UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

TRANSLATING CLOJURE TO ACL2 FOR VERIFICATION

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

RYAN LEE RALSTON
Norman, Oklahoma
2016

TRANSLATING CLOJURE TO ACL2 FOR VERIFICATION

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Rex L. Page, Co-Chair

Dr. Dean F. Hougen, Co-Chair

Dr. Christopher E. Weaver

Dr. David P. Miller

Dr. Matthew L. Jensen

Acknowledgements

Completing a dissertation is a long and difficult process. Throughout the process, I have been fortunate to have the support and encouragement from many others. I have known my adviser Rex Page for 14 years since my first semester as an undergraduate student. He has guided me through my bachelor's, master's, and doctoral education. I appreciate the advice and opportunities he has given me over the years. I also thank Dean Hougen for serving as my co-chair following Rex's retirement. Your suggestions helped me overcome several stumbling blocks in the research. I thank the rest of my committee of Chris Weaver, David Miller, and Matt Jensen for their time and advice. I thank my good friend Jonathan Boston for introducing me to Clojure and encouraging my initial research ideas. I thank my friends Chris Allen, Richard Gillock, Nick Stringer, and Javier Elizondo for listening to my frustrations and otherwise being supportive.

I thank Mom and Dad for instilling in me a desire for knowledge and a strong work ethic, in addition to their overwhelming support to pursue higher education. I thank my siblings Chad, Trevor, Zane, and Kaitlyn so that they don't feel left out, even though they probably deserve to be for all the hard times they have given me every Christmas that I was still in school.

Finally, I thank Kristi for her love and support while I worked on something that was much harder and took far longer than I ever expected.

Contents

List of Figures

Abstract

Software spends a significant portion of its life-cycle in the maintenance phase and over 20% of the maintenance effort is fixing defects. Formal methods, including verification, can reduce the number of defects in software and lower corrective maintenance, but their industrial adoption has been underwhelming. A significant barrier to adoption is the overhead of converting imperative programming languages, which are common in industry, into the declarative programming languages that are used by formal methods tools. In comparison, the verification of software written in declarative programming languages is much easier because the conversion into a formal methods tool is easier. The growing popularity of declarative programming — evident from the rise of multi-paradigm languages such as Javascript, Ruby, and Scala — affords us the opportunity to verify the correctness of software more easily.

Clojure is a declarative language released in 2007 that compiles to bytecode that executes on the Java Virtual Machine (JVM). Despite being a newer, declarative programming language, several companies have already used it to develop commercial products. Clojure shares a Lisp syntax with ACL2, an interactive theorem prover that is used to verify the correctness of software. Since both languages are based on Lisp, code written in either Clojure or ACL2 is easily converted to the other. Therefore, Clojure can conceivably be verified by ACL2 with limited overhead assuming that the underlying behavior of Clojure code matches that of ACL2. ACL2 has been previously used to reason about Java programs through the use of formal models of the JVM. Since Clojure compiles to JVM bytecode, a similar approach is taken in this dissertation to verify the underlying implementation of Clojure.

The research presented in this dissertation advances techniques to verify Clojure

code in ACL2. Clojure and ACL2 are declarative, but they are specifically functional programming languages so the research focuses on two important concepts in functional programming and verification: arbitrary-precision numbers ("bignums") and lists. For bignums, the correctness of a model of addition is verified that addresses issues that arise from the unique representation of bignums in Clojure. Lists, in Clojure, are implemented as a type of sequence. This dissertation demonstrates an abstraction that equates Clojure sequences to ACL2 lists. In support of the research, an existing ACL2 model of the JVM is modified to address specific aspects of compiled Clojure code and the new model is used to verify the correctness of core Clojure functions with respect to corresponding ACL2 functions. The results support the ideas that ACL2 can be used to reason about Clojure code and that formal methods can be integrated more easily in industrial software development when the implementation corresponds semantically to the verification model.

Chapter 1

Introduction

A successful project spends a significant portion of its life cycle in the maintenance phase with estimates of maintenance costing as low as 32% (McKee, 1984) to as high as 90% (Erlikh, 2000) of the total cost of software development. The cost of fixing a defect during maintenance is between 2 and 100 times more expensive than during the specification phase, depending on the type of software (Boehm & Basili, 2001; Shull et al., 2002). Corrective maintenance, which removes defects from released software, is estimated to only be 21% of the maintenance cost (Abran & Nguyenkim, 1991), but developers are the least productive at it compared to other maintenance activities (Basili et al., 1996; Graves & Mockus, 1998; Nguyen et al., 2009). *Formal methods*, which is the application of mathematical logic to the specification and development of software (Schumann, 2001), can reduce the defects in software and, by extension, the cost of maintenance.

Hoare (1996) attributes software's reliability in the absence of formal methods to modern processes and tools addressing the most common issues. The *uncommon* issues, though, represent a significant portion of the defects in released software. For example, infrequently executed code in applications disproportionately contributes to the number of defects discovered during maintenance (Hecht et al., 1997). Formal methods identify those defects more effectively than traditional testing because they consider all possible execution paths equally (Holzmann, 2001). Case studies at Amazon (Newcombe et al., 2015) and NASA (Havelund et al., 2001) report developers finding, through the use of formal methods tools, subtle bugs that their engineers

did not believe would be found otherwise.

Formal methods tools are generally built on top of declarative programming languages. Declarative languages implement software by "stating *what* is to be computed, but not necessarily *how* it is to be computed" (Lloyd, 1994), so they make effective specification languages. The syntax for declarative languages is based on mathematical notation and programs written in them can be analyzed using classical mathematical logic. One class of formal methods tool are interactive theorem provers (ITPs) that mechanize mathematical logic to create proof assistants, such as HOL (Gordon & Melham, 1993), Isabelle (Nipkow & Paulson, 1992), PVS (Owre et al., 1992), Coq (Théry et al., 2006), and ACL2 (Kaufmann et al., 2000). Through the use of these tools, an engineer can write a formal specification, simulate its execution, and verify that it exhibits desirable properties.

Since industry has preferred imperative languages for implementation, researchers have embedded the semantics of imperative languages within ITPs. For example, C (Tuch, 2008), C# (Börger et al., 2005), and Java (Strecker, 2002) have each been formalized in an ITP to verify properties of programs written in those languages. The programs that are verified can be significant, such as a seL4 microkernel written in C (Klein et al., 2009). Results like these are impressive, but are still motivated largely by academia. Industrial adoption of formal methods has been slow with part of the reason being related to the tools using declarative languages. Newcombe (2014) reports Amazon developers specifically complaining about the cognitive difficulties of learning a language that is only used during the design phase when formal methods are often applied. The complaint is empirically evident as well: developers take 20% more time to develop in unfamiliar languages (Boehm, 1987) and are 40% less productive when their work is fragmented evenly across two languages (Krein et al., 2010). However, improvements in tools and processes are the most significant cause of increases in productivity over the last 40 years (Nguyen et al., 2011) and the use of modern tools

and processes is highly correlated with programming language (Maxwell et al., 1996). Therefore, if the industrial adoption of declarative programming increased, industry could more easily adopt existing formal methods tools, the application of ITPs to programs would be easier, and it would lead to higher quality software.

There are reasons to believe declarative programming is more acceptable to industry now. Declarative semantics are being added to Java (first-order function objects, generics) and C# (anonymous functions, delegates) and newer languages like Javascript, Ruby, and Scala are designed to be multiple paradigm programming languages. This dissertation focuses on Clojure: a functional language, which is a specific type of declarative language, with a Lisp syntax that compiles to Java Virtual Machine (JVM) bytecode and runs on the JVM. It is a viable functional programming language for modern development that has been used by over 200 companies including Atlassian, Walmart, and Facebook (Miller, 2016). In this research, Clojure is formalized in ACL2, a mechanical logic for an applicative subset of Common Lisp (Kaufmann et al., 2000).

Thesis Statement

The bytecode of Clojure's core functions can be modeled in the ACL2 logic so that the functional composition of those core functions may be reasoned about in ACL2.

This thesis is supported by the formal verification in ACL2 of the underlying Clojure system that culminates in the verification of a user-defined function that combines the results. The ACL2 logic and the Clojure programming language are described in more detail in Chapter 2. A simple model of arbitrary-precision integers, or *bignums*, based on the Clojure implementation is verified in Chapter 3. The rest of the dissertation uses a complex model, described in Chapter 4, to reason about Clojure code. In the model, programs are represented in objects that mimic Java classes,

which is reasoned about using an abstraction layer presented in Chapter 5. Our model is used to formalize Clojure sequences, in Chapter 6, and apply the sequence formalization to the verification of a user-defined recursive function in Chapter 7. The remainder of this chapter identifies the contributions of the dissertation.

1.1 Contributions

Moore (2002) challenged the community to develop a *verified stack* of components that can be combined to create a practical computing device. At each layer of the stack, the semantics of the input language must be formalized, which is non-trivial for high-level programming languages. A novelty of our choice of Clojure is that ACL2 is already a formalization of its Lisp-based semantics. Despite that benefit, it is still important to verify that the Clojure implementation is correct with respect to the ACL2 formalization of Lisp and that ACL2 can be used to reason about Clojure code. By doing so, it creates a verified user interface — the Clojure programming language — for a end-user engineer to develop and verify new systems. This research makes four contributions to the current literature to verify the Clojure programming language: (1) a verified bignum addition that addresses the unique representation of Clojure's bignums, (2) a model of the JVM that specifically addresses aspects of verifying features of Clojure's underlying implementation, (3) a big-step verification style that effectively reasons about large programs, and (4) an abstraction of Clojure's sequences, verification of Clojure's sequence functions, and a demonstration that the results can be applied to verify a user-defined sequence function. Contribution (1) is also significant as a verification of a widely distributed Java bignum class that may be used to develop software security features. Contributions (2) and (3) are also applicable to any verification approach that is based on machine models.

1.1.1 Verification of Bignum Addition

A *bignum* is an unbounded number that is implemented as an array of words, where each word is a fixed-sized number in the implementation language. Theorem provers can reason about infinite types like natural numbers. When theorem provers are used to reason about code written in programming languages with fixed-sized numbers, researchers either represent the modeled language's integers with bignums, as in Hardin (2015), or define a new sized number type, as in Klein et al. (2009). The use of bignums is easier and it simplifies verification because existing, extensive proof libraries for infinite types are available. New types are harder to implement and verify, but the results are more accurate. Since Clojure supports bignums, it is possible to write Clojure code that can be accurately and easily verified, if the underlying arithmetic is verified. We verify an addition function based on Clojure's implementation.

Other languages have verified bignum arithmetic including the Piton language (Moore, 1989), a formal C-like language named C0 (Fischer, 2007), Ada (Berghofer, 2012), and machine code (Affeldt, 2013; Myreen & Curello, 2013). The logic of bignum arithmetic is also similar to hardware-based arithmetic on bit-vectors by considering the word size to be 1-bit. Arithmetic has been verified as well for different hardware (Hunt, 1994; Swords & Davis, 2011; Russinoff, 2005). In each of those examples, the representation of bignums is conducive to inductive reasoning because the least significant component of the number is operated on first. Clojure uses Java's `BigInteger` class to support bignums, which represents numbers with the most significant component first. The verification effort is significantly affected by the changes. In Chapter 3, we convert a proof similar to that of Moore (1989) to apply to the more difficult `BigInteger` representation.

The bignum verification is also a contribution independent of its use by Clojure. Bignums are used in cryptography (Denis & Rose, 2006) and one of the existing im-

plementations has been used for that purpose (Berghofer, 2012). `BigInteger`, unlike the other examples in the literature, is a widely distributed Java implementation of bignum arithmetic and security is an important area of interest for verification.

1.1.2 JVM Model for Compiled Clojure

A significant portion of the Clojure programming language is implemented in Java. The remaining non-Java portion is the Clojure `core` library that is implemented using Clojure. The `core` library defines the 448 standard functions of the language. Both portions of Clojure compile to the JVM. In this dissertation, we verify the underlying Java code and `core` functions for a specific Clojure data structure. To do so, we contribute a formal model of the JVM and an abstraction layer for programs that run on the model.

Our model of Clojure is named *MC* and is based on an existing formal model of the JVM (Moore & Porter, 2002). The existing model lacks support for interfaces, but Clojure uses interfaces extensively to invoke methods and check the type of objects. MC adds support for the bytecode instructions INVOKEINTERFACE and CHECKCAST and updates support for the instruction INSTANCEOF to recognize interface types. MC is described in Chapter 4.

MC is not sufficient by itself to verify a large program like Clojure. In a JVM model similar to MC, Liu (2006) finds the direct use of the model to scale poorly for programs with many classes. In addition to MC, we contribute an abstraction layer, described in Chapter 5, that can be used to reason about larger programs.

1.1.3 Big-Step Style Verification

Models are reasoned about in both *small-step* and *big-step* semantics, where the difference between the two is the amount of computation performed in a single transition. MC is a small-step semantic that is used to reason about programs with transitions

that execute a single bytecode instruction. Small-step semantics are capable of verifying programs of any size, but big-step semantics are easier to apply to large programs. Since Clojure is a large program, we introduce an ACL2 macro, in Chapter 4, to reason about programs in MC using big-step semantics where each transition executes an entire method. Our `->` macro combines series of small-step transitions into a single step to equate machine states that would eventually converge if both were executed indefinitely. It is based on the implementation given by Manolios & Moore (2003) for an earlier model of the JVM, but we modify it to improve the reliability that theorems are useful rewrite rules and apply it to the verification of a large system.

1.1.4 Verification of a Recursive Sequence Clojure Function

The final contribution is the verification of a recursive Clojure function that operates on sequences, which is a Clojure data structure similar to a list. Recursive functions are typically verified using inductive reasoning, which requires the inputs to be well-ordered. Sequences are well-ordered when created and operated on by Clojure functions, but MC operates on sequences as references to objects in a heap. From that perspective, MC cannot guarantee that sequences are well-ordered. We contribute an abstraction, which is described in Chapter 6, of sequences that can be used to guarantee that they are well-ordered. Using the sequence abstraction, we also verify the correctness of seven Clojure functions that operate on sequences. In Chapter 7, the abstraction and verified functions are combined together in an inductive proof of a sequence recursive function. The contribution is similar to the verified Java list implementation in Moore (2003), but that version is a small "toy" example that uses two Java classes and three methods. Clojure's implementation of sequences requires 17 classes and 37 methods.

1.2 Summary

A verified Clojure `core` library, including the verification of the underlying Java bytecode, enables developers to verify software more easily and the integration of verification into development processes is more flexible than traditional formal methods approaches. Since formal methods prioritize specification, the costs of using formal methods in software development are paid early in the life-cycle and the return on the investment is not gained until the maintenance phase. Since Clojure and ACL2 share a syntax and semantics, much of the overhead in applying formal methods tools is avoided because one implementation is applicable to both systems. In a traditional formal methods process, ACL2 can be used to formally specify the software and then applied to Clojure for implementation. However, a strength of our approach is the system can be developed in Clojure without a formal specification, as is common in development, and verified later using the Clojure code in ACL2. This shifts the cost of applying formal methods later into the process and closer to the maintenance phase when the risks and rewards are more noticeable.

Chapter 2

Research Methodology

The concept of verified correctness is appealing, but the process is not straightforward. A process must formally specify desired properties and define how to model a system to verify those properties. The process we use is significantly informed by Moore (1999). Moore constructed a model of the JVM in ACL2 that executed bytecode generated by compiling Java programs. The behavior of bytecode programs was verified to be equivalent to the behavior of "semantically close" ACL2 programs. Since the semantics of the implementations were close, additional properties of the Java programs could be analyzed by reasoning about the equivalent ACL2 programs. However, the semantic similarities were artificially enforced by the implementation of the Java version of the program and the perspective of the developer. In this research, the semantics are closer between the Clojure and ACL2 implementations and the closeness is inherent to the language paradigm.

This dissertation, like Moore (1999), verifies the underlying imperative behavior of Java bytecode programs is equivalent to similar ACL2 programs for the purpose of reasoning about the ACL2 programs. In our case the programs are the compiled form of the core Clojure functions. If the behavior of a set of Clojure functions is equivalent to ACL2 versions, the composition of those Clojure functions is equivalent to the same composition in ACL2. Our process is to model the JVM and the underlying bytecode generated from Clojure functions, verify the functions are equivalent to corresponding ACL2 functions, and show that new functions can be composed in Clojure and reasoned about in ACL2. This chapter begins with definitions for key

terms. Later in the chapter, ACL2 and Clojure are defined in more depth and our methodology is explained.

2.1 Definitions

Several terms are used frequently in this dissertation that have more than one definition. Those terms are defined and discussed in this section to avoid ambiguity.

2.1.1 Models

Verification is performed on *models* of the software being analyzed. Models, however, can vary significantly depending on the investigation. In this dissertation, three different types of models are used. The simplest type of model is a formal, mathematical definition of an algorithm. It works as a specification of the algorithm that can ignore most of the implementation concerns that affect production quality software. The verification of bignums is performed on such a model.

The other two types of models are related. The first of these is the formal implementation of the JVM that is constructed in Chapter 4. This JVM model, named *MC* in this dissertation for *model of Clojure*, can simulate the execution of JVM bytecode programs. The bytecode programs are the third type of model discussed in this dissertation. The following subsections define terms related to models of machines like the JVM that execute programming languages. Such models are categorized across three dimensions: program semantics, style of embedding, and step semantics.

Program Semantics

Program semantics formally define the meaning of a program. Different types of semantics exist that capture different concepts. In this work, the model is implemented using *operational semantics*. Operational semantics define the "meaning of a pro-

gramming language by specifying how it executes on an abstract machine" (Winskel, 1993). Our model is an abstract JVM that defines the operational semantics of JVM instructions as transition functions from a start state to an end state. The presented work is limited to operational semantics, but denotational (Scott & Strachey, 1971) and axiomatic (Hoare, 1969) semantics are both well-established alternatives to describing languages.

Deep and Shallow Embeddings

Approaches to embedding programs in logic are divided into *deep embeddings* and *shallow embeddings*. A deep embedding represents programs as abstract data types and defines functions in the logic that give the programs meaning. A shallow embedding defines the behavior of the program as functions in the logic (Boulton et al., 1992). Our model is a deep embedding of the JVM that represents programs as data in a structure based on the specification for JVM class files. Since Clojure and ACL2 are semantically similar languages, the shallow embedding of a Clojure function is the corresponding ACL2 function.

Shallow embeddings are easier to implement, but the analysis performed in the logic is less likely to apply to the implemented system. It can be much more difficult to verify properties in a deep embedding but the results reflect the properties of the system more reliably. Our verification of the deep embedding in MC is intended to increase the likelihood that a shallow embedding of a Clojure function describes its actual behavior.

Step Semantics

Operational semantics are implemented as a transition from configuration s to s'. Commonly, the transitions are implemented in either a *small-step* or *big-step* style. Small-step semantics are single step transitions applied repeatedly to define program

behavior. In the context of the JVM, a single step is the execution of an instruction. Small-step semantics are able to reason about non-terminating programs and cases where an application behaves incorrectly (Leroy, 2009). Big-step semantics are transitions from an initial configuration to a final configuration. Big-step semantics are effective at verifying compilers because programs can be decomposed into smaller programs (Ciobâcă, 2013). Both step styles are used in this research.

2.1.2 Conventions

In this research, the behavior of Clojure functions are equated to the behavior of corresponding ACL2 functions. The analysis is performed by reasoning about Java programs and their compiled bytecode. To be as clear as possible, the following terminology is used precisely:

- A *function* is a Clojure or ACL2 function.
- A *method* is either a Java method or the MC representation of Java bytecode.
- An *operation* is a term to describe program behavior that can be used to refer to methods and functions.
- A Clojure function is *mapped* to an ACL2 function by equating the behavior of the Clojure function's bytecode method to the behavior of the ACL2 function.

