CODE-CARRYING THEORY

By

Aytekin Vargun

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major Subject: Computer Science

Approved by the
Examining Committee:

David R. Musser, Thesis Adviser

Selmer Bringsjord, Member

Mukkai Krishnamoorthy, Member

Paliath Narendran, Member

Carlos Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2006
(For Graduation December 2006)
CODE-CARRYING THEORY

By

Aytekin Vargun

An Abstract of a Thesis Submitted to the Graduate Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major Subject: Computer Science

The original of the complete thesis is on file in the Rensselaer Polytechnic Institute Library

Examinining Committee:

David R. Musser, Thesis Adviser
Selmer Bringsjord, Member
Mukkai Krishnamoorthy, Member
Paliath Narendran, Member
Carlos Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2006
(For Graduation December 2006)
CONTENTS

LIST OF FIGURES ................................................................. vi

ACKNOWLEDGMENTS ............................................................ vii

ABSTRACT ............................................................................. ix

1. Introduction ...................................................................... 1
   1.1 Contributions ............................................................... 7
   1.2 Outline of the thesis ...................................................... 9

2. Related Work .................................................................... 11
   2.1 Proof-Carrying Code (PCC) ........................................... 11
      2.1.1 Type-specialized proof-carrying code ....................... 11
         2.1.1.1 Creating the native code with annotations .......... 11
         2.1.1.2 Generating the verification condition ............... 12
         2.1.1.3 Finding a proof of the verification condition ....... 12
         2.1.1.4 Verifying the proof ....................................... 13
      2.1.2 Foundational proof-carrying code ........................... 13
         2.1.2.1 Defining and proving safety ............................ 14
      2.1.3 Language-level differences between PCC and CCT .... 15
   2.2 Machine-checkable and human-readable proofs ................. 17

3. Code-Carrying Theory (CCT) ............................................. 19
   3.1 Code-Carrying Theory steps ......................................... 19
      3.1.1 Step 1: Defining Requirements (Code Consumer) ....... 19
      3.1.2 Step 2: Defining a New Function (Code Producer) .... 21
      3.1.3 Step 3: Proving Termination (Code Producer) .......... 21
      3.1.4 Step 4: Proving the Additional Consistency Condition (Code Producer) ................................ 29
      3.1.5 Step 5: Proving Correctness (Code Producer) .......... 31
      3.1.6 Step 6: Termination Checking (Code Consumer) ....... 32
      3.1.7 Step 7: Additional Consistency Checking (Code Consumer) ........................................ 32
      3.1.8 Step 8: Checking Correctness (Code Consumer) ....... 32
      3.1.9 Step 9: Code Extraction (Code Consumer) .............. 33
   3.2 Ensuring correctness .................................................... 35
<table>
<thead>
<tr>
<th></th>
<th>Case Study: CCT Steps for a Nongeneric Memory-Observing Function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>Case Study: CCT Steps for a Nongeneric Memory-Updating Function</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td>5.1 Nongeneric definition of memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.2 Memory-range theory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.3 Memory-updating operations and nongeneric reverse-range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.4 Note on safety properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.5 Dealing with memory updates</td>
</tr>
<tr>
<td></td>
<td>6. Generic Proof Writing</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>6.1 Case study: CCT steps for a generic memory-observing function</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.1 Definition of axioms with ATHENA functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.2 Organizing theorems and proofs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.3 Definitions of efficient generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.4 Proof methods in CCT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.5 Correctness proof of generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.6 Instantiating generic definitions and proofs of sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.6.1 Addition of naturals in a list via generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.6.2 Multiplication of naturals in a list via generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.6.3 Joining list elements via generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.7 Termination and consistency of generic sum-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.7.1 Proofs for sum-list-compute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.1.7.2 Proofs of sum-list</td>
</tr>
<tr>
<td></td>
<td>6.2 Case study: CCT steps for a generic memory-updating function</td>
<td>90</td>
</tr>
<tr>
<td>7.</td>
<td>A Larger Example of the CCT Approach</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>7.1 A generic rotate function</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.1 Specification file: rotate-specification.ath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.2 Implementation file: rotate-implementation.ath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.3 Unit test file: rotate-implementation_unittest.ath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.4 Generated Oz code: rotate.oz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.1.5 Unit test file for generated code: rotate_unittest.oz</td>
</tr>
<tr>
<td></td>
<td>7.2 A generic find-if function</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.1 Specification file: find-if-specification.ath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.2 Implementation file: find-if-implementation.ath</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.2.3 Unit test file: find-if-implementation_unittest.ath</td>
</tr>
</tbody>
</table>
7.2.4 Generated Oz code: find-if.oz ........................................ 139
7.2.5 Unit test file for generated code: find-if.unittest.oz .......... 140
7.3 Testing the PartialPartition function ............................... 142

8. Conclusions ................................................................. 144
9. Future Work ............................................................... 146

LITERATURE CITED ............................................................. 148
A. Athena ................................................................. 152
  A.1 Primitive deduction methods ....................................... 153
  A.2 Some example proofs ................................................. 156
  A.3 Inductive definitions of new types .................................. 161
  A.4 Term rewriting methods ............................................. 163
  A.5 Proof by induction ................................................... 164
     A.5.1 Basis case proof ................................................ 166
     A.5.2 Induction step proof ............................................ 166
B. Natural Numbers and Lists in Athena .............................. 168
C. Range Induction ......................................................... 173
  C.1 A proof by range induction ......................................... 178
D. Formalization of Memory .............................................. 183
  D.1 Memory theory ....................................................... 183
     D.1.1 Definition of Memory Concept in CCT ....................... 183
     D.1.2 CCT theorems about memory and some selected proofs .... 184
  D.2 Memory-range Theory ............................................... 191
     D.2.1 Definition of memory range concept in CCT ............... 191
     D.2.2 CCT theorems about memory range and some selected proofs 192
E. Formalization of Iterators and Valid Range ....................... 200
  E.1 Iterator theory: formalization of iterators in Athena ......... 200
     E.1.1 Definition of iterator theory in CCT ....................... 200
     E.1.2 CCT theorems about iterators and some selected proofs ... 202
  E.2 Range theory: formalization of valid range property in Athena 213
     E.2.1 Definition of range theory in CCT ......................... 213
     E.2.2 CCT theorems about valid ranges and some selected proofs 215
F. Proof of sum-list-compute-relation .................................. 235
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Proof-Carrying Code, Successfully Checked and Delivered</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Proof Carrying Code, Tampered-With But Checked and Prevented from Acceptance</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Code-Carrying Theory, Successfully Checked and Delivered (FDA stands for Function Defining Axioms)</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Necula and Lee’s Type-Specialized Proof-Carrying Code</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Code-Carrying Theory (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Code-Carrying Theory, Tampered-With But Checked and Prevented from Acceptance (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Code-Carrying Theory, Tampered-With But Checked and Prevented from Acceptance (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Code-Carrying Theory, Tampered-With But Checked and either Prevented from Acceptance or Guaranteed Correctness (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>Generic Axiom or Theorem Definitions in CCT</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>General Structures of Proof Methods in CCT</td>
<td>74</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

My interaction with my advisor, Dave Musser, started with the programming languages class I took at Rensselaer. The challenge problems I solved in the C++ competition in his class were simple but important since they sparked even more interactions. In the beginning of 2002, while trying to find my way between two poles I decided to go with the “proofs.” Dave Musser, Code-Carrying Theory, logic, proofs, correctness, recursion and generic concepts became an important part of my life during the next five years.

I had the privilege to be supervised by one of the world’s top researchers who taught me to be precise and patient. I will always remember him as a meticulous person about every detail of the code and documents we wrote. He was always reachable, his door was always open. We held hundreds of meetings, sometimes discussing for hours either in his office or through email. There were times that he even worked harder on this project than I did. He is a good friend, a thoughtful person, and a real problem solver. I will always miss his sense of humor and friendship.

I would also like to thank to the other members of my PhD committee: Selmer Bringsjord, Mukkai Krishnamoorthy, Paliath Narendran, and Carlos Varela. I am much indebted to them for their encouragement and for spending their precious time to read my thesis and provide comments, advice, and questions.

I think there are many ways of learning the system you are working with. One of the best ways, in my opinion, is to get help from the system’s implementer. Kostas Arkoudas was always available whenever we needed help for ATHENA. He kindly provided us with new versions of ATHENA, code examples, documents, and invaluable comments for my thesis.

I know that I have been helped by many people in the computer science department at Rensselaer. But there were two that I interacted with more than the others: Terry Hayden and Christine Coonrad. There were many times I have visited their offices for various problems. I do not remember any single occasion that my problems were not solved. Students and the Rensselaer administration are really
very lucky to have them both.

I also wish to express my sincere thanks to one of my former advisors, Lloyd D. Fosdick. Our friendship started in Turkey and continued in the US. Perhaps I would not be here pursuing my PhD degree if he did not provide me with guidance and help. With this opportunity I also want to thank Irem Ozkarahan, former chair of the computer science department of the Dokuz Eylul University (DEU) in Turkey. I would not possibly have attempted to come to the US if DEU had not offered me a full scholarship for my education.

I doubt that I could have reached this stage of my academic life without having my two little treasures by my side: my wife, Kadriye, and our lovely daughter, Zeynep. I am extremely grateful to Kadriye for her constant support, patience, love, and assistance during my research. Her contributions in my life are endless. I am also deeply thankful to my parents for their outstanding support of my study and taking care of my chores in Turkey.
ABSTRACT

Code-Carrying Theory (CCT) is an alternative to the Proof-Carrying Code (PCC) approach to secure delivery of code. With PCC, code is accompanied by assertions and a proof of its correctness or of other required properties. The code consumer does not accept delivery unless it first succeeds in generating theorems, called verification conditions, from the code and assertions and checking that the supplied proof proves these theorems. With CCT, instead of transmitting both code and proof explicitly, only assertions and proofs are transmitted to the consumer. If proof checking succeeds, code is then obtained by applying a simple tool called CODEGEN to the resulting theory. This thesis first explains the design and implementation of CCT and shows how it can be used to achieve secure delivery of code with required correctness or safety properties. All the tools used in the verification steps are implemented in ATHENA, which is both a traditional programming language and a deduction language. In addition, we present examples of generic and nongeneric proofs, which play an important role in our design. We show how critical it is to organize theories and proofs to reduce the amount of information transmitted between the code producer and consumer and to ease the development of code-carrying theories.
CHAPTER 1
Introduction

Today’s ever increasing trends toward pervasive computing include the phenomenon of mobile code, which must be installed and run on a computer host even though it is delivered through an insecure network or from an untrusted source. Important examples of such mobile code include Web applets, actor-based distributed system software [32], and updates to embedded computers. During delivery, code might be corrupted, or a malicious hacker might change the code. Potential problems can be categorized as security problems (i.e., unauthorized access to data or system resources), safety problems (i.e., illegal operations or illegal access to memory), or functional incorrectness (i.e., the delivered code fails to satisfy a required relation between its input and output). Proof-Carrying Code (PCC) [23] and Code-Carrying Theory (CCT) are two alternatives among other solutions to these problems. In comparison with other solutions, these two alternatives can generally provide stronger assurance of secure delivery of code with all required security, safety, or functional correctness properties preserved.

A quick picture of the main idea of the PCC approach can be seen in Figure 1.1 and Figure 1.2. The basic idea of PCC is that a code consumer does not accept delivery of new code unless it is accompanied by a formal proof of required safety, security, or correctness properties that can be checked by the code consumer. One way of doing this is to attach to the code an easily-checkable proof at the code producer’s site. This proof must be checked by the code consumer and should prove that the code does not violate predefined requirements.

The transmitted proof can be regarded as a form of certification that it is safe to execute the code. In Necula and Lee’s PCC approach [23], code is annotated with assertions and a program called a verification condition generator is applied to produce a set of verification conditions (VCs), which are propositions whose validity implies that the code satisfies the stated assertions. Both the annotated code and the proof of its VCs must be sent to the consumer, who regenerates the VCs, checks that
the proof proves them, and accepts delivery of the code only if this proof-checking step succeeds. In Appel’s PCC approach [1], which, like Necula and Lee’s, is focused on safety properties, definitions of safe machine instructions are encoded within a suitable logic. The safety conditions are included in these definitions implicitly. Therefore, a VC generator is not necessary, resulting in a smaller “trusted code base,” i.e., a smaller amount of code subjected only to manual inspection, not formal checking. A program is simply defined as a sequence of machine instructions which are represented by integers. Then a safety theorem can be constructed based on the idea that the program counter points to an illegal instruction if any unsafe instruction is encountered during the execution of the code. As in Necula and Lee’s approach the proof of the safety theorem is transmitted with the code to the consumer.

Code-Carrying Theory (CCT), is a new alternative to PCC with similar goals and technology, but it is based on the idea of proof-based program synthesis [21, 9, 12, 8] rather than program verification. That is, instead of starting with program code in a conventional programming language, annotating it with assertions, generating VCs
and proving them, one instead starts with axioms that define functions (generally with equations and induction) and proves that the defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus all that has to be transmitted from producer to consumer is a \textit{theory} \footnote{There are some exceptions permitted to the requirement that functions be total, as discussed in Section 3.1.3.} (a set of axioms and theorems) and a set of proofs of the theorems; there is no need to transmit code explicitly or to apply a VC generator. An obvious concern with this approach is preventing the acceptance of new axioms that could introduce a logical inconsistency (after which anything could be proved). To this end our approach only admits new axioms if they are of a form that defines new functions and obey a “definitional principle” \cite{28} that guarantees, in essence, that they only define total mathematical functions. The consumer checks that the proofs actually \textit{do} prove these theorems, and only if that step succeeds does the consumer apply a “code extractor” to the function-defining axioms to obtain the
executable code. A variant on this approach would be to extract code directly from proofs based on their structure in terms of how inference rules of different types are used [21]. In this thesis, we do not explore this second possibility of code extraction further. Figure 1.3 illustrates the basic CCT approach.

As previously mentioned, requirements on a program can take the form of functional correctness requirements (e.g., an algorithm sorts its input), safety requirements (e.g., no out-of-bounds array-indexing), or security requirements (e.g., no unauthorized access to classified data). In the case of safety, the requirements are stated as a set of “safety rules” and are also called a “safety policy.” One can define different safety policies for accessing memory or other system resources. PCC has been developed and applied primarily as a method for achieving safety, but in principle it can be used with other forms of requirements (though, under the cur-
rent state of the art of program verification and theorem proving technology, with a lesser degree of automation than is achievable with safety properties). CCT could also be used with any of the three forms of requirements, but in this thesis we focus on applying CCT to functional correctness (also called “input-output correctness”). In conjunction with either PCC or CCT, one could employ additional certification techniques, such as encrypted signatures or check-sums; Devanbu et al. [13] discuss such combinations as well as the use of “trusted hardware.”

We have implemented a code extractor program called CodeGen that is capable of producing code that implements both memory-observing and memory-updating functions. By memory-observing functions we mean ones that examine data structures but do not make any changes, such as searching or element-counting functions. In contrast, memory-updating functions make in-place changes to data structures, a simple example being a reverse function that replaces a sequence of elements in memory with the same sequence in reverse order.\(^2\) We first present in Chapter 4 a simple example of theories, proofs, and extracting code from function-defining axioms, one that results in a memory-observing function named sum-list that computes the sum of the natural numbers in a (Lisp-like) list. We present axioms defining an embedded-recursion version of sum-list that can be viewed as the specification (the requirements) of a function that a code consumer would like to have. We then present the theory of a tail-recursive (and therefore more efficient) sum-list function that can be viewed as the implementation to be delivered and from which code is extracted. This example is also used to outline the CCT “protocol” between a code producer and a code consumer that is designed to ensure secure code delivery.

