| = |s|+|t|.
Proof.
  intros. induction t. simpl. omega. simpl. omega.
Qed.

Theorem equal_len_prefixes_1:
  forall (X:Type) (a b c d: string X),
    (a**b) = (c**d) -> (|b| = |d|) -> b = d.
Proof.
  induction b.
    intros. simpl in H0. symmetry in H0. symmetry.
      apply len_0_implies_empty. assumption.
    intros. destruct d. simpl in H0. inversion H0.
      simpl in H0. inversion H0. clear H0. simpl in H.
      inversion H. clear H.
      assert (b=d). apply IHb with (c:=c). assumption.
      assumption. rewrite -> H. reflexivity.
Qed.

Theorem equal_len_prefixes_2:
  forall (X:Type) (a b c d: string X),
    (a**b) = (c**d) -> (|b| = |d|) -> a = c.
Proof.
  induction b.
    intros. simpl in H0. symmetry in H0.
      apply len_0_implies_empty in H0.
      rewrite -> H0 in H. simpl in H. assumption.
    intros. destruct d. simpl in H0. inversion H0.
      simpl in H. inversion H. clear H. simpl in H0.
      inversion H0. clear H0. apply IHb with (d:=d).
      assumption. assumption.
Qed.

Theorem equal_implies_equal_len:
  forall (X:Type) (a b: string X), a = b -> |a| = |b|.
Proof.
  intros. rewrite -> H. reflexivity.
Qed.

Theorem ext_of_concat: forall (X:Type) (s t: string X) (x: X),
  ext (s ** t) x = s ** (ext t x).
Proof.
  intros. simpl. reflexivity.
Qed.

Notation "<< x >>" :=
  (ext empty_string x) (at level 50): nat_scope.

Theorem stringleton_not_empty:
```

```
  forall (X:Type) (x:X), <<x>> <> @empty_string X.
Proof.
  congruence.
Qed.

Theorem stringleton_len_1: forall (X:Type) (x:X), |<<x>>| = 1.
Proof.
  simpl. reflexivity.
Qed.

Theorem stringleton_inj:
  forall (X:Type) (x y: X), <<x>> = <<y>> -> x = y.
Proof.
  congruence.
Qed.

Theorem len_1_implies_stringleton:
  forall (X:Type) (s: string X),
    |s|=1 -> exists x:X, s = <<x>>.
Proof.
  intros. destruct s. simpl in H. inversion H.
  simpl in H. inversion H. apply len_0_implies_empty in H1.
    rewrite -> H1. exists x. reflexivity.
Qed.

Theorem concat_stringleton_is_ext:
  forall (X:Type) (s: string X) (x: X),
    (s ** <<x>>) = (ext s x).
Proof.
  intros. simpl. reflexivity.
Qed.

Fixpoint rev (X:Type) (s: string X): string X :=
  match s with
  | empty_string => empty_string
  | ext t x => <<x>> ** (rev X t)
  end.

Implicit Arguments rev [[X]].

Theorem rev_stringleton:
  forall (X:Type) (x: X), (rev (<<x>>)) = <<x>>.
Proof.
  simpl. reflexivity.
Qed.

Theorem rev_concat:
  forall (X:Type) (s t: string X),
    (rev (s ** t)) = (rev t) ** (rev s).
Proof.
  induction t. simpl. rewrite -> concat_left_id. reflexivity.
```

```
  simpl. rewrite -> IHt. rewrite -> concat_assoc. reflexivity.
Qed.

Theorem rev_rev_cancels:
  forall (X:Type) (s: string X), (rev (rev s)) = s.
Proof.
  induction s. simpl. reflexivity.
    simpl. rewrite -> rev_concat. rewrite -> IHs.
    rewrite -> rev_stringleton.
    rewrite -> concat_stringleton_is_ext. reflexivity.
Qed.

Theorem rev_transfer:
  forall (X:Type) (s t: string X), (rev s) = t -> s = (rev t).
Proof.
  intros. rewrite <- H. rewrite -> rev_rev_cancels.
  reflexivity.
Qed.

Theorem rev_equals_rev_implies_equal:
  forall (X:Type) (s t: string X), (rev s) = (rev t) -> s = t.
Proof.
  intros. apply rev_transfer in H.
  rewrite -> rev_rev_cancels in H. assumption.
Qed.

Theorem equal_implies_rev_equals_rev:
  forall (X:Type) (s t: string X), s = t -> (rev s) = (rev t).
Proof.
  intros. apply rev_transfer. rewrite -> rev_rev_cancels.
  assumption.
Qed.

Theorem concat_left_cancel:
  forall (X:Type) (a b c: string X), a ** b = a ** c -> b = c.
Proof.
  induction a. intros. rewrite -> concat_left_id in H.
    rewrite -> concat_left_id in H. assumption.
  intros. rewrite <- concat_stringleton_is_ext in H.
    rewrite -> concat_assoc in H.
    rewrite -> concat_assoc in H. apply IHa in H.
    apply equal_implies_rev_equals_rev in H.
    rewrite -> rev_concat in H. simpl in H.
    rewrite -> rev_concat in H. simpl in H.
    inversion H. apply rev_equals_rev_implies_equal.
    assumption.
Qed.