References to code in the text are written in a monospace font and follow naming conventions. Java classes are written in Pascal case so every word in an identifier is capitalized including the first. Java methods are written in Camel case, where every word in an identifier is capitalized except the first. For example, the Java class `LinkedList` has an `addFirst` method. Clojure and ACL2 functions are written in lowercase characters with a "`-`" separating words. For example, Clojure has a `replace-first` function in its core library.

2.2 ACL2

ACL2 is a functional programming language and a mechanical theorem prover. Functions defined in the programming language are introduced into the logic database, often referred to as the *world*, of the theorem prover. This section is an introduction to the ACL2 programming language, a brief overview of the reasoning system, and a description of the process of guiding ACL2 to prove theorems.

2.2.1 The Programming Language

Common Lisp is a popular, standard dialect of the functional programming language Lisp. Lisp expressions are contained within parenthesis and written in prefix notation. For example, adding `x` and `y` is expressed in Lisp as `(+ x y)`. The following code shows the definition of a Fibonacci function in Common Lisp,

```
1 (defun fibonacci (n)
2   (if (<= n 0)
3       0
4     (if (equal n 1)
5         1
6       (+ (fibonacci (- n 1))
7          (fibonacci (- n 2))))))
```

The ACL2 programming language is a subset of Common Lisp that removes features of the language that cause side-effects. Specifically, the language removes access to a global state and destructive operations that modify data in-place. The lack of side-effects means functions are referentially transparent, so expressions within a formula can be rewritten to equivalent expressions without changing the meaning of the formula. The mechanical theorem prover relies on functions being referentially transparent.

The ACL2 logic is one of total recursive functions. As such, functions defined in ACL2 are *total* meaning that they terminate on all inputs. The theorem prover

verifies that a new function is a total function before accepting it into the world. Under certain conditions, it is possible to define and reason about partial functions that may not terminate for some inputs (Manolios & Moore, 2003), which we do to define our big-step function in Chapter 4. However, programs containing partial functions can not be executed so they are avoided whenever possible.

Programs written in ACL2 can be compiled by any compliant Common Lisp implementation. Features have been added to the language to support efficient run-time execution. In some cases, code written in ACL2 has been shown to be comparable in performance to C code (Wilding et al., 2001). Despite its potential, there are no known cases of ACL2 being used to develop production code. The ACL2 programming language is specifically used to implement and reason about models of programs.

2.2.2 The Theorem Prover

ACL2 extends the syntax of Common Lisp to support the definition of theorems. The ACL2 theorem prover searches for a proof that verifies the truth of a theorem submitted to it. Once a theorem is verified by ACL2, it is added to the world and can be applied by the prover to verify new theorems. Kaufmann et al. (2000) provides a detailed description of the techniques that the theorem prover uses. In this section, we focus on the structure of ACL2 theorems, a general overview of rewriting, the available rule classes, and induction.

Conjectures are supplied to the theorem prover using `defthm`. If the theorem prover can ascertain the veracity of the conjecture, it is added as a theorem to the world. To illustrate, consider a function `absolute(x)` that computes the absolute value of `x`. `absolute` is idempotent because applying it multiple times is the same as applying it once. The idempotency of `absolute` is submitted to ACL2 using the theorem,

```
(defthm idempotency-of-absolute
  (implies (integerp x)
           (equal (absolute (absolute x))
                  (absolute x))))
```

Since ACL2 functions are total, the theorem prover will, by default, attempt to verify conjectures for any type of input. Idempotency is a property of `absolute` for numeric values, so the hypothesis of `idempotency-of-absolute` specifies `x` is an integer.

The ACL2 theorem prover mechanically verifies theorems by applying a powerful rewriting system to formulas. If `idempotency-of-absolute` is a theorem in the world, the theorem prover may apply it during the proof attempt of a new theorem. Imagine the theorem prover produces a subgoal that contains the expression `(absolute (absolute y))`. The expression is matched to the equivalent expression in the theorem's conclusion. If the theorem prover can establish that `y` is an integer, it can apply the rule and rewrite the expression `(absolute (absolute y))` to `(absolute y)`.

One of the strengths of the ACL2 theorem prover is its ability to apply induction on recursive functions definitions. ACL2's "definitional principal" (Kaufmann et al., 2000) requires recursive functions to continuously decrease a parameter for each recursive call until reaching a terminating state. Based on how the parameter is decreased, ACL2 associates induction schemes with the function in its database. The theorem prover uses induction schemes to generate the base and inductive subgoals and then applies its other techniques to verify the subgoals. The two most common induction schemes are (1) decrementing a number towards zero and (2) reducing a list towards the empty list. Expert ACL2 users learn to write function definitions that produce inductive subgoals that are easy to solve.

2.2.3 The Method

ACL2 automatically proves significantly complex conjectures, but it is designed to be an *interactive* theorem prover. Its expected use is as a proof assistant where a user guides the theorem prover towards a solution. Expert users learn how to guide the theorem prover by adding lemmas, managing the world, and directing proof attempts using hints. Those individual techniques are most successful when users apply *The Method* (Kaufmann et al., 2000), a process for proving complex theorems in ACL2.

At its heart, The Method is the strategy of creating a proof sketch, implementing it in ACL2, and using the ACL2 output to fill in the messing pieces. For example, imagine a user that desires to prove a theorem T. Using The Method, that user sketches a proof that states that T is provable from lemmas T_1, T_2, and T_3. Once the proof is sketched, the user implements `(defthm T-1 t)` in ACL2 to correspond to the lemma T_1. If the attempt is successful, the user moves on to T_2, T_3, and eventually T. If the proof attempt of T_1 fails, the user inspects the output to determine why. In some cases, ACL2 does not "know" enough to prove T_1 and the proof sketch needs to be expanded with more lemmas. In other cases, the user may be wrong and T_1 is not true, in which case, the proof sketch must be fixed. Eventually, the user implements enough knowledge, in the form of theorems, to attempt to verify the desired theorem T. If the attempt fails, the process continues in the same way as verifying the lemmas. The ACL2 output is examined and used to determine why the attempt failed. At this point, it is possible to discover that T is not a theorem and be forced to reconsider the entire problem. Importantly, The Method is a recommendation that users informally sketch out a proof by hand and refine it using ACL2.

2.3 Clojure

Clojure is a Lisp programming language that compiles to JVM bytecode instructions. A large portion of the language, including data structures and base functionality, is implemented using Java. The remaining portion that defines the fundamental functions is implemented in Clojure. Code from both portions is analyzed in this research.

The Java portion of the Clojure code implements the low-level details required to create a functional language that targets the JVM. Functions, for example, are objects in Clojure. The features of functions are implemented in Java using the abstract class `AFunction` and several interfaces. When a function definition is compiled, a new class is generated that extends the abstract class and implements the appropriate interfaces. All Clojure data structures are partially implemented as Java classes.

The Java code also implements a layer that performs basic operations on the low-level data structures. Clojure's run-time class `RT` is the primary link between code written in Clojure and the underlying Java portions of the implementation. `RT` implements static methods that route behavior to the appropriate data structures. For example, `RT` implements methods to create a sequence and access its data. Sequences are implemented using more than one class, so the `RT` methods determine, based on the type of the inputs, what class should execute the operation.

The `core` library, written in Clojure, glues the language to the underlying Java code. The Clojure functions that call Java methods do so by using the `.` operator. The `cons` function, shown below, calls the static `cons` method in the `RT` class.

```
1 (def
2  ^{:arglists '([x seq])
3    :static true}
4  cons (fn* ^:static
5            cons [x seq] (. clojure.lang.RT (cons x seq))))
```

The `core` library contains 579 definitions which includes 448 function definitions.

In this research, the JVM bytecode generated from each portion of the system is analyzed with respect to the sequence data structure. The classes that implement the sequence structure are analyzed in Chapters 5 and 6. The `core` functions that operate on sequences, as well as the `RT` methods that those functions call, are analyzed in Chapter 6. The complete analysis equates Clojure functionality on sequences to that of lists in ACL2.

2.4 Methodology

We will investigate a system that is written in Clojure and compiled to Java bytecode. We will reason about the system in ACL2. The Clojure and ACL2 syntaxes are both inspired by Common Lisp so the mapping between the two is straightforward. As an illustration of the intent of this research, let us consider an implementation of Fibonacci. Figure 2.1 shows a Clojure implementation of Fibonacci and an ACL2 implementation.

Despite small differences in syntax, there is a clear mapping from the Clojure implementation to the ACL2 model. But the Clojure function is compiled to Java bytecode before execution, so the mapping is only valid if the compiled bytecode is equivalent to the ACL2 function. For example, the `+` operator in Clojure must be equivalent to ACL2's `+` function. This is particularly important because Clojure must incorporate the JVM's type system in the compilation so a math operator is implicitly boxing and unboxing numerics. It is a goal of this research to verify functions in the Clojure core library so a developer can assume equivalence between the two languages.

Our research models the behavior of programs developed in Clojure version 1.7.0. The analysis is performed in an ACL2 model of the JVM that is named *MC* for *model of Clojure*. MC is a modification of the M5 model (Moore & Porter, 2002). It

```
; Clojure implementation
(defn fibonacci [n]
  (if (<= n 0)
      0
    (if (equal n 1)
        1
        (+ (fibonacci (- n 1))
           (fibonacci (- n 2))))))

; ACL2 model
(defun fibonacci (n)
  (if (<= n 0)
      0
    (if (equal n 1)
        1
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2))))))
```

Figure 2.1: Fibonacci Functions

is described in Chapter 4. The ACL2 code is developed in the ACL2 Sedan version 1.1.7.1 (Dillinger et al., 2007) in Compatibility mode using ACL2 version 7.0.0.

The compiled class files that execute on the JVM are translated into MC class declarations. A series of standard functions are defined for each class included in an analysis. The functions simplify the process for configuring a state in a manner that allows ACL2's logic to reliably apply knowledge defined in the class declarations. The format for the standard class functions is defined in Chapter 5.

The research analyzes two important features of functional programming as it relates to the implementation of Clojure: structural recursion and arbitrary-precision numbers. For structural recursion, a template for a sequence recursive function is defined. Sequences are modeled in ACL2 as an abstraction on top of the standard class functions. The function dependencies of the template are mapped from Clojure's implementation to ACL2. This is described in detail in Chapter 6. In Chapter 7, an example recursive function is implemented in Clojure, compiled to MC, and mapped

to ACL2.

For arbitrary-precision numbers, the algorithm for adding magnitudes is implemented in ACL2 and verified to produce the correct magnitude. The proof is described in Chapter 3.

Chapter 3

Big-Endian Bignum Arithmetic

Similar to software libraries, proof libraries encapsulate work that can be reused to simplify the verification of more complex problems. Many proof libraries that exist for theorem provers are built to reason about infinite number sets such as the naturals or mathematical integers. Theorem provers that are integrated into functional languages use bignums — numeric types that can represent arbitrarily large numbers — to represent numbers from an infinite set. When a theorem prover is used to verify a software system, researchers must either define numeric types that mimic the types used in the system or ignore the difference. Defining specific numeric types results in a more accurate model of the system, but it can also limit the applicability of existing proof libraries.

Fortunately, Clojure supports bignums, so systems developed with Clojure can be accurately verified without sacrificing the use of existing bignum proof libraries. However, since Clojure runs on the JVM, the representation of bignums and the associated arithmetic operations are implemented in Java code. Clojure's underlying bignum representation, which relies on Java's `BigInteger` class, needs to be verified to claim bignum operations are accurately modeled.

Abstractly, a bignum is stored in memory as an array of words where each word is a primitive numeric type in the implementation language. Individual arithmetic operations iterate over the arrays of two bignums to calculate a result. Concrete implementations of bignum arithmetic have previously been verified for the Piton language (Moore, 1989), a formal version of C (Fischer, 2007), and machine code (Af-

feldt, 2013; Myreen & Curello, 2013). For each of those implementations, the bignum representation was designed to be conducive to inductive reasoning. In comparison, `BigInteger` is a widely available bignum implementation that is distributed with the Java Development Kit (JDK). It was not designed with this verification effort in mind and its internal representation of bignum magnitudes adds additional complexity that complicates reasoning about its arithmetic operations. The specification of `BigInteger` and the complications are explained in Section 3.1.

In this chapter, an ACL2 model of `BigInteger`'s magnitude addition is verified to correctly add natural numbers. Section 3.1 defines a specification for the algorithm based on a code analysis of `BigInteger`. The verification is performed in ACL2 by following The Method, a process that begins with sketching out a proof first. The proof sketch for verifying `BigInteger` operations, which is detailed in Section 3.2, adds a single element of complexity at each layer in the proof tree. The implementation and verification of each layer is presented in Sections 3.3-3.5. This chapter serves an additional purpose as an introduction to using the ACL2 theorem prover, so some technical details regarding the configuration of ACL2 are described.

3.1 BigInteger Specification

`BigInteger` represents the magnitude of a bignum as an integer array. The magnitude is interpreted in big-endian order so the integer in the zeroth index of the array is the most significant integer. Addition is the operation that will be verified because it is the simplest operation that illustrates the complications of `BigInteger`'s representation. The code for `BigInteger`'s `add` method is shown in Figure 3.1.

The method splits the addition into five pieces: (1) swap, (2) add, (3) propagate the carry, (4) copy, and (5) expand. The pieces are highlighted in Figure 3.1 by comments in the code. The `BigInteger` specification introduces two challenges

```
1  private static int[] add(int[] x, int[] y) {
2    // Swap the longest array into the x variable
3    if (x.length < y.length) {
4      int[] tmp = x;
5      x = y;
6      y = tmp;
7    }
8
9    int xIndex = x.length;
10   int yIndex = y.length;
11   int result[] = new int[xIndex];
12   long sum = 0;
13
14   // Add common parts of both numbers
15   while(yIndex > 0) {
16     sum = (x[--xIndex] & LONG_MASK) +
17           (y[--yIndex] & LONG_MASK) + (sum >>> 32);
18     result[xIndex] = (int)sum;
19   }
20
21   // Propagate carry
22   boolean carry = (sum >>> 32 != 0);
23   while (xIndex > 0 && carry)
24     carry = ((result[--xIndex] = x[xIndex] + 1) == 0);
25
26   // Copy remainder of longer number
27   while (xIndex > 0)
28     result[--xIndex] = x[xIndex];
29
30   // Expand result if necessary
31   if (carry) {
32     int bigger[] = new int[result.length + 1];
33     System.arraycopy(result, 0, bigger, 1, result.length);
34     bigger[0] = 0x01;
35     return bigger;
36   }
37   return result;
38 }
```

Figure 3.1: `BigInteger` Add Magnitudes

that have not been addressed in previous research. First, since Java does not have unsigned primitive types, each word in the magnitude is a *signed* `int`. The words are interpreted as unsigned 32-bit integers by applying a mask that casts the value to the larger `long` type but preserves the bit values of the original `int`. Second, the magnitude is stored in big-endian order rather than little-endian order. Little-endian order is easier to analyze because the arrays are inherently aligned at index 0 and the magnitude of a word at index i can be calculated from the magnitude of the word at $i-1$. In comparison, in big-endian order, the arrays must be explicitly aligned at the largest index in each array and the magnitude of a word at index i changes depending on the length of the array. The proof sketch addresses each challenge separately to verify bignum addition.

3.2 Proof Sketch

Our focus is on the addition of magnitudes as implemented in `BigInteger`. We will demonstrate that coercing the result of a `BigInteger` magnitude operation to an ACL2 natural is equivalent to executing the corresponding operation on naturals. Our approach, which addresses the complications of the signed, big-endian representation, is to begin by verifying our desired property on an unsigned, little-endian representation of the operation. We will then define a signed, little-endian implementation and a signed-to-unsigned translation and verify the desired property still holds. We finish by implementing the `BigInteger` specification and equating it to the reverse of the signed, little-endian results.

For this proof, imagine that a function $\eta_r(m)$ converts a magnitude array m into the natural number it represents if interpreted as being stored in representation r. Assume x_r and y_r are arrays formatted according to r that represent natural numbers x and y respectively. Then the goal of the proof at each layer is to verify that

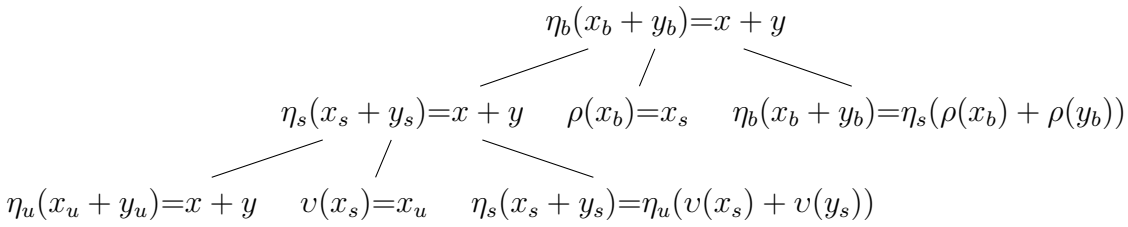

Figure 3.2: Bignum Addition Proof Sketch

$\eta_r(x_r + y_r) = x + y$. The proof sketch considers three representations which are identified with the subscripts: u for unsigned, little-endian, s for signed, little-endian, and b for signed, big-endian. The proof sketch is a three-layered approach shown in Figure 3.2.

Each node in the proof sketch represents a theorem that must be admitted to ACL2. The function $\upsilon(m)$ converts a signed array m into an unsigned array and the function $\rho(m)$ reverses the array m. The top node in the proof is the desired theorem that verifies the bignum addition operation implemented by `BigInteger`. The proof begins in the bottom left node that states $\eta_u(x_u + y_u) = x + y$, which is a theorem similar to those in the literature. Our implementation is most similar to the version presented by Moore (1989). The remaining nodes on the bottom layer translate that proof into one that verifies the signed, little-endian representation. The nodes in the middle layer convert the signed, little-endian representation into the goal theorem by reversing the elements. The lemmas to verify each node are explained in the following sections.

3.3 Unsigned, Little-Endian Add

The proof sketch requires each representation to define (1) a function $\eta_r(m)$ that converts a magnitude array to a natural and (2) a function that adds two magnitude arrays formatted in the representation. Since the informal description is similar for all iterations of adding magnitudes, the implementations specify the representation

in the function names: `ule-` functions are unsigned, little-endian, `sle-` functions are signed, little-endian, and `sbe-` functions are signed, big-endian. The prefixes are also used in the text to reference the representations. The function `ule-to-nat` converts a ule magnitude into a natural number.

```
(defun ule-to-nat (xs)
  (if (endp xs)
    0
    (+ (car xs)
       (* (expt 2 32)
          (ule-to-nat (cdr xs))))))
```

This function recursively walks the list from the least to most significant integer and multiples the recursive call by 2^{32} to shift the value.

The difference between the first two representations in the proof sketch is the signedness of the words. For representations using signed words, each word is still interpreted as unsigned during calculations. Therefore, the difference between the first two representations is limited to the calculations performed on individual words. Specifically, words are added together to calculate a new word and to determine if there is a carry. The ACL2 code for both calculations is shown in Figure 3.3. It introduces three additional functions: `uint-p`, `loghead`, and `logtail`. The predicate `uint-p` recognizes unsigned 32-bit integers. `loghead` returns the lower n bits of its input where, in this case, n is 32. It is used in the add function to truncate the result to 32-bits. The carry is calculated by `logtail`, which returns all of the higher bits after index n.

The add operation is implemented in ACL2 with the functions `ule-add` and `ule-propagate`. Figure 3.4 shows the code for both functions. The `ule-add` function adds the common elements until reaching the end of the shorter list before calling `ule-propagate` on the longer list. If a carry exists, `ule-propagate` propagates it over the longer list and expands the resulting list if necessary. Otherwise, `ule-propagate`

```
(defun add-uint (x y carry)
  (let* ((i (if (uint-p x) x 0))
         (j (if (uint-p y) y 0)))
    (loghead 32 (+ i j carry))))

(defun add-uint-carry (x y carry)
  (let* ((i (if (uint-p x) x 0))
         (j (if (uint-p y) y 0)))
    (logtail 32 (+ i j carry))))
```

Figure 3.3: Unsigned Add and Carry Functions

copies the remaining elements in the longer list into the resulting list. Recall that the `add` method in `BigInteger` splits the operation into five steps: (1) swap, (2) add, (3) propagate the carry, (4) copy, and (5) expand. `ule-add` addresses step 2 and `ule-propogate` addresses steps 3, 4, and 5. Step 1, which is a swap that guarantees the longest list is the `x` variable, simplifies the Java code for steps 3 and 4 by removing the need to reference the `y` array. The ACL2 code gets the same effect by splitting steps 3 and 4 into a separate function.