In Section 6.1 we introduce a version of the sum-list example that is generic.\(^2\) This example is presented in Chapter 5. Some authors use the terms “nondestructive” for memory-observing and “destructive” for memory-updating, but “destructive” can leave an impression of undesirable behavior that is at odds with the actual usefulness of such operations. Another issue of terminology is whether memory-updating “functions” should be called functions at all, since they do not obey all the requirements of functions as defined in traditional mathematics. We chose to stick with calling them functions since this usage is widespread in computer science, but in stating axioms and proving theorems about memory-updating functions we must be careful to address the issue that they can violate one of the most basic principles of the logic we use, the substitution principle of equality. How we handle this issue is discussed in Section 5.5.
in that the list elements may belong to any type that models a monoid theory (i.e., having an associative binary operation and an identity element). This generic sum-list can be instantiated with other numeric types besides natural numbers, or even with nonnumeric types such as sequences (such as lists of lists, with append and the empty list as the Monoid operations). In doing this, we simulate type-parameterization simply by parameterizing functions and methods by functions that carry operator mappings.

Another important dimension with generic functions is generalizing to work with a family of data structures, as popularized in the C++ Standard Template Library (STL) [29, 22]. We have defined a generalization of sum-list called sum-range that is parameterized by STL-like iterators (objects with abstract properties generalizing pointers). It can be applied to any sequence representation that obeys a certain range theory that captures the essential properties needed by this function and many useful functions similar to those included in the STL, such as accumulate, max_element, copy, find, etc. Rather than present the details of sum-range, we illustrate the issues and complications involved with this kind of genericity with a reverse-range function for reversing the order of elements in a sequence delimited by iterators. Since reverse-range is an in-place operation, this example also serves to introduce our approach for dealing with correctness of memory-updating functions, and, to a limited extent, with memory safety properties.

Another principle that the STL illustrates is that small, well-designed, and interoperable library components can be easily composed to construct more complex components of an application. Thus, although the examples of theories and extracted functions presented in this thesis are quite small, their generality and other STL-like design characteristics enable a degree of usefulness that goes far beyond actual code size. Chapter 7 presents a case study of a code extraction of a partitioning function composed of calls of several of the functions whose correctness has been formally proved and whose associated theories should provide a practical basis for its more complex correctness proof.

The implementation tool we use is ATHENA [2], which provides a language both for ordinary computation and for logical deduction. As a functional programming
language, it provides higher-order functions and is lexically scoped in the tradition of Scheme and ML. As a deductive programming language, Athena makes it possible to formulate and write proofs as function-like constructs called methods, whose executions either result in theorems if they properly apply rules of inference, or stop with an error condition if they do not. This high-level abstraction of proofs allows the user to call the functions and apply methods defining them in many different ways. Therefore, the same proof does not have to be redeveloped for different parameters. A generic library of proofs can be constructed with this ability. Especially in systems like PCC and CCT, proof size is a major problem which may cause to slow transmissions between the producer and the consumer. In some cases, the size of the proof may grow exponentially with respect to the functions being constructed. Writing generic proofs and organizing them properly via theories can help to overcome this problem. Once the generic proofs are available at the consumer’s site, proper instantiations of them may reduce the amount of extra transfers from the producer. In Section 6.1.6, we present a working example of instantiating generic proofs and show how it can dramatically reduce proof sizes by multiple instantiations of the same general property and its proof.

1.1 Contributions
Correctness and safety of software obtained from remote systems has become a major problem. Especially with the heavy use of mobile code, it is vital to have an efficient, general, and flexible mechanism to verify the correctness of code, which is generally transferred through an insecure network or from an untrusted source. Code consumers need assurance that the software is not corrupted or harmful, whether intentionally or inadvertently. Furthermore, requirements should be satisfied without requiring run-time checks, if possible, in order to avoid performance degradation.

In this thesis, we propose a general framework called Code-Carrying Theory (CCT) to provide strong assurance to code consumers who receive, install, and execute new or updated code. CCT is flexible and general. Code consumers can decide on and define correctness requirements without reference to the specifics of machine instructions or architectures or any language-specific type systems.
In CCT, the code producer constructs axioms that define the functions required by the consumer. In addition to axiomatic definitions, the producer is also responsible for providing proofs of termination, consistency, and correctness conditions. In CCT, the producer does the time-consuming and hard work. Proof checking is relatively easy and can be done very quickly. The consumer only checks the proofs and extracts code automatically using a small code extractor program. If proof checking fails, the consumer does not extract code. Since CCT is simple and has a small trusted computing base it requires only modest amounts of memory and computing power, in comparison with approaches that require automated proof search. The relatively small memory and computing-time requirements for checking proofs and executing the code extractor are especially important in cases where the code consumer is a small embedded system, such as heart pacemakers, cell phones, PDAs, military-related systems, etc.

Some code consumers, including some embedded systems, require frequent updates. An important issue in CCT and previous approaches is the size of the proofs being transmitted. Proofs can be very large. The second contribution of this thesis is to provide a methodology that makes it easy to write generic proofs. The producer defines axioms with parameterized generic property functions. Similarly, generic proofs are constructed in the form of parameterized programs. Once the consumer receives these generic property functions and proofs, they can be instantiated over and over, reducing the need for further transfers from the producer. In CCT this does not impose any additional burden on the consumer. Once again, the hard work of writing the generic definitions, theorems, and proofs is done by the producer.

Previous approaches in the area are based on automatic generation of proofs which are not human readable. While writing both generic and nongeneric proofs we also aimed to write proofs that are both machine checkable and human readable. We tested our approach by constructing numerous generic and nongeneric proofs, which are easy to understand and are maintainable. For this to be possible, selecting the right theorem prover that suits our goals became an important task. We believe Athena provides more help than any other existing interactive theorem prover for
writing readable and generic proofs that are also machine checkable.

In a system like CCT, code producers prove hundreds of theorems about programs resulting in thousands of lines of proof. Generally they prove some reusable lemmas and use them in the proofs of other theorems when needed. The last contribution of this thesis is to present a set of functions that can be used in organizing generic axioms, theorems, and proofs. In this way, related properties can be grouped together under the same theories, with properties that are common to several theories represented only once. An important benefit of this organization is that searches for a stored theorem and its proof are usually limited to a small fraction of the proof library.

1.2 Outline of the thesis

The next chapter reviews existing PCC approaches and discusses different approaches to formal proof, especially relating to human-readability of proofs. In Chapter 3, we discuss the CCT approach in detail. Chapter 4 illustrates the CCT steps with a nongeneric example: sum defined on lists, sum-list. In Chapter 5 we first show how we define, access, and update memory in CCT. We then present a nongeneric memory-updating function that reverses the order of elements in an STL range, reverse-range. After discussing how to construct generic proofs and organize theorems and their proofs in Chapter 6, we show another application of CCT with a generic reverse-range function. We formally prove the correctness of more functions with CCT in Chapter 7. In this chapter we first develop formally correct components of a complex function and then we show how the main function can be constructed afterwards with the use of the definition of its components. In Chapter 8 we give a brief summary of our research. Finally, our future goals with open problems to be solved in this project are discussed in Chapter 9.
Figure 1.4: Necula and Lee’s Type-Specialized Proof-Carrying Code
CHAPTER 2
Related Work

2.1 Proof-Carrying Code (PCC)

2.1.1 Type-specialized proof-carrying code

Figure 1.4 shows the original proof-carrying code approach developed by Necula and Lee. Appel refers to this type of PCC as *type-specialized PCC* [1]. The architecture is composed of a compiler, verification condition generator (VCGen), theorem prover and a proof checker. This technique is fundamentally based on the idea of defining the safety policy in terms of *types* and achieving safety by *type checking*.

In general the predefined safety policy consists of memory safety rules, typing rules, predicates defining the semantics of the machine instructions, and rules of first-order logic. Some examples of the rules which could be defined to achieve safety would be: specific memory regions are readable, specific memory regions are writable, if a variable’s type is natural numbers then it is readable, and so on.

All the rules in the safety policy are expressed by using the higher-order LF syntax [17]. Representing the rules of safety policy in a formal logic is crucial to generate and check the safety proof during the later steps. Since type checking is used to decide the safety policy, all the typing rules are required to be introduced as predicates by using the LF logic. An example of such a predicate is “expression $e$ has type $t$.”

An important note is that both the code producer and the consumer will obtain the safety policy before the code transfer. In the remaining part of this section we explain the details of type-specialized PCC.

2.1.1.1 Creating the native code with annotations

In the first step, the source code written in a type-safe subset of C is translated by a compiler called Touchstone [24] into annotated native code and type specifications. Touchstone is an optimizing compiler whose target language is assembly.
The native code will be transferred to the consumer along with a proof of safety. The code annotations provide more information about the code to be sent by the producer. This extra information can be used to make the VCGen’s job easier, to enable optimizations, or to resolve ambiguities.

2.1.1.2 Generating the verification condition

The next step is to construct a verification condition as the safety theorem. The code producer applies the VCGen to the code and the annotations to construct a predicate in the LF-style first-order logic, defining the meaning of the source code. While doing this it uses the loop invariants, function preconditions and postconditions as well as the typing rules. Each of the created verification conditions is a Floyd-style verification condition [15] which is a predicate expressed in first-order logic.

The VCGen is a simple and fast program written in C. The code consumer will use the VCGen and create the same verification conditions later for the purpose of checking the proof of safety.

2.1.1.3 Finding a proof of the verification condition

After the VCGen generates the verification condition, the code producer uses a theorem prover to get a proof of this condition. As it is seen from Figure 1.4, the theorem prover uses the axioms defined as part of the safety policy. The proof is the certification that the code is both type safe and memory safe which means it can be executed safely.

The proof must be encoded formally so that it can be checked by the consumer automatically. In type-specialized PCC, the proofs are done by using the Elf [26] theorem prover which produces LF-style representation of proofs. It is important that the proofs are represented in LF because when the consumer uses the proof checker to check the proof it assumes this pre-negotiated framework.

In general, proving the verification condition is a time-consuming and complex process to be done by the producer. But it is the beauty of this technique that this difficult job is not being done by the code consumer during any steps of PCC.

Finally this proof is submitted with the native code to the consumer.
2.1.1.4 Verifying the proof

The code consumer is required to check the proof before loading the executable submitted by the producer. He runs the VCGen to find the verification condition as the producer did previously. The next step is to see if the submitted proof is really the proof of the verification condition. After the proof check succeeds the consumer can load the executable safely.

One important reason for using LF-style encoding of proofs is to use the type checking in LF to verify the proofs. To be able to use this facility one should define all the predicates as LF types and write the proof as an LF expression. More formally: let \( P \) be a predicate and \( \text{pf} \ P \) be a proof of this predicate. Assume also that we have the following declarations:

\[
\begin{align*}
\text{pred} &: \text{Type} \\
\text{pf} &: \text{pred} \rightarrow \text{Type}
\end{align*}
\]

Having \( \text{pf} \ P \) means that we actually have the proof of \( P \) and this proof should have a type. Based on this information we use the verification condition as the predicate \( P \) and want to see if LF type checks the proof of the verification condition \( \text{pf} \ P \) provided by the code producer. If it succeeds then the proof validation is completed. After this step the code can be executed safely.

2.1.2 Foundational proof-carrying code

Even though a VCGen is simple and fast, Appel points out that its size is too large [1] and it needs to be verified itself. If there were a bug in either the VCGen or the typing rules then the trusted base would be vulnerable. In fact League [19] found an error in the typing rules of Special J [11], which is another certifying compiler that generates machine code from Java source code. Needless to say this bug affects the overall safety in type-specialized PCC. Since Appel avoids using a VCGen the trusted base is smaller.

Foundational PCC is based on the idea of defining the semantics of the machine instructions and the safety rules using only a foundational mathematical logic (i.e., a logic that itself has no programming-language–specific axioms or safety rules). This approach does not rely on a particular type system since it does not introduce
any type constructors as primitives or prove lemmas about these constructors as in type-specialized PCC. As in type-specialized PCC the code producer must send both the executable code and its safety proof.

### 2.1.2.1 Defining and proving safety

A specific von Neumann machine such as a Sparc processor is required to be modeled for a program \( P \) implemented in a machine-language, with the model consisting mainly of a register bank and a memory. Each of these can be thought of as a function from integers (addresses) to integers (contents). The foundational logic used to define instructions and instruction executions is a higher-order logic. Each execution step is specified by a step relation \( (r, m) \rightarrow (r', m') \) where \( r \) and \( r' \) are registers and \( m \) and \( m' \) are memory units. Then by executing one instruction the machine state is changed from \( (r, m) \) to \( (r', m') \). As an example (from [1]), one could define the add(\( r_1, r_2, r_3 \)) instruction \( (r_1 \leftarrow r_2 + r_3) \) as

\[
(r, m) \rightarrow (r', m') \equiv r'(1) = r(2) + r(3) \land (\forall x \neq 1. r'(x) = r(x)) \land m' = m
\]

An important property is that some steps do not have any successor steps. In these cases the program counter points to an illegal instruction which should never exist in a safe program. In addition, if there is an undefined instruction it will be considered to be an illegal instruction that violates the safety policy. Having this in mind the safety policy is defined as: A given state is safe if, for any state reachable in the Kleene closure of the step relation, there is a successor state. Here is the formal definition:

\[
safe-state(r, m) \equiv \forall r', m'. (r, m) \rightarrow^* (r', m') \Rightarrow \exists r'', m''. (r', m') \rightarrow (r'', m'')
\]

where \( \rightarrow^* \) denotes zero or more steps.

Note that this definition is used for formulating the verification condition for each program. The code producer should decide on an invariant which should hold during the execution of the code satisfying \( safe-state(r, m) \) for each step. If any step reaches an illegal instruction then the safety policy is violated and the program will
not be accepted by the consumer.

Defining different safety policies is possible by using a different form of step relation. Let us define a policy saying “only the contents of memory addresses between 0 and 1000 are readable.” We first define the predicate \( \text{readable} \) as follows:

\[
\text{readable}(x) = 0 \leq x < 1000
\]

We next define a general \texttt{load} instruction \( r_i \leftarrow m[r_j + c] \)

\[
(\text{load}(i, j, c) \equiv \lambda r, m, r', m'.(r'(i) = m(r(j) + c)) \land (\forall x \neq i.r'(x) = r(x)) \land m' = m)
\]

Now we just need to change the semantics of the \texttt{load} with an extra conjunct:

\[
\text{load}(i, j, c) \equiv \lambda r, m, r', m'.
\begin{align*}
& r'(i) = m(r(j) + c) \land (\forall x \neq i.r'(x) = r(x)) \land m' = m \\
& \land \text{readable}(r(j) + c)
\end{align*}
\]

When executing the program if the program counter points to a \texttt{load} instruction which wants to read the content of memory address 1001, the state relation will not relate this state to any other state.