Theorem length_is_sum_implies_concat:
  forall (X:Type) (s: string X) (m n: nat),
    |s| = (m + n) -> exists! t: string X, exists! u: string X,
```

```
    s = t ** u /\ |t| = m /\ |u| = n.
Proof.
  induction s. intros. simpl in H. assert (m=0). omega.
  assert (n=0). omega. clear H. exists empty_string.
  unfold unique. split. exists empty_string. split. split.
  simpl. reflexivity. split. simpl. symmetry. assumption.
  simpl. symmetry. assumption. intros. decompose [and] H.
  rewrite -> concat_left_id in H2. assumption.
  intros. destruct H. decompose [and] H. clear H.
  rewrite -> H0 in H2. apply len_0_implies_empty in H2.
  symmetry. assumption.
  intros. simpl in H.
  case_eq (beq_nat n 0).
    intros. apply beq_nat_true in H0. assert (S (|s|) = m).
    omega. clear H. exists (ext s x). unfold unique. split.
    exists empty_string. split. split.
    rewrite -> concat_right_id. reflexivity. split. simpl.
    assumption. simpl. omega. intros. decompose [and] H.
    rewrite -> H0 in H5. apply len_0_implies_empty in H5.
    symmetry. assumption. intros. destruct H.
    decompose [and] H. clear H. rewrite -> H0 in H6.
    apply len_0_implies_empty in H6. rewrite -> H6 in H4.
    rewrite -> concat_right_id in H4. assumption.
    intros. apply beq_nat_false in H0.
    assert (|s| = m + (n-1)). omega. clear H.
    assert (exists! t : string X, exists! u : string X,
              s = t ** u /\ |t | = m /\ |u | = n-1).
    apply IHs. assumption. destruct H. unfold unique in H.
    decompose [and] H. clear H. destruct H2.
    decompose [and] H. clear H. exists x0. unfold unique.
    split. exists (ext x1 x). split. split. rewrite -> H5.
    simpl. reflexivity. split. assumption. simpl.
    rewrite -> H7. omega. intros. decompose [and] H.
    clear H. rewrite -> H5 in H6.
    rewrite -> ext_of_concat in H6.
    apply concat_left_cancel in H6. assumption. intros.
    destruct H. decompose [and] H. clear H.
    rewrite -> H5 in H9. rewrite -> ext_of_concat in H9.
    apply equal_len_prefixes_2 with (b:=(ext x1 x)) (d:=x2).
    assumption. simpl. rewrite -> H7. rewrite -> H11. omega.
Qed.

Theorem rev_length:
  forall (X:Type) (a: string X), |a|=|rev a|.
Proof.
  intros. induction a. simpl. reflexivity.
    simpl. rewrite -> concat_sums_lengths. simpl.
    rewrite -> IHa. reflexivity.
Qed.

Fixpoint prt_btwn (X:Type) (m n: nat)
```

```
                  (s: string X): string X :=
  match s with
  | empty_string => empty_string
  | ext s' x =>
      if (andb (Compare_dec.leb (|s'|) m)
               (negb (beq_nat (|s'|) m)))
      then empty_string
      else if (andb (Compare_dec.leb m (|s'|))
                    (andb (Compare_dec.leb (|s'|) n)
                          (negb (beq_nat (|s'|) n))))
           then (ext (prt_btwn X m n s') x)
           else prt_btwn X m n s'
  end.

Implicit Arguments prt_btwn [[X]].

Theorem prt_btwn_past_end:
  forall (X:Type) (n: nat) (s: string X),
    (n >= |s|) -> (prt_btwn 0 n s = s).
Proof.
  induction s as [| s']. intros. simpl. reflexivity.
  intros. simpl.
  assert (n >= |s'|).
    simpl in H. omega.
  assert (prt_btwn 0 n s' = s').
    apply IHs'. assumption.
  clear IHs'. clear H0. rewrite -> H1.
  assert (andb (Compare_dec.leb (|s'|) 0)
               (negb (beq_nat (|s'|) 0)) = false).
    destruct s'. simpl. reflexivity. simpl in H. reflexivity.
  rewrite -> H0. clear H0.
  assert (Compare_dec.leb (|s'|) n = true).
    simpl in H.
  assert (|s'| <= n). omega.
  apply leb_correct in H0. assumption.
  rewrite -> H0. clear H0.
  assert (negb (beq_nat (|s'|) n) = true).
    simpl in H. apply Bool.negb_true_iff.
    assert (n <> (|s'|)). omega. apply beq_nat_false_iff.
    omega. rewrite -> H0. clear H0. simpl. reflexivity.
Qed.