With definitions for the components, we turn our attention to formalizing, in ACL2, the node in the proof sketch that states $\eta_u(x_u + y_u) = x + y$. Before ACL2 can prove that statement, it must be "taught" a few facts. For starters, the carry of an addition is at most 1. The predicate `carry-p` recognizes acceptable values for the carry. ACL2 can prove a theorem that states that the function `add-uint-carry` always returns an acceptable carry value.

```
(defthm add-uint-carry-is-0-or-1
  (implies (and (uint-p x)
                (uint-p y)
                (carry-p carry))
           (carry-p (add-uint-carry x y carry))))
```

ACL2 uses this theorem during induction to "know" that the calculated `carry` is acceptable for each recursive call.

```
(defun ule-propagate (xs carry)
  (if (endp xs)
    (if (< 0 carry)
      (cons carry nil)
      nil)
    (cons (add-uint (car xs) nil carry)
          (ule-propagate (cdr xs)
                         (add-uint-carry (car xs) nil
                           carry)))))

(defun ule-add (xs ys carry)
  (if (or (endp xs) (endp ys))
    (ule-propagate (if (endp xs) ys xs) carry)
    (cons (add-uint (car xs) (car ys) carry)
          (ule-add (cdr xs)
                   (cdr ys)
                   (add-uint-carry (car xs)
                                   (car ys)
                                   carry)))))
```

Figure 3.4: Unsigned, Little-Endian Add

We anticipate ACL2 will generate subgoals about the behavior of `ule-propagate`. Since the carry is either 0 or 1, we can address `ule-propagate` with two cases. For the case when the carry is 0, `ule-propagate` is an identity function.

```
(defthm ule-propagate-identity
  (implies (uint-listp xs)
           (equal (ule-propagate xs 0) xs)))
```

For the case when the carry is 1, the following theorem is admitted to ACL2:

```
(defthm propagate-carry-works
  (implies (and (uint-listp xs)
                (= carry 1))
           (equal (ule-to-nat (ule-propagate xs carry))
                  (+ (ule-to-nat xs) carry))))
```

Using the previous theorems as lemmas, ACL2 can now accept a theorem that states $\eta_u(x_u + y_u) = x + y$. In ACL2, it is implemented as,

```
(defthm add-lists-works
  (implies (and (uint-listp xs)
                (uint-listp ys)
                (carry-p carry))
           (equal (ule-to-nat (ule-add xs ys carry))
                  (+ (ule-to-nat xs)
                     (ule-to-nat ys)
                     carry))))
```

The theorem `add-lists-works` is a proof of the bottom-left node of the proof sketch. The next section uses this theorem to prove the middle node of the sketch.

3.4 Signed, Little-Endian Add

The signed, little-endian (sle) representation changes the storage of the words to signed integers. According to the proof sketch, the verification of the ule representation is translated into a proof for sle by verifying that $\upsilon(x_s) = x_u$ and $\eta_s(x_s + y_s) = \eta_y(\upsilon(x_s) + \upsilon(y_s))$, where $\upsilon(m)$ converts a sle magnitude into a ule magnitude. Both of those properties are verified as lemmas in this section and used to verify the theorem that an sle addition function correctly adds magnitudes in the sle format.

The only difference between the sle and ule representations is the signedness of the words. Signed 32-bit integers, which correspond to Java `ints`, are recognized with `jint-p`. Signedness is only relevant to the storage of the values; the calculations performed on words still interpret the values as unsigned. In the Java code, the interpretation is performed by applying a longmask that casts an `int` to a `long` but preserves the bit-order. In ACL2, the function `longmask` applies the conversion in the same manner. The following theorem, `jint-longmask-is-uint`, verifies that applying a longmask to a signed int results in an unsigned int.

```
(defthm jint-longmask-is-uint
  (implies (jint-p x)
           (uint-p (longmask x))))
```

The theorem is necessary to verify both of our lemmas.

The proof sketch defines a function $\upsilon(m)$ that converts m from an sle magnitude to a ule magnitude. The function `sle-to-ule` is an ACL2 implementation of $\upsilon(m)$.

```
(defun sle-to-ule (xs)
  (if (endp xs)
    nil
    (cons (longmask (car xs))
          (sle-to-ule (cdr xs)))))
```

The sketch requires that the conversion be verified to work correctly by the node that states the property $\upsilon(x_s) = x_u$. In the sketch, x_s and x_u are related to the natural number x by definition. For verification of the lemma, the magnitudes are formally converted into natural numbers using the existing `to-nat` functions.

```
(defthm sle-to-nat-is-ule-to-nat
  (implies (jint-listp xs)
           (equal (sle-to-nat xs)
                  (ule-to-nat (sle-to-ule xs)))))
```

The theorem `sle-to-nat-is-ule-to-nat` verifies the lemma $\upsilon(x_s) = x_u$ from the proof sketch. Similary, a `ule-to-sle` function is also defined and verified.

The sketch also requires a proof of the statement: $\eta_s(x_s + y_s) = \eta_u(\upsilon(x_s) + \upsilon(y_s))$. Informally, it states that the result of adding two sle magnitudes is equivalent to adding ule conversions of the magnitudes. The property is verified by demonstrating the relationship between the differences of the two add functions. In the previous section, `ule-add` was implemented with the calculations for adding words and determining the carry separated into the functions `add-uint` and `add-uint-carry`. The `sle-add` function is identical to `ule-add` except it depends on signed versions of those functions, which are shown in Figure 3.5. The function `logext`, seen in the definition of `add-jint`, converts an unsigned integer into a signed integer of size n bits. Along with the code, Figure 3.5 shows the theorems that confirm that the signed calculations are equivalent to unsigned calculations on converted inputs. With that

knowledge, ACL2 accepts that `sle-add` is equivalent to `ule-add` on converted lists.

```
(defthm sle-add-is-ule-add
  (implies
    (and (jint-listp xs)
         (jint-listp ys)
         (carry-p carry))
    (equal
      (sle-add xs ys carry)
      (ule-to-sle
        (ule-add (sle-to-ule xs)
                 (sle-to-ule ys)
                 carry)))))
```

The bottom layer of the proof sketch defines three nodes that must be verified to convert a proof of ule correctness into a proof of sle correctness. The first node was verified in the previous section. The remaining two nodes were verified in this section. When ACL2 is configured with all three lemmas, it accepts the theorem `sle-add-lists-works` that corresponds to the node in the the middle of the sketch that states: $\eta_s(x_s + y_s) = x + y$.

```
(defthm sle-add-lists-works
  (implies
    (and (jint-listp xs)
         (jint-listp ys)
         (carry-p carry))
    (equal (sle-to-nat (sle-add xs ys carry))
           (+ (sle-to-nat xs) (sle-to-nat ys) carry))))
```

In the next section, the sle representation is translated into a signed, big-endian (sbe) representation to verify the correctness of the model of the `BigInteger add` method.

3.5 Signed, Big-Endian Add

The little-endian representations use definitions that can be reasoned about using list induction. In contrast, the big-endian representation requires functions to know the length of the original list and interpret values starting at the last index. For the

```
(defun add-jint (x y carry)
  (let* ((i (if (jint-p x) x 0))
         (j (if (jint-p y) y 0)))
    (logext 32
            (loghead 32
                     (+ (longmask i)
                        (longmask j)
                        carry)))))

(defthm add-jint-is-add-uint
  (implies
    (and (jint-p x)
         (jint-p y)
         (carry-p carry))
    (equal
      (add-jint x y carry)
      (logext 32
              (add-uint (longmask x)
                        (longmask y)
                        carry)))))


(defun add-jint-carry (x y carry)
  (let* ((i (if (jint-p x) x 0))
         (j (if (jint-p y) y 0)))
    (logtail 32 (+ (longmask i) (longmask j) carry))))

(defthm add-jint-carry-is-add-uint-carry
  (implies
    (and (jint-p x)
         (jint-p y)
         (carry-p carry))
    (equal (add-jint-carry x y carry)
           (add-uint-carry (longmask x)
                           (longmask y)
                           carry))))
```

Figure 3.5: Signed Add and Carry Functions

previous results to be related to a big-endian implementation, the list-based induction schemes must be converted to index-based induction schemes. For demonstration of this process, we will focus on equating the little-endian and big-endian natural number conversions.

The function `sle-to-nat-i` is an indexed-based implementation to convert signed, little-endian lists into natural numbers. The index of the big-endian implementation will decrease from the length of the list to 0, so the function is defined with a parameter i that also decreases so the inputs to the little-endian and big-endian functions are aligned. The index in the little-endian implementation is calculated by subtracting i from the length of the list.

```
(defun sle-to-nat-i (xs i)
  (if (zp i)
    0
    (+ (longmask (nth (- (len xs) i) xs))
       (* (expt 2 32)
          (sle-to-nat-i xs (1- i))))))
```

The result of calling `sle-to-nat-i` with an i value of `(len xs)` should be equal to calling `sle-to-nat` on the same list. To prove that in ACL2 requires coordinating the induction schemes of the two functions. ACL2 chooses induction schemes by examining the induction schemes used to admit the functions found in a theorem. In this case, the induction scheme for the original definition is based on walking the list and the induction scheme for the indexed version is based on decrementing the index. The theorem `sle-idx-to-sle` supplies a hint that instructs ACL2 to apply a combination of both induction schemes.

```
(defthm sle-idx-to-sle
  (implies (jint-listp xs)
           (equal (sle-to-nat-i xs (len xs))
                  (sle-to-nat xs)))
  :hints
  (("Goal" :induct (cdr-dec-induct xs (1- (len xs))))))
```

With an index-based implementation of the sle functions, the verification can proceed towards the remaining lemmas required by the proof sketch.

Similar to the bottom layer of the sketch, the middle layer converts the sle proof into a sbe proof by introducing a conversion function. In the sketch, the function is named $\rho(m)$, but sle is converted to sbe by reversing the elements so we use ACL2's built-in `rev` function. We include `rev`'s definition here, because we need to prove an important lemma about it to continue.

```
(defun rev (x)
  (if (consp x)
      (append (rev (cdr x)) (list (car x)))
      nil))
```

When a list `xs` is reversed, the elements are in the opposite order, but that description does not say anything about the position of specific elements. For our purposes, we need to know *where* an element moved in the reversed list. `rev` reverses a list recursively by appending the first element to the end. With that in mind, the theorem `distance-moved` verifies that the element appended to the end of a list `xs` is at an index equal to the length of `xs`.

```
(defthm distance-moved
  (implies
    (and (consp xs)
         (atom y))
    (equal (nth (len xs) (append xs (list y)))
           y)))
```

The length of a list and the length of its reverse are the same. Therefore, when `distance-moved` is combined with the definition of `rev`, it states that an element moves the length of the remaining elements in the list. ACL2 mechanically combines the theorem and definition during proof attempts to verify both lemmas required by the proof sketch. Figure 3.6 shows both ACL2 theorems. The theorems `sbe-to-nat-is-sle-nat` and `sbe-add-is-sle-add` correspond to the nodes

```
(defthm sbe-to-nat-is-sle-to-nat
  (implies (and (jint-listp xs)
                (<= i (len xs)))
           (equal (sbe-to-nat-i xs i)
                  (sle-to-nat-i (rev xs) i))))

(defthm sbe-add-is-sle-add
  (implies
    (and (jint-listp xs)
         (jint-listp ys)
         (carry-p carry)
         (<= xi (len xs))
         (<= yi (len ys)))
    (equal
      (sbe-add-i xs ys carry xi yi)
      (rev (sle-add-i (rev xs) (rev ys) carry xi yi)))))
```

Figure 3.6: Relationship between Big- and Little-Endian Representations

$\rho(x_b) = x_s$ and $\eta_b(x_b + y_b) = \eta_s(\rho(x_b) + \rho(y_b))$, respectively.

Once the indexed versions of `sbe-add` and `sle-add` are equated, the functions `sbe-add-w` and `sle-add-w` are written to wrap the definitions and explicitly initialize the indexes to the lengths of the lists. All of the theorems in this chapter guide ACL2 to accept the theorem `sbe-add-w-works`.

```
(defthm sbe-add-w-works
  (implies
    (and (jint-listp xs)
         (jint-listp ys)
         (carry-p carry))
    (equal (sbe-to-nat-w (sbe-add-w xs ys carry))
           (+ (sbe-to-nat-w xs)
              (sbe-to-nat-w ys)
              carry))))
```

The theorem `sbe-add-w-works` is an ACL2 artifact that verifies that a model of `BigInteger`'s `add` specification correctly adds bignums. The theorem corresponds to the top node in the proof sketch.

3.6 Summary

The code in this chapter implements an ACL2 model of the algorithm used to perform addition in Java's `BigInteger` class, which is the basis for Clojure's bignums. The theorems presented verify that the addition algorithm correctly implements bignum addition. The layered approach to the verification began with a proof of a simpler representation of bignums and added additional complexity one component at a time. The same approach can be applied to the remaining arithmetic operations that are implemented in `BigInteger` to complete a bignum arithmetic library based on its specific representation.

By modeling Clojure's versions of bignums in ACL2, existing proof libraries can be more easily applied to industrial software without sacrificing the accuracy of the model. However, Clojure also supports the use of Java's primitive number types. The primitive types are more efficient to store in memory so the use of bignums, even in Clojure, may not be common. Despite possibly limited use, the verification of bignums is still significant because many cryptography libraries operate on arbitrarily large numbers and security software is an important domain for formal verification (Denis & Rose, 2006).

The verified code in this chapter is a model built from an inspection of Java code. Beginning in the next chapter, the remainder of the dissertation considers a deeper analysis of Java code. Instead of building a model from inspection, a model of the JVM is constructed in ACL2. The JVM model can then be used to simulate the execution of Java bytecode.

Chapter 4

The Java Virtual Machine Model

Software is an abstraction that directs the execution of a hardware system. Applications are a single layer of software that sits atop many others. Each piece of software, and the underlying hardware, all affect the execution of the application. Therefore, a formal analysis of a system must decide what layers to model and how precisely the model reflects the system.

Early research focused on building models of hardware to reason about the compiled machine code of software. Boyer & Yu (1996) formalizes a significant portion of the Motorola MC68020 processor and gives a rationale for reasoning about programs at the machine code level, which includes the points that (1) the semantics of the machine are what determine the run-time behavior of a program and (2) hardware is generally defined with more formal semantics than programming languages. The Java Virtual Machine (JVM) is a higher level abstraction than hardware but it does have a well-defined semantics (Lindholm et al., 2013) that describe the run-time execution of software. The JVM has two additional benefits to verification compared to hardware. First, the JVM specifies a format for valid programs that is more structured than required by hardware. Second, the use of memory — including format of objects, storage, and access — are defined by the JVM and enforced by the semantics of the instructions. For these reasons, the JVM and code that runs on it have been extensively studied.

The JVM has been studied in ACL2 through a series of six JVM models, named M1 through M6. Of the six models, three have been featured in the literature. The

earliest version, M1, only implements 17 instructions but it was used to demonstrate that the JVM code of a Java method could be mapped to an ACL2 function (Moore, 1999). M5, which is the next model featured in publication, is a significantly more sophisticated model that implements 138 instructions. M5 has been used to analyze a non-terminating, multithreaded Java program (Moore & Porter, 2002). The final model in the series, M6, is sufficiently sophisticated to reason about properties of the JVM including class initialization (Liu & Moore, 2003) and the bytecode verifier (Liu, 2006). We introduce a new ACL2 model of the JVM that is targeted towards the verification of bytecode programs generated from Clojure source code.

The model is named *MC* for *model of Clojure*. MC is a modification of M5 that adds additional features to support the verification of Clojure programs. Clojure uses interfaces, which are not supported in M5, to check types and invoke methods. MC adds two new instructions and extends the implementation of a third to add support for interfaces. M5 is modified instead of the newer, more sophisticated M6 because M5 is the most sophisticated model that is publicly available through the ACL2 community books (Moore & Porter, 2001). M6 has still influenced the development of MC. For example, support for invoking native methods is added to MC based on the implementation in M6. Also, this research project verifies programs that require reasoning about more interacting classes than the examples that have been published for the M1 through M5 models. Since M6 was used to verify properties of the JVM rather than JVM programs, Liu & Moore (2004) formalized abstract properties of the JVM as rewrite rules in ACL2 to reduce the complexity of later proofs. This dissertation formalizes several similar properties, beginning in this chapter with the introduction of our big-step semantics that is used to abstract the execution of method calls.

The ACL2 JVM models implement the semantics as transition functions that each perform a single JVM instruction. The models can be used to reason about programs

naturally using small-step semantics by simulating execution of the model. On the other hand, big-step semantics, which define transitions that encompass many JVM instructions, are easier to use in the verification of large systems because each step covers more logic. The Clojure core library is a large system. Even though this dissertation only verifies the portions of Clojure related to sequences, the verification still requires the analysis of several interconnected classes. In this dissertation, methods are verified initially using small-step semantics, but the resulting theorems are applied as big-steps. In addition to defining MC, this chapter explains how methods are invoked in MC and how methods are verified to apply as big-steps.

4.1 Structures

The state is a call stack, a heap, and a class table that binds class names to class declarations. The heap is represented as a list of integer addresses mapped to instance objects, where instance objects specify the class name of the instance and its fields' values as well as any parent classes and fields. References are stored as `(REF i)` and dereferenced by selecting the heap object at the i-th address. The call stack is a list of frames where the currently executing frame is at the top. A frame has a program counter, the list of local objects, the operand stack, a set of instructions, and the name of the current class.

Class declarations define the name, list of superclasses, list of implemented interfaces, member and static fields, constant pool, and list of methods. Identically to M5, MC defines a function to access each element in a declaration. MC also adds an abstraction layer to the access functions for the list of superclasses and interfaces in the form of a function that retrieves a class declaration by name from a class-table.

```
1 (defun class-decl-superclasses (dcl)
2   (nth 1 dcl))
```

```
(defun class-superclasses (class-name class-table)
  (class-decl-superclasses
    (bound? class-name class-table)))
```

This abstraction layer is combined with class loading predicates to restrict information ACL2 actively considers at each subgoal.

Each method is a name, list of parameters, flag that identifies whether the method is native or not, and list of instructions that comprise the method program. The constant pool of a Java .class file contains a lot of information including data to look up methods and class declarations. MC stores the method and class information directly in the instructions. The simplified constant pool of an MC class declaration only stores references to constant values.