In order to prove that a program \( P \) is safe what we need to do is to prove \( \text{safe-state}(r, m) \) for steps = \( S_0, S_1, \ldots, S_n \) for \( n \) steps. This can be done by using proof by induction on \( n \). Note that if we succeed in finding a proof this means that after executing the program \( P \) we get a safe state.

In foundational PCC the proofs are currently done by hand and represented using the LF syntax. Therefore if one has the proof of a program \( P \) the proof validation would be achieved by just type-checking the given proof in LF. The theorem prover used in this process is Twelf \cite{27}.

2.1.3 Language-level differences between PCC and CCT

We are working in a higher level programming language than one finds being used in most of the PCC work. Of course, in the case of CCT the “source language”
is actually a logic as defined and implemented in the proof checking system, so when we talk about the programming language involved we mean the language in which the extracted code is expressed. Previous research such as Necula and Lee’s [23] and Appel’s [1] has been carried out with very low level languages (machine code or Java byte code). We believe the best support available for CCT or other proof-based program synthesis approaches is a Denotational Proof Language (DPL) [2, 3, 4], which provides separate but intertwined languages for ordinary programming and logical deduction. The DPL system we are using, ATHENA [5], provides an ML-like programming language (but with Scheme-like syntax) in which programs are typically expressed as collections of recursive function definitions, and a structurally similar language for expressing deductive methods whose executions carry out proofs using both primitive inference methods of first order logic with equality (of which there are about a dozen, such as modus ponens, existential generalization, universal specialization, etc.) and “calls” of other deductive methods previously defined by the user or imported from a deductive-method library. Both of these languages are high level by most programming language standards, offering, for example, higher-order functions (and methods)—the ability to pass functions/methods to a function/method or return them as results.

Moreover, we are experimenting with the use of both of these programming and deductive language facilities at a substantially higher level of abstraction than in most programming activity. We have already found elegant ways to use higher-order functions and methods in ATHENA to express generic requirements specifications much like the theory specifications of research languages such as Isabelle [34], Imps [14], or Maude [10]. There are two major benefits that stem from expressing proofs at this high abstraction level. First, once a proof of a theorem is written at this level, the functions and methods defining it can be called in many different ways, which means that the proof does not have to be redeveloped when its conclusion is needed.

\[3\] A friendlier syntax is in the works for ATHENA as outlined in [6], where Arkoudas notes, “…some find it odd that a language whose primary design goal was proof readability would use s-expression syntax. In fact there is nothing odd about it, provided that one appreciates the important distinction between concrete and abstract syntax. Most of the effort in the design of ATHENA went into the development of the abstract syntax and the formal semantics of the language. Concrete syntax was viewed as something that could be settled later.”
in a different setting. Second, such high-level theory or concept specifications serve
as a replacement for template or other generic parameterization mechanisms in
programming languages, in that one can extract from a single source “code” many
different useful specific versions by plugging in other suitable components. Thus
the substantial effort required in constructing such proofs can be amortized over the
many repeated uses of both proofs and the generic software components that can
be extracted from them.

2.2 Machine-checkable and human-readable proofs

Some theorem proving systems strive for complete automation—as in resolu-
tion provers or in Boyer-Moore provers that use highly developed heuristics to con-
struct induction proofs—while others strive mainly for human readability of proofs
and require the human user to provide almost all of the proof structure and only
check for following rules of the logic—the Mizar system [30] being perhaps the
extreme case. In the spectrum between these extremes, there is always a tension
between automation and human-readability goals. In our opinion, Athena deals
with this tension better than any other system we have encountered. Its format of
methods operating on an assumption base provides an elegant framework for ex-
pressing proofs in great detail where that is desired—and we believe it is desirable
for proofs in generic theories, since then one can learn much about how such proofs
work and often use them as guides to construction of other proofs. It is also de-
sirable for proofs that must be efficiently checkable, without resorting to extensive
search, as is likely to be the case for the embedded system type of code consumer.5
On the other hand, the hooks that Athena provides to Vampire (and to other
highly-tuned resolution provers such as S Pass [33] and Otter [20]) support more
productive human activity and could serve well even in the CCT setting in cases
where the consumer has sufficient computational resources.

Athena method bodies are both human-readable and efficiently machine-

4In a system supporting higher order logic, it wouldn’t be necessary to rerun the proofs, but
rerunning them in Athena is not expensive anyway.

5Note, however, that even the detailed Athena-only proofs do not require the reader to examine
each step in terms of a small set of primitive axioms and inference rules; it is easy and natural to
use previously proved theorems as lemmas.
checkable. Another possibility might be Isabelle/Isar’s format [34], which is implemented as a front end to Isabelle [25]. Isabelle itself provides executable proof methods called tactics, which operate on a sequent representation of propositions. Both tactics and the sequent representation are typical of a large number of formal proof checking systems but unfortunately they fail to provide readable proofs; to understand such a proof, one has to replay it and examine the intermediate goals produced as it executes step by step. Isabelle/Isar proofs are intended to be human-readable, but this is achieved by an extra level of mechanism that does not always prevent the underlying tactic and sequent issues from surfacing when writing and debugging proofs, so one still has to learn to deal with them.

Also crucial to the soundness and expressiveness of Athena’s proof checking capability is its strong type system for terms and propositions, which allows declarations of domains, structures, and arity of symbols (operators), with subsequent type-checking via the same Hindley-Milner style of type inference used in ML.
CHAPTER 3
Code-Carrying Theory (CCT)

Figure 3.1 illustrates the basic Code-Carrying Theory approach in detail. With CCT, instead of transmitting code, assertions, and proofs explicitly, as in PCC, only assertions and proofs are transmitted. The assertions transmitted are limited to axioms that define new function symbols and can be readily checked by the software consumer to do so properly as discussed in the next section. The proofs, expressed in a Denotational Proof Language \[2, 3, 4\], take the form of executable methods, which, when executed in the presence of the new axioms and ones already held by the consumer, prove additional propositions as theorems.

From the resulting theory—axioms and theorems—code is extracted by a simple process that perfectly preserves the semantics expressed in the theory.\(^6\) While PCC can be said to be a variation on the traditional program verification approach, CCT can be viewed as a variation on the traditional proof-based program synthesis approach. When compared to ordinary methods of software development, the proof-based synthesis approach has much higher demands on human time and expertise than seem justified for most software. However, in light of the new demands of mobile code with high-stakes requirements, such as those of safety-critical embedded systems or code to be distributed to millions of consumers, investing in the time and resources required to write theorems and generate the proofs is much more justifiable.

3.1 Code-Carrying Theory steps

3.1.1 Step 1: Defining Requirements (Code Consumer)

The code consumer is responsible for the construction of requirements, which are of two types: general and application-specific. General requirements contain declarations and axiomatic definitions of some common symbols which are being

\(^6\)An exception to the exact semantic correspondence is the code for memory-updating operations, as explained in Section 5.5.
used frequently in CCT. We have also proved new theorems from this initial set and built theory libraries which are ready to be loaded as part of the general requirements. Both consumers and producers have the same set of general requirements initially. This reduces redundant transfers of a large set of definitions and proofs between them.

Code consumers use application-specific requirements to describe new functions. These requirements include a new symbol for each new function and propositions specifying its correctness property or properties. Initially code producers do not have these types of requirements and receive them from the consumers later in the process. After a code consumer specifies a function by using application-specific requirements he sends them to the producer.
3.1.2 Step 2: Defining a New Function (Code Producer)

After the code producer receives the specifications of a function requested by the consumer, it enters the symbol declarations into the theorem prover (Athena[2] in our case) and sets up the specified properties as goals to be proved.

The code producer wants to send to the code consumer an efficient implementation of the specified function and a proof that it satisfies the required specification. Rather than sending the efficient function as actual code, the producer will send only definitions and proofs.

3.1.3 Step 3: Proving Termination (Code Producer)

Looking ahead, after the consumer receives the function definition in the form of a set of axioms, he must decide whether to accept the axioms or not. In general the consumer cannot accept and assert arbitrary axioms and propositions. If this were allowed, the producer or a hacker would be able to use them to prove anything.

Now the question is, how can the consumer recognize certain kinds of propositions as constituting a definition of a function, and accept them and add them to its assumption base? It is the producer’s responsibility to show that the new definitions do not introduce any inconsistency into the assumption base. This can be done by proving that the new function-defining axioms satisfy a definitional principle. In logics in which a function is defined by a single recursive equation, such as the Boyer-Moore computational logic [28], the definitional principle takes the form of a requirement that the equation defines the function totally, i.e., the function terminates for all inputs. We enforce this requirement also for function definitions in the form of a set of axioms, but in this case there is an additional consistency requirement: that the axioms together do not impose such strong requirements that there could not exist a function satisfying them. Here we discuss how we deal with termination; the additional consistency requirement is discussed in the next step.

We will show a very simple example of how we deal with termination, but first we give a very brief introduction to Athena’s term and proposition languages. Terms can be either constants, variables, or function applications, where the latter is of the form \((f \ x_1 \ldots x_n)\), in which \(f\) is a function symbol and each \(x_i\) is a constant,
variable, or function application. Each term has a type, which is some *domain* (a type without prescribed constructors) or *datatype* (a type with a fixed set of constructors for which a corresponding induction principle is automatically generated by *Athena*). Function symbols are declared to have specific domain and range types with declarations of the form \((\text{declare } f \ (-\rightarrow (d_1 \ldots d_n) \ r))\) corresponding to the usual mathematical notation \(f : d_1 \times \ldots \times d_n \rightarrow r\). Constants are just functions of no arguments, declared (equivalently) as \((\text{declare } c \ (-\rightarrow () \ r))\) or \((\text{declare } c \ r)\). For example, if \(\text{Nat}\) is a datatype (one which, in fact, we have axiomatized to represent the natural numbers), then \((\text{declare } f \ (-\rightarrow (\text{Nat}) \ Boolean))\) expresses that \(f\) is a function taking a natural number argument and returning a \(\text{Boolean}\), and \((\text{declare } \text{one} \ \text{Nat})\) expresses that \(\text{one}\) is a constant of type \(\text{Nat}\). Types of arguments in a function application must match the corresponding types in the function symbol’s definition, and the type of the application is the declared range type of the function symbol.

*Athena* propositions are then either constants or variables of type \(\text{Boolean}\), function applications of type \(\text{Boolean}\), or expressions of one of the following forms:

\[
\begin{align*}
\text{(not } p\text{)} & \quad \text{(negation, conventionally written } \neg\text{)} \\
\text{(and } p_1 \ p_2\text{)} & \quad \text{(conjunction, } \land\text{)} \\
\text{(or } p_1 \ p_2\text{)} & \quad \text{(disjunction, } \lor\text{)} \\
\text{(if } p_1 \ p_2\text{)} & \quad \text{(implication, } \supset\text{)} \\
\text{(iff } p_1 \ p_2\text{)} & \quad \text{(equivalence, } \equiv\text{)} \\
\text{(forall } x_1 \ldots x_n \ p\text{)} & \quad \text{(universal quantification, } \forall\text{)} \\
\text{(exists } x_1 \ldots x_n \ p\text{)} & \quad \text{(existential quantification, } \exists\text{)}
\end{align*}
\]

where \(p, p_1, p_2\) are propositions and the \(x_i\) are variables (which in this context are always written with a preceding “\(\forall\)”). For example the proposition that would be written in traditional logical notation as

\[
\forall x, y(p(x, y) \supset \exists z(q(y, z) \lor \neg r(z)))
\]

would be written
\[(\forall x \ y. (\text{if } p \ x \ y \text{ then } (\exists z. (\text{or } q \ y \ z \text{ and } \neg r \ z))))\]

(where we have used line breaks and indentation, which, though unnecessary, helps show the structure of the expression). Appendices A through C explain ATHENA in more detail and describe the term rewriting and specialized induction methods we have implemented.

Returning now to CCT, it is the code producer’s responsibility to define new functions by means of axioms from which a terminating function can be extracted. Although termination is in general undecidable, it is easily proved in special cases by showing there is a well-founded ordering, such as \(<\) on the natural numbers, such that the size, length or any other similar measure of the arguments decreases in the ordering with each recursive call of the function. One approach to mechanizing proofs of termination is to require that each new recursive function definition be accompanied by an ordering relation such as \(<\) and a formal proof that each step in the recursion produces a smaller value in the ordering. Since the ordering is well-founded, by definition any sequence of decreasing values is finite; hence any sequence of recursive steps from the initial function arguments is finite, and the function computed is total. In systems supporting automatic proof search, such as the Boyer-Moore provers, this proof can in many cases be found automatically, or with help only in the form of guiding lemmas. In the CCT context, however, where we try to avoid the need for automatic proof search, it is up to the producer to construct a termination proof in sufficient detail that the consumer needs only a limited proof-checking capability.

The approach we have adopted for proving termination is similar to the ordering-relation approach, but instead of requiring a separate order relation we construct the proof of termination as a proof by induction that mirrors the recursion structure in the axioms. That is, we generate for each new function being defined a termination condition (TC), and from each axiom of the function’s definition we generate a corresponding termination axiom. Using the resulting set of termination axioms, we construct a proof of the TC, with the structure of the proof being a
proof by induction on the function’s arguments. That these proofs have essentially
the same form as proofs by induction of correctness properties (as will be seen in
the examples in the next chapter) is perhaps the main advantage of our approach
over the ordering-relation approach, which has to introduce the extra complication
of an ordering relation and a somewhat different style of proof.

We wrote a program called TCGEN which takes a list of function-defining
axioms and produces the corresponding termination axioms. In addition to these
predicates, it creates the termination condition (TC). We now give the steps of how
TCGEN transforms the given axioms and constructs the termination axioms and
condition from the axioms defining a new function symbol \( f \).

We require each axiom to be either an equation or conditional equation in
variables that are universally quantified. (As a special case the axiom might contain
no variables and no quantifiers.) For a function \( f : d_1 \times \ldots \times d_k \rightarrow r \), an equational
axiom must have the form

\[
\forall x_1 \ldots x_j (f(t_1 \ldots t_k) = e),
\]

where each \( t_i \) is a term of type \( d_i \) and \( e \) is a term of type \( r \), and a conditional axiom must have the form

\[
\forall x_1 \ldots x_j (p \supset f(t_1 \ldots t_k) = e),
\]

for some Boolean expression \( p \); \( x_1 \ldots x_j \) are the free variables in the equation (and
in \( p \), if present).

Corresponding to each function or predicate symbol \( f : d_1 \times \ldots \times d_k \rightarrow r \) we
introduce a new predicate symbol \( f.t : d_1 \times \ldots \times d_k \rightarrow \text{Boolean} \) for the specific
purpose of asserting that \( f \) terminates.\(^7\) The termination condition (TC) for \( f \) is
then expressed as

\[
\forall x_1 \ldots x_k f.t(x_1 \ldots x_k).
\]

In order to define the termination axioms for \( f \), we first define a function \( \tau \)
that maps any term \( u \) into a proposition \( \tau(u) \) that asserts that computation of \( u

\(^7\)The \( t \) is mnemonic for termination, and we assume the underscore character is not otherwise
allowed in function names so that there is no conflict with this naming convention.
terminates. The definition of $\tau$ is by induction on the term structure of $u$, and assumes there is a predefined set of function symbols $S$ that represent known total functions.