Inductive occurs_ct (X:Type): X -> string X -> nat -> Prop :=
  | empty_occurs:
      forall (v: X), occurs_ct X v empty_string 0
  | nonempty_occurs_in:
      forall (v x: X) (s: string X) (n: nat),
        occurs_ct X v s n ->
          x = v -> occurs_ct X v (ext s x) (n + 1)
  | nonempty_occurs_out:
      forall (v x: X) (s: string X) (n: nat),
```

```
      occurs_ct X v s n -> x <> v ->
        occurs_ct X v (ext s x) n.

Implicit Arguments occurs_ct [[X]].

Theorem occurs_stringleton_1: forall (X:Type) (x y: X),
  x = y -> occurs_ct x (<<y>>) 1.
Proof.
  intros. apply nonempty_occurs_in with (n:=0).
  apply empty_occurs. congruence.
Qed.

Theorem occurs_stringleton_0: forall (X:Type) (x y: X),
  x <> y -> occurs_ct x (<<y>>) 0.
Proof.
  intros. apply nonempty_occurs_out.
  apply empty_occurs. congruence.
Qed.

Theorem occurs_concat:
  forall (X:Type) (x: X) (s t: string X) (n1 n2: nat),
    occurs_ct x s n1 ->
      occurs_ct x t n2 -> occurs_ct x (s**t) (n1 + n2).
Proof.
  intros. induction H0.
    simpl. rewrite -> plus_0_r. assumption.
    simpl. rewrite -> plus_assoc. apply nonempty_occurs_in.
      apply IHoccurs_ct. assumption. assumption.
    simpl. apply nonempty_occurs_out. apply IHoccurs_ct.
    assumption. assumption.
Qed.

Theorem occurs_to_rev:
  forall (X:Type) (x: X) (s: string X) (n: nat),
    occurs_ct x s n -> occurs_ct x (rev s) n.
Proof.
    intros. induction H.
      simpl. apply empty_occurs.
      simpl. rewrite -> plus_comm. apply occurs_concat.
        apply nonempty_occurs_in with (n:=0).
        apply empty_occurs. assumption. assumption.
      simpl. apply occurs_concat with (n1:=0).
        apply nonempty_occurs_out.
        apply empty_occurs. assumption. assumption.
Qed.

Theorem occurs_from_rev:
  forall (X:Type) (x: X) (s: string X) (n: nat),
    occurs_ct x (rev s) n -> occurs_ct x s n.
Proof.
  intros. apply occurs_to_rev in H.
```

```
  rewrite -> rev_rev_cancels in H. assumption.
Qed.


Fixpoint power (X: Type) (s: string X) (n: nat): string X :=
  match n with
  | 0 => empty_string
  | S n' => (power X s n') ** s
  end.

Implicit Arguments power [[X]].

Theorem sum_exponents:
  forall (X:Type) (s: string X) (m n: nat),
    power s (m + n) = power s m ** power s n.
Proof.
  induction n.
    simpl. replace (m+0) with m. reflexivity. omega.
    replace (m + S n) with (S (m + n)). simpl. rewrite -> IHn.
      rewrite <- concat_assoc. reflexivity. omega.
Qed.

Theorem empty_power: forall (X:Type) (n: nat),
  power (@empty_string X) n = (@empty_string X).
Proof.
  induction n.
    simpl. reflexivity.
    simpl. rewrite -> IHn. reflexivity.
Qed.

Theorem length_of_power:
  forall (X:Type) (s: string X) (n: nat),
    |power s n| = n * |s|.
Proof.
  induction n. simpl. reflexivity.
    simpl. rewrite -> concat_sums_lengths.
    rewrite -> IHn. omega.
Qed.

Theorem pow_concat_eq_concat_pow:
  forall (X:Type) (s: string X) (n: nat),
    power s n ** s = s ** power s n.
Proof.
  induction n. simpl. rewrite -> concat_left_id. reflexivity.
    simpl. rewrite <- concat_assoc. rewrite -> IHn.
    reflexivity.
Qed.

Theorem power_of_rev: forall (X: Type) (s: string X) (n: nat),
  rev (power s n) = power (rev s) n.
Proof.
```

```
  induction n. simpl. reflexivity.
  simpl. rewrite <- IHn. rewrite <- rev_concat.
    apply equal_implies_rev_equals_rev.
    apply pow_concat_eq_concat_pow.
Qed.

Inductive prefix (X:Type): string X -> string X -> Prop :=
  | is_prefix: forall (s t: string X),
      ((exists u: string X, (t = s ** u)) -> prefix X s t).

Implicit Arguments prefix [[X]].

Theorem prefix_refl:
  forall (X:Type) (s: string X), prefix s s.
Proof.
  intros. apply is_prefix. exists empty_string.
  rewrite -> concat_right_id. reflexivity.
Qed.

Theorem prefix_antisym: forall (X:Type) (s t: string X),
  prefix s t -> prefix t s -> s = t.
Proof.
  intros. inversion H. clear H2 H3. inversion H0. clear H3 H4.
  destruct H1. destruct H2.
  assert (x = empty_string). rewrite -> H1 in H2.
    apply equal_implies_equal_len in H2.
    rewrite -> concat_sums_lengths in H2.
    rewrite -> concat_sums_lengths in H2.
    assert (|x| = 0). omega. apply len_0_implies_empty.
    assumption.
  rewrite -> H3 in H1. rewrite -> concat_right_id in H1.
  symmetry. assumption.
Qed.