MC implements 138 JVM 7 instructions, but does not implement instructions that throw exceptions or operate on floats or doubles. A Java primitive integer is represented as an ACL2 integer, and a Java primitive long is represented as a 0 followed by an ACL2 integer. As an example, the ALOAD instruction copies a value from the locals onto the stack. The instruction is represented as a list `(ALOAD i)` in a program. The `execute-ALOAD` function will modify state s by updating the program counter `pc` and pushing onto the stack the value in the i-th location.

```
(defun execute-ALOAD (inst s)
  (modify
   s
   :pc (+ (inst-length inst) (pc (top-frame s)))
   :stack (push (nth (arg1 inst)
                     (locals (top-frame s)))
                (stack (top-frame s)))))
```

MC adds support for interfaces which requires extending `execute-INSTANCEOF` to check the interface list of the class of an instance object. The function `class-types` appends the instance's class name, its list of parent classes, and its interfaces into a single list. The `instance-of` logic is separated from `execute-INSTANCEOF` into its

own function that searches the list of class types for the desired type.

```
(defun class-types (class-name class-table)
  (let*
    ((obj-supers
      (cons class-name
            (class-superclasses class-name
                                class-table))))
    (append obj-supers
            (class-interfaces class-name
                              class-table))))

(defun instance-of (type ref heap class-table)
  (let*
    ((obj (deref ref heap))
     (obj-class (caar obj))
     (obj-types (class-types obj-class
                             class-table)))
    (if (nullrefp ref)
      0
      (if (member-equal type obj-types)
        1
        0))))
```

The `execute-INSTANCEOF` function pushes the result onto the stack and modifies the state.

```
(defun execute-INSTANCEOF (inst s)
  (let* ((ref (top (stack (top-frame s)))))
    (modify
     s
     :pc (+ (inst-length inst) (pc (top-frame s)))
     :stack (push (instance-of (arg1 inst)
                               ref
                               (heap s)
                               (class-table s))
                  (pop (stack (top-frame s)))))))
```

4.2 Method Invocation

Each method executes within its own frame. When the JVM encounters an invoke instruction, a new frame is created containing the method's program, an empty operand stack, and a program counter set to index 0. For each parameter of the method, the calling frame pops an object off its operand stack and pushes it onto the new frame's locals. The parameters in the calling frame are in *reverse order* to the method signature so that the order of the locals match the order of the signature. When an instance method is invoked, a reference to the instance is pushed onto the top of the locals.

The JVM Specification defines four invocation instructions:

- *invokestatic* calls a static method,
- *invokeinterface* calls an interface method,
- *invokevirtual* calls an instance method
- *invokespecial* calls a constructor, superclass method, or private method

The bytecode format in the JVM is **INVOKESTATIC index-byte1 index-byte2**, except for INVOKEINTERFACE. The index-bytes are combined to form a integer index into the class's constant pool that should resolve to a method reference. The method reference is resolved to find the number of parameters and corresponding program code. The INVOKEINTERFACE instruction is similar, but has two additional legacy bytes to each instruction that do not change its behavior but do change its instruction length. The invoke instructions are modeled in MC as `(invoke* "method-name" "object-name" param-count)`. For INVOKEINTERFACE, the object name is the interface; otherwise, it is a class name. The parameter count indicates how many values to push onto the locals, so each long parameter must increment the count by two to account for the 2-value representation.

The general lookup procedure is to check the class definition of the invoked class for a method that matches the instruction parameters. If a method is not found, the class's immediate parent is checked and the process is repeated until a matching method is found or no more parent classes exist.

```
(defun lookup-method-in-superclasses (name
                                      classes
                                      class-table)
  (cond
    ((endp classes) nil)
     (t
       (let*
         ((class-name (car classes))
          (class-decl (bound? class-name
                              class-table))
          (method
            (bound? name
                    (class-decl-methods class-decl))))
       (if method
           method
           (lookup-method-in-superclasses
             name
             (cdr classes)
             class-table))))))

(defun lookup-method (name class-name class-table)
  (lookup-method-in-superclasses
    name
    (cons class-name
          (class-decl-superclasses
            (bound? class-name class-table)))
    class-table))
```

The function `lookup-method` searches through the `class-table` for a method `name` in a base class named `class-name`. The function constructs a list of class names containing `class-name` and the class's parents. The `lookup-method-in-superclasses` function finds the applicable method by recursively checking the declarations of each class in the list of class names. The base class-name used to look up a method differs slightly depending on the invocation being executed. The instructions INVOKEIN-

TERFACE and INVOKEVIRTUAL invoke methods based on the class of the instance of the method that is invoked, whereas INVOKESPECIAL and INVOKESTATIC specify the base class to begin the lookup procedure.

Java supports native methods, which are interfaces to code that run outside of the JVM. M5 represents native methods as a method with a nil program but does not implement native invocation. M6, the immediate successor to M5, implements native invocation as a lookup table for a set of native method names. Each method name in the table is mapped to a ACL2 function that modifies the state consistently with the behavior of the native method. MC implements the function `execute-native` based on the M6 implementation.

The function `execute-INVOKEINTERFACE` is described in detail as an example.

```
(defun execute-INVOKEINTERFACE (inst s)
  (let*
    ((method-name (arg2 inst))
     (nformals (arg3 inst))
     (obj-ref
      (top (popn nformals (stack (top-frame s)))))
     (obj-class-name
      (class-name-of-ref obj-ref (heap s)))
     (closest-method
      (lookup-method method-name
                     obj-class-name
                     (class-table s)))
     (prog (method-program closest-method))
     (s1 (modify s
                 :pc
                 (+ (inst-length inst)
                    (pc (top-frame s)))
                 :stack
                 (popn (+ nformals 1)
                       (stack (top-frame s))))))
    (modify
     s1
     :call-stack
     (push
      (make-frame
       0
```

```
27          (reverse
28           (bind-formals (+ nformals 1)
29                         (stack (top-frame s))))
30          nil
31          prog
32          (arg1 inst))
33         (call-stack s1)))))
```

Lines 5 through 13 retrieve the class name of the instance the instruction is operating on and initiates the method lookup procedure on that class name. Lines 14—20 increment the pc of the calling frame and remove the formals from the stack prior to pushing the new frame on top. Lines 21—33 construct a new frame and push it onto the call stack of the existing state. The other invocation instructions are similar.

A method terminates once it reaches a return instruction. MC supports four of the JVM return instructions. RETURN instructions terminate methods that do not return a value. They pop the frame off of the call stack. The instructions IRETURN, LRETURN, and ARETURN return values of type integer, long, and reference. Each one retrieves the top element off the returning method's stack, pops the frame off of the call stack, and pushes the element onto the stack of the next frame. The `execute-ARETURN` function shows one of the return implementations.

```
1 (defun execute-ARETURN (inst s)
2   (declare (ignore inst))
3   (let* ((val (top (stack (top-frame s))))
4          (s1 (modify s
5                      :call-stack (pop (call-stack s)))))
6     (modify s1
7             :stack (push val (stack (top-frame s1))))))
```

4.3 Stepping the Machine

When verifying properties of a method in Java, we simulate the machine as it steps from the invocation of the method to the eventual return. In past work, clock functions have been used that calculate the number of steps necessary to terminate. Clock

functions require knowledge of the state to calculate the number of steps an execution will take, which complicates the use of any theorems as lemmas. Consider a method m_1 that calls a method m_2 and both methods have a corresponding m_n-*clock* function. The m_1-*clock* must be aware of the m_2-*clock* as well as any changes to the state prior to the m_2 call. The logic of the clocks will increase in complexity as fast as the programs, which will either make reasoning about the programs impossible or require effort introducing lemmas to ACL2 to reason about the clocks. What we need is a notion of a big-step that can be used to configure effective rewrite rules in ACL2.

ACL2 is a logic of total functions but it can be used to build theories about partial functions. To do so, one defines a witness where the witness is a total function that has the desired properties. The desired properties are configured as theorems in ACL2. The witness and the theorems are defined in ACL2 within an encapsulation event but only the theorems are exported. Manolios & Moore (2003) demonstrate that tail recursive partial functions are always able to be witnessed and create the `defpun` macro in ACL2 to quickly define partial functions. Like Manolios and Moore, we are only interested in programs that terminate so we borrow their halting predicate and `big-step` function.

```
(defun haltedp (s)
  (equal (step s) s))

(defpun big-step (s)
  (if (haltedp s)
    s
    (big-step (step s))))
```

The machine does not need to terminate even though the program must. If a machine runs to state s_n then executes method program p that terminates at s_{n+1}, the properties of method program p hold for the transition from s_n to s_{n+1} regardless of whether or not the machine continues running after s_{n+1}. The benefit of this intuition is that we do not need to prove termination to reason about it. Instead, we verify properties

hold on a machine by showing that big-stepping s_1 is equivalent to big-stepping s_2. Manolios & Moore (2003) implemented this concept as a function `==`, but we found using the macro resulted in more successful rewrite rules. The equivalence relation between s_1 and s_2 is symmetric, but it is intended to be used in theorems to generate ACL2 rewrite rules, which are triggered by the terms in the left-hand side of the conclusion. Therefore, we name the macro `->` to imply a convention that s_1 is an earlier state in the execution than s_2. The `->` macro is shown here,

```
(defmacro -> (s1 s2)
  `(equal (big-step ,s1)
          (big-step ,s2)))
```

If a machine is proven to terminate, the big-step can be eliminated. The theorem `big-step-halts-all` states that if s_2 halts, so does s_1.

```
(defthmd big-step-halts-all
  (implies (and (haltedp (big-step s2))
                (-> s1 s2))
           (haltedp (big-step s1))))
```

The theorem `halting-is-s` eliminates the `big-step` term from a halted state.

```
(defthmd halting-is-s
  (implies (haltedp s)
           (equal (big-step s) s)))
```

ACL2 does not rewrite `(big-step s)` to `(big-step (step s))` naturally, which we need it to do. When enabled, the following theorem `big-step-opener-infinite` will rewrite instances of `(big-step s)` until it reaches the maximum call depth for ACL2's theorem prover.

```
(defthmd big-step-opener-infinite
  (equal (big-step s) (big-step (step s))))
```

It is introduced to ACL2 using `defthmd`, which immediately disables the rewrite rule associated with the theorem. `big-step-opener-infinite` is an expensive theorem

because it is triggered by the term `big-step` and equates to a term that contains `big-step`. Therefore, the rule needs to be weakened with a hypothesis that eventually falsifies so that the proof does not apply `big-step` indefinitely. Since the machine must at least consider every instruction is the code being verified, ideally, we want ACL2 to step the program as long as it knows what the next step is. To do just that, we define a predicate `inst-known-p` that recognizes all instructions that have been implemented in MC.

```
(defun inst-known-p (s)
  (let* ((inst (next-inst s))
         (op (op-code inst)))
    (or (equal op 'AALOAD)
        ... All other instructions ...
        (equal op 'SWAP))))
```

The theorem `big-step-opener` weakens the theorem `big-step-opener-infinite` to only apply when the next instruction is known.

```
(defthm big-step-opener
  (implies (inst-known-p s)
           (equal (big-step s) (big-step (step s))))
  :hints (("Goal" :use big-step-opener-infinite)))
```

MC admits to ACL2 theorems that describe the behavior of a single instruction being stepped, which is consistent with the example from Manolios & Moore (2003). For example, here is a theorem that big-steps a program poised to execute ALOAD_0.

```
(defthm ->-execute-ALOAD
  (implies
   (equal (next-inst s) '(ALOAD_0))
   (-> s
       (modify s
               :pc (+ 1 (pc (top-frame s)))
               :stack
               (push (nth 0
                          (locals (top-frame s)))
                     (stack (top-frame s)))))))
```

Most of these theorems are simple to construct but save ACL2 the work of applying rules for `do-inst`, `execute-*`, and `inst-length` at each step.

4.4 Summary

MC defines the formal semantics of our deep embedding of the JVM into ACL2. A deep embedding also requires that programs written in the embedded language be represented as objects in the logic. A basic structure of programs is described in this chapter. The basic structure has been sufficient for all of the previously published examples that use any of the models in the ACL2 series. However, the systems analyzed in this dissertation are larger in terms of the number of interacting classes and the length of the call stacks. Liu (2006) concluded from using M6 that a more robust class representation would be beneficial. In the next chapter, an abstraction layer is introduced to simplify some of the complexity required to reason about classes in MC.

Chapter 5

Java Class Models

In Java, the class defines both the code and the structure of instances. MC supports a basic class declaration that mimics the format of the files that a traditional JVM reads. The declarations are used to load code into a new frame and to create instances that are stored in the heap. The basic declaration structure, which is comparable to the structures used in M5 and M6, is sufficient for verifying small projects (Moore & Porter, 2002; Moore, 2003, 2006). However, when relying on the basic format, the ability to reason about a system in MC scales poorly with the number of classes in the system, which is consistent with results for a similar model (Liu, 2006). To reason about any Clojure program requires the consideration of several classes because every Clojure function generates a new class, including the functions defined in the Clojure core library. For MC to be applicable to reasoning about Clojure, it must be capable of scaling efficiently. Based on the verification of a bytecode verifier in M6, Liu (2006) suggests large verification projects should (1) organize the proofs into groups that limit the information that is exposed outside of the group and (2) "identify an effective strategy" to define and set up new data structures. We identify such a strategy in the form of an abstraction layer on top of the basic class declaration that hides information from the ACL2 logic except at key points during proof attempts. The abstraction layer is described in this chapter as a set of functions and theorems that will be defined for every class to simplify the common access patterns of the internal structure of a declaration.

5.1 Structure of the Class Declaration

A class declaration is responsible for defining its dependencies, the structure of objects of that class, and the access to its methods. For each of these responsibilities for each class, we define convenience functions to identify the properties explicitly by name. Theorems are introduced into ACL2 to rewrite those convenience functions to base unit components that MC instructions access. The benefit of this process is two-fold. First, the theorems written using the convenience functions reference the components and concepts by name, making the theorems more intuitive to read and write. Second, by rewriting only the base unit components for an operation, the ACL2 output is much easier to read. In this chapter, we will build the book for Clojure's `PersistentList` class to demonstrate.

`PersistentList` is a class in the Java portion of the Clojure core library and it is written in an Object Oriented fashion. The components in the abstraction layer apply to any Java class, but certain components are specifically emphasized because of the structure of classes that are generated by Clojure functions. Clojure uses static fields and interfaces extensively so the abstraction simplifies accessing static fields, identifying the interface types of instances, and invoking interface methods. `PersistentList`, despite falling on the pure Java side of the Clojure implementation, does use a static field and implements interfaces.

The `PersistentList` declaration in MC is shown in Figure 5.1. In addition to the static field `EMPTY`, `PersistentList` has three member fields `_first`, `_rest`, and `_count`. It implements a constructor and three accessor methods for its three member fields.

The symbols for the class functions follow a naming convention. The names are prefixed with the name of the class within |'s. If the function accesses an element of the class, it is included in the prefix after a ":", so a function describing the method

```
(defconst *clojure.lang.PersistentList*
  (make-class-decl
   ; class name
   "clojure.lang.PersistentList"
   ; Parent Classes
   '("clojure.lang.ASeq"
     "clojure.lang.Obj"
     "java.lang.Object")
   ; Interfaces
   '(
     ; PersistentList Interfaces
     "clojure.lang.IPersistentList"
     "clojure.lang.IReduce"
     "clojure.lang.List"
     "clojure.lang.Counted"
     ; ASeq Interfaces
     "clojure.lang.ISeq"
     "clojure.lang.Sequential"
     "java.util.List"
     "java.io.Serializable"
     "clojure.lang.IHashEq"
     )
   '("_first"
     "_rest"
     "_count")
   '("EMPTY")
   '()

   (list
    *clojure.lang.PersistentList-<init>*
    *clojure.lang.PersistentList-first*
    *clojure.lang.PersistentList-next*
    *clojure.lang.PersistentList-count*
    )
   '(REF -1)))
```

Figure 5.1: `PersistentList` MC Declaration

`run` in class `App` would be prefixed with |`App:run`|. The prefix is followed by a "`-`" and a description of the operation. The naming convention of the specific operations is self-evident. Long class names are abbreviated in prefixes that also contain a method or field. For example, the function that accesses the `EMPTY` field in `PersistentList` is abbreviated |`PL:EMPTY`|.

5.2 Static Fields

Except in situations that require interfacing directly with Java, Clojure functions are the composition of other functions. The other functions are *function dependencies* to the definition. When the definition is compiled, the resulting class stores references to the function dependencies as static fields. During the execution of the function, the dependencies are loaded into a frame using the GETSTATIC instruction and invoked upon. When verifying a function, it is necessary to configure its dependencies. Therefore, accessor functions are defined for each static field in a class that allow referencing the fields by name. Theorems are verified for each accessor that guarantee the correct static field is loaded onto the operand stack of the current frame.

`PersistentList` has a single static field `EMPTY` of type `EmptyList`, which is a sequence object that evaluates to null. `EMPTY` is returned by sequence operations that return empty. `EMPTY` is constructed when the class declaration is loaded. We define an accessor function to get `EMPTY`'s value from the heap.

```
1 (defun |PL:EMPTY|-get (heap class-table)
2   (static-field-value
3    "clojure.lang.PersistentList"
4    "EMPTY"
5    heap
6    class-table))
```

For the theorems written using the accessor to be reused as lemmas, ACL2 must know to introduce the term while stepping through a machine state. We construct a

theorem that will add the term when the field is accessed. The `EMPTY` field is accessed by the GETSTATIC instruction. We define a poised function that recognizes system states that are poised to execute GETSTATIC to access `EMPTY` on its next instruction.

```
(defun |PersistentList:EMPTY|-poised (s)
  (equal
   (next-inst s)
   '(GETSTATIC "clojure.lang.PersistentList"
               "EMPTY"
               NIL)))
```

The theorem `PersistentList-EMPTY-is-EMPTY` adds the term `|PL:EMPTY|-get` to the resulting state.

```
(defthm PersistentList-EMPTY-is-EMPTY
  (implies
   (|PL:EMPTY|-poised s)
   (-> s
       (modify
        s
        :pc (+ 3 (pc (top-frame s)))
        :stack
        (push
         (|PL:EMPTY|-get (heap s)
                         (class-table s))
         (stack (top-frame s)))))))
```

With this theorem, future theorems about code that uses `PersistentList` can be configured to specify the instance stored in the `EMPTY` field using the `|PL:EMPTY|-get` function.

5.3 Loading and Dependencies

The first step to reasoning about a snippet of bytecode is stating that the code is loaded. For each class that is simulated in our model, we define a `loaded?` predicate that confirms the class table of a state contains the declaration and its dependencies. `PersistentList` extends `ASeq` and has a static field of type `EmptyList` so its `loaded?`

function confirms both of those classes are also loaded.

```
(defun |PersistentList|-loaded? (s)
  (and
   (|ASeq|-loaded? s)
   (|EmptyList|-loaded? s)
   (loaded? s
            "clojure.lang.PersistentList"
            *clojure.lang.PersistentList*)
   (|EmptyList|-p
    (|PL:EMPTY|-get (heap s)
                    (class-table s))
    (heap s))))
```

The function |EmptyList|-p recognizes EmptyList instances. Instances are described in more detail in Section 5.4. For now, it is important to note that the JVM executes the code that initializes the static members. The loaded? function ensures that the class exists in the state that occurs after the class is loaded in the JVM.

The loaded? predicate is not recursive, so ACL2 expands the definition during simplification of a theorem, which causes a significant usability problem. When the definition is expanded, the constant *clojure.lang.PersistentList* is also expanded to show the entire class declaration in the ACL2 output. For systems with multiple classes containing many methods, each subgoal becomes several pages long, making it difficult to find relevant information indicating why theorems do not succeed. The key to solving this issue is disabling |PersistentList|-loaded?, but doing so hides relevant information from ACL2. We solve the issue by admitting theorems to ACL2 that introduce specific information from the loaded? predicate. For example, we want ACL2 to know that if PersistentList is loaded, that PersistentList's dependencies are also loaded.

```
(defthm |PersistentList|-dep
  (implies (|PersistentList|-loaded? s)
           (and (|ASeq|-loaded? s)
                (|EmptyList|-loaded? s)))
  :rule-classes :forward-chaining)
```

This rule is admitted as a forward chaining rule. Forward chaining rules do not add terms to the goal, but rather add the information to a context that ACL2 assumes during a proof attempt. Once all of the abstraction layer components described in this chapter are defined for a class declaration, the `loaded?` predicate for the class is disabled. At that point, proof attempts in ACL2 rely on the forward chaining rules and the remaining components to introduce class information to the logic.