- If $u$ is a constant or variable, $\tau(u)$ is true.
- If $u$ is a function application, $f(t_1 \ldots t_k)$, where $f \not\in S$, then $\tau(u)$ is the conjunction
  \[
  \tau(t_1) \land \ldots \land \tau(t_k) \land f \not\in \tau(t_1 \ldots t_k),
  \]
- If $f \in S$, the $\tau(f(t_1 \ldots t_k))$ is just $\tau(t_1) \land \ldots \land \tau(t_k)$.
- If any subexpression of $\tau(u)$ has the form $Q \land \text{true}$ or $\text{true} \land Q$, replace that subexpression by $Q$.

Then the termination axiom corresponding to an equational axiom

\[
\forall x_1 \ldots x_j \ (f(t_1 \ldots t_k) = e)
\]

is, in the most general case,

\[
\forall x_1 \ldots x_j \ (\tau(t_1) \land \ldots \land \tau(t_k) \land \tau(e) \supset f \not\in \tau(t_1 \ldots t_k)), \quad \text{(3.1)}
\]

In the most common case, where each argument term $t_i$ is composed of constructors for its type and each of these constructors belongs to $S$ (i.e., they are assumed to be total functions), formula 3.1 simplifies to

\[
\forall x_1 \ldots x_j \ (\tau(e) \supset f \not\in \tau(t_1 \ldots t_k))
\]

This is the case for all of the CCT examples we present in this thesis.

For example, for the following axioms for an exponentiation function with a natural number exponent $\mathbf{n}$,
(forall ?x
   (= (power ?x zero) one))

(forall ?x ?n
   (= (power ?x (succ ?n))
      (Times ?x (power ?x ?n))))

we get the following termination axioms:

(forall ?x
   (power_t ?x zero))

(forall ?x ?n
   (if (and (power_t ?x ?n)
             (Times_t ?x (power ?x ?n)))
       (power_t ?x (succ ?n))))

Or, if we assume Times is a total function, the second termination axiom would simply be

(forall ?x ?n
   (if (power_t ?x ?n)
       (power_t ?x (succ ?n))))

For defining the termination axiom for a conditional axiom we also need to define a function $T$ similar to $\tau$ but for application to propositions instead of terms: for any proposition $P$, $T(P)$ asserts that computation of $P$ terminates. Again, the definition is by induction on the structure of $P$.

- If $P$ is a constant or variable, $T(P)$ is true.
- If $P$ is a predicate function application, $p(t_1\ldots t_m)$, where $p \not\in S$, then $T(P)$ is the conjunction

$$\tau(t_1) \land \ldots \land \tau(t_k) \land p_t(t_1\ldots t_m),$$

where $p_t : d_1 \times \ldots \times d_m \to \text{Boolean}$ is a new predicate symbol.
- If $p \in S$, the $T(p(t_1\ldots t_k))$ is just $\tau(t_1) \land \ldots \land \tau(t_k)$.  

• If $P$ is $P_1 \circ P_2$, where $\circ$ is one of the logic connectives $\land$, $\lor$, $\supset$ or $\equiv$, then $T(P)$ is the conjunction $T(P_1) \land T(P_2) \land P$.

• If $P$ is $\neg Q$, then $T(P)$ is $T(Q) \land P$.

• If any subexpression of $T(P)$ has the form $Q \land \text{true}$ or $\text{true} \land Q$, replace that subexpression by $Q$.

Then the termination axiom corresponding to the conditional equation

$$\forall x_1 \ldots x_j \ (p \supset f(t_1 \ldots t_k) = c)$$

is, in the most general case,

$$\forall x_1 \ldots x_j \ (T(p) \land \tau(t_1) \land \ldots \land \tau(t_k) \land \tau(e) \supset f \_\_ (t_1 \ldots t_k)), \quad (3.2)$$

but under the same assumptions as before about constructors, formula 3.2 simplifies to

$$\forall x_1 \ldots x_j \ (T(p) \land \tau(e) \supset f \_\_ (t_1 \ldots t_k)) .$$

For example, suppose now that the type of the variable $?n$ in the exponentiation function introduced earlier is integer rather than natural number, and we have these axioms:

(forall ?x
  (= (power ?x zero) one))

(forall ?x ?n
  (if (>= ?n zero)
      (= (power ?x (succ ?n))
        (Times ?x (power ?x ?n)))))

From the second, conditional axiom we obtain (assuming as before that Times is total) the following conditional termination axiom:
Although in some cases we will have conditional termination axioms with opposite conditions and thus can prove totality of the function, here we only have termination for nonnegative integers. To maintain our stance that we only allow definitions of total functions, we view this in terms of functions that are total on a restricted domain, a subset of the whole domain which is well-defined by a predicate. In other terms, we say that \((\geq ?n \text{ zero})\) is a precondition for termination, and we will allow such partial functions to be used in CCT as long as we can prove that the precondition for termination is satisfied in every call of the function. This point is discussed further in Section 5.3.

The next step is to construct a proof of the termination condition, after asserting the termination axioms into the assumption base. If the producer cannot construct a proof of the termination condition, the function definition might not terminate for all inputs, and it must be revised into a definition for which a proof of its termination condition can be constructed.

A natural question to ask is, why is it any safer to assert the termination axioms into the assumption base than it would be to go ahead and assert the original axioms? Why couldn’t the termination axioms themselves be inconsistent? To see why they can’t be, first note their form: under the given assumptions about the form of the original axioms, the termination axioms are horn clauses, i.e., of the form

\[
\forall x_1 \ldots x_n \ (p_1 \land \ldots \land p_m \supset f_t(x_1 \ldots x_n))
\]

where the consequent of the implication is a positive literal. From the conjunction of horn clauses, it is impossible to derive false and thus impossible to derive a contradiction. The only other possibility is that a pre-existing axiom could be combined with these horn clauses to derive false, but that is ruled out by the fact that \(f\), and therefore \(f_t\), is a new symbol and thus there can be no interaction with pre-existing axioms.
Thus, if the definition of \( f \) does not terminate then we wouldn’t be able to prove the TC but there still wouldn’t be any way of deriving false from the termination axioms. Just consider the single axiom \( \forall x (f(x) = f(x)) \), from which we get the termination axiom \( \forall x (f \downarrow(x) \supset f \downarrow(x)) \) (which is tautological and thus useless for proving anything else). We can’t prove the termination condition \( \forall x f \downarrow(x) \), but there is nothing falsifying it either. To be able to falsify it we’d have to introduce additional, non horn-clause axioms for the purpose of proving nontermination. But even in cases like this where we can see that a function as defined would be nonterminating, we do not need to prove that it’s nonterminating; we just need to replace the original axioms with ones that do define a terminating function and from them—that is, from the corresponding termination axioms—prove termination.

3.1.4 Step 4: Proving the Additional Consistency Condition (Code Producer)

We also wrote a program called CCGEN which takes a list of function-defining axioms and produces a predicate that we call the (additional) consistency condition (CC). As with the termination condition, the producer must construct a proof of this condition.

CCGEN introduces new variables \( w_1 \ldots w_k \) to transform an equational axiom

\[
\forall x_1 \ldots x_j (f(t_1 \ldots t_k) = e),
\]

into an equivalent proposition in which the application of \( f \) on the left hand side is only to variable arguments:

\[
\forall x_1 \ldots x_j w_1 \ldots w_k \ (w_1 = t_1 \land \ldots \land w_k = t_k \supset f(w_1 \ldots w_k) = e).
\]

A conditional equational axiom

\[
\forall x_1 \ldots x_j \ (p \supset f(t_1 \ldots t_k) = e)
\]
is similarly transformed into
\[
\forall x_1 \ldots x_j w_1 \ldots w_k (w_1 = t_1 \land \ldots \land w_k = t_k \land p \supset f(w_1 \ldots w_k) = e).
\]

Having thus normalized the left hand side of the equation in each axiom, we can uniformly replace \( f(w_1 \ldots w_k) \) by an existentially quantified variable \( y \) and combine the resulting propositions into the following single proposition:
\[
\forall x_1 \ldots x_j w_1 \ldots w_k \exists y (\,
(w_1 = t_{11} \land \ldots \land w_k = t_{1k} \land p_1 \supset y = e_1) \land
(w_1 = t_{21} \land \ldots \land w_k = t_{2k} \land p_2 \supset y = e_2) \land \\
\ldots \land
(w_1 = t_{n1} \land \ldots \land w_k = t_{nk} \land p_n \supset y = e_n))
\]

where \( \land p_i \) is not present when the \( i \)-th axiom is unconditional. This is the additional consistency condition produced by CCGEN, except that it is simplified when an argument term \( t_{ij} \) is a variable (it must be one of the variables \( x_1 \ldots x_j \)), since no new variable needs to be introduced for it, and thus no corresponding equation either; other simplifications are done by renaming variables.

This condition expresses that it’s possible to define a function that satisfies the conditions in the axioms, because it says that corresponding to every sequence of values \( x_1 \ldots x_k \) in the function domain there exists a range value \( y \) that we can associate with \( x_1 \ldots x_k \) to form a function (considered as a set of ordered tuples \((x_1 \ldots x_k, y)\)). That is, the axioms are not so strong that they rule out the existence of a function satisfying them.

As a simple example, suppose we define a function \( f \) of two arguments by the following two axioms:
\[
\text{(forall \( \forall x \ (= (f \ ?x \ zero) \ one)))}
\]

\[
\text{(forall \( x \ ?y \ (if \ (not \ (= \ ?y \ zero)) \ (= (f \ ?x \ ?y) \ two)))}
\]

The first axiom is transformed into
\[
(f \forall x \ w (\text{if } (= w \ 0) \\
\quad (= (f \ x \ w) \ 1)))
\]

and the second (just by renaming \(y\) to \(w\)) into

\[
(f \forall x \ w (\text{if } (\neg (= w \ 0)) \\
\quad (= (f \ x \ w) \ 2)))
\]

Then \((f \ ?x \ ?w)\) is replaced by \(y\) and the consistency condition is formed as

\[
(f \forall x \ w (\exists y \ (\text{and } (\text{if } (= w \ 0) \\
\quad (= y \ 1)) \\
\quad (\text{if } (\neg (= w \ 0)) \\
\quad (= y \ 2)))))
\]

The proof of this consistency condition is just a simple case analysis, and even for more complex sets of axioms the consistency condition proof is usually straightforward (but tedious) to construct. For these proofs in our CCT experiments to date, we have relaxed our requirement of proofs that can be checked within ATHENA’s own inference system; instead, we fall back on ATHENA’s ability to dispatch propositions to an external automatic resolution prover, SPASS or VAMPIRE. To enable the consumer to entirely avoid using this capability, the producer would have to construct ATHENA-only proofs, but the assumption that the consumer could use SPASS or VAMPIRE just for this purpose is not as big an assumption as it would be to rely on such resolution provers to do more complex proofs.

Both the termination and additional consistency proofs have to be done before proceeding with the next step. If they cannot be done then the producer has to return to Step 2 (Section 3.1.2) and revise the function-defining axioms or write new ones. But if both of these conditions are proved then the new axioms can be asserted and used safely.

### 3.1.5 Step 5: Proving Correctness (Code Producer)

Even the consistency and termination requirements are satisfied, the new definitions will not be accepted by the consumer without a correctness proof. The code producer has to convince the consumer that the new function satisfies the correctness conditions that the consumer requires.
In this step, after asserting the function-defining axioms, the producer attempts to construct a proof of each of the required correctness conditions using those axioms, along with any axioms and theorems in the theory library that are known by the producer to be available also in the consumer’s theory library. Sometimes the producer needs to state some intermediate properties as lemmas and prove them too before proving the correctness condition. In this case he has to send the consumer the proofs of these lemmas too.

Once all the proofs are ready, the producer can send them to the consumer along with the function-defining axioms and proofs of the termination and additional consistency condition. Thus the producer does not send actual code but the theory which carries it.

3.1.6 Step 6: Termination Checking (Code Consumer)

On receiving the theory and proofs for a new function the consumer first runs TCGEN to obtain the termination condition (TC). Since both the consumer and producer use TCGEN to get this condition, the termination proof sent by the producer must prove it, unless it has been corrupted in transmission or hacked by an intruder, in which case the consumer will reject the theory immediately, not allowing code to be generated from it. If the termination proof checks, the consumer proceeds to the next step.

3.1.7 Step 7: Additional Consistency Checking (Code Consumer)

The consumer next runs CCGEN and recreates the same additional consistency condition as the producer did; again, unless it has been changed in transmission, the consistency proof provided by the producer will prove this condition. (In our current CCT experiments, the transmitted “proof” just takes the form of a single command to the external SPASS prover to automatically search for the proof.) Otherwise, the function-defining axioms will be rejected.

3.1.8 Step 8: Checking Correctness (Code Consumer)

If we reach this point it is safe to assert the new function-defining axioms into ATHENA’s assumption base. The consumer then checks the proofs supplied by the
producer of the required correctness properties. If each such proof is accepted, the next and final step of generating code from the axioms can be carried out.

Note that proving theorems (manually or by automated search) is generally much harder and more time-consuming than checking a given proof. In CCT, generally the producer is responsible for providing the proofs whereas the code consumer is only responsible for checking them. If the producer sends all the proofs as manual proofs (Athena-only, with no use of an external automated prover), this reduces the consumer’s workload. Also the trusted computing base is smaller since the consumer relies on less software (no need for an automated theorem prover). On the other hand, if the consumer has enough resources, some simple properties like CC and TC can be proved automatically with an external automated prover like SPASS [33]. Our experience shows that proving correctness conditions is usually harder than proving the termination or additional consistency conditions. Therefore, we believe the producer should always provide manual proofs for the correctness conditions; for TC’s and CC’s, we leave it as optional to use automation since their proofs can be found fairly quickly anyway.

3.1.9 Step 9: Code Extraction (Code Consumer)

The consumer now knows that the new definitions are acceptable. The form of the function-defining axioms is such that it is easy to extract executable code from them. In general the axioms and theorems are in the form of universally quantified, and possibly conditional, equations. Each axiom is used in extraction of one line of code. The left hand side of each equation is an expression in which the function is applied to arguments of a specific form. The code extractor generates cases by checking how arguments are structured or if they have any symbolic patterns. The right hand side of equations represents the expression to be returned. These expressions may contain calls to other recursive functions. If there are any conditional axioms, CODEGEN first checks different argument patterns and creates cases. After this step, conditions are translated into “if-then” expressions accordingly. When there are two if-then expressions with opposite conditions, they are combined into one “if-then-else” expression.
The target language of CodeGen is currently Oz [31]. Many programming languages support one particular paradigm while Oz is a multiparadigm language that supports multiple programming paradigms, including object-oriented, functional, logic, concurrent, and distributed programming. Once developers learn how to program in Oz, they can decide what paradigm they want to use and build code accordingly without worrying about different language syntax and semantics supporting the paradigm in which they are interested. Oz is a higher-order language which also distinguishes between stateful and stateless data. Variables are stateless whereas cells are stateful. Explicit states can be defined through the use of cells which are updatable. Lists are provided as a primitive data type. The empty list is represented with nil. To cons (as in Lisp) a new element onto an existing list, the | operator is used, as in 5|[6 7 8]. Pattern matching can be done with the use of a case statement and by decomposing its expression according to the patterns defined with | in the branches. In addition, Oz has an optimization called tail-call optimization that enables it to execute tail-recursive functions in constant stack size, so that tail-recursive functions are as memory-efficient as while-loops in other languages. In CCT, if the producer writes axioms that can be used in the extraction of tail-recursive functions, he obtains the benefit of this optimization.