Theorem prefix_trans: forall (X:Type) (s t u: string X),
  prefix s t -> prefix t u -> prefix s u.
Proof.
  intros. inversion H. clear H2 H3. inversion H0. clear H3 H4.
  destruct H1. destruct H2. rewrite -> H1 in H2.
  rewrite -> concat_assoc in H2.
  apply is_prefix. exists (x ** x0). assumption.
Qed.

Theorem unique_suffix: forall (X:Type) (s t: string X),
  prefix s t -> exists! u: string X, t = s ** u.
Proof.
  intros. inversion H. clear H1 H2. destruct H0.
  exists x. unfold unique. split. assumption.
  intros. rewrite -> H0 in H1.
  apply concat_left_cancel in H1. assumption.
Qed.
```

```
Theorem prefix_smaller: forall (X:Type) (s t: string X),
  prefix s t -> |s| <= |t|.
Proof.
  intros. inversion H. clear H1 H2. destruct H0.
  rewrite -> H0. rewrite -> concat_sums_lengths. omega.
Qed.
```

Appendix B: Resolve Solutions For The First Verified Software Competition

All problem statements are quoted from [45]. The solution to Problem Five was presented in an improved form in Chapter 4. The solution shown here is the one submitted for the publication of [45], and it includes some infelicities such as the lack of a representation field for the length of the `Queue`, and dead code at the beginning of `Dequeue`.

Problem One

> Given an `N`-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array. Prove the postcondition that `sum <= N * max`.

```
1  contract ArrayOfUnboundedIntegerAsStringFacility
2
3      uses UnboundedIntegerFacility
4
5      definition DIFFER_ONLY_AT (
6              s1: string of integer,
7              s2: string of integer,
8              d: finite set of integer
9          ) : boolean
10         is
11             |s1| = |s2|  and
12             for all i: integer
13                     where (i is not in d)
14                 (substring (s1, i, i + 1) = substring (s2, i, i + 1))
15
16     math subtype ARRAY_MODEL is (
17             lb: integer,
```

```
            ub: integer,
            s: string of integer
        )
        exemplar a
        constraint
            a.lb <= a.ub + 1  and
            |a.s| = a.ub - a.lb + 1

    type ArrayOfInteger is modeled by ARRAY_MODEL
        exemplar a
        initialization ensures
            a = (1, 0, empty_string)

    procedure SetBounds (updates a: ArrayOfInteger,
                         restores lower: Integer,
                         restores upper: Integer)
        requires
            lower <= upper
        ensures
            a.lb = lower  and  a.ub = upper

    procedure SwapItem (updates a: ArrayOfInteger,
                        restores i: Integer,
                        updates x: Integer)
        requires
            a.lb <= i  and  i <= a.ub
        ensures
            a.lb = #a.lb  and
            a.ub = #a.ub  and
            DIFFER_ONLY_AT (a.s, #a.s, {i - a.lb})  and
            substring (a.s, i - a.lb, i - a.lb + 1) = <#x>  and
            substring (#a.s, i - #a.lb, i - #a.lb + 1) = <x>

    function LowerBound (restores a: ArrayOfInteger): Integer
        ensures
            LowerBound = a.lb

    function UpperBound (restores a: ArrayOfInteger): Integer
        ensures
            UpperBound = a.ub

end ArrayOfUnboundedIntegerAsStringFacility
```

```
contract Get enhances ArrayOfUnboundedIntegerAsStringFacility

    procedure Get (restores a: ArrayOfInteger,
                   restores i: Integer,
                   replaces value: Integer)
        requires
            a.lb <= i and
            i <= a.ub
```

```
        ensures
            there exists s, t: string of integer
                (a.s = s * <value> * t and
                 |s| = i - a.lb)

end Get
```

```
contract FindMaxAndSum enhances ArrayOfUnboundedIntegerAsStringFacility

    definition SUM (s: string of integer): integer
        satisfies
            if (s = empty_string)
            then (SUM(s) = 0)
            else
                there exists x: integer, t: string of integer
                    (s = <x> * t and
                     SUM(s) = x + SUM(t))

    definition IS_MAX_OF (s: string of integer, i: integer): boolean
        is
            i is in elements(s) and
            for all j: integer where (j is in elements(s))
                (j <= i)

    procedure FindMaxAndSum (restores a: ArrayOfInteger,
                             replaces max: Integer,
                             replaces sum: Integer)
        requires
            a.s /= empty_string
        ensures
            sum = SUM(a.s) and
            IS_MAX_OF(a.s, max)

end FindMaxAndSum
```

```
realization Iterative implements FindMaxAndSum for
    ArrayOfUnboundedIntegerAsStringFacility

    uses Add for UnboundedIntegerFacility
    uses Get for ArrayOfUnboundedIntegerAsStringFacility

    procedure FindMaxAndSum (restores a: ArrayOfInteger,
                             replaces max: Integer,
                             replaces sum: Integer)
        variable count: Integer
        variable hi: Integer