5.4 Class Instances

MC stores instances as a list of class structure data. The class structure data is the class name followed by a list of fields. Each field is a pair containing a name and the field's value. The instance is ordered from the most derived class structure data to its earliest parent, which is always `java.lang.Object`. The instance type is the class's most derived class, so the class name is the `caar` of the instance.

```
(defun j-instance-classname (instance)
  (caar instance))

(defun j-type-p (class ref heap)
  (let* ((instance (deref ref heap))
         (class-name (j-instance-classname instance)))
    (equal class-name class)))

(defun |PersistentList|-p (ref heap)
  (and (not (nullrefp ref))
       (j-type-p "clojure.lang.PersistentList"
                 ref
                 heap)))
```

The type of an instance effects control flow. Based on the instance type, the INVOKEINTERFACE and INVOKEVIRTUAL instructions load methods and the INSTANCEOF instruction branches. Both cases require MC to know a class's parents and the interfaces it implements. Once the `loaded?` predicate is disabled, neither will be visible to ACL2 so we need to define rewrite rules that introduce the informa-

```
(defthm |PersistentList|-superclasses
  (implies
   (|PersistentList|-loaded? s)
   (equal
    (class-superclasses "clojure.lang.PersistentList"
                        (class-table s))
    (list "clojure.lang.ASeq"
          "clojure.lang.Obj"
          "java.lang.Object"))))

(defthm |PersistentList|-interfaces
  (implies
   (|PersistentList|-loaded? s)
   (equal
    (class-interfaces "clojure.lang.PersistentList"
                      (class-table s))
    (list "clojure.lang.IPersistentList"
          "clojure.lang.IReduce"
          "clojure.lang.List"
          "clojure.lang.Counted"
          "clojure.lang.ISeq"
          "clojure.lang.Sequential"
          "java.util.List"
          "java.io.Serializable"
          "clojure.lang.IHashEq"))))
```

Figure 5.2: `PersistentList` Superclasses and Interfaces

tion. The `|PersistentList|-superclasses` and `|PersistentList|-interfaces` theorems, which are shown in Figure 5.2, rewrite the superclass and interface lookup functions in MC for instances of type `PersistentList`.

The INSTANCEOF instruction leads to an immediate branching decision. By default, ACL2 splits branches into two subgoals and attempts to verify both. This will almost always fail because one branch is expecting a different instance type. ACL2 frequently realizes that one of the branches is not relevant, but as a practice, we verify the behavior of the INSTANCEOF instruction explicitly. The following theorem confirms that a `PersistentList` instance is an `ASeq`.

```
(defthm PersistentList-instanceof-ASeq
  (implies
   (and (|PersistentList|-loaded? s)
        (equal ref (top (stack (top-frame s))))
        (|PersistentList|-p ref (heap s))
        (equal (next-inst s)
               '(INSTANCEOF "clojure.lang.ASeq")))
   (equal (step s)
          (modify s
                  :pc (+ 3 (pc (top-frame s)))
                  :stack
                  (push 1
                        (pop (stack (top-frame s))))))))
```

We also define accessors for the instance fields that are stored in the class data structure. These are similar to the static field accessors, but operate on a instance rather than the loaded class declaration. For each field in a class, accessor functions are defined to get and set the field values. For example, the `PersistentList` class has a `_first` field. The functions to get and set the value of `_first` on a `PersistentList` object are:

```
(defun |PersistentList:_first|-get (ref heap)
  (let* ((instance (deref ref heap)))
    (field-value "clojure.lang.PersistentList"
                 "_first"
                 instance)))

(defun |PersistentList:_first|-set (instance value)
  (set-instance-field "clojure.lang.PersistentList"
                      "_first"
                      value
                      instance))
```

The get function for instance fields is used to configure theorems in the same way as the static field get functions. However, unlike static fields, instance fields in Clojure programs may be set or modified during execution. The set function abstracts that modification so that the theorem's configuration can be updated by MC during a proof attempt.

5.5 Method Lookup

The methods in the declaration are defined as individual constants. Theorems are introduced into ACL2 that rewrite calls to the lookup procedure for the method into the method constant. Recall from Section 4.2 that the lookup procedure is a two function process. The top-level function `lookup-method` constructs a list of class names containing the name of the base class and all of its children. The bottom-level function `lookup-method-in-superclasses` iterates over the list searching for the method in each class. For methods invoked by the instructions INVOKESTATIC or INVOKESPECIAL, it is acceptable to write the theorem to rewrite instances of the top-level lookup function because the method is called on a specific class name. The `PersistentList` constructor is always invoked by INVOKESPECIAL, so its method lookup theorem is

```
(defthm |PersistentList:<init>|-method
  (implies
   (|PersistentList|-loaded? s)
   (equal (lookup-method "<init>"
                         "clojure.lang.PersistentList"
                         (class-table s))
          *clojure.lang.PersistentList-<init>*)))
```

Methods that are invoked by the INVOKEINTERFACE or INVOKEVIRTUAL instructions will not be found if the theorem rewrites the top-level function because those instructions do not specify an exact class name. For example, if a child class extends `PersistentList` and a `PersistentList` method is invoked on an object of the child, `lookup-method` will be evaluated on the class name of the child. In this case, the rewrite rule would not match and ACL2 would not apply it to the term. Instead, these methods are found using theorems that rewrite the bottom-level function `lookup-method-in-superclasses`. `PersistentList`'s `first` method is run by executing INVOKEINTERFACE on `ISeq`, so an example of a theorem that

rewrites the bottom-level function is shown using it.

```
(defthm |PersistentList:first|-method
  (implies
   (and (|PersistentList|-loaded? s)
        (equal (car classes)
               "clojure.lang.PersistentList"))
   (equal (lookup-method-in-superclasses "first"
                                         classes
                                         (class-table s))
          *clojure.lang.PersistentList-first*)))
```

`PersistentList`'s `first` method is just an accessor to the `_first` field. As shown in the previous section, the abstraction layer defines ACL2 accessor functions for all fields in a class declaration to simplify the configuration of theorems. Since `first` is an accessor to a field with an ACL2 accessor, we verify that executing the `first` method on an instance behaves identically to executing the `|PL:_first|-get` on a reference to the instance. The poised function `|PL:first|-poised` specifies that the state `s` is poised to invoke the `first` method.

```
(defthm |PL:first|=_first
  (implies
   (and (|PersistentList|-loaded? s)
        (|PL:first|-poised s))
   (-> s
       (modify
        s
        :pc (+ 5 (pc (top-frame s)))
        :stack
        (push
         (|PL:_first|-get (top (stack (top-frame s)))
                          (heap s))
         (pop (stack (top-frame s))))))))
```

The theorem verifies that a system executing the `first` method pushes the instance's value of `_first`, as defined by the ACL2 accessor function, onto the stack.

5.6 Constructors

The JVM separates the construction of an instance into two steps. The first step allocates an empty object on the heap. This step is issued on the JVM with the NEW instruction. The second step invokes the class's constructor to execute the initialization logic. This step is issued on the JVM with the INVOKESPECIAL instruction. The function |PersistentList|-new models the NEW instruction.

```
1  (defun |PersistentList|-new ()
2    (build-an-instance
3     (list "clojure.lang.PersistentList"
4           "clojure.lang.ASeq"
5           "clojure.lang.Obj"
6           "java.lang.Object")
7     (make-class-def
8      (list *clojure.lang.Obj*
9            *clojure.lang.ASeq*
10           *clojure.lang.PersistentList*))))
```

The NEW instruction allocates a new reference from the heap and adds the reference to the stack. The theorem new-PL-creates-new-PL shows the heap allocating a reference by executing NEW in lines 10 through 12. MC only adds elements to the heap, so the allocation is to the next index as determined by the length of the current heap. The lines 8 and 9 show the reference being returned.

```
1  (defthm new-PL-creates-new-PL
2    (implies
3     (and (poised-to-new s "clojure.lang.PersistentList")
4          (|PersistentList|-loaded? s))
5     (-> s
6         (modify s
7                 :pc (+ 3 (pc (top-frame s)))
8                 :stack (push (list 'REF (len (heap s)))
9                              (stack (top-frame s)))
10                :heap (bind (len (heap s))
11                            (|PersistentList|-new)
12                            (heap s))))))
```

The second step of the initialization executes the <init> method.

```
(defthm |PL:init|-initializes
  (implies
   (and (|PersistentList|-loaded? (class-table s)
                                  (heap s))
        (|PL:<init>|-poised s)
        (equal (deref (top (pop (stack (top-frame s))))
                      (heap s))
               (|PersistentList|-new)))
   (-> s
       (modify
        s
        :pc (+ 3 (pc (top-frame s)))
        :stack (popn 2 (stack (top-frame s)))
        :heap
        (bind (cadr (top (pop (stack (top-frame s)))))
              (|PersistentList|-init
               (deref (top (pop (stack (top-frame s))))
                      (heap s))
               (top (stack (top-frame s))))
              (heap s))))))
```

In the programs analyzed in this dissertation, the static field values are only set once during class loading. MC is not used to reason about the JVM's class loading behavior, so the static fields are specified in the `loaded?` predicates for each class. For example, the `loaded?` predicate for `PersistentList` specifies the value of `EMPTY` after the class declaration is loaded. If the values defined by `loaded?` are not valid, the proofs that depend on the static fields would also be invalid. The specified value for the static field `EMPTY` is justifiable, though. The `EmptyList` class declaration is created using the same paradigm for class development that is described in this chapter. It includes the constructor functions and corresponding proofs that verify that new objects are allocated correctly.

5.7 Summary

In Java, classes are responsible for defining the storage of static and instance fields, referencing class dependencies and methods, and defining the structure of instances of the class. In short, the behavior of a Java program is almost entirely defined by the set of class declarations in the program. Previous results have indicated a need for applying higher-level abstractions to class declarations in a JVM model when reasoning about large programs (Liu, 2006). In this chapter, each of the responsibilities of a class was separated into specific functions for the responsibility and theorems were verified that enable ACL2 to execute the responsibility independent of the rest of the declaration. In addition to the generic needs of Java programs, additional emphasis was added to abstracting the access of static fields and referencing a class's interfaces because most compiled Clojure functions use both features.

All class declarations that are reasoned about in this dissertation have been structured using the abstractions presented in this chapter. Two of the theorems verified in this chapter, `|PL:first|=_first` and `|PL:init|-initializes`, are examples of the type of bytecode analysis MC is built to perform. In both cases, the theorem configures a system poised to execute a piece of code containing multiple instructions. The conclusion applies the big-step equivalence macro `->` to verify the behavior of running all of the instructions in the method. In the remaining chapters, a Clojure program is separated into method-sized portions of code and reasoned about in the same way using ACL2.

Chapter 6

Sequences as Lists

The Clojure core library implements the features for the entire Clojure language. It is a large library containing 579 Clojure definitions that is supported by an underlying Java system. In this dissertation, we focus on modeling in MC a single, comprehensive section of the core library and demonstrating that Clojure functions programmed using the modeled section can be analyzed in MC. We choose to focus on Clojure's sequence data structure because it compares to list, which is a fundamental data structure in Common Lisp. Additionally, functions that operate on lists are commonly recursive. Recursive functions are a staple of functional programming and reasoning about recursive functions using induction is a strength of ACL2.

Modeling a specific data structure in MC requires abstractions for three components: (1) the underlying Java classes that implement the structure, (2) the allocation of instances on the heap, and (3) the Clojure core functions that operate on the structure. The abstractions are built in MC with the explicit intention of applying them to the inductive reasoning of recursive functions. Inductive reasoning, however, poses a problem. Recursive functions that operate on lists can be reasoned about inductively because lists are immutable and well-ordered. Sequences are also immutable and well-ordered when constructed and operated on by Clojure functions, but those properties cannot be inferred by the representation of a sequence in MC. Abstractions for (1) and (2), which are described in Section 6.1 and Section 6.2, address the well-ordered property of sequences.

The immutable property of sequences is inherently enforced by only applying the

model to functions that depend on *acceptable functions*. In this situation, acceptable functions are those that have a Common Lisp analog. For component (3), the seven acceptable core library functions are verified to behave correctly with respect to their Common Lisp analog. Functions in the core library are defined in two ways and both are analyzed in this chapter. Core functions either directly call the underlying Java code or are the composition of other functions. Four examples of functions that call Java are verified in Section 6.3. A remaining function, and its dependencies, are verified in Section 6.4. Section 6.4 also details the format of a compiled Java function.

The functions verified in this chapter represent the basis of a verified `core` library. The functions are proven correct against their logical specification, as defined by the Clojure documentation and the ACL2 semantics of the functions. New Clojure functions, defined using the verified set presented in this chapter, can be reliably reasoned about directly in ACL2 without including MC. In the next chapter, a recursive function that combines these verified functions is verified using MC to validate this claim.

6.1 Sequence Overview

In Common Lisp, the cons cell is a basic data type that stores an ordered pair. The first element of the pair is the `car` and the second element is the `cdr`. A list is a cons cell that has a cons cell or nil in the `cdr`. In Clojure, the sequence data structure is a generic type that represents a superset of list. Sequences are actually implemented in the underlying Java code as a set of classes: `PersistentList`, `Cons`, and `EmptyList`. Sequence classes implement the `ISeq` interface which defines contracts for methods `first`, `next`, `more`, and `cons`. In this dissertation, *sequence* refers to the Clojure structure and *list* refers exclusively to the ACL2 structure. This section explains how sequences are related to lists. The code in this section reference functions `seq-first`

and `seq-more`, which retrieve a sequence's `first` and `more` fields respectively. The definitions will be introduced later in the chapter.

Sequences are reasoned about in ACL2 as references to sequence instances. The predicate `seq-p` recognizes a sequence reference given a heap.

```
(defun seq-p (ref heap)
  (and (not (nullrefp ref))
       (or (|Cons|-p ref heap)
           (|PersistentList|-p ref heap)
           (|EmptyList|-p ref heap))))
```

A sequence is traversed by calling `seq-more` on a sequence, which returns a reference to the next instance in the sequence. The Clojure function used to construct a sequence will not create one with cycles, but there is no type information stored for a sequence reference that can guarantee that it does not contain a cycle. A sequence with cycles would not be a well-ordered set so it would be incompatible with induction. Sequences must be constrained to ordered lists. Our approach is to define a list of sequence references such that each sequence's `more` field is the next reference in the list. The `more` field of the last reference in the list must be the static `EMPTY` field. The predicate `seq-listp` recognizes these lists.

```
(defun seq-listp (xs heap class-table)
  (if (endp xs)
    (equal xs nil)
    (and (seq-p (car xs) heap)
         (not (|EmptyList|-p (car xs) heap))
         (if (endp (cdr xs))
           (equal (seq-more (car xs) heap class-table)
                  (|PL:EMPTY|-get heap
                                  class-table))
           (equal (seq-more (car xs) heap class-table)
                  (cadr xs)))
         (seq-listp (cdr xs) heap class-table))))
```

Note that the list does not contain any `EmptyList` sequences. We use this feature in the next chapter to verify the base case in an inductive proof.

The inductive step to verify a property $P(xs)$ for sequences is $P(more(xs)) \Rightarrow P(xs)$. In most cases, the Clojure functions that are verified are overloaded to operate on types other than the sequence types modeled in this chapter. Therefore, our lemmas only reason about a function by assuming the input is a sequence. Such lemmas will only apply as rewrite rules in the inductive step if `seq-more` returns a sequence reference. The definition of `seq-listp` is sufficient to verify that property. If `seq-more` is called on sequence from a sequence list, the result is also sequence. The following theorem verifies.

```
(defthm seq-more-of-seq-is-seq
  (implies
   (and (|Cons|-loaded? class-table heap)
        (|PersistentList|-loaded? class-table heap)
        (not (endp xs))
        (seq-listp xs heap class-table))
   (seq-p (seq-more (car xs) heap class-table) heap))
```

6.2 Sequence Allocation

A Clojure function that recursively constructs a sequence will allocate an object for every element in the result. When the function is mapped to an ACL2 function, it is necessary to demonstrate that the result of the ACL2 function is allocated in MC. Our approach follows the one taken by Moore (2003) to reason about a Java insertion sort method in M5. Moore defines a heap invariant and theorems that demonstrate it holds after allocation. In this section, an allocation function is defined for cons allocation and two properties of Moore's invariant are described and verified. A third invariant that is unique to our examples is described and verified as well.

The code for the `cons` method is described in detail in Section 6.3.4, but it is a function that constructs a sequence element from a value `x` and an existing sequence `coll`, which is short for *collection*. An individual allocation initiated by `cons` creates

a new `Cons` instance if `coll` is non-null; otherwise, it creates a `PersistentList` instance. The `alloc` function implements the logic.

```
(defun alloc (x coll heap)
  (if (null coll)
    (bind (len heap)
          (|PersistentList|-init (|PersistentList|-new) x)
          heap)
    (bind (len heap)
          (|Cons|-init (|Cons|-new) x coll)
          heap)))
```

The `alloc-list` function takes a list of sequences and allocates a new sequence for each element. Sequences allocated recursively using `cons` allocate objects from the last element forward. Therefore, `alloc-list` performs allocations in that order and calculates the reference index for the `coll` input as the sum of the length of the heap and the length of the list minus 1.

```
(defun alloc-list (xs heap)
  (if (endp xs)
    heap
    (let* ((x (seq-first (car xs) heap)))
      (if (endp (cdr xs))
        (alloc x nil heap)
        (alloc x
               (list 'REF
                     (+ (len heap) (len (cdr xs)) -1))
               (alloc-list (cdr xs) heap))))))
```

In ACL2, association lists are recognized by `alistp`. Association lists are lists of ordered pairs that interpret the first element in the pair as a key and the second element as a value. The heap is an association list where the keys are references and the values are objects. The first invariant property verifies that the heap is still an association list after allocating a sequence.

```
(defthm alistp-alloc-list
  (implies (alistp heap)
           (alistp (alloc-list xs heap))))
```

The NEW instruction is implemented in MC to always add the instance at a reference index equal to the length of the initial heap. MC does not garbage collect heap objects, so there is no mechanism for reducing the heap. If the initial heap only consists of reference indexes less than the length, then allocations in MC are additive and the heap grows predictably. The function `all-smallp` is used to recognize this property. `all-smallp` recognizes heaps where the index of every reference in the heap is less than a maximum value.

```
(defun all-smallp (heap max)
  (cond
    ((endp heap) t)
     (t (and (integerp (caar heap))
             (<= 0 (caar heap))
             (< (caar heap) max)
             (all-smallp (cdr heap) max)))))
```

The function `all-smallp` was originally defined in the ACL2 community book associated with Moore (2003) as part of the heap invariant. The `all-smallp` property holds after allocating a sequence.

```
(defthm all-smallp-alloc-list-heap
  (implies
    (and (alistp heap)
         (all-smallp heap (len heap)))
    (all-smallp (alloc-list xs heap)
                (len (alloc-list xs heap)))))
```

The third invariant property is the preservation of the heap configuration. Basically, the type and values of existing heap references must not change after an allocation. This invariant is verified with a set of theorems that can be organized into three groups. Examples of each are shown in Figure 6.1. The first group of theorems verifies, for every type, that references to instances of that type are still that type after an allocation. The theorem `EmptyList-persists` demonstrates this property for the `EmptyList` type. The second group verifies that every static field maintains its value

```
(defthm EmptyList-persists
  (implies
   (and (|EmptyList|-p ref heap)
        (alistp heap)
        (all-smallp heap (len heap)))
   (|EmptyList|-p ref
                  (alloc-list xs heap))))

(defthm |PL:EmptyList|-persists
  (implies
   (and (alistp heap)
        (all-smallp heap (len heap)))
   (equal (|PL:EMPTY|-get (alloc-list xs heap)
                          class-table)
          (|PL:EMPTY|-get heap class-table))))

(defthm PersistentList-loaded-persists
  (implies
   (and (|PersistentList|-loaded? class-table heap)
        (alistp heap)
        (all-smallp heap (len heap)))
   (|PersistentList|-loaded? class-table
                             (alloc-list xs heap))))
```

Figure 6.1: Heap Configuration Preserved by Allocation

after allocation. An example of the second group is |PL:EmptyList|-persists, which shows that PersistentList's EMPTY member is preserved. The third group verifies that if a class is loaded before an allocation, it is still loaded after an allocation. For this group, a theorem is verified in ACL2 for each loaded? predicate. The theorem PersistentList-loaded-persists is an example of this group. Interestingly, since |PersistentList|-loaded? defines |PL:EMPTY|-get as an EmptyList, it relies on the first two theorems as lemmas.