It does not appear that it would be difficult to retarget CodeGen to another language that supports tail-call optimization, such as Scheme; it would mainly require generating code to do the pattern matching for which Oz has native support. Retargetting to a language lacking tail-call optimization, such as Common Lisp or C, would require additional extensions to generate iterative loops rather than tail-recursive functions. We discuss this issue further in relation to memory-updating functions in Section 5.5.
Figure 3.2: Code-Carrying Theory, Tampered-With But Checked and Prevented from Acceptance (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)

3.2 Ensuring correctness

In CCT, both the producer and the consumer have to run CCGen and TCGen to get the same conditions. Therefore, there is no way of changing the verification conditions (CC or TC). Nevertheless, an intruder might change their proofs during transmission. Figure 3.2 shows this scenario where correctness is still guaranteed since either the proofs will not go through or are valid but not the proofs of the verification conditions generated by CCGen or TCGen. Similarly, if an intruder attempts to change the proof of a correctness condition, the axioms will be rejected if the proof does not check or is not the proof of the required correctness conditions (which are decided on beforehand by the consumer).

Figure 3.3 illustrates another scenario where an intruder changes the function-
Figure 3.3: Code-Carrying Theory, Tampered-With But Checked and Prevented from Acceptance (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)

defining axioms. In this case, the original TC, CC, and correctness conditions all have to be proved by the intruder. If any of them has not been proved, the axioms will be rejected. If the proofs are provided, then there is no danger in using the axioms introduced by the intruder, at least in terms of correctness. In another scenario, an intruder might change the application-specific requirements during their transmission from the consumer to the producer. Again, the producer might be able to prove all verification conditions. But since the consumer has the

---

8One potential kind of attack is to degrade the computing-time or memory performance of the delivered code through substitution of axioms that compute the same function but with a much less efficient algorithm. To protect against such an attack, the set of requirements established by the consumer would have to include some form of computing-time and memory bounds, but that is an extension we leave for future work.
Figure 3.4: Code-Carrying Theory, Tampered-With But Checked and either Prevented from Acceptance or Guaranteed Correctness (CC stands for Additional Consistency Condition, TC is Termination Condition, CTC is Correctness Condition, and FDA is Function Defining Axioms)

original requirements, not the modified ones, the proofs of the modified requirements will generally not serve to prove the requirements held by the consumer.

Figure 3.4 shows the last scenario. In this case, an intruder changes both the axioms and the proofs. Since the axioms are different, CCGen and TCGen will generate a TC and a CC that differ from the ones generated by the producer. If the proofs are corrupted and do not prove the new CC and TC, the new axioms will be rejected and will not be asserted by the consumer. If the changed proofs are valid, however, and prove both the new TC and CC, the consumer can assert the changed axioms safely (i.e., without fear of introducing an inconsistency in the assumption base), and proceed to check the proof of the correctness condition. If this check is successful too, correctness is guaranteed and the consumer can use the axioms in
code extraction even if they are not the ones provided by the producer. Otherwise, the changed axioms will be rejected.

We also have not formally verified our implementation of CCT (TCGen, CC-Gen and CodeGen) but we note that the total amount of Athena code is less than 1000 lines, a size much smaller than has been reported for the VCGen program (23,000 lines) in type-specialized PCC [1], making it much more amenable to manual inspection.
CHAPTER 4
Case Study: CCT Steps for a Nongeneric Memory-Observing Function

For a first illustration of the CCT approach, we present in this chapter a case study in which a small theory is developed from which a memory-observing function for summing natural numbers in a list is extracted. In the next chapter we present an example involving a theory of sequence reversal from which an in-place (memory-updating) reversal function is extracted. To simplify the discussion, we are presenting in these chapters versions of these functions that are nongeneric. In Section 6.1, we return to the summation example to show how it can be redeveloped as a generic function, in that the list elements may belong to any type that models a Monoid theory (i.e., having an associative binary operation and an identity element). In the case of the reversal function, we will see in Section 6.2 that it can be redeveloped as a generic function applicable even to different representations of sequences, resulting in a version with the same level of genericity as one finds in the C++ Standard Template Library.

Here we use lists which can be introduced with List-Of data type whose definition in Athena is given as follows:

\[(\text{datatype} (\text{List-Of} \ T) \ \text{Nil} \ (\text{Cons} \ T \ (\text{List-Of} \ T)))\]

where \(T\) is a type variable that can be instantiated with any type. It has two constructors: Nil, which denotes an empty list, and Cons, which represents a function that puts an element in the beginning of a list. Interested readers can find more information about lists in Appendix B.

- **Step 1: Defining Requirements:** The consumer declares an operator \texttt{sum-list@} and defines it by axioms, as the specification of the function, and sends them to the producer.\(^9\)

\(^9\)In this case, the specification itself is executable; i.e., code-extraction could be applied to it though the function that would result would not be optimally efficient since it would use embedded
(declare sum-list@ (-> ((List-Of Nat)) Nat))

(define sum-list-empty-axiom
  (= (sum-list@ Nil) zero))

(define sum-list-nonempty-axiom
  (forall ?L ?x
    (= (sum-list@ (Cons ?x ?L))
      (Plus ?x (sum-list@ ?L)))))

The consumer also declares the function to be defined by the producer and states the correctness condition that this function must satisfy:

(declare sum-list (-> ((List-Of Nat)) Nat))

(define sum-list-correctness
  (forall ?L
    (= (sum-list ?L)
      (sum-list@ ?L))))

• **Step 2: Defining the New Function Axiomatically:** The producer starts with defining sum-list with the intension of using it to obtain a more efficient way of computing the same function as sum-list@.

(define sum-list-empty
  (= (sum-list Nil)
    zero))

(define sum-list-nonempty
  (forall ?L ?x
    (= (sum-list (Cons ?x ?L))
      (sum-list-compute ?L ?x))))

(define sum-list-axioms [sum-list-empty sum-list-nonempty])

recursion. In general, requirements specifications defined by the consumer do not need to be executable, as illustrated by the example of Section 5.3.
The producer constructs `sum-list-nonempty` axiom in a way that the `sum-list` function makes a call to a tail-recursive function called `sum-list-compute` which is first declared and then defined axiomatically as follows:

```
(declare sum-list-compute (-> ((List-Of Nat) Nat) Nat))

(define sum-list-compute-empty
  (forall ?x
    (= (sum-list-compute Nil ?x) ?x)))

(define sum-list-compute-nonempty
  (forall ?L ?x ?y
    (= (sum-list-compute (Cons ?y ?L) ?x)
        (sum-list-compute ?L (Plus ?x ?y))))))

define sum-list-compute-axioms
  [sum-list-compute-empty sum-list-compute-nonempty])
```

**Step 3: Proving Termination:** Termination must be proved for each of the two implementing functions, `sum-list` and `sum-list-compute`. The code producer first declares a new predicate function symbol, `sum-list-compute_t`, in order to express whether `sum-list-compute` terminates or not:

```
(declare sum-list-compute_t (-> ((List-of Nat) Nat) Boolean))
```

The termination condition (TC) produced by TCGEN from `sum-list-compute-axioms` is

```
(forall ?L ?x
  (sum-list-compute_t ?L ?x))
```

and the termination axioms, as produced by applying a function we call `termination-axiom` to each member of `sum-list-compute-axioms`, are
(forall ?x
  (sum-list-compute_t Nil ?x))

(forall ?L ?x ?y
  (if (sum-list-compute_t ?L (Plus ?x ?y))
      (sum-list-compute_t (Cons ?y ?L) ?x)))

In computing these propositions, termination-axiom treats certain function symbols, in this case Plus and Cons, as known to be terminating, as determined by entries in a predefined but augmentable table.

After asserting these termination axioms into the assumption base, the code producer must then construct a proof of the termination condition, which in this case is a simple induction proof based on cases defined by ATHENA from the form of the List-Of datatype:

(by-induction
  sum-list-compute-termination
  (Nil
    (pick-any x
      (!uspec sum-list-compute_t-empty x)))
  ((Cons y L)
    (dlet ((induction-hypothesis
      (forall ?x (sum-list-compute_t L ?x))))
      (pick-any x
        (!mp (!uspec* sum-list-compute_t-nonempty [L x y])
          (!uspec induction-hypothesis (Plus x y)))))))

ATHENA provides the built-in command by-induction which can be used if the type of the first quantified variable of the predicate to be proved is defined with datatype. There are two cases in the above example. For the basis case the given predicate is implicitly instantiated with Nil, producing (forall ?x (sum-list-compute_t Nil ?x)). The proof of this case then consists simply of using pick-any to name an arbitrarily chosen value x and instantiating the first termination axiom with x using uspec (universal specialization).
To prove the induction step, the predicate is instantiated with \((\text{Cons } y \ L)\), producing

\[(\forall x (\text{sum-list-compute}_t \ (\text{Cons } y \ L) \ x)).\]

While this proof is in progress \textsc{Athena} automatically constructs and enters into its assumption base the proposition

\[(\forall x (\text{sum-list-compute}_t \ L \ x)).\]

This is the induction hypothesis, which we identify clearly in the proof by giving a name to it with \texttt{dlet} (which is \textsc{Athena}'s deductive-language counterpart of traditional \texttt{let} constructs in functional languages). The proof continues with another use of \texttt{pick-any} to choose an arbitrary value \(x\) (which, though it has the same name as in the first \texttt{pick-any}, is independent under \textsc{Athena}'s scope rules). There are two specializations here. For the first, we use the method call

\[(\texttt{uspec}^* P [t_1 \ldots t_n])\]

which replaces the first \(n\) quantified variables in \(P\) with the terms \(t_1, \ldots, t_n\), to specialize \texttt{sum-list-compute}_t \texttt{-nonempty}. This produces an implication

\[(\text{if } (\text{sum-list-compute}_t \ L \ (\text{Plus } x \ y))
 (\text{sum-list-compute}_t \ (\text{Cons } y \ L) \ x)),\]

which has the desired conclusion, so we only need to discharge the assumption \((\text{sum-list-compute}_t \ L \ (\text{Plus } x \ y))\). We do so with the second specialization and combine it with the implication to obtain the conclusion using another of \textsc{Athena}'s primitive inferences, \texttt{mp} (\textit{modus ponens}).

Similarly, for proving termination of \texttt{sum-list} itself, the termination axioms are

\[(\text{sum-list}_t \ \text{Nil}))\]

\[(\forall \ L \ x
 (\text{if } (\text{sum-list-compute}_t \ ?L \ ?x)
 (\text{sum-list}_t \ (\text{Cons } ?x \ ?L))))\]
and the termination condition and its proof are

\[
\text{(define sum-list-termination}
\text{(forall ?L}
\text{ (sum-list_t ?L)))}
\text{(by-induction}
\text{ sum-list-termination}
\text{(Nil (!claim sum-list_t-empty))}
\text{((Cons y L)
\text{ (!mp (!uspec* sum-list_t-nonempty [L y])
\text{ (!uspec* sum-list-compute-termination [L y])))))}
\]

where we use the already established \text{sum-list-compute-termination} property in the proof of the inductive step.

- **Step 4: Proving Consistency**: For the proof of the (additional) consistency condition for \text{sum-list-compute}, application of CCGEN to the \text{sum-list-compute} axioms produces

\[
\text{(forall ?y ?L ?x ?L2}
\text{ (exists ?result}
\text{ (and (if (= ?L2 (Cons ?y ?L))}
\text{ (= ?result}
\text{ (sum-list-compute ?L (Plus ?x ?y))))}
\text{ (if (= ?L2 Nil)
\text{ (= ?result ?x))))})
\]

This condition is easily proved with a call to the external automated prover SPASS.

\[
(!\text{prove-from} \ (\text{consistency-condition sum-list-compute-rules})
\text{(datatype-axioms "List-0f"))}
\]

SPASS carries out this proof using only the predefined axioms for the \text{List-0f} data type.

Similarly, for \text{sum-list} the condition and its proof are as follows:
(define sum-list-consistency
  (forall ?y ?L ?x ?L2
    (exists ?result
      (and (if (= ?L2 (Cons ?y ?L))
        (= ?result (sum-list-compute ?L ?x)))
      (if (= ?L2 Nil)
        (= ?result zero)))))

(!prove-from (consistency-condition sum-list-rules)
  (datatype-axioms "List-Of")

• Step 5: Proving Correctness: Having proved both termination and the additional consistency condition for each of the implementation functions, their axioms can now be asserted into the assumption base.

  (assert sum-list-compute-axioms)
  (assert sum-list-axioms)

These axioms are now available for proving the consumer’s required correctness condition, as stated in Step 1.

(define sum-list-correctness
  (forall ?L
    (= (sum-list ?L)
      (sum-list@ ?L))))

The proof can be programmed by the producer as follows.

(by-induction
  sum-list-correctness
  (Nil
    (dseq
      (!setup left (sum-list Nil))
      (!setup right (sum-list@ Nil))
      (!reduce left zero sum-list-empty)
      (!reduce right zero sum-list@-empty)
      (!combine left right))))
This proof is done by a combination of induction and application of rewriting steps. The sum-list-correctness property is a quantified equation, and in this example each case of the inductive proof is an unquantified equation. The main technique for proof of equational propositions is rewriting, in which one performs a series of rewrites — applications of the substitution property of equality — to the term on the left hand side of the equation and another series of rewrites to the right hand side until obtaining two terms that are syntactically identical. We implemented a simple form of rewriting by defining in ATHENA’s method language the method setup for isolating the left and right hand side terms of an equation, reduce and expand for performing a single rewriting step, and combine for deducing the original equation once the left and right terms have been rewritten to the same term. These methods are described more fully in Appendix A.4, but we note here that they are defined as compositions of ATHENA’s primitive inferences; they do not require assuming any new axioms.

In the proof of the inductive step one of the rewriting steps applies a lemma,
with the aid of lemmas. Once a lemma’s proof has been constructed, it is available for use whenever the lemma is needed in any other proof. Factoring of proofs via lemmas is important in reducing the size of proofs that must be transmitted between producer and consumer.

After completing this proof, the producer can send the consumer the axioms defining `sum-list` and `sum-list-compute`, and the proofs of the termination, additional consistency, and correctness conditions, including proofs of any lemmas used if they are not already in the consumer’s proof library.

- **Step 6: Termination Checking:** The consumer runs TCGen and generates the same TCs and termination rules shown at Step 3. He first asserts the termination rules. He then enters their proofs into ATHENA, and since the proofs check the consumer will go on to the next step.

- **Step 7: Consistency Checking:** The consumer first runs CCGEN and generates the same CC shown at Step 4 after receiving the theorems and proofs from the producer. As in the previous step he enters the proof of the TC and ATHENA accepts it.

- **Step 8: Proving Correctness:** Since the termination and additional consistency checks succeeded the consumer asserts the function-defining axioms first. He then checks the proof of the correctness condition given in Step 5.