        Clear (max)
        Clear (sum)
        count := LowerBound (a)
        hi := UpperBound (a)
```

```
17
18          Get (a, count, max)
19          Get (a, count, sum)
20          Increment (count)
21
22          loop
23              maintains
24                  a = #a and
25                  hi = a.ub and
26                  a.lb <= count and
27                  count <= hi + 1 and
28                  sum = SUM(substring(a.s, 0, count-a.lb)) and
29                  IS_MAX_OF(substring(a.s, 0, count-a.lb), max)
30              decreases
31                  a.ub - count + 1
32          while (not IsGreater (count, hi)) do
33              variable value: Integer
34              Get (a, count, value)
35              if (IsGreater (value, max)) then
36                  max := Replica (value)
37              end if
38              Add (sum, value)
39              Increment (count)
40          end loop
41
42      end FindMaxAndSum
43
44  end Iterative
```

Problem Two

Invert an injective (and thus surjective) array `A` of `N` elements in the subrange from `0` to `N-1`. Prove that `B[A[i]] = i` for `0 <= i < N`.

```
1   contract ArrayOfIntegerAsSetOfPairsFacility
2
3       uses UnboundedIntegerFacility
4
5       definition IS_PARTIAL_FUNCTION (
6               m: finite set of (i: integer, x: integer)
7           ): boolean
8           is
9               for all i1, i2, x1, x2: integer
10                  where ((i1, x1) is in m  and  (i2, x2) is in m)
11              (if i1 = i2 then x1 = x2)
12
13      definition IS_VALID_ARRAY (
14              lb: integer,
```

```
15              ub: integer,
16              table: finite set of (i: integer, x: integer)
17          ): boolean
18          is
19              IS_PARTIAL_FUNCTION (table)  and
20              for all i: integer
21                  (there exists x: integer
22                       (((i, x) is in table)  iff
23                        (lb <= i  and  i <= ub)))
24
25      math subtype ARRAY_MODEL is (
26              lb: integer,
27              ub: integer,
28              table: finite set of (i: integer, x: integer)
29          )
30          exemplar a
31          constraint
32              a.lb <= a.ub + 1 and
33              IS_VALID_ARRAY (a.lb, a.ub, a.table)
34
35      definition DIFFER_ONLY_AT (
36              a1: ARRAY_MODEL,
37              a2: ARRAY_MODEL,
38              i: integer
39          ): boolean
40          is
41              a1.lb = a2.lb  and
42              a1.ub = a2.ub  and
43              for all j: integer, x: integer
44                      where (j /= i)
45                  (((j, x) is in a1.table)  iff  ((j, x) is in a2.table))
46
47      definition ALL_INITIAL (
48              a: ARRAY_MODEL
49          ): boolean
50          is
51              for all i: integer
52                  ((i, 0) is in a.table iff (a.lb <= i and i <= a.ub))
53
54      type Array is modeled by ARRAY_MODEL
55          exemplar a
56          initialization ensures
57              a = (1, 0, empty_set)
58
59      procedure SetBounds (updates a: Array,
60                           restores lower: Integer,
61                           restores upper: Integer)
62          ensures
63              a.lb = lower  and
64              a.ub = upper  and
65              ALL_INITIAL (a)
```

```
66
67      procedure SwapItem (updates a: Array,
68                          restores i: Integer,
69                          updates x: Integer)
70          requires
71              a.lb <= i  and  i <= a.ub
72          ensures
73              (i, x) is in #a.table  and
74              (i, #x) is in a.table  and
75              DIFFER_ONLY_AT (a, #a, i)
76
77      function LowerBound (restores a: Array): Integer
78          ensures
79              LowerBound = a.lb
80
81      function UpperBound (restores a: Array): Integer
82          ensures
83              UpperBound = a.ub
84
85  end ArrayOfIntegerAsSetOfPairsFacility
```

```
1   contract InvertInjection enhances ArrayOfIntegerAsSetOfPairsFacility
2
3       definition VALUES_IN_BOUNDS (a: ARRAY_MODEL): boolean
4           is
5               IS_VALID_ARRAY (a.lb, a.ub, a.table) and
6               for all i, x: integer where ((i, x) is in a.table)
7                   (a.lb <= x and x <= a.ub)
8
9       definition IS_INVERTED_UP_TO (
10              c: integer, a: ARRAY_MODEL, b: ARRAY_MODEL): boolean
11          is
12              IS_VALID_ARRAY (a.lb, a.ub, a.table) and
13              IS_VALID_ARRAY (b.lb, b.ub, b.table) and
14              a.lb = b.lb and
15              a.ub = b.ub and
16              for all i, x: integer where ((i, x) is in a.table)
17                  ((i < c and (x, i) is in b.table) or
18                   (i >= c and (x, 0) is in b.table))
19
20      procedure InvertInjection (updates a: Array)
21          requires
22              VALUES_IN_BOUNDS(a)
23          ensures
24              a.lb = #a.lb and a.ub = #a.ub and
25              IS_INVERTED_UP_TO (a.ub+1, #a, a)
26  end InvertInjection
```