The reference to an allocation is generally returned on the stack as the result. Therefore, it must be possible to derive the last reference value allocated. An all-smallp heap grows predictably by allocating a single element for each object in the sequence so the new heap is the combined length of the initial heap and xs.

```
(defthm len-alloc-list-heap
  (implies
    (and (alistp heap)
         (all-smallp heap (len heap)))
    (equal (len (alloc-list xs heap))
           (+ (len xs) (len heap)))))
```

The returned reference index is the length of the new sequence minus one, but ACL2 accounts for the difference without a specific rewrite rule. Theorems that allocate a sequence from the elements in a `seq-listp` use this theorem to determine the returned reference.

6.3 Static Run-Time Methods

Clojure implements several operations as static methods in the Run-Time class `RT`. The core functions defined in Clojure are wrappers that compile to direct calls of those `RT` methods. The functions `cons`, `first`, and `rest` are all defined as wrappers to `RT` methods. The `empty?` function is defined using the functions `seq` and `not`. The `seq` function is also a wrapper for a `RT` method call. In this section, the functions `seq`, `first`, `rest`, and `cons` are mapped to the corresponding logic in ACL2. The theorems presented in this section apply the sequence structure to small operations and are used as rewrite rules later for more complex functions.

The functions in this and the next section are key operations performed on sequences. It covers type-checking, access, traversal, and construction of sequences. In addition, the `not` function verifies boolean negation, which is important for implementing branching. These functions can be combined to create common function patterns that iterate over the elements in a sequence or iteratively construct new sequences.

```
1  static public ISeq seq(Object coll){
2    if(coll instanceof ASeq)
3      return (ASeq) coll;
4    else { /* Additional cases */ }
5    else
6      return seqFrom(coll);
7  }
8
9  static ISeq seqFrom(Object coll){
10   if(coll instanceof Seqable)
11     return ((Seqable) coll).seq();
12   else if(coll == null)
13     return null;
14   else {  /* Additional cases */ }
15 }
```

Figure 6.2: `RT` `seq` and `seqFrom` Methods

6.3.1 Seq

The `seq` function coerces an input into a sequence or null if the input cannot be coerced. It is a wrapper for the `seq` method, which depends on the `seqFrom` method to perform the type coercion. The `seq` and `seqFrom` methods are shown in Figure 6.2. For this investigation, the inputs to `seq` are either a sequence or null. The `seq` function acts as an identity function for inputs of type `Cons`, `PersistentList`, and null. The `Cons` and `PersistentList` classes extend `ASeq` so the `seq` method returns the input on line 3 of Figure 6.2. A null input is identified and returned at lines 13 and 14. The `EmptyList` does not extend `ASeq` but does implement `Seqable`. Line 13 in Figure 6.2 shows that `seqFrom` returns the result of calling the instance method `seq` on the object. `EmptyList`'s implementation of the `seq` method returns null.

The `seq` behavior is defined in two theorems in ACL2, which are shown in Figure 6.3. The first, `|RT:seq|-null-input`, verifies the behavior when the input is null. The second, `|RT:seq|-sequence-input`, verifies the behavior when the input is a sequence.

```
(defthm |RT:seq|-null-input
  (let* ((coll (top (stack (top-frame s)))))
    (implies
     (and (|RT|-loaded? (class-table s)
                        (heap s))
          (|RT:seq|-poised s)
          (nullrefp coll))
     (-> s
         (modify s
                 :pc (+ 3 (pc (top-frame s)))
                 :stack
                 (push (nullref)
                       (pop (stack (top-frame s)))))))))

(defthm |RT:seq|-sequence-input
  (let* ((coll (top (stack (top-frame s)))))
    (implies
     (and (|RT|-loaded? (class-table s)
                        (heap s))
          (|RT:seq|-poised s)
          (seq-p coll (heap s)))
     (-> s
         (modify s
                 :pc (+ 3 (pc (top-frame s)))
                 :stack
                 (push (if (|EmptyList|-p coll (heap s))
                           (nullref)
                           coll)
                       (pop (stack (top-frame s)))))))))
```

Figure 6.3: Verified Correctness of `seq`

6.3.2 First

The `RT` method for `first` returns the `car` of the sequence.

```
static public Object first(Object x){
  if(x instanceof ISeq)
    return ((ISeq) x).first();
  ISeq seq = seq(x);
  if(seq == null)
    return null;
  return seq.first();
}
```

The `seq-first` function implements the logic in ACL2. It takes a reference and a heap. If the reference points to a `Cons` or `PersistentList` instance, it returns the element the instance stores in its `first` field. If it is neither, it returns a null reference.

```
(defun seq-first (ref heap)
  (if (|Cons|-p ref heap)
    (|Cons:_first|-get ref heap)
    (if (|PersistentList|-p ref heap)
      (|PL:_first|-get ref heap)
      (nullref))))
```

The theorem `|RT:first|=first` verifies that running the bytecode produced by compiling the `first` method pushes the result of `seq-first` onto the stack.

```
(defthm |RT:first|=first
  (let* ((x (top (stack (top-frame s)))))
    (implies
     (and (|RT|-loaded? (class-table s) (heap s))
          (|RT:first|-poised s)
          (seq-p x (heap s)))
     (-> s
         (modify s
                 :pc (+ 3 (pc (top-frame s)))
                 :stack
                 (push
                   (seq-first x (heap s))
                   (pop (stack (top-frame s)))))))))
```

6.3.3 Rest

The `rest` function returns the `cdr` of a sequence. It is a wrapper for `RT`'s `more` method, shown here:

```
static public ISeq more(Object x){
  if(x instanceof ISeq)
    return ((ISeq) x).more();
  ISeq seq = seq(x);
  if(seq == null)
    return PersistentList.EMPTY;
  return seq.more();
}
```

The `more` method is very similar to `first`. If the input is not a sequence, an `EmptyList` is returned. The `seq-more` function implements the logic in ACL2.

```
(defun seq-more (ref heap class-table)
  (if (|EmptyList|-p ref heap)
    ref
    (if (nullrefp ref)
      (|PL:EMPTY|-get heap class-table)
      (if (|Cons|-p ref heap)
        (if (nullrefp (|Cons:_more|-get ref heap))
          (|PL:EMPTY|-get heap
                          class-table)
          (|Cons:_more|-get ref heap))
        (if
          (or (= (|PL:_count|-get ref heap) 1)
              (nullrefp (|PL:_rest|-get ref heap)))
          (|PL:EMPTY|-get heap class-table)
          (|PL:_rest|-get ref heap))))))
```

The verification of `more` is split into two theorems, shown in Figure 6.4. The theorem `|RT:more|=seq-more` verifies that running the bytecode produced by compiling the `more` method pushes the result of `seq-more` onto the stack. The theorem `seq-more-of-seq-is-seq` in Section 6.1 states that `seq-more` returns a sequence reference. It is necessary to reason inductively about recursive functions because it guarantees the type of the input does not change on the recursive call. By proving

```
(defthm |RT:more|=seq-more
  (implies
   (and (|RT|-loaded? (class-table s)
                      (heap s))
        (|RT:more|-poised s)
        (seq-p (|RT:more|-param1 s) (heap s)))
   (-> s
       (modify s
               :pc (+ 3 (pc (top-frame s)))
               :stack
               (push
                 (seq-more (|RT:more|-param1 s)
                           (heap s)
                           (class-table s))
                 (pop (stack (top-frame s))))))))

(defthm |RT:more(null)|=null
  (implies
   (and (|RT|-loaded? (class-table s)
                      (heap s))
        (|RT:more|-poised s)
        (nullrefp (|RT:more|-param1 s)))
   (-> s
       (modify s
               :pc (+ 3 (pc (top-frame s)))
               :stack
               (push (seq-more (|RT:more|-param1 s)
                               (heap s)
                               (class-table s))
                     (pop (stack (top-frame s))))))))
```

Figure 6.4: Verified Correctness of `more`

```
static public ISeq cons(Object x, Object coll){
  if(coll == null)
    return new PersistentList(x);
  else if(coll instanceof ISeq)
    return new Cons(x, (ISeq) coll);
  else
    return new Cons(x, seq(coll));
}
```

Figure 6.5: `RT cons` Method

|`RT:more`|=`seq-more` in terms of `seq-more`, the value of `seq-more-of-seq-is-seq` is applicable to proofs about `more`. |`RT:more(null)`|=`null`, also shown in Figure 6.4, verifies the behavior of `more` when executed on a null reference.

6.3.4 Cons

The `RT cons` method, in Figure 6.5, takes two parameters: `x` and `coll`. If `coll` is null, `cons` creates a `PersisentList` instance with a `first` field containing `x`. If `coll` is not null and it is an instance of a class that implements `ISeq`, the `cons` function creates a `Cons` instance with a `first` field containing `x` and a `more` field containing `coll`. The `alloc` function defined in the previous section implements this behavior. The theorem |`RT:cons`|-`alloc`, which is shown in Figure 6.6, verifies that invoking the `cons` method allocates the appropriate instance and returns a reference to it. It specifically introduces the term `alloc` into the logic when a program executes the `cons` method.

6.4 Function Classes

The functions in the previous section are basically function wrappers to the underlying Java code. Unlike those functions, most Clojure functions, especially those defined in a functional style, are defined as the composition of other functions. The compiled

```
(defthm |RT:cons|-alloc
  (let* ((x (top (pop (stack (top-frame s)))))
         (coll (top (stack (top-frame s)))))
  (implies
   (and (|RT|-loaded? (class-table s) (heap s))
        (|RT:cons|-poised s)
        (or (nullrefp coll)
            (and (seq-p coll (heap s))
                 (not (null coll)))))
   (-> s
       (modify s
               :pc (+ 3 (pc (top-frame s)))
               :stack
               (push (list 'REF (len (heap s)))
                     (popn 2 (stack (top-frame s))))
               :heap
               (alloc x
                      (if (nullrefp coll)
                        nil
                        coll)
                      (heap s)))))))
```

Figure 6.6: Verified Correctness of `cons`

version of such a function is more complex than the function wrappers already verified because composite functions must load references to the objects that implement its function dependencies. In this section, the structure of a compiled composite function is explained along with our approach to configuring the structure in theorems. All user-defined functions composed of our verified `core` library will be compiled to Java classes that match this format, so our structure for configuring the function dependencies is a Clojure-specific component that fits with our class abstraction layer from Chapter 5.

All Clojure functions are compiled to Java classes that extend the `AFunction` class. `AFunction` implements the `IFn` interface that defines signatures for `invoke` methods of different arity. The `AFunction` implementation of each signature throws an `ArityException` error. A compiled Clojure function overrides the `invoke` method of the appropriate arity. Wrapper functions are compiled into classes that have an `invoke` method that calls the corresponding `RT` static method. Composite functions, in contrast, compile to classes that load other `AFunction` objects that implement its function dependencies. Those dependencies must be configurable in MC and the configuration must be capable of applying the proofs about those dependencies to the verification of the composite function. This capability is demonstrated in this section through the verification of the `empty?` Clojure function.

The Clojure `empty?` function is the composition of the functions `not` and `seq`.

```
1 (defn empty?
2   {:static true}
3   [coll] (not (seq coll)))
```

The function evaluates an input `coll` to a `Boolean`. If `coll` is null or an `EmptyList`, `empty?` evaluates to true. Any other type of sequence evaluates to false. Figure 6.7 shows the code for the class that implements the `empty?` logic. At a high level, Clojure stores an `AFunction` instance as the `root` field of a `Var` instance. `Var` instances are

```
1  public final class core$empty_QMARK_ extends AFunction {
2      public static final Var const__0 =
          (Var)RT.var("clojure.core", "not");
3      public static final Var const__1 =
          (Var)RT.var("clojure.core", "seq");
4
5      public core$empty_QMARK_() {
6      }
7
8      public Object invoke(Object coll) {
9          IFn var10000 = (IFn)const__0.getRawRoot();
10         IFn var10001 = (IFn)const__1.getRawRoot();
11         Object var10002 = coll;
12         coll = null;
13         return var10000.invoke(var10001.invoke(var10002));
14     }
15 }
```

Figure 6.7: The `Empty` Class

stored in a hashmap and referenced by name within `Namespaces`. `Namespaces` are stored in a hashmap as well. Line 2 shows a `Var` being loaded that is named "not" in namespace "clojure.core" and line 3 shows a `Var` being loaded for "seq".

At a low level, function loading is implemented using concurrency and Java's `Unsafe` class wrapper for direct memory management, neither of which is supported by MC. Also, functions are loaded statically during class loading, which is not a period of operation considered in this research. Therefore, the function loading is assumed to create a `Var` object with the correct `root`. The information is added to the `loaded?` predicate function for `AFunction` classes. The `loaded?` predicate for the `Empty` class is shown in Figure 6.8.

The function |`Var:root`|`-get` accesses the `root` field directly. The predicates |`not`|`-p` and |`seq`|`-p` recognize instances of `AFunction` classes that implement the `not` and `seq` logic. `AFunction` classes are loaded into the logic of an `invoke` method by executing `Var`'s `getRawRoot` method. The theorem |`Var:getRawRoot`|`-is-root`

```
(defun |empty|-loaded? (class-table heap)
  (and (|Var|-loaded? class-table)
       (|not|-loaded? class-table heap)
       (|seq|-loaded? class-table heap)
       (loaded? class-table
                "clojure.core$empty_QMARK_"
                *clojure.core$empty_QMARK_*)
       (|Var|-p (|empty:not|-get heap class-table)
                heap)
       (|Var|-p (|empty:seq|-get heap class-table)
                heap)
       (|not|-p
         (|Var:root|-get
           (|empty:not|-get heap class-table)
           heap)
         heap)
       (|seq|-p
         (|Var:root|-get
           (|empty:seq|-get heap
                            class-table)
           heap)
         heap)))
```

Figure 6.8: Configuration of `Empty` Class Object

verifies that invoking `getRawRoot` returns the `root` field.

```
(defthm |Var:getRawRoot|-is-root
  (implies
   (and (|Var|-loaded? (class-table s))
        (|Var:getRawRoot|-poised s)
        (|Var|-p (top (stack (top-frame s)))
                 (heap s)))
   (-> s
       (modify
        s
        :pc (+ 3 (pc (top-frame s)))
        :stack
        (push
          (|Var:root|-get (top (stack (top-frame s)))
                          (heap s))
          (pop (stack (top-frame s))))))))
```

`AFunction invoke` methods are executed by INVOKEINTERFACE instructions. The poised functions include the object type that the instruction is being invoked upon. The poised function for invoking the `Empty` class is

```
(defun |empty:invoke|-poised (s)
  (and
   (equal (next-inst s)
          '(INVOKEINTERFACE "clojure.lang.IFn"
                            "invoke"
                            1))
   (|empty|-p (top (pop (stack (top-frame s))))
              (heap s))))
```

If the function references are defined, the analysis of the code precedes similarly to the previous examples. The theorems in Figure 6.9 prove the correctness of the `empty?`. Theorem `|empty|-sequence-is-false` verifies that `empty?` evaluates to false for non-`EmptyList` sequences and `|empty|-null-or-EmptyList-is-true` verifies that `empty?` evaluates to true when the input is either an `EmptyLists` reference or null.

```
(defthm |empty|-sequence-is-false
  (let* ((coll (top (stack (top-frame s)))))
    (implies
     (and (|empty|-loaded? (class-table s) (heap s))
          (|empty:invoke|-poised s)
          (|Boolean|-loaded? (class-table s) (heap s))
          (seq-p coll (heap s))
          (not (|EmptyList|-p coll (heap s))))
     (-> s
         (modify
          s
          :pc (+ 5 (pc (top-frame s)))
          :stack
          (push (|Boolean:FALSE|-get (heap s)
                                     (class-table s))
                (popn 2 (stack (top-frame s))))))))

(defthm |empty|-null-or-EmptyList-is-true
  (let* ((coll (top (stack (top-frame s)))))
    (implies
     (and (|empty|-loaded? (class-table s) (heap s))
          (|empty:invoke|-poised s)
          (or (nullrefp coll)
              (|EmptyList|-p coll (heap s))))
     (-> s
         (modify
          s
          :pc (+ 5 (pc (top-frame s)))
          :stack
          (push (|Boolean:TRUE|-get (heap s)
                                    (class-table s))
                (popn 2 (stack (top-frame s))))))))
```

Figure 6.9: Verified Correctness of `empty?`

6.5 Summary

Clojure executes on the JVM, so MC reasonably is modeled after the JVM. However, at the JVM level, sequences do not have the requisite properties to analyze their use inductively. This chapter introduced an abstraction of Clojure sequences that allow inductive reasoning. In addition, the Clojure core functions that operate on sequences were verified to be correct. In the next chapter, a recursive function is verified that combines all of the concepts presented in this chapter.

Chapter 7

Sequence Recursion

This dissertation is defending the thesis that new Clojure functions defined using verified `core` functions can be reasoned about in ACL2. In support of the thesis, a new Clojure function is defined and verified in ACL2 that uses the `core` sequence functions that were verified in the previous chapter. The verification requires combining most of the previous results into a new proof. The new function compiles to a Java class that has many class dependencies so it requires the abstraction layer from Chapter 5 to shrink the search space for a proof. It operates on sequences so it requires the sequence abstraction and the verified properties of the sequence functions from Chapter 6. Finally, it combines the verified properties of the sequence functions by using the big-step equivalence macro `->` from Chapter 4. A recursive function is chosen as the new function because it is common in functional programming but adds complexity to the verification process.

Recursion is a concise method for specifying an algorithm that can be reasoned about with mathematical induction. Successful inductive proofs of recursive functions structure the definitions so that the inductive hypothesis introduces the information necessary to simplify the desired conclusion. This can happen naturally in ACL2 when the terms used for the inductive hypothesis match the terms used for the recursive call within a function definition. However, recursion in bytecode does not naturally provide matching terms because the machine configuration for each state differs. In this chapter, an inductive proof of a recursive Clojure function is described that maps the Clojure function to a corresponding ACL2 function.

As an example recursive function, we use an exercise from a library designed for teaching software engineering students. The `every-other` function constructs a sequence that keeps the first element and removes every other element after it.

```
(defn every-other [xs]
  (if (empty? xs)
    nil
    (cons (first xs)
          (every-other (rest (rest xs))))))
```

The structure of `every-other` is typical of functional code that constructs a sequence from an existing sequence. The `every-other` example is used because it is a non-identity function that uses only the sequence operations verified in the previous chapter.

Clojure compiles `every-other` into an `EveryOther` class that extends `AFunction`. The function dependencies `empty?`, `cons`, `first`, `every-other`, and `rest` are stored in `Var`'s `const__0`, `const__1`, `const__2`, `const__3`, and `const__4`. The root of `const__3` is an instance of `EveryOther`. Throughout the chapter, it is referred to as `self` in the text and as `(|eo:every_other|-get heap class-table)` in the code.

Figure 7.1 shows the Java code gathered by decompiling the Java class file generated by the Clojure compiler. The code is annotated to indicate which function is referred to by each `const` object. The `invoke` method allocates to the heap a new sequence of objects that contain the result and returns a reference to the initial element in the sequence. To map `every-other` into an ACL2 `every-other` function, an inductive proof is constructed in ACL2 that verifies that the `every-other` function infers the new heap and return reference.