- **Step 9: Code Extraction:** The consumer passes each list of axioms defining `sum-list-compute` and `sum-list` separately to CODEGEN which generates the following two Oz functions:
fun {SumListCompute L X}
  case L
  of nil then X
  [\ Y|L then {SumListCompute L (X + Y)}
  end
end

fun {SumList L}
  case L
  of nil then 0
  [\ X|L then {SumListCompute L X}
  end
end

These functions are efficient, because the first is tail recursive and thus is
equivalent to a definition by a while loop rather than by embedded recursion.
The second function does not have any recursion at all. In the body of ex-
tracted functions we use a case statement to do pattern-matching. This Oz
statement has the form

  case E
  of Pattern1 then S1
  [] Pattern2 then S2
  [] ...
  else S end

where E is an expression. If the result of evaluation of E matches one of the
patterns then the corresponding statement is executed. Both SumList and
SumListCompute have two case branches. L is a list which can be either nil
or a nonempty list, which can be decomposed as the head of the list and the
rest of it (tail). In the SumList function, case decomposes list L according to
the pattern X|L in the second branch, where X is the first element of L, and
the new L is a list that contains the rest of the elements in original L.

A code generation issue is whether recursion should be eliminated in favor of an
imperative style of loops with assignment statements. What we are relying on with
the current extractor is that transformation to the imperative style will be done by
the target-language processor, at least for tail-recursive functions. This assumption
is true of all Scheme interpreters and compilers (since the Scheme standard man-
dates it), and really should be true of more mainstream language implementations.
Unfortunately it is not required by the C or C++ language standards, but Bailey
and Weston [7] have shown that transformation of tail-recursive C or C++ func-
tions to imperative style is both worthwhile and reasonably simple to carry out, so
perhaps it will someday be required by the languages’ standards. In the meantime,
if the extractor targets a language that does not require tail-recursion removal, it
should be extended to do that job itself.

We have developed a similar example in which the sequence of values to be
summed is accessed through a pair of STL-like iterators that form a valid range, i.e.,
(range i j) such that an n-fold application of the increment operation ++ to i is
equal to j, for some natural number n. It is assumed that a dereference operation,
(* i), yields a memory location, whose contents are then obtainable with an Access
operation: (Access M (⋆ i)) where M represents the memory. We omit the details
of this sum-range example here, but the way in which iterators and ranges are used
is amply illustrated in the memory-updating example in the next chapter.
In this chapter, we introduce `reverse-range`, an in-place reversal operation defined on a valid range. Besides the introduction of memory-updating, this example also shows the use of conditional equations, expressed with ATHENA logical implications (if \( P \rightarrow Q \)).

We will first create a memory model in ATHENA before showing how the CCT approach is applied to this nongeneric `reverse-range` function.

### 5.1 Nongeneric definition of memory

In CCT, the code consumer decides requirements and expresses them formally in ATHENA. In most cases, these requirements contain specifications of memory operations. Similarly the code producer may prove new theorems about some operational behavior of memory. In order for this to be possible, we defined a memory model as follows:

- Memory is introduced as a new domain and is composed of cells whose locations are unique.

\[
\text{(domain Memory)}
\]

\[
\text{(domain (Loc T))}
\]

Note the type parameter \( T \). We are actually doing things in this discussion generically to the extent that we do have polymorphic types like \( \text{Loc} \) and polymorphic functions like \( \text{Access} \) (and `reverse-range` for that matter), but not in the full sense of genericity that is presented later in Chapter 6.

- \( \text{(Access } M \ i \text{)} \) returns the content of memory location \( i \) in memory \( M \).
• \((\text{Assign } M \ i \ x)\) updates the content of memory location \(i\) in memory \(M\) with value \(x\) and returns the updated memory. Note that \text{Assign} returns a new memory.

• \((\text{Swap } M \ i \ j)\) exchanges the content of memory location \(i\) in memory \(M\) with the content of memory location \(j\) and returns the updated memory.

Here is how we define these operators in CCT:

\[
\text{(declare Access ((T) ->)} \\
\quad \text{(Memory (Loc T))} \\
\quad \text{T))}
\]

\[
\text{(declare Assign ((T) ->)} \\
\quad \text{(Memory (Loc T) T)} \\
\quad \text{Memory))}
\]

\[
\text{(declare Swap ((T) ->)} \\
\quad \text{(Memory (Loc T) (Loc T))} \\
\quad \text{Memory))}
\]

In addition to these definitions, the following axioms are used in the model:

• Axiom 1: Accessing the content of a memory location \(i\) after updating its content with some value \(x\) results in the same value.

\[
\text{(forall ?M ?i ?x} \\
\quad (= (Access (Assign ?M ?i ?x) ?i) \\
\quad \quad ?x))
\]

• Axiom 2: If \(i\) and \(j\) are two distinct memory locations, updating the content of location \(i\) does not change the content of location \(j\).

\[
\text{(forall ?M ?i ?j ?x} \\
\quad \text{(if (not (= ?i ?j))} \\
\quad \quad (= (Access (Assign ?M ?i ?x) ?j) \\
\quad \quad \quad (Access ?M ?j))))
\]
• Axiom 3: The content of memory location \( i \) after exchanging its content with some memory location \( j \) is the value previously held by \( j \).

\[(\forall M \, i \, j \quad (= (\text{Access} \ (\text{Swap} \ M \ i \ j) \ i) \quad \text{(Access} \ M \ j)))\]

• Axiom 4: The content of memory location \( j \) after exchanging its content with some memory location \( i \) is the value previously held by \( i \).

\[(\forall M \, i \, j \quad (= (\text{Access} \ (\text{Swap} \ M \ i \ j) \ j) \quad \text{(Access} \ M \ i)))\]

• Axiom 5: If \( k \) is a location different from both \( i \) and \( j \), exchanging the contents of \( i \) and \( j \) does not change the content of \( k \).

\[(\forall M \, i \, j \, k \quad (\text{if} \ (\text{and} \ (\text{not} \ (= ?k \ i))) \quad \text{(not} \ (= ?k \ j)))\quad (= (\text{Access} \ (\text{Swap} \ M \ i \ j) \ k) \quad \text{(Access} \ M \ k))))\]

Moreover, we grouped these axioms and developed a Memory-theory and elaborated this theory by adding new theorems whose proofs are constructed in ATHENA manually. It is enough to use only this theory whenever a code consumer or a receiver needs to prove a new theorem based on any memory rules and even refine the theory with the new theorem. We also provide generic declarations and axiomatic definitions of these operators in Chapter 6.1.2.

### 5.2 Memory-range theory

There are two operations we defined for accessing and updating a range of elements pointed to by iterators, much as in the STL style.

• \((\text{Access-range} \ M \ (\text{range} \ i \ j))\) returns a list of the contents of memory locations in memory \( M \) in the range from \( i \) to \( j \).
• (Assign-range M i L) updates the contents of memory locations starting from i in memory M with the members of list L and returns the updated memory. The number of locations being updated after (and including) i is limited to the size of L. Note that Assign-range returns a new memory.

(declare Access-range ((T) ->
    (Memory (Range (Iterator T)))
    (List-Of T)))

(declare Assign-range ((T) ->
    (Memory (Iterator T) (List-Of T))
    Memory))

The axioms we assume for these functions are as follows:

• Access empty range axiom: Accessing the content of an empty range produces Nil.

    (forall ?M ?i
        (= (Access-range ?M (range ?i ?i))
            Nil))

• Access nonempty range axiom: The result of accessing a nonempty (range ?i ?j) in memory ?M is equal to the list constructed by prefixing the content of the memory location pointed to by iterator ?i to the list produced from the contents of the memory locations pointed to by the iterators in (range (++ ?i) ?j).

    (forall ?M ?i ?j
        (if (not (= ?i ?j))
            (= (Access-range ?M (range ?i ?j))
                (Cons (Access ?M (* ?i))
                    (Access-range ?M (range (++ ?i) ?j)))))

• Assign empty range axiom: Updating the content of memory locations starting from ?i with Nil results in the same memory.

    (forall ?M ?i
        (= (Assign-range ?M ?i Nil)
            ?M)))
• Assign nonempty range axiom: Updating the content of memory locations starting from \$i\$ with a nonempty list is equal to assigning the first element of the list to the memory location pointed to by \$i\$ and assigning the rest of the elements to the memory locations that start from the address pointed to by \((++\ i)\).

\[
\text{(forall } \forall M \ i \ ?L \ x \ \\
\text{ (\textbf{=} } (\text{Assign-range } M \ i \ (\text{Cons } x \ ?L)) \\
\text{ (Assign-range (Assign } M \ (\text{\* } i) \ x) (\text{++ } i) \ ?L)))
\]

5.3 Memory-updating operations and nongeneric reverse-range

The main property to be proved—the correctness condition—of reverse-range is the following relation to reverse (which operates on lists):

\[
\text{(define reverse-range-correctness} \\
\text{(forall } \forall i \ j \\
\text{ (if } (\text{valid (range } i \ j) \ \\
\text{ (forall } \forall M \ \\
\text{ (\textbf{=} } (\text{Access-range } M \ (\text{reverse-range } M \ (\text{range } i \ j)) \\
\text{ (range } i \ j)) \\
\text{ (\text{reverse (Access-range } M \ (\text{range } i \ j)))))))))
\]

This reverse-range-correctness property states that accessing the memory locations in a given range after reversing the content of memory locations in this range will return the same list of elements as if we first accessed the content of this range and applied reverse to the resulting list.

In fact, this property serves as the specification of the function to be expected from the producer. For this example, the consumer does not provide any new axioms like those used to define sum-list@ in the sum-list example. Note that reverse has already been defined and is part of the “general requirements” which should exist both at the producer and at the consumer. It takes a list and produces a new list with the same elements but with their order reversed. Appendix B contains more information on reverse.

Here is an axiomatic definition of this function constructed by the producer:
(define reverse-empty-range-axiom
  (forall ?i ?M
    (= (reverse-range ?M (range ?i ?i)) ?M)))

(define reverse-nonempty-range-axiom1
  (forall ?i ?j ?M
    (if (and (not (= ?i ?j))
            (= (++ ?i) ?j))
        (= (reverse-range ?M (range ?i ?j)) ?M))))

(define reverse-nonempty-range-axiom2
  (forall ?i ?j ?M
    (if (and (valid (range ?i ?j))
             (not (= ?i ?j))
             (not (= (++ ?i) ?j)))
        (= (reverse-range ?M (range ?i ?j))
            (reverse-range (swap ?M (* ?i) (* (-- ?j)) (range (++ ?i) (-- ?j)))))))

These axioms assume that Access and Swap return an entire memory as a value distinct from their memory argument. This semantics is the correct one for the classical first-order logic in which our proofs are done, where all functions are considered to be pure; we are not resorting to a dynamic logic. On the other hand, it is certainly not the semantics we want for the extracted code, where we want Assign and Swap to update a single memory. In section 5.5 we discuss how we deal with this semantic difference.

In the proof of reverse-range-correctness we needed and first proved the following lemma: that reverse-range has no effect on memory outside of the range to which it is applied:

(define reverse-range-has-no-effect-outside
  (forall ?i ?j
    (if (valid (range ?i ?j))
        (forall ?k ?M
          (= (reverse-range ?M (range ++ ?i)) ?M))))
(if (not (in ?k (range ?i ?j)))
  (= (Access (reverse-range ?M (range ?i ?j))
      (* ?k))
      (Access ?M (* ?k))))))))

The proof here is by range2-induction, a specialized induction schema we defined and proved within Athena, by a translation to applications of its by-induction method on the natural number data type. It has two basis cases, for an empty range and a range of one element, and an induction step from (range (++ i) (-- j)) to (range i j).\textsuperscript{10} Aside from making the proofs simpler and more readable, using range induction also helps in reducing the size of the proofs.

Extraction of Oz code from the axioms for reverse-range produces:

```oz
fun {ReverseRange M R }
    case R
    of range(I I) then M
     [] range(I J) then
      if {Not (I == J)} then
        if {('++' I} == J) then M
        else {ReverseRange {Swap M {'*' I} {'*' {'--' J}}} range({'++' I} {'--' J})} end
      end
    end
end
```

CodeGen provides a method called extract-code, which is used for code generation. We show below how to extract ReverseRange using extract-code.

\textsuperscript{10}We also developed a simpler range-induction method, with one basis case for an empty range and an induction step from (range (++ i) j) to (range i j). It is used in proofs about the sum-range function mentioned in Chapter 4 and about the find-if function in Section 7.2. Its definition is given in Appendix C.
(define reverse-range-axioms
  [reverse-empty-range-axiom
    reverse-nonempty-range-axiom1
    reverse-nonempty-range-axiom2])

(extract-code
  (reverse-range ?M ?r)
  reverse-range-axioms
  (valid (range ?i ?j))
  "reverse-range.oz")

extract-code prototype P fname
extracts a function in Oz whose header is constructed from the term given with prototype and body is constructed from a list of axioms, A, which define the function to be extracted. P is a precondition that must be satisfied whenever the function is called and which is used in eliminating checks of the condition from the extracted code. It is the predicate defining a restricted domain on which the function is defined (simply true for total functions). The function name and parameters given with prototype will be translated, if necessary, into proper Oz identifiers. Extracted code is written to the file specified with fname.

For the reverse-range example, the precondition is (valid (range ?i ?j)), which eliminates the valid range check from the last branch of the case statement. If the consumer passed true instead, CODEGEN would produce (Valid range(I J)) additionally combined with the other conditions given after the elseif part. Here this would cause a problem since there is no practical implementation of Valid. Elimination of the check in the generated code is safe as long as the code is only called in contexts where the precondition is satisfied.

5.4 Note on safety properties
Although we have not given special attention to safety properties, the theory of memory ranges, as local sections of a global memory delimited by iterators,
provides a basis for simple cases of proving memory safety properties. In fact, the astute reader may have noticed that the lemma proved in the previous section, `reverse-range-has-no-effect-outside`, is such a property (although our original motivation for stating and proving it was its essential role in establishing a functional correctness property).

5.5 Dealing with memory updates

Another important issue is how we deal with memory-updating. In order to generate efficient code from the `reverse-range` axioms (and for memory-updating functions in general), our original design for the code extractor called for it to carry out the conversion of tail-recursion into while loops with assignments, so that passing of a parameter like `(assign M i x)` for memory `M` would become explicit as an assignment statement `M ← (assign M i x)`, which could then be converted to a more efficient assignment statement `M[i] ← x`. In fact, however, the extractor can simply leave functions in tail-recursive form, even when memory-updating functions are involved. The key idea that makes things work out is that memory arguments can be passed by reference, thus not requiring expensive copying operations: when `(assign M i x)` is passed for `M`, the assignment `M ← (assign M i x)` is actually done, but it is just assigning to `M` a reference returned by the `assign` function, not a copy of the memory. The only real work that is done to update a memory location, `M[i] ← x`, is carried out within the `assign` operation itself. This idea works out readily in Oz where one can represent memory by an array and take advantage of the fact that arrays are passed (to functions like `Array.put` and `Array.get`) by reference.\footnote{Similarly, in Scheme one can represent memory by a `vector` and pass vectors (to functions like `vector-assign!` and `vector-ref`) by reference. Though `vector-assign!` does not return the reference, one can encapsulate it in an `assign!` function that does.}

Of course, this pass-by-reference semantics in the target programming language differs fundamentally from the pass-by-value semantics assumed in first or-
der logic with equality (as reflected in its equality substitution proof rules), so we have to be careful to limit the contexts in which pass-by-reference is used to those where the difference cannot be detected by subsequent operations. Principally this means that a memory variable should not appear more than once in a term if it occurs as argument to an \texttt{assign} or \texttt{swap} call within the term. For example, 

\begin{verbatim}
(f (Assign M i x) (Access M))
\end{verbatim}

is ruled out since in the copy-semantics of the logic, the change in memory effected by the first argument (assuming it is evaluated first) is not seen when the second argument is evaluated, but it is seen assuming reference semantics.
CHAPTER 6
Generic Proof Writing

The sum-list function presented in Chapter 4 can only be used for summing natural numbers in a list. With the way it is defined, the operands of the Plus operator in the axiomatic definitions cannot have any domain types other than Nat. If one intends to use the same operator with another numeric type such as integers, another operator has to be defined since ATHENA does not support operator overloading. Similarly one may want to assign a completely different meaning like concatenation of strings to the Plus operator. In all these cases we need different operators and have to define the sum-list function with different axioms. Most importantly, similar proofs have to be redeveloped each time, which requires more space to store them and is a time-consuming process.