```
1   realization Iterative implements InvertInjection for
2       ArrayOfIntegerAsSetOfPairsFacility
3
```

```
4       procedure InvertInjection (updates a: Array)
5           variable counter: Integer
6           variable hi: Integer
7           variable temp: Array
8           counter := LowerBound (a)
9           hi := UpperBound (a)
10          SetBounds (temp, counter, hi)
11          loop
12              maintains
13                  a = #a and
14                  hi = #hi and
15                  temp.lb = a.lb and
16                  temp.ub = a.ub and
17                  a.lb <= counter and
18                  counter <= a.ub + 1 and
19                  IS_INVERTED_UP_TO (counter, a, temp)
20              decreases
21                  a.ub - counter + 1
22          while (not IsGreater (counter, hi)) do
23              variable value: Integer
24              variable counterCopy: Integer
25              SwapItem (a, counter, value)
26              counterCopy := Replica (counter)
27              SwapItem (temp, value, counterCopy)
28              SwapItem (a, counter, value)
29              Increment (counter)
30          end loop
31          a :=: temp
32      end InvertInjection
33
34  end Iterative
```

Problem Three

Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number `i` equal to the length of the list if there is no such element. Otherwise, the element at index `i` must be equal to zero, and all the preceding elements must be non-zero.

```
1   contract ListOfIntegerFacility
2
3       uses UnboundedIntegerFacility
4
5       math subtype LIST_MODEL is (left : string of integer,
6                                   right : string of integer)
7
8       type List is modeled by LIST_MODEL
```

```
9           exemplar l
10          initialization ensures
11              l.left = empty_string and l.right = empty_string
12
13      procedure Insert (updates s: List, clears x: Integer)
14          ensures
15              s.left = #s.left and
16              s.right = <#x> * #s.right
17
18      procedure Remove (updates s: List, replaces x: Integer)
19          requires
20              s.right /= empty_string
21          ensures
22              s.left = #s.left and
23              #s.right = <x> * s.right
24
25      procedure Advance (updates s: List)
26          requires
27              s.right /= empty_string
28          ensures
29              s.left * s.right = #s.left * #s.right and
30              |s.right| = |#s.right| - 1 and
31              |s.left| = |#s.left| + 1
32
33      procedure Reset (updates s: List)
34          ensures
35              |s.left| = 0 and
36              s.right = #s.left * #s.right
37
38      procedure AdvanceToEnd (updates s: List)
39          ensures
40              |s.right| = 0 and
41              s.left = #s.left * #s.right
42
43      function LeftIsEmpty (restores s: List): control
44          ensures
45              LeftIsEmpty = (|s.left| = 0)
46
47      function RightIsEmpty (restores s: List): control
48          ensures
49              RightIsEmpty = (|s.right| = 0)
50
51      function LeftLength (restores s: List): Integer
52          ensures
53              LeftLength = |s.left|
54
55      function RightLength (restores s: List): Integer
56          ensures
57              RightLength = |s.right|
58
59  end ListOfIntegerFacility
```

```
contract FindEntry enhances ListOfIntegerFacility

    procedure FindEntry (updates  s: List,
                         restores x: Integer,
                         replaces i: Integer)
        ensures
            s.left * s.right = #s.left * #s.right and
            x is not in elements(substring(s.left * s.right, 0, i)) and
            (if x is in elements(s.left * s.right) then
             substring(s.left * s.right, i, i+1) = <x>)

end FindEntry
```

```
realization Iterative implements FindEntry for ListOfIntegerFacility

    uses BooleanFacility

    local procedure MoveFenceTo (updates s: List, restores x: Integer)
        ensures
            s.left * s.right = #s.left * #s.right and
            x is not in elements(s.left) and
            (if (x is in elements(s.right)) then
                (substring (s.right, 0, 1) = <x>))

        variable value: Integer
        Reset (s)

        if not RightIsEmpty (s) then
            Remove (s, value)
            loop
                maintains
                    x = #x and
                    s.left * <value> * s.right =
                        #s.left * <#value> * #s.right and
                    x is not in elements (s.left)
                decreases
                    |s.right|
            while not RightIsEmpty (s) and not AreEqual (x, value) do
                Insert (s, value)
                Advance (s)
                Remove (s, value)
            end loop

            if AreEqual (x, value) then
                Insert (s, value)
            else
                Insert (s, value)
                Advance (s)
            end if
        end if
    end MoveFenceTo
```

```
39
40      procedure FindEntry (updates s: List,
41                           restores x: Integer,
42                           replaces i: Integer)
43          MoveFenceTo (s, x)
44          i := LeftLength (s)
45      end FindEntry
46  end Iterative
```

Problem Four

Write and verify a program to place N queens on an N×N chess board so that no queen can capture another one with a legal move. If there is no solution, the algorithm should indicate that.