7.1 Cutpoints

`EveryOther` is reasoned about in segments and those segments are combined to verify the mapping. The first segment is the code that executes for `EmptyList` or null inputs,

```
public Object invoke(Object xs) {
    // const__0 = empty?
    Object var10000 =
      ((IFn)const__0.getRawRoot()).invoke(xs);
    if(var10000 != null) {
        if(var10000 != Boolean.FALSE) {
            var10000 = null;
            return var10000;
        }
    }

    // const__1 = cons
    IFn var2 = (IFn)const__1.getRawRoot();
    // const__2 = first
    Object var10001 =
       ((IFn)const__2.getRawRoot()).invoke(xs);
    // const__3 = every-other
    IFn var10002 = (IFn)const__3.getRawRoot();
    // const__4 = rest
    IFn var10003 = (IFn)const__4.getRawRoot();
    IFn var10004 = (IFn)const__4.getRawRoot();
    Object var10005 = xs;
    xs = null;
    var10000 =
      var2.invoke(var10001,
        var10002.invoke
        (
          var10003.invoke
          (
            var10004.invoke(var10005)
          )
        ));
    return var10000;
}
```

Figure 7.1: `EveryOther invoke` Java Code

which is in lines 3-10 in Figure 7.1. The recursive call is executed on line 25. For sequence inputs, the code prior to the recursive call and the code after the recursive call returns are two other segments. The recursive call is treated separately, similar to a fourth segment.

The state configuration changes to reason about segments of methods. In previous chapters, theorems were proved using states poised to invoke the method under verification. The first step of the machine in those theorems pushed a frame containing the method instructions onto the call stack. For the every-other verification, the strategy instead uses states poised to execute instructions at specific pc values. The pc values are *cutpoints* and are defined for the entry and exit of the method as well as each segment of code. The segments are combined by verifying that each cutpoint steps correctly to the next. The cutpoints are defined as a transition from a base frame configured at the first instruction in invoke. The predicate `eo-base-frame` recognizes the base frame.

```
1 (defun eo-base-frame (frame heap)
2   (and (|every_other|-p (top (locals frame)) heap)
3        (equal (pc frame) 0)
4        (equal (stack frame) nil)
5        (equal (cur-class frame) "clojure.lang.IFn")
6        (equal
7          (program frame)
8          (method-program *examples$every_other-invoke*))))
```

The base frame is at pc 0 with an empty operand stack. The frame's method contains the instructions to run `EveryOther`'s invoke method. The first local variable is an instance of `EveryOther`.

State configurations are constructed for each cutpoint from a base frame, a list of sequences, and a state with `EveryOther` and its dependencies loaded. The function `eo-start-state` configures a state at the method's entry. It pushes a base frame onto the call stack and modifies the locals to include the first sequence in `xs` as the

input to `every-other`.

```
(defun eo-start-state (frame xs s)
  (let*
    ((s1 (modify s :call-stack (push frame (call-stack
       s))))
     (xs-ref (if (endp xs)
                 (|PL:EMPTY|-get (heap s)
                                 (class-table s))
                 (car xs))))
    (modify
     s1
     :locals
     (list (car (locals frame)) xs-ref))))
```

The start state configuration can be defined from the method signature and knowledge of the JVM. The method's bytecode must be analyzed to define the end state configuration. The bytecode for `EveryOther`'s invoke method is shown in Figure 7.2. Even though the method written in Java has two return statements, the bytecode only has a single return instruction in line 39. The Java method returns null on empty inputs but the bytecode method implements that behavior with the instructions on lines 12 (ACONST_NULL) and 13 (GOTO 78). Those instructions load null onto the operand stack and jump to the return instruction at line 39. Therefore, the end state configuration is always at that final instruction, which is pc 104, regardless of the input. The locals may be modified as well. Initially, locals store the reference to the input sequence or, if the input evaluates to empty, the `EmptyList`. However, lines 33 and 34 overwrite non-empty sequences with a null pointer before exiting. The end state accounts for the change to locals, but the external behavior is unaffected. The function `eo-end-state`, shown in Figure 7.3, modifies a base frame to correspond to the entire behavior of the method and pushes the result onto an MC state `s`.

The correctness properties of the heap and the return value are the interesting portions of the end state. The ACL2 `every-other` function specifies the correct behavior of the method, so the heap and return value must be described in terms of

```
(defconst *examples$every_other-invoke*
  '("invoke" ((CLASS "java.lang.Object"))
    (GETSTATIC "every_other" "const__0" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (ALOAD_1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 1)
    (DUP)
    (IFNULL 13)
    (GETSTATIC "java.lang.Boolean" "FALSE" NIL)
    (IF_ACMPEQ 8)
    (ACONST_NULL)
    (GOTO 78)
    (POP)
    (GETSTATIC "every_other" "const__1" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (GETSTATIC "every_other" "const__2" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (ALOAD_1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 1)
    (GETSTATIC "every_other" "const__3" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (GETSTATIC "every_other" "const__4" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (GETSTATIC "every_other" "const__4" NIL)
    (INVOKEVIRTUAL "clojure.lang.Var" "getRawRoot" 0)
    (CHECKCAST "clojure.lang.IFn")
    (ALOAD_1)
    (ACONST_NULL)
    (ASTORE_1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 1)
    (INVOKEINTERFACE "clojure.lang.IFn" "invoke" 2)
    (ARETURN)))
```

Figure 7.2: `EveryOther invoke` Bytecode

```
(defun eo-end-state (frame xs s)
  (let*
    ((s1 (modify
          s
          :call-stack
          (push frame (call-stack s))))
     (result (if (endp xs)
                 nil
                 (every-other xs)))
     (result-ref (if (endp result)
                     (nullref)
                     (list 'REF (+ (len (heap s))
                                   (len result)
                                   -1))))
     (local-1 (if (endp result)
                  (|PL:EMPTY|-get (heap s)
                                  (class-table s))
                  (nullref))))
    (modify
     s1
     :pc 104
     :locals (list (car (locals frame)) local-1)
     :stack (push result-ref (stack frame))
     :heap (alloc-list result (heap s)))))
```

Figure 7.3: `eo-end-state`

that function. To do so, the end state calculates a result sequence using `every-other`. The heap is updated by allocating the result onto the original heap using `alloc-list`. The method returns a reference to the first sequence element in the resulting sequence. Since the heaps are constrained to sequential indexes without garbage collection, the end state calculates the reference index by adding the maximum index in the initial heap to the length of the result.

Our goal is to verify that a machine configured at the start state will run to the end state. Two additional cutpoints are necessary to sketch out the high-level proof: a recursive call cutpoint at the instruction that invokes the recursive call and a recursive end cutpoint at the instruction immediately after. The goal is verified using proof by induction. The inductive step is broken into three segments: a prelude, postlude, and internal segment. The prelude describes the behavior of the method from the beginning of the method until the recursive call cutpoint and the postlude describes the behavior from the recursive end cutpoint until the end of the method. As a roadmap for the remaining sections of the chapter, the cases are enumerated.

1. *Base*: For an empty sequence, the start state runs to the end state.
2. *Prelude*: For a non-empty sequence, the start state runs to the recursive call.
3. *Postlude*: For a non-empty sequence, the recursive end state runs to the end state.
4. *Internal Segment*: For a non-empty sequence, the recursive call cutpoint runs to the recursive end cutpoint.

7.2 Base Case

The `EveryOther` method, which operates on sequences, is being mapped to an ACL2 `every-other` function, which operates on lists. The base case for each type must be

verified, but it is accomplished by verifying the base case on an empty sequence list. A sequence list has two properties pertaining to this:

1. A sequence list does not contain references to `EmptyList` sequences
2. The `more` field of the last sequence in the list is the `EmptyList`

The start frame and end frame functions interpret an empty list as an `EmptyList` reference. Therefore, the base cases for the two types are aligned for a sequence list. The following theorem verifies the base case:

```
1 (defthm every-other-base-case
2   (implies
3    (and (|every_other|-loaded? (class-table s) (heap s))
4         (eo-base-frame frame (heap s))
5         (seq-listp xs (heap s) (class-table s))
6         (endp xs))
7    (-> (eo-start-state frame xs s)
8        (eo-end-state frame xs s))))
```

7.3 Prelude

The method prelude is the code prior to the recursive call. The prelude includes the logic that reacts to empty inputs, but that behavior is covered by the base case. The behavior for non-empty inputs begins on line 14 in Figure 7.2 and ends at the call to invoke `self` on line 37. In between those lines, the stack is modified multiple times. The net change in the stack is as follows:

1. The `Empty` reference is popped
2. A reference to `self` is pushed
3. The result of executing `First`'s invoke method is pushed
4. A reference to `Cons` is pushed

```
(defun prelude-end-state (frame xs s)
  (let*
    ((s1 (modify
           s
           :call-stack
           (push frame (call-stack s))))
     (xs-ref (car xs)))
    (modify
     s1
     :pc 94
     :locals
     (list (car (locals frame)) (nullref))
     (push
      (seq-more (seq-more xs-ref
                          (heap s)
                          (class-table s))
                (heap s)
                (class-table s))
      (push (|Var:root|-get
             (|eo:every_other|-get (heap s)
                                   (class-table s))
             (heap s))
            (push (seq-first xs-ref (heap s))
                  (push (|Var:root|-get
                         (|eo:cons|-get (heap s)
                                        (class-table s))
                         (heap s))
                        (stack frame)))))))))
```

Figure 7.4: `prelude-end-state`

5. The result of executing `Rest`'s invoke method twice is pushed

The second element of the locals is set to the null reference. The `prelude-end-state` function modifies a base frame to construct a state that reflects the prelude behavior. The function is shown in Figure 7.4. The following theorem verifies that the prelude is correct:

```
(defthm prelude-is-correct
  (implies
   (and (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (not (endp xs)))
   (-> (eo-start-state frame xs s)
       (prelude-end-state frame xs s)))
```

It states that if `EveryOther` is invoked on a non-empty sequence, the start state runs to the recursive call.

7.4 Postlude

The postlude must define a *where* and a *when*. The where is easy to identify from bytecode analysis; it starts at the instruction immediately following the recursive call. Defining when the postlude starts requires making a choice between two points in the process of executing a method: invocation or return. The invocation updates the stack and pc on the calling frame before pushing the new frame onto the call stack. If the called method returns a value, when the new frame reaches an exit, the top element on its operand stack is pushed onto the calling frame's operand stack.

We choose to define the postlude as the state immediately after the recursive call is invoked because it simplifies the transition from the prelude. The function `postlude-start-state` constructs a postlude state from a base frame, list of sequences, and a state.

```
(defun postlude-start-state (frame xs s)
  (let*
    ((s1 (modify
           s
           :call-stack
           (push frame (call-stack s))))
     (xs-ref (car xs)))
    (modify
     s1
     :pc 99
     :locals (list (car (locals frame)) (nullref))
     :stack
     (push (seq-first xs-ref (heap s))
           (push (|Var:root|-get
                  (|eo:cons|-get (heap s)
                                 (class-table s))
                  (heap s))
                 (stack frame))))))
```

The call stack of the postlude state does not contain the newly invoked method. Instead, a recursive start state is constructed using the postlude state.

```
(defun recursive-start (frame xs s)
  (eo-start-state frame
                  (cddr xs)
                  (postlude-start-state frame xs s)))
```

The base frame requires that the top value on the locals is a reference to an instance of the `EveryOther` class. In the case of the recursive call, the reference is specifically `self`. The theorem `prelude-to-postlude`, shown in Figure 7.5, explicitly declares the reference is `self` in lines 7 through 11.

The prelude ends with the result of invoking `First` on the top of the stack. When the recursive call exits, a reference to a sequence is pushed onto the stack. The postlude concludes execution of the method by invoking `Cons` on those two references. The theorem `every-other-postlude-ends` in Figure 7.5 verifies that if the recursive call results in an end state, the calling frame will run to an end state.

```
(defthm prelude-to-postlude
  (implies
   (and (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (not (endp xs))
        (equal (car (locals frame))
               (|Var:root|-get
                (|eo:every_other|-get (heap s)
                                      (class-table s))
                (heap s))))
   (-> (prelude-end-state frame xs s)
       (recursive-start frame xs s))))


(defthm every-other-postlude-ends
  (implies
   (and (alistp (heap s))
        (all-smallp (heap s) (len (heap s)))
        (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (not (endp xs)))
   (-> (eo-end-state frame
                     (cddr xs)
                     (postlude-start-state frame xs s))
       (eo-end-state frame xs s))))
```

Figure 7.5: Postlude Starts and Ends

7.5 Internal Segment

Our goal is to prove that a Clojure every-other function maps to an ACL2 every-other function. The internal segment is an instance of the goal that is specific to the method being invoked on `self`. A proof of correctness for the internal segment resolves the inductive step in the goal theorem, but it requires a special induction scheme.

The conclusion of the goal theorem affirms that `(eo-start-state frame xs s)` runs to `(eo-end-state frame xs s)`. The induction is on `xs` and sequences have been modeled in ACL2 to facilitate inducting on `xs` as a list. Frame is always a base frame that is configured for a specific cutpoint. It is neither referenced nor modified by the machine, so it does not change for a recursive call. The state `s`, however, is changed for each recursive call because a new frame is pushed onto the call stack. We need to define an induction scheme in ACL2 that acknowledges the change to the state for the recursive call.

ACL2 verifies that a function terminates before accepting it into the logic. ACL2 associates an induction rule with every recursive function based on the induction used to admit it. Theorems in ACL2 can be configured to apply a specific induction rule. The function `eo-ind` is defined to introduce an induction rule that can be used to verify the correctness of the internal segment.

```
1 (defun eo-ind (xs frame s)
2   (if (endp xs)
3     (list xs frame s)
4     (list
5       (eo-ind (cddr xs)
6               frame
7               (postlude-start-state frame
8                                     xs
9                                     s)))))
```

The function reduces `xs` by two elements to match `every-other`'s behavior and pushes the postlude state onto the call stack. Other than those two properties, the

behavior of the function is unimportant.

The induction scheme is recommended to ACL2 as a hint, which can be seen in the theorem `every-other-internal-segment`.

```
(defthm every-other-internal-segment
  (implies
   (and (alistp (heap s))
        (all-smallp (heap s) (len (heap s)))
        (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (equal (car (locals frame))
               (|Var:root|-get
                (|eo:every_other|-get (heap s)
                                      (class-table s))
                (heap s))))
   (-> (eo-start-state frame xs s)
       (eo-end-state frame xs s)))
  :hints
  (("Goal" :induct (eo-ind xs frame s))))
```

The proof of correctness of the internal segment is used as a lemma to resolve the inductive step in the final proof.

7.6 Proof of Correctness

The previous sections verified the base case, prelude, postlude, and the internal segment. In this section, those results are combined into a proof of correctness that Clojure `every-other` maps to ACL2 `every-other`. The verification of the internal segment is limited to invocations called on `self`. The process for removing the `self` constraint from the theorem is straight-forward and aligns with the original strategy.

In this section, an internal frame is one that is operating on `self`. The strategy is to prove the following properties:

1. For a non-empty sequence, the start state runs to a state poised to run an internal frame in the start state.

2. A state that is exiting an internal frame runs to the end state.

3. An internal frame in the start state runs to an internal frame in the end state.

An internal frame is created using

```
(defun internal-frame (frame heap class-table)
  (make-frame
   (pc frame)
   (list
    (|Var:root|-get
     (|eo:every_other|-get heap class-table)
     heap))
   (stack frame)
   (program frame)
   (cur-class frame)))
```

Before verifying the three properties, it must be demonstrated that an internal frame is a base frame. The frame constructed by `internal-frame` is identical to the input frame except the first element in locals is set to `self`. The `|every_other|-loaded?` predicate defines `self` as an `EveryOther` object. If `every_other` is loaded and the internal frame is constructed from a base frame, then the internal frame is a base frame.

```
(defthm internal-frame-is-base-frame
  (implies
   (and (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s)))
   (eo-base-frame (internal-frame frame
                                  (heap s)
                                  (class-table s))
                  (heap s))))
```

Property 1 affirms that a start state runs to the recursive state with an internal frame on the top of the call stack. The theorem `prelude-is-correct` in Section 7.3 verifies a start state runs to the end of the prelude, so we only need to verify that stepping past the prelude's end state will push an internal frame onto the call stack. The function `eo-internal-start` is similar to `eo-recursive-start` except it pushes

an internal frame onto the call stack of the start state. The following theorem verifies property 1.

```
(defthm ext-prelude->internal-start
  (implies
   (and (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (not (endp xs)))
   (-> (prelude-end-state frame xs s)
       (eo-internal-start frame xs s))))
```

Property 2 affirms that an internal frame in the end state runs to the end state of the calling frame.

```
(defthm internal-end->end
  (implies
   (and (alistp (heap s))
        (all-smallp (heap s) (len (heap s)))
        (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s))
        (not (endp xs)))
   (->
     (eo-end-state
       (internal-frame
         frame
         (heap (postlude-start-state frame xs s))
         (class-table (postlude-start-state frame xs s)))
       (cddr xs)
       (postlude-start-state frame xs s))
     (eo-end-state frame xs s))))
```

Property 3 affirms that an internal frame in the start state runs to the end state. It is verified below by the theorem `internal-start->internal-end`. The theorem is similar to the theorem `every-other-internal-segment`, but it removes the reference to `self` from the hypothesis. The theorem is still specific to instances invoked on `self`, but it is constrained to those instances by the function `internal-frame` in the conclusion.

```
(defthm internal-start->internal-end
  (implies
   (and (alistp (heap s))
        (all-smallp (heap s) (len (heap s)))
        (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s)))
   (-> (eo-start-state (internal-frame frame
                                       (heap s)
                                       (class-table s))
                       xs
                       s)
       (eo-end-state (internal-frame frame
                                     (heap s)
                                     (class-table s))
                     xs
                     s)))
```

The final proof of correctness verifies that the Clojure `every-other` maps to the ACL2 `every-other` when invoked on an arbitrary instance of `EveryOther`.

```
(defthm every-other-is-every-other
  (implies
   (and (alistp (heap s))
        (all-smallp (heap s) (len (heap s)))
        (|every_other|-loaded? (class-table s) (heap s))
        (eo-base-frame frame (heap s))
        (seq-listp xs (heap s) (class-table s)))
   (-> (eo-start-state frame xs s)
       (eo-end-state frame xs s))))
```

The effort to prove `every-other-is-every-other` required the analysis of 17 classes, 37 methods, and bytecode totaling a length of 346 JVM instructions. The final analysis required over 500 theorems to be admitted to ACL2. The ACL2 source code, including the theorems, can be found online (Ralston, 2016).

Chapter 8

Conclusion

Correctness is a pursuable quality using current state-of-the-art formal methods. However, there are still considerable difficulties in applying formal methods in industrial development. Functional languages are chosen industrially for many reasons, but verification is not yet seen as a motivating factor. Therefore, there is a need to adapt existing verification tools to the languages being used in industry. Towards that goal, this dissertation presents research that maps Clojure code into the ACL2 theorem prover. In this chapter, the contributions are summarized and limitations are discussed. The contributions are presented in order of significance, with the most significant presented first. The chapter concludes with a discussion of future work.

8.1 Verification of a Recursive Sequence Clojure Function

Recursion is an important pattern in functional programming and verification. Results presented in this dissertation led to a recursive sequence function being mapped to a recursive list function. The mapping required the verification of Clojure core functions as well as abstractions for sequences and the heap. The core functions `empty?`, `cons`, `first`, `rest`, `seq`, and `not` were verified and the associated theorems were used as rewrite rules for the recursive proof.

The concept of a sequence list was introduced to relate sequences to lists. A sequence list enforces a well-ordered property on sequence references. Recursive sequence functions can be reasoned about using induction by limiting the input to sequence lists. The verified recursive function is an example of induction being used

on a sequence list. The recursive proof also required reasoning about the allocation of a sequence onto the heap. The `alloc-list` function abstracts the logic of allocating a sequence if it is constructed in a function based on our recursive template. Heap invariants from Moore (2003) were verified about `alloc-list` and a new set of invariants was introduced to prove persistence of the existing objects on the heap prior to the allocation.

A limitation of our results is the assumption that the compiled form of a Clojure function correctly loads its dependencies. A compiled Clojure function references its function dependencies as static fields that are initialized during class loading. A verification of function loading could strengthen the results. It was not attempted here because the implementation of function loading relies on the Java `Unsafe` class. `Unsafe` is a wrapper for native calls that directly modify memory. The behavior is undefined in the JVM specification and the documentation of `Unsafe` is minimal and its use is discouraged. MC could reason about some restricted use of `Unsafe` but each class declaration would need to define how instances of the class are stored in memory. A more complete model of `Unsafe` would require a more sophisticated JVM model than MC.