The key to avoiding such a proliferation of functions and proofs is to define and use them in their most abstract, or generic, form; that is, in terms of operations and constants that satisfy minimal requirements for correctness. For sum-list, the minimal requirements are that Plus is an associative operator and there is a constant Zero that serves as its identity element, which are exactly the requirements of the familiar algebraic structure, or theory, monoid. The natural numbers form a monoid under addition with the identity element 0. Similarly, multiplication of natural numbers with the identity element 1 models a monoid. The integers with addition or multiplication as the binary operation is another example. In the case of strings, concatenation is associative and the empty string serves as the identity element.

We are motivated, therefore, to parameterize functions like sum-list with a type parameter M and to require that any type instantiating M model a monoid. In some programming languages, such parameterization can be partially achieved with constructs such as function templates (as in C++) or generics (as in Ada or Java), and these constructs are of central importance in implementing generic libraries like the Standard Template Library. In current versions of these languages, however,
there is too little support for expressing requirements on type parameters; in particular there is no support for semantic requirements like the associativity property of \texttt{Plus}.

\textsc{Athena}'s programming and proof languages lack type parameterization; in fact they are untyped (although \textsc{Athena}'s proposition language is strongly typed), which might give the impression that they are totally unsuited for programming generic components. On the other hand, \textsc{Athena}'s languages are higher order: functions and methods take other functions or methods as parameters and return functions or methods, and it is this facility that we can take advantage of to express generic functions and proofs.

For example, we can parameterize the \texttt{sum-list} function with an operator mapping, the domain of which includes \texttt{Plus} and \texttt{Zero}, to obtain a generic function that can be instantiated with different operator mappings to produce the different instances already mentioned and many other instances. But we must also parameterize proofs about \texttt{sum-list} with the same operator mapping, and moreover, we must write the proofs in terms of the minimal properties upon which their inference steps depend.

In our design, generic property definitions and proofs are constructed in the form of programs that are parameterized with operator mappings. We use functions to pack together related axioms whereas proofs need to be defined with \textsc{Athena} methods which must contain a series of proper deductions. In both cases, functions which perform operator mappings must also be provided. If an execution of a proof method fails, an error condition results and nothing is asserted into \textsc{Athena}'s assumption base. Otherwise, the result is a theorem, the specialized result of instantiating the generic proof with a particular operator mapping, and \textsc{Athena} enters this theorem, like any other, into its assumption base.

More formally, we define a generic property function to be a function that contains a sequence of name/property pairs \((n_1 p_1) \ldots (n_k p_k)\), where a name is a quoted identifier and a property is a proposition that is parameterized by an operator map. The function takes as inputs a name \(n\) and an operator map \(m\) and returns

\footnotesize
12Java generics have limited support for expressing syntactic requirements via interface inheritance. For C++, a proposal \cite{konstantinou2003} for adding \textit{concepts} is under consideration, which would allow requirements on template parameters to be expressed, but only syntactic requirements.
the proposition $p_i$ for which $n = n_i$, instantiated with $m$. An error condition results if $n$ is not equal to any of the $n_i$ or if the instantiation fails (as may happen, for example, if $m$ does not have a symbol required by $p_i$ in its domain).

A *generic theorem* can then be defined as a generic property function that contains a single property, and for which there is an associated *generic proof*. A generic proof for a generic theorem $T$ is a proof method $M$ that contains a proof $P$, such that $M$ takes as inputs a name $n$ and an operator map $m$ and returns the proposition $p = (T n m)$, provided the instantiation of $P$ with $m$ proves $p$. An error condition results if $n$ is not the name held by $T$, or if instantiation or execution of the proof fails, or if the proof succeeds but proves a proposition different from $(T n m)$.

6.1 Case study: CCT steps for a generic memory-observing function

6.1.1 Definition of axioms with Athena functions

Figure 6.1 shows the structure of generic property definitions in CCT. There are three parts. The first part is the function header with two parameters, a property name and operator mapping. The second part is the place where the local declarations, if any, are done. Operator mappings, and some other bindings that will increase the readability of the definition and reduce its size, are all done in this part. And finally the last part is where we write the generic axioms or theorems with a match special form, described below.

We now show how to define a generic version of the *sum-list@* example and again step through the CCT protocol in this more general setting. The consumer starts with defining *sum-list@* with a generic property function as follows:
As an example of an operator mapping that could be passed for the `ops` parameter, we could define
(define (Monoid-ops op)
  (match op
    ('Plus Plus)
    ('Zero zero)))

The match special form has the general structure

\[
\text{match } E (\pi_1 E_1) \ldots (\pi_n E_n)
\]

It matches the patterns \(\pi_1, \pi_2, \ldots\) with the value of \(E\) until it finds a match. If any \(\pi_i\) matches then \(E_i\) is evaluated and its value is returned as the result. Execution halts with an error if no match is found.

Thus if \(op\) is \'Plus then Monoid-ops returns Plus, and it returns zero if \(op\) is \'Zero. Similarly we could define the following function to represent monoid operation Times with the identity element one.

\[
\text{(define (Times-ops op)}
\text{  (match op}
\text{    ('Plus Times)
\text{    ('Zero one)))}
\]

The next part of the sum-list@-definition is

\[
\text{(let ((Plus (ops 'Plus))}
\text{  (Zero (ops 'Zero)))}
\]

In ATHENA, let is used for local bindings as define is used for global ones. In the first line ops returns an operator symbol and the result is stored in the local variable Plus. The second activation of ops binds the identity element to Zero.

The next part matches the axiom name given with the name parameter. For example, if the name \'sum-list@-empty is passed it is stored in variable name and matched with the first axiom, resulting in

\[
(= (sum-list@ Nil)
    zero),
\]

while

\[
(\text{sum-list@-definition 'sum-list@-nonempty Monoid-ops})
\]

produces the second axiom
\[(\forall ?L \ ?y \\( \
(= \ (\text{sum-list}\_\@ \ (\text{Cons} \ ?y \ ?L)) \\)
(\text{Plus} \ ?y \ (\text{sum-list}\_\@ \ ?L))))).\]

Now suppose we want to produce axioms that define \text{sum-list}\_\@ as a function to compute the product of the natural numbers in a list. If the list is empty the result is \textit{one}. The instantiation

\[(\text{sum-list}\_\@\text{-definition }'\text{sum-list}\_\@\text{-empty Times-ops})\]

produces

\[(= \ (\text{sum-list}\_\@ \ \text{Nil}) \\)
\text{one},\]

and

\[(\text{sum-list}\_\@\text{-definition }'\text{sum-list}\_\@\text{-nonempty Times-ops})\]

produces

\[(\forall ?L \ ?y \\( \)
(= \ (\text{sum-list}\_\@ \ (\text{Cons} \ ?y \ ?L)) \\)
(\text{Times} \ ?y \ (\text{sum-list}\_\@ \ ?L))))\]

The results of these instantiations are all propositions that can, for example, be asserted into the assumption base and used in constructing proofs.

### 6.1.2 Organizing theorems and proofs

As we shall see, quite extensive theories (sets of axioms and theorems) and proofs need to be developed and made available for use in CCT. While we should generally strive to have only a relatively small number of axioms in the theories, we should not feel constrained to keep the number of theorems small. As previously noted, besides the “main theorems" in theory development, it is frequently useful to introduce numerous lemmas for the purpose of simplifying proof structure and sharing significant proof parts among many separate proofs. Thus, useful theories can be quite large, numbering at least in the hundreds and eventually in the thousands of propositions, most of which are theorems and thus must be accompanied by proofs — at least in CCT where we want to avoid expensive proof search by having available pre-programmed proofs that only need to be checked.
As a consequence, while a theory can be considered abstractly as an unstructured set or list of propositions, it is essential to provide a more efficient representation for their storage and retrieval. For CCT, we must also be able to efficiently retrieve the proof of any theorem we want to use.

The data structure we have developed, which we call a *structured theory* (but often shorten to just *theory*), while certainly not optimal in efficiency, has been adequate for the size of theories developed for CCT to date (a few hundred theorems and proofs) and will probably suffice for theories that are at least one order of magnitude larger.

We define a *structured theory* as an abstract data type with the following functions and proof methods defined on it:

- A function `theory` for creating a structured theory from a generic property function containing axioms;
- A function `evolve` for extending an existing structured theory with a new generic theorem and its proof;
- A function `refine` for creating a new structured theory as a composition of one or more existing structured theories and a generic property function.
- A function `get-property` for retrieving an instance of a generic property function.
- A method `property` for retrieving an instance of a generic property function, which might be an axiom or a theorem; in the latter case it also retrieves the theorem’s proof and executes it if the property instance is not already in the assumption base.
- A method `property-test`, which is the same as `property` except that it does not first check if the retrieved proposition \( p \) is already in the assumption base; it requires that a generic proof be supplied and always executes it.

More details about these operations follow below, but first it may be useful to point out a few features of the currently implemented representation of a structured
theory. The representation is simply an Athena cell holding a list, each element of which is either

- a two-element list \([F \ P]\), where \(F\) is a generic property function and \(P\) is either a generic proof or the symbol \texttt{Axiom}, or

- (the representation of, recursively) a structured theory.

Cycles are not allowed, so that this representation is essentially an acyclic graph structure that can be searched beginning at any node of the graph and continuing along all paths from that node. Although a search must continue until a match is found or all paths from the initial node are examined, nodes not on any of those paths are not included in the search. This means that it is usually unnecessary to search through more than a small fraction of the total number of nodes in the graph.

It is also essential to understand that the representation is not a fixed structure, because an updatable cell holding a list is used rather than a pure list. Suppose a structured theory \(T_1\) is an element of a structured theory \(T\). If the list in \(T_1\) is updated (with \texttt{evolve}) to hold an additional element, then property retrievals starting at \(T\) as well as \(T_1\) will include this element in their search. This means that one does not have to construct a structured theory completely before beginning to use it; it can be incrementally developed with additions of new theorems and proofs at any time to any structured theory within it.

We now describe the structured theory operations in more detail and illustrate them with small examples.

```plaintext
theory [D Axiom]
```

A function that creates a new theory whose only entry is \([D \ Axiom]\), where \(D\) is a generic property function holding definitions or axioms; the keyword \texttt{Axiom} denotes that no generic proof is given with \(D\); the propositions instantiated from \(D\) must already be in the assumption base when used in a proof (by having been previously asserted or proved).

A typical usage of this function is
where we define a new structured theory called \texttt{Memory-theory}. Here, \texttt{Memory} is a generic property function that collects together and parameterizes with an operator mapping all of the axioms we previously discussed in Chapter 5.1.

\begin{verbatim}
(define (Memory name ops)
  (let ((access (ops 'access))
        (assign (ops 'assign))
        (swap (ops 'swap)))
    (match name
      ('assign-axiom1
        (forall ?M ?a ?x
          (= (access (assign ?M ?a ?x) ?a)
              ?x)))
      ('assign-axiom2
        (forall ?M ?a ?b ?x
          (if (not (= ?a ?b))
            (= (access (assign ?M ?a ?x) ?b)
                (access ?M ?b))))
      ('swap-axiom1
        (forall ?M ?a ?b
          (= (access (swap ?M ?a ?b) ?a)
              (access ?M ?b)))
      ('swap-axiom2
        (forall ?M ?a ?b
          (= (access (swap ?M ?a ?b) ?b)
              (access ?M ?a)))
      ('swap-axiom3
        (forall ?M ?a ?b ?c
          (if (and (not (= ?c ?a))
                  (not (= ?c ?b)))
            (= (access (swap ?M ?a ?b) ?c)
                (access ?M ?c))))
    )))
\end{verbatim}
Semigroup is a mathematical structure which consists of a set that is closed under an associative binary operation. Our second example shows how we define Semigroup as a theory:

(define (Semigroup name ops)
  (match name
    ('Associative
      (let ((Plus (ops 'Plus)))
        (forall ?x ?y ?z
          (= (Plus (Plus ?x ?y) ?z)
            (Plus ?x (Plus ?y ?z)))))))

(define Semigroup-theory (theory [Semigroup Axiom]))

evolve $T$ $[D P]$

updates an existing structured theory $T$ with a new entry $[D P]$, where $D$ is a generic theorem and $P$ is its generic proof.

For an example, consider:

(evolve Memory-theory [Double-assign Double-assign-proof])

where Memory-theory is an existing theory, and Double-assign-proof proves the Double-assign property.

refine $[T_1 \ldots T_k]$ $[D$ Axiom$]$

creates a new structured theory from the list of structured theories $[T_1 \ldots T_k]$ and the pair $[D$ Axiom$]$, where $D$ is a generic property function.

As examples consider the following sequence of generic property functions and structured theory refinements:

(define (Right-Identity name ops)
  (match name
    ('Right-Identity
(let ((Plus (ops 'Plus))
     (Zero (ops 'Zero)))
  (forall ?x
    (= (Plus ?x Zero) ?x))))

(define Monoid-theory
  (refine [Semigroup-theory] [Right-Identity Axiom]))

(define (Right-Inverse name ops)
  (match name
    ('Right-Inverse
     (let ((Plus (ops 'Plus))
           (Zero (ops 'Zero))
           (Negate (ops 'Negate)))
      (forall ?x (= (Plus ?x (Negate ?x)) Zero))))))

(define Group-theory
  (refine [Monoid-theory] [Right-Inverse Axiom]))

An example of refinement of more than one structured theory is the following:

(define Memory-range-theory
  (refine [Memory-theory Range-theory] [Memory-range Axiom]))

where Memory-theory and Range-theory have already been defined. We inherit all properties from these theories and add the set of axioms held in the generic property function Memory-range.

get-property propname ops T

A function that retrieves the generic property named propname, instantiated with operator map ops, from structured theory T. An error condition results if the search for propname or the instantiation is unsuccessful.
Before showing an application of \texttt{get-property} we first create another theory by using \texttt{Monoid-theory} and the generic definition of \texttt{sum-list@} we discussed in the previous section.

\begin{verbatim}
(define Sum-list-theory
 (refine
  [Monoid-theory]
  [sum-list@-definition Axiom]))
\end{verbatim}

If typed at ATHENA’s top level, the output of

\begin{verbatim}
(get-property ’sum-list@-nonempty Times-ops Sum-list-theory)
\end{verbatim}

would be:

\textbf{Proposition:} (forall ?L

  (forall ?y
   (= (sum-list@ (Cons ?y ?L))
    (Times ?y
     (sum-list@ ?L))))))

Note that the result is labeled a \textit{proposition} since \texttt{get-property} does not check whether it is in the assumption base. The way that \texttt{get-property} is frequently used is to assert a proposition into the assumption base:

\begin{verbatim}
(assert
 (get-property ’sum-list@-nonempty Times-ops Sum-list-theory))
\end{verbatim}

\texttt{get-property \texttt{propname} \texttt{ops} \texttt{T}}

A method that retrieves both the generic property named \texttt{propname}, instantiated with operator map \texttt{ops}, and its proof, if any, from structured theory \texttt{T}. Returns the resulting property instance \texttt{p} and furthermore ensures that \texttt{p} is in the assumption base — if it is not there already, it is placed there as a result of executing the retrieved proof. An error condition results if the search for \texttt{propname} or the instantiation is unsuccessful, or if \texttt{p} is not already in the assumption base and there is no accompanying proof (i.e., if in place of a proof we only have the symbol \texttt{Axiom}), or if the retrieved proof fails to prove \texttt{p}. 
This method is the same as property except that it does not first check if the retrieved proposition \( p \) is already in the assumption base; it requires that a generic proof be supplied and always executes it.