```
1   contract SequenceOfIntegerWithSubstringFacility
2
3       uses UnboundedIntegerFacility
4
5       type Sequence is modeled by string of integer
6           exemplar t
7           initialization ensures
8               t = empty_string
9
10      procedure Add (updates t: Sequence,
11                     restores pos: Integer,
12                     restores x: Integer)
13          requires
14              0 <= pos  and  pos <= |t|
15          ensures
16              |t| = |#t| + 1 and
17              substring(t, 0, pos) = substring(#t, 0, pos) and
18              substring(t, pos, pos+1) = <x> and
19              substring(t, pos+1, |t|) = substring(#t, pos, |#t|)
20
21      procedure Remove (updates t: Sequence,
22                        restores pos: Integer,
23                        replaces x: Integer)
24          requires
25              0 <= pos  and  pos < |t|
26          ensures
27              |t| = |#t| - 1 and
28              substring(t, 0, pos) = substring(#t, 0, pos) and
29              substring(#t, pos, pos+1) = <x> and
30              substring(t, pos, |t|) = substring(#t, pos+1, |#t|)
31
32      procedure Swap (updates t: Sequence,
33                      restores pos: Integer,
34                      updates x: Integer)
```

```
35          requires
36              0 <= pos  and  pos < |t|
37          ensures
38              |t| = |#t| and
39              substring(t, 0, pos) = substring(#t, 0, pos) and
40              substring(t, pos, pos+1) = <#x> and
41              substring(t, pos, pos+1) = <x> and
42              substring(t, pos+1, |t|) = substring(#t, pos+1, |#t|)
43
44      function Length (restores t: Sequence): Integer
45          ensures
46              Length = |t|
47
48      function IsEmpty (restores t: Sequence): control
49          ensures
50              IsEmpty = (t = empty_string)
51
52  end SequenceOfIntegerWithSubstringFacility
```

```
1   contract FindNQueensPlacement
2       enhances SequenceOfIntegerWithSubstringFacility
3
4       definition COLUMNS_ARE_CONSISTENT (
5           i: integer, j: integer,
6           board: string of integer
7       ): boolean
8           is
9               there exists q1, q2: integer
10                  (val(board, i, q1) and val(board, j, q2) and
11                   q1 /= q2 and
12                   |q1 - q2| /= |i - j|)
13
14      definition PREFIX_IS_CONSISTENT_WITH (
15              len: integer, i: integer,
16              board: string of integer): boolean
17          is
18              len > 0 and
19              for all j: integer where (0 <= j and j < len)
20                  (COLUMNS_ARE_CONSISTENT (i, j, board))
21
22      definition IS_A_BOARD_PREFIX
23              (board_size: integer, board: string of integer): boolean
24          is
25              |board| <= board_size and
26              for all q: integer
27                  where (q is in elements(substring(board, 0, |board|)))
28                  (0 <= q and q < board_size)
29
30      definition CONSISTENT_BOARD_EXISTS_WITH
31              (n: integer, board: string of integer): boolean
32          is
```

```
33              there exists board_rest: string of integer
34                  (|board| + |board_rest| = n and
35                   IS_A_BOARD_PREFIX (n, board * board_rest) and
36                   for all i, j: integer
37                      where (0 <= i and i < n and
38                             0 <= j and j < n and i /= j)
39                      (COLUMNS_ARE_CONSISTENT (i, j, board * board_rest)))
40
41      procedure FindNQueensPlacement
42              (restores n: Integer, updates board: Sequence)
43          requires
44              |board| <= n
45          ensures
46              if CONSISTENT_BOARD_EXISTS_WITH (n, #board) then
47                  (|board| = n and
48                   CONSISTENT_BOARD_EXISTS_WITH (n, board))
49              else
50                  (board = #board)
51
52  end FindNQueensPlacement
```

```
1   realization Recursive implements FindNQueensPlacement
2       for SequenceOfIntegerWithSubstringFacility
3
4       uses BooleanFacility
5       uses Subtract for UnboundedIntegerFacility
6       uses AbsoluteValue for UnboundedIntegerFacility
7
8       local procedure IsConsistent (restores board: Sequence,
9                                     restores pos: Integer,
10                                    replaces result: Boolean)
11          requires
12              pos >= 0 and pos <= |board|
13          ensures
14              result = PREFIX_IS_CONSISTENT_WITH(pos, pos, board)
15
16          variable q: Integer
17          Clear (result)
18          loop
19              maintains
20                  board = #board and
21                  pos = #pos and
22                  q <= pos and
23                  result = PREFIX_IS_CONSISTENT_WITH(q, pos, board)
24              decreases
25                  pos - q
26          while IsGreater (pos, q) and IsTrue (result) do
27              variable qHeight: Integer
28              variable posHeight: Integer
29              variable horizDist: Integer
30              variable vertDist: Integer
```