The results are also limited by the assumption that the compiled form of the analyzed Clojure code is representative of the form generated by compiling similar Clojure code. A verified proof of the Clojure compilation process would significantly improve the value of this work. To accomplish this goal, a formal specification would need to be created from code analysis, and it would need to be verified that the Clojure compiler implements the specification.

8.2 Big-Step Style Verification

MC uses small-step semantics but we reasoned about programs in it using a big-step style. The `->` macro equated states that eventually converge if both are executed indefinitely. It is based on an implementation in Manolios & Moore (2003) for an earlier ACL2 model of the JVM, but it is modified to improve the reliability that theorems written using it are applied as rewrite rules. `->` is used effectively to prove theorems about methods using MC's natural small-step semantics, but apply the theorems as lemmas in a big-step style in the verification of larger programs.

In our experience, the use of `->` does come at a cost. Rewrite rules were defined using `->` for each JVM instruction, but they significantly lengthened ACL2's proof time because the rules were so frequently attempted. Therefore, the rules were disabled by default. The rules for individual instructions were enabled after determining, through analysis of the bytecode, their relevance to a specific proof. A new system, Stateman (Moore, 2015), may eventually resolve this issue. Stateman is an ACL2 system that incorporates generic proof techniques for verification using models like MC. Stateman has produced promising results, but it does not currently support state equality so it could not be applied to our big-step analysis.

8.3 JVM Model for Compiled Clojure

Our research investigated the feasibility of mapping Clojure functions to corresponding ACL2 functions. We introduced MC, a new model of the JVM in ACL2, that was developed from the existing M5 model to specifically address our research question. Clojure relies heavily on interfaces, which are not supported in M5, so support was added in MC. It required modifying the class declaration, implementing INVOKEINTERFACE and CHECKCAST, and extending support of the INSTANCEOF instruc-

tion. Clojure is also a large program and MC does not scale well to programs with many classes. Based on experiences with a similar model to MC, Liu (2006) suggests a robust representation of Java classes. We followed that advice by developing an abstraction layer for each class declaration that hides information from ACL2's theorem prover unless it is needed. The abstraction layer improves the ability to prove theorems about bytecode in MC.

While MC is a sophisticated model of the JVM, it does leave out many features. Many of the excluded features only require adding the operational semantics of their instructions. MC implements 138 of the 204[1] bytecode instructions in the JVM 7 Specification (Lindholm et al., 2013). The instructions for floats and doubles are not implemented, but account for 57 of the 66 unimplemented instructions. ACL2 has been used to verify models of IEEE compliant floating point models (Russinoff, 1998; Moore et al., 1998) that can be used as a basis for adding floats and doubles to MC. The remaining nine unimplemented instructions are ATHROW, INVOKEDYNAMIC, LOOKUPSWITCH, MONITORENTER, MONITOREXIT, TABLESWITCH, and WIDE. M6 implements all of those instructions except INVOKEDYNAMIC and WIDE (Liu, 2006), and could be used as a basis for support. The WIDE instruction extends the size of the reference into locals from 8-bit to 16-bit, so it is only necessary for methods that require more than 256 local variables. Functional code is usually written in small, concise functions that compile to small methods in Java, so the need is minimal. Finally, Clojure does not currently support INVOKEDYNAMIC.

Some features are unsupported because there is not a clear corresponding concept to relate it to in ACL2. MC does not support accessibility flags, overloaded methods, and exceptions but support can be added based on the implementations

[1]Count does not include instructions BREAKPOINT, IMPDEP1, IMPDEP2 that are available for use by debuggers.

in M6 (Liu, 2006). Adding overloaded methods is necessary to reason about Clojure functions with optional parameters, but ACL2 does not support optional parameters, so the one-to-one correspondence of the code would be weakened. However, it is still possible, but an ACL2 function would need to be defined for each possible arity of the Clojure function. Exception handling and accessibility modifiers, on the other hand, are imperative programming features that are not common in functional programming, so their exclusion is based on the language semantics being researched. M5 and M6 both support multi-threaded programs, but MC removes it based on the experience reported in Liu (2006) that verifying sequential programs takes an unnecessary, and significant, performance hit by including it. ACL2 does not support multi-threaded code, so it lacks a corresponding syntax to relate multi-threaded Clojure. However, future support is worthwhile for other research questions because Clojure implements a *software transactional memory* (STM) system that shares immutable objects between threads.

MC also does not support garbage collection but its absence is not considered a significant limitation because when garbage collection is run is generally nondeterministic. However, adding support for it to MC could be worthwhile for analysis of Clojure. Studies have shown that, compared to Java, Clojure allocates more objects, that the objects are smaller, and that their lifespans are shorter (Li et al., 2013; Sarimbekov et al., 2013). The JVM does not specify the behavior of garbage collection, but systems that allocate a lot of objects with short lifespans will generally trigger the garbage collector frequently.

8.4 Verification of Bignum Addition

Bignum arithmetic has been represented in little-endian format in prior work. Java's `BigInteger` class represents bignums in big-endian format. The change in represen-

tation complicates reasoning about bignum values because the interpretation of an individual element changes based on the length of the array. A big-endian bignum addition function was verified in this dissertation. It is an early result in the process of verifying `BigInteger` operations.

The verified function is not yet attached to the `BigInteger` bytecode. The bytecode implements the behavior with multiple loops and a native method call that extends the length of the result when necessary. Similar to the `every-other` verification, the loops are treated as segments. The exit of a loop must run to the entry of the next. Each loop requires an invariant to be defined that demonstrates the correct value is accumulated during each iteration. We are in the process of defining those invariants.

The `BigInteger` class was considered in this dissertation because Clojure supports bignums using it. Bignums are also used extensively in cryptography (Denis & Rose, 2006). Since Java is a commonly used language and it represents bignums uniquely, the verification of `BigInteger` is a worthwhile goal by itself.

8.5 Summary

This research pursues a verified Clojure `core` library in ACL2. With a verified `core` library, new Clojure functions that are defined can be reasoned about directly in ACL2 with a significant degree of reliability. The degree of reliability is based on the properties verified about the Clojure code and the fidelity of MC with respect to the desired properties. In this dissertation, the properties that are verified state that the behavior of the Clojure code is equivalent to corresponding ACL2 code. Since ACL2 is formal Common Lisp, it is a reliable specification of the proper behavior of the Clojure functions. MC is also a trustworthy model when considering the behavior of Clojure functions that are capable of being transcribed to ACL2. Most of the missing

features can be implemented based on existing examples in the literature. However, many of the missing features do have not a corresponding implementation in ACL2 so their exclusion does not detract from the research question, which focuses on the verification of Clojure functions with corresponding implementations in ACL2.

The research question, ultimately, is pursued towards the goal of simplifying the adoption of formal methods in industrial software development. Formal methods prioritize the specification phase, which helps detect defects earlier (Clarke & Wing, 1996), but also means the overhead of using formal methods is felt early in the project while the benefit is not realized until the maintenance phase. A benefit of using Clojure with ACL2 is that the similarity in the code makes it easier to apply verification at a later stage of development, even after some of the system has been developed. This shifts the overhead of formal methods closer to the benefit and increases the chances it will be seen as worthwhile. Eventually, this can lead to lower maintenance costs for companies and higher quality software for users.

Bibliography

Abran, A., & Nguyenkim, H. (1991). Analysis of maintenance work categories through measurement. In *Proceedings of the Conference on Software Maintenance* (pp. 104–113). doi:10.1109/ICSM.1991.160315.

Affeldt, R. (2013). On construction of a library of formally verified low-level arithmetic functions. *Innovations in Systems and Software Engineering*, *9*, 59–77. doi:10.1007/s11334-013-0195-x.

Basili, V. R., Briand, L. C., Condon, S. E., Kim, Y., Melo, W. L., & Valett, J. D. (1996). Understanding and predicting the process of software maintenance release. In *Proceedings of the 18th International Conference on Software Engineering* ICSE '96 (pp. 464–474). Washington, DC, USA: IEEE Computer Society.

Berghofer, S. (2012). Verification of Dependable Software using SPARK and Isabelle. In J. Brauer, M. Roveri, & H. Tews (Eds.), *6th International Workshop on Systems Software Verification* (pp. 15–31). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik volume 24 of *OpenAccess Series in Informatics (OASIcs)*. doi:10.4230/OASIcs.SSV.2011.15.

Boehm, B., & Basili, V. R. (2001). Software defect reduction top 10 list. *Computer*, *34*, 135–137. doi:10.1109/2.962984.

Boehm, B. W. (1987). Improving software productivity. *Computer*, *20*, 43–57. doi:10.1109/MC.1987.1663694.

Börger, E., Fruja, N. G., Gervasi, V., & Stärk, R. F. (2005). A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, *336*, 235–284. doi:10.1016/j.tcs.2004.11.008.

Boulton, R. J., Gordon, A., Gordon, M. J. C., Harrison, J., Herbert, J., & Tassel, J. V. (1992). Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience* (pp. 129–156). North-Holland Publishing Co.

Boyer, R. S., & Yu, Y. (1996). Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, *43*, 166–192. doi:10.1145/227595.227603.

Ciobâcă, Ş. (2013). From small-step semantics to big-step semantics, automatically. In *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings* (pp. 347–361). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-38613-8_24.

Clarke, E. M., & Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys*, *28*, 626–643. doi:10.1145/242223.242257.

Denis, T. S., & Rose, G. (2006). *BigNum math — implementing cryptographic multiple precision arithmetic*. Syngress.

Dillinger, P. C., Manolios, P., Vroon, D., & Moore, J. S. (2007). ACL2s: "The ACL2 Sedan". In *Companion to the Proceedings of the 29th International Conference on Software Engineering* ICSE COMPANION '07 (pp. 59–60). Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICSECOMPANION.2007.14.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, *2*, 17–23. doi:10.1109/6294.846201.

Fischer, S. (2007). *Formal Verification of a Big Integer Library Including Division*. (Master's thesis). Saarland University.

Gordon, M. J. C., & Melham, T. F. (Eds.) (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. New York, NY, USA: Cambridge University Press.

Graves, T. L., & Mockus, A. (1998). Inferring change effort from configuration management databases. In *5th IEEE International Software Metrics Symposium , March 20-21, 1998, Bethesda, Maryland, USA* METRICS 1998 (p. 267). doi:10.1109/METRIC.1998.731253.

Hardin, D. S. (2015). Reasoning about LLVM code using Codewalker. In *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.* (pp. 79–92). doi:10.4204/EPTCS.192.7.

Havelund, K., Lowry, M., & Penix, J. (2001). Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, *27*, 749–765. doi:10.1109/32.940728.

Hecht, H., Hecht, M., & Wallace, D. R. (1997). Toward more effective testing for high-assurance systems. In *2nd High-Assurance Systems Engineering Workshop* HASE '97 (pp. 176–181). Washington, DC, USA: IEEE Computer Society. doi:10.1109/HASE.1997.648060.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*, 576–580. doi:10.1145/363235.363259.

Hoare, C. A. R. (1996). How did software get so reliable without proof? In M.-C. Gaudel, & J. Woodcock (Eds.), *FME'96: Industrial Benefit and Advances in Formal Methods* book section 1. (pp. 1–17). Springer Berlin Heidelberg volume 1051 of *Lecture Notes in Computer Science*. doi:10.1007/3-540-60973-3_77.

Holzmann, G. J. (2001). Economics of software verification. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* PASTE '01 (pp. 80–89). New York, NY, USA: ACM. doi:10.1145/379605.379681.

Hunt, W. A., Jr. (1994). *FM8501: A Verified Microprocessor* volume 795 of *Lecture Notes in Computer Science*. Springer. doi:10.1007/3-540-57960-5.

Kaufmann, M., Moore, J. S., & Manolios, P. (2000). *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., & Winwood, S. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* SOSP '09 (pp. 207–220). New York, NY, USA: ACM. doi:10.1145/1629575.1629596.

Krein, J. L., MacLean, A. C., Knutson, C. D., Delorey, D. P., & Eggett, D. L. (2010). Impact of programming language fragmentation on developer productivity: A sourceforge empirical study. *International Journal of Open Source Software Processes*, *2*, 41–61. doi:10.4018/jossp.2010040104.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, *52*, 107–115. doi:10.1145/1538788.1538814.

Li, W. H., White, D. R., & Singer, J. (2013). JVM-hosted languages: They talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* PPPJ '13 (pp. 101–112). New York, NY, USA: ACM. doi:10.1145/2500828.2500838.

Lindholm, T., Yellin, F., Bracha, G., & Buckley, A. (2013). *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional.

Liu, H. (2006). *Formal Specification and Verification of a JVM and Its Bytecode Verifier*. (Doctoral dissertation). University of Texas.

Liu, H., & Moore, J. S. (2003). Executable JVM model for analytical reasoning: a study. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators* IVME '03 (pp. 15–23). New York, NY, USA: ACM. doi:10.1145/858570.858572.

Liu, H., & Moore, J. S. (2004). Java program verification via a JVM deep embedding in ACL2. In K. Slind, A. Bunker, & G. Gopalakrishnan (Eds.), *Theorem Proving in Higher Order Logics* book section 14. (pp. 184–200). Springer Berlin Heidelberg volume 3223 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-540-30142-4_14.

Lloyd, J. W. (1994). Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94* (pp. 18–30).

Manolios, P., & Moore, J. (2003). Partial functions in ACL2. *Journal of Automated Reasoning*, *31*, 107–127. doi:10.1023/B:JARS.0000009505.07087.34.

Maxwell, K. D., Wassenhove, L. V., & Dutta, S. (1996). Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering*, *22*, 706–718. doi:10.1109/32.544349.

McKee, J. R. (1984). Maintenance as a function of design. In *Proceedings of the July 9-12, 1984, National Computer Conference and Exposition* AFIPS '84 (pp. 187–193). New York, NY, USA: ACM. doi:10.1145/1499310.1499334.

Miller, A. (2016). Clojure Companies. URL: http://clojure.org/community/companies.

Moore, J. S. (1989). A mechanically verified language implementation. *Journal of Automated Reasoning*, *5*, 461–492. doi:10.1007/BF00243133.

Moore, J. S. (1999). Proving theorems about Java-like byte code. In E.-R. Olderog, & B. Steffen (Eds.), *Correct System Design* book section 7. (pp. 139–162). Springer Berlin Heidelberg volume 1710 of *Lecture Notes in Computer Science*. doi:10.1007/3-540-48092-7_7.

Moore, J. S. (2002). A grand challenge proposal for formal methods: A verified stack. In *Formal Methods at the Crossroads. From Panacea to Foundational Support* (pp. 161–172). doi:10.1007/978-3-540-40007-3_11.

Moore, J. S. (2003). Proving theorems about Java and the JVM with ACL2. In *Models, Algebras and Logic of Engineering Software* (pp. 227–290). IOS Press.

Moore, J. S. (2006). Inductive assertions and operational semantics. *International Journal on Software Tools for Technology Transfer*, *8*, 359–371. doi:10.1007/s10009-005-0180-2.

Moore, J. S. (2015). Stateman: Using metafunctions to manage large terms representing machine states. In *Proceedings Thirteenth International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, 1-2 October 2015.* (pp. 93–109). doi:10.4204/EPTCS.192.8.

Moore, J. S., Lynch, T. W., & Kaufmann, M. (1998). A mechanically checked proof of the AMD5$_{\mathrm{k}}$86$^{\mathrm{tm}}$ floating point division program. *IEEE Transactions on Computers*, *47*, 913–926. doi:10.1109/12.713311.

Moore, J. S., & Porter, G. (2001). M5 ACL2 arithmetic book (source code). https://github.com/acl2/acl2/tree/master/books/models/jvm/m5.

Moore, J. S., & Porter, G. (2002). The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, *24*, 193–216. doi:10.1145/514188.514189.

Myreen, M., & Curello, G. (2013). Proof pearl: A verified bignum implementation in x86-64 machine code. In G. Gonthier, & M. Norrish (Eds.), *Certified Programs and Proofs* book section 5. (pp. 66–81). Springer International Publishing volume 8307 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-319-03545-1_5.

Newcombe, C. (2014). Why Amazon chose TLA+. In Y. Ait Ameur, & K.-D. Schewe (Eds.), *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings* (pp. 25–39). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-662-43652-3_3.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., & Deardeuff, M. (2015). How Amazon web services uses formal methods. *Communications of the ACM*, *58*, 66–73. doi:10.1145/2699417.

Nguyen, V., Boehm, B., & Danphitsanuphan, P. (2009). Assessing and estimating corrective, enhancive, and reductive maintenance tasks: A controlled experiment. In *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific* (pp. 381–388). doi:10.1109/APSEC.2009.49.

Nguyen, V., Huang, L., & Boehm, B. (2011). An analysis of trends in productivity and cost drivers over years. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering* Promise '11 (pp. 3:1–3:10). New York, NY, USA: ACM. doi:10.1145/2020390.2020393.

Nipkow, T., & Paulson, L. C. (1992). Isabelle-91. In D. Kapur (Ed.), *Proceedings of the 11th International Conference on Automated Deduction* (pp. 673–676). Springer Berlin Heidelberg volume 607 of *Lecture Notes in Computer Science*. doi:10.1007/3-540-55602-8_201.

Owre, S., Rushby, J. M., & Shankar, N. (1992). PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction* CADE-11 (pp. 748–752). London, UK, UK: Springer-Verlag.

Ralston, R. (2016). Translating Clojure to ACL2 source code. https://github.com/rlralston/clojure-acl2.

Russinoff, D. M. (1998). A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, *1*, 148–200. doi:10.1112/S1461157000000176.

Russinoff, D. M. (2005). A formal theory of register-transfer logic and computer arithmetic. URL: http://www.russinoff.com/libman/index.html [Online; accessed 22-May-2015].

Sarimbekov, A., Podzimek, A., Bulej, L., Zheng, Y., Ricci, N., & Binder, W. (2013). Characteristics of dynamic JVM languages. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* VMIL '13 (pp. 11–20). New York, NY, USA: ACM. doi:10.1145/2542142.2542144.

Schumann, J. M. (2001). *Automated Theorem Proving in Software Engineering*. Berlin: Springer-Verlag Berlin Heidelberg.

Scott, D., & Strachey, C. (1971). *Toward a Mathematical Semantics for Computer Languages*. Programming Research Group Technical Monograph PRG-6 Oxford University Computing Lab.

Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., & Zelkowitz, M. (2002). What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on* (pp. 249–258). doi:10.1109/METRIC.2002.1011343.

Strecker, M. (2002). Formal verification of a Java compiler in Isabelle. In A. Voronkov (Ed.), *Proceedings of the 18th International Conference on Automated Deduction* CADE-18 (pp. 63–77). Springer-Verlag. doi:10.1007/3-540-45620-1_5.

Swords, S., & Davis, J. (2011). Bit-Blasting ACL2 Theorems. In D. Hardin, & J. Schmaltz (Eds.), *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications* (pp. 84–102). Open Publishing Association volume 70 of *Electronic Proceedings in Theoretical Computer Science*. doi:10.4204/EPTCS.70.7.

Théry, L., Letouzey, P., & Gonthier, G. (2006). Coq. In F. Wiedijk (Ed.), *The Seventeen Provers of the World* book section 6. (pp. 28–35). Springer Berlin Heidelberg volume 3600 of *Lecture Notes in Computer Science*. doi:10.1007/11542384_6.

Tuch, H. (2008). *Formal Memory Models for Verifying C Systems Code*. (Doctoral dissertation). The University of New South Wales.

Wilding, M., Greve, D., & Hardin, D. (2001). Efficient simulation of formal processor models. *Formal Methods in System Design*, *18*, 233–248. doi:10.1023/A:1011217102270.

Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press.