```
31
32              Swap (board, q, qHeight)
33              Swap (board, pos, posHeight)
34
35              if AreEqual (qHeight, posHeight) then
36                  Negate (result)
37              else
38                  horizDist := Replica (pos)
39                  Subtract (horizDist, q)
40                  vertDist := Replica (posHeight)
41                  Subtract (vertDist, qHeight)
42                  AbsoluteValue (vertDist)
43                  if AreEqual (vertDist, horizDist) then
44                      Negate (result)
45                  end if
46              end if
47
48              Swap (board, q, qHeight)
49              Swap (board, pos, posHeight)
50
51              Increment (q)
52          end loop
53      end IsConsistent
54
55      procedure FindNQueensPlacement (restores n: Integer,
56                                      updates board: Sequence)
57          decreases n - |board|
58
59          variable boardLength: Integer
60          boardLength := Length (board)
61          if not AreEqual (n, boardLength) then
62              variable done: Boolean
63              variable i: Integer
64              Add (board, boardLength, i)
65              loop
66                  maintains
67                      n = #n and
68                      i <= n and
69                      boardLength = #boardLength and
70                      substring(board, 0, boardLength) =
71                          substring(#board, 0, boardLength) and
72                      (if done then
73                          |board| = n and
74                          CONSISTENT_BOARD_EXISTS_WITH(n, board))
75                  decreases
76                      n - i
77              while IsGreater (n, i) and not IsTrue (done) do
78                  variable iCopy: Integer
79                  variable result: Boolean
80                  iCopy := Replica (i)
81                  Swap (board, boardLength, iCopy)
```

```
82                  IsConsistent (board, boardLength, result)
83                  if IsTrue (result) then
84                      variable newBoardLength: Integer
85                      FindNQueensPlacement (n, board)
86                      newBoardLength := Length (board)
87                      if AreEqual (newBoardLength, n) then
88                          Negate (done)
89                      end if
90                  end if
91                  Increment (i)
92              end loop
93              if not IsTrue (done) then
94                  Remove (board, boardLength, i)
95              end if
96          end if
97      end FindNQueensPlacement
98
99  end Recursive
```

Problem Five

An applicative queue with a good amortized complexity can be implemented using a pair of linked lists, such that the front list joined to the reverse of the rear list gives the abstract queue. The queue offers the operations `Enqueue(item:T)` to place an element at the rear of the queue, `Tail()` to return the queue without the first element, and `Front()` to return the first element of the queue. The implementation must maintain the invariant `queue.rear.length <= queue.front.length` (prove this). Also, show that a client invoking the above operations observes an abstract queue given by a sequence.

```
1   contract QueueTemplate (type Item)
2
3       uses UnboundedIntegerFacility
4
5       math subtype QUEUE_MODEL is string of Item
6
7       type Queue is modeled by QUEUE_MODEL
8           exemplar q
9           initialization ensures
10              q = empty_string
11
12      procedure Enqueue (updates q: Queue, clears x: Item)
13          ensures
14              q = #q * <#x>
15
16      procedure Dequeue (updates q: Queue, replaces x: Item)
```

```
        requires
            q /= empty_string
        ensures
            #q = <x> * q

    function Length (restores q: Queue): Integer
        ensures
            Length = |q|

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

end QueueTemplate
```

```
realization StackRealization implements QueueTemplate

    uses StackTemplate
    uses Reverse for StackTemplate
    uses Concatenate for StackTemplate
    uses Add for UnboundedIntegerFacility

    facility StackFacility is StackTemplate (Item)

    type representation for Queue is (
            front: Stack,
            back: Stack
        )
        exemplar q
        convention
            |q.back| <= |q.front|
        correspondence function
            q.front * reverse (q.back)
    end Queue

    local procedure FixThings (
            updates s1: Stack,
            updates s2: Stack
        )
        ensures
            if |#s1| < |#s2| then
                (s1 = #s1 * reverse(#s2) and s2 = empty_string)
            else
                (s1 = #s1 and s2 = #s2)

        variable l1, l2: Integer
        l1 := Length (s1)
        l2 := Length (s2)
        if IsGreater (l2, l1) then
            Reverse (s2)
            Concatenate (s1, s2)
```

```
37          end if
38      end FixThings
39
40      procedure Enqueue (updates q: Queue, clears x: Item)
41          Push (q.back, x)
42          FixThings (q.front, q.back)
43      end Enqueue
44
45      procedure Dequeue (updates q: Queue, replaces x: Item)
46          if IsEmpty (q.front) then
47              Concatenate (q.front, q.back)
48              Reverse (q.front)
49          end if
50          Pop (q.front, x)
51          FixThings (q.front, q.back)
52     end Dequeue
53
54      function Length (restores q: Queue): Integer
55          variable len: Integer
56          Length := Length (q.front)
57          len := Length (q.back)
58          Add (Length, len)
59      end Length
60
61      function IsEmpty (restores q: Queue): control
62          IsEmpty := IsEmpty (q.front) and IsEmpty (q.back)
63      end IsEmpty
64
65  end StackRealization
```