For example:

\[
\text{(assert}
\begin{align*}
&\text{(get-property 'sum-list@-empty Monoid-ops Sum-list-theory)} \\
&\text{(get-property 'sum-list@-nonempty Monoid-ops Sum-list-theory)}
\end{align*}
\text{)}
\]

\[
(!\text{property 'sum-list@-empty Monoid-ops Sum-list-theory})
\]

This returns \((= (\text{sum-list@ Nil}) \text{zero})\), since this property instance is in the assumption base as a result of the preceding assert command. Had it not been asserted, an error condition would result, since there is no associated proof. (If this method call is done at the top level in ATHENA, the output is actually labeled “Theorem”:

\[
\text{Theorem: } (= (\text{sum-list@ Nil}) \text{zero})
\]

ATHENA considers any proposition in its assumption base to be a theorem, whether it got there as a result of having been asserted or having been proved.)

Similarly,

\[
(!\text{property 'sum-list@-nonempty Monoid-ops Sum-list-theory})
\]

returns

\[
(\forall \text{?L \?y} \\
\begin{align*}
&= (\text{sum-list@ (Cons \?y \?L))} \\
&(\text{Plus ??y (sum-list@ \?L)))
\end{align*}
\]

We provide a generic proof for a generic lemma called \text{sum-list-compute-relation} in Appendix F. After adding this property and its proof to \text{Sum-list-theory} we could use property as follows to test if the proof works as expected.

\[
(!\text{property 'sum-list-compute-relation Monoid-ops Sum-list-theory})
\]
6.1.3 Definitions of efficient generic sum-list

As in the nongeneric example the producer defines an efficient \texttt{sum-list} function that implements \texttt{sum-list@}. But this time the definitions are generic.

\begin{verbatim}
(define (sum-list-definition name ops)
  (let ((Plus (ops 'Plus))
        (Zero (ops 'Zero)))
    (match name
      ('sum-list-empty
       (= (sum-list Nil)
           Zero))
      ('sum-list-nonempty
       (forall ?L ?x
           (= (sum-list (Cons ?x ?L))
               (sum-list-compute ?L ?x))))))

(define (sum-list-compute-definition name ops)
  (let ((Plus (ops 'Plus)))
    (match name
      ('sum-list-compute-empty
       (forall ?x
           (= (sum-list-compute Nil ?x)
               ?x)))
      ('sum-list-compute-nonempty
       (forall ?L ?x ?y
           (= (sum-list-compute (Cons ?y ?L) ?x)
               (sum-list-compute ?L (Plus ?x ?y)))))))
\end{verbatim}

The logic is exactly the same. The \texttt{sum-list} function calls a helper function \texttt{sum-list-compute} which makes a tail-recursive call.

\begin{verbatim}
(define (sum-list-compute-definition name ops)
  (let ((Plus (ops 'Plus)))
    (match name
      ('sum-list-compute-empty
       (forall ?x
           (= (sum-list-compute Nil ?x)
               ?x)))
      ('sum-list-compute-nonempty
       (forall ?L ?x ?y
           (= (sum-list-compute (Cons ?y ?L) ?x)
               (sum-list-compute ?L (Plus ?x ?y)))))))
\end{verbatim}

The definitions are now ready to be used in the proofs. We can add them to the \texttt{Sum-list-theory} created in the previous section.

\begin{verbatim}
(evolve Sum-list-theory [sum-list-compute-definition Axiom])

(evolve Sum-list-theory [sum-list-definition Axiom])
\end{verbatim}
This theory will be updated and used by the producer while introducing new definitions and proving theorems.

### 6.1.4 Proof methods in CCT

Figure 6.2 shows the structure of our generic proof methods in CCT. There are three parts. The first part is the method header containing its name and the same two parameters as for generic property functions, a property name and an operator mapping. The second part, which is optional, contains local declarations of operator mappings, cells left and right (if the proof will be done with rewriting steps), and some other bindings that will increase the readability of the code and reduce its size. And, finally, the last part is the body of the method where we write the steps of the proof.
6.1.5 Correctness proof of generic sum-list

Our goal in this section is to explain the general structure of proof methods in CCT with an example and illustrate how to write generic proofs. For this reason, we defer the discussion of how to prove generic versions of the termination condition (TC) and additional consistency condition (CC) to the generic reverse-range example to be presented in the next section and concentrate on the correctness proof of sum-list.

Suppose both the TC and CC have been proved, and now the producer must prove the following correctness property (which the consumer has specified among his requirements):

\[
\text{(define (sum-list-correctness name ops)}
\]
\[
\text{(forall ?L}
\]
\[
\text{(= (sum-list ?L)
\]
\[
\text{(sum-list0 ?L))))}
\]

We will divide the proof into two parts and explain its differences from the concrete sum-list-correctness presented in Chapter 4. The proof begins as follows:

\[
\text{(define (sum-list-correctness-proof name ops)}
\]
\[
\text{(dlet ((Zero (ops 'Zero))}
\]
\[
\text{(left (cell true))}
\]
\[
\text{(right (cell true))}
\]
\[
\text{(prop (method (name)}
\]
\[
\text{(!property name ops Sum-list-theory)))}
\]
\[
\text{(theorem (sum-list-correctness proof name ops)))}
\]

There are two new local bindings in this first part: prop and theorem. In generic proofs we frequently use the method property to find an axiom or a theorem in a specific theory. As we will see shortly, this proof needs several axioms and a theorem from the Sum-list-theory to do necessary reductions in rewriting steps. What we only need to provide differently in each case to the property method is an axiom or a theorem name; the theory name and operator mapping will be the same. Therefore, we define a local method called prop with one parameter. In this way we also increase the readability of the proof and reduce the proof size. We also bind theorem as the proposition to be proved.
Now follows the main part of the proof:

(\texttt{by-induction theorem (Nil (dseq (!setup left (\texttt{sum-list Nil}))) (!setup right (\texttt{sum-list@ Nil})))
  (!reduce left Zero (!prop 'sum-list-empty))
  (!reduce right Zero (!prop 'sum-list@-empty))
  (!combine left right)))

((\texttt{Cons x L})
  (dseq
    (!setup left (\texttt{sum-list (Cons x L)}))
    (!setup right (\texttt{sum-list@ (Cons x L)}))
    (!reduce left (\texttt{sum-list-compute L x})
      (!prop 'sum-list-nonempty))
    (!reduce right (\texttt{sum-list-compute L x})
      (!prop 'sum-list-compute-relation))
    (!combine left right))))

The proof is again by induction. It uses the same rewriting steps as in the non-generic \texttt{sum-list} example. However, there are two differences. First, we use \texttt{Zero}, representing a generic identity element, instead of \texttt{zero}, which only represents an element of the natural numbers. The second difference is that we used concrete axioms in the proof steps of nongeneric \texttt{sum-list} but here we use the \texttt{prop} method with axiom names instead. Since it is possible to pass different \texttt{ops} functions, the result of \texttt{prop} will be different in different instantiations.

This proof uses as a lemma a theorem called \texttt{sum-list-compute-relation}, which defines a relation between the helper function and the specification given by the consumer:
(define (sum-list-compute-relation name ops)
  (match name
    ('sum-list-compute-relation
      (forall ?L ?x
        (= (sum-list@ (Cons ?x ?L))
            (sum-list-compute ?L ?x)))))
)

For the proof of this lemma, see Appendix F. In the nongeneric example, this lemma had to be proved before being used in the proof of the main theorem, but there is more flexibility in ordering of proofs in the generic setting. If sum-list-correctness-proof is executed before sum-list-compute-relation-proof then the needed instance of sum-list-compute-relation will not be in the assumption base, but the property method will retrieve its proof and execute it to put it there.

As a last note, since our main proof looks for sum-list-compute-relation in Sum-list-theory, both the lemma and its proof are required to be added to Sum-list-theory as follows:

(evolve Sum-list-theory [sum-list-compute-relation
                         sum-list-compute-relation-proof])

(evolve Sum-list-theory [sum-list-correctness
                         sum-list-correctness-proof])

6.1.6 Instantiating generic definitions and proofs of sum-list

We will show how beneficial it is to use generic definitions and proofs in CCT instead of sending similar concrete definitions and proofs to the consumer. As we have mentioned before, different instantiations of proofs of sum-list are possible. The definitions and proofs are constructed in a way that passing a proper ops function will be sufficient to generate proofs for different operations on a list. We should just make sure before each instantiation that we implement an operator mapping function that defines a Monoid operation correctly, assert the right combinations of axioms, and instantiate both functions and proofs with the right arguments.

Suppose now the consumer created Sum-list-theory with the generic specification of the sum-list function and correctness condition, and the producer updated it with the generic definitions and proofs for sum-list and sum-list-compute.
The consumer can now extract various different sum-list functions by instantiating the same generic definitions and proofs in Sum-list-theory. Here we demonstrate this capability with three example instantiations.

### 6.1.6.1 Addition of naturals in a list via generic sum-list

This example shows the construction of a sum-list function that does addition of natural numbers with Plus and zero as the Monoid operations.

```scheme
(define (Monoid-ops op)
  (match op
    ('Plus Plus)
    ('Zero zero)))

(define sum-list@-axioms
  (map (function (name)
    (get-property name Monoid-ops Sum-list-theory))
    ['sum-list@-empty 'sum-list@-nonempty]))
```

The consumer enters `sum-list@-axioms` as a specification of the function to be defined by the consumer. Athena’s `map` function is used as a higher-order function that applies a given function to a list of elements and produces a list of results. In this definition the list elements are the names of the properties to be matched in Sum-list-theory. If we typed `sum-list@-axioms` into Athena, we would see the following output:

```
List: [(= (sum-list@ Nil) zero)
       (forall ?L ?y (= (sum-list@ (Cons ?y ?L)) (Plus ?y (sum-list@ ?L))))]
```

This list contains two concrete properties involving zero and Plus. Assuming the corresponding termination and consistency conditions have been proved, which we do not show here, the consumer can assert these properties as axioms.

```scheme
(assert sum-list@-axioms)
```

The consumer next constructs the specialized axioms for the helper function,
(define sum-list-compute-axioms
  (map (function (name)
    (get-property name Monoid-ops Sum-list-theory))
    ['sum-list-compute-empty 'sum-list-compute-nonempty]))

and applies the termination and consistency checks. Again, we do not show these checks here but they are successful, and it is safe to assert the sum-list-compute-axioms:

List: [(forall ?x
  (= (sum-list-compute Nil ?x) ?x))

(forrall ?L ?x ?y
  (= (sum-list-compute (Cons ?y ?L) ?x)
    (sum-list-compute ?L (Plus ?x ?y))))]

(assert sum-list-compute-axioms)

We can now instantiate the proof of sum-list-compute-relation:

(!property 'sum-list-compute-relation Monoid-ops Sum-list-theory)

which produces

Theorem: (foral ?L ?x
  (= (sum-list@ (Cons ?x ?L))
    (sum-list-compute ?L ?x)))

The consumer is done with the helper function and defines sum-list-axioms as follows:

(define sum-list-axioms
  (map (function (name)
    (get-property name Monoid-ops Sum-list-theory))
    ['sum-list-empty 'sum-list-nonempty]))

which stores the following list of properties:

List: [(= (sum-list Nil) zero)

(forrall ?L ?x (= (sum-list (Cons ?x ?L))
    (sum-list-compute ?L ?x)))]
Once again, `sum-list-axioms` cannot be asserted unless the termination and consistency checks succeed, but those checks are omitted here.

\[(\text{assert } \text{sum-list-axioms})\]

Finally, we can instantiate the proof of `sum-list-correctness`.

\[(!\text{property } \text{sum-list-correctness})\]
\[\text{Monoid-ops Sum-list-theory}\]

whose output is the correctness theorem for the `sum-list` function that computes addition of natural numbers with `Plus` and `zero`.

**Theorem:** (forall ?L
\[\text{=} (\text{sum-list } ?L)
\[\text{(sum-list@ } ?L))\]

In the last step `CodeGen` is applied to `sum-list-compute-axioms` and `sum-list-axioms`, producing the following Oz code:

fun `{SumListCompute L X}`
    case [L X]
    of [nil X] then X
        [] [Y|L X] then `{SumListCompute L (X + Y)}
    end

fun `{SumList L}`
    case L
    of nil then 0
        [] X|L then `{SumListCompute L X}
    end
end

### 6.1.6.2 Multiplication of naturals in a list via generic `sum-list`

Our second example shows how the consumer constructs the `sum-list` function that computes the product of natural numbers with `Times` and `one` as monoid operations.
(define (Monoid-ops op)
  (match op
    ('Plus Times)
    ('Zero one)))

(define sum-list@-axioms
  (map (function (name)
          (get-property name Monoid-ops Sum-list-theory))
       ['sum-list@-empty 'sum-list@-nonempty]))

(assert sum-list@-axioms)

In this case, sum-list@-axioms specifies axioms for computing the product of list elements as follows:

List: [(= (sum-list@ Nil) one)
       (forall ?L ?y
            (= (sum-list@ (Cons ?y ?L))
                (Times ?y (sum-list@ ?L))))]

This list contains two concrete properties where one and Times are used as operators.

The consumer asserts these properties as axioms and defines the helper function.

(define sum-list-compute-axioms
  (map (function (name)
            (get-property name Monoid-ops Sum-list-theory))
       ['sum-list-compute-empty 'sum-list-compute-nonempty]))

where sum-list-compute-axioms contains the following two axioms:

(forall ?x
  (= (sum-list-compute Nil ?x) ?x))

(forall ?L ?x ?y
  (= (sum-list-compute (Cons ?y ?L) ?x)
      (sum-list-compute ?L (Times ?x ?y))))

After the termination and consistency checks, the consumer asserts the axioms defining the helper function and instantiates the generic proof as follows:
The correctness proof is instantiated in the same way for the multiplication of list elements after the termination and consistency checks are done for the axioms returned by the following definition:

```
(define sum-list-axioms
  (map (function (name)
    (get-property name Monoid-ops Sum-list-theory))
    ['sum-list-empty 'sum-list-nonempty]))
```

which is a list that contains two properties:

```
List: [(= (sum-list Nil) one)
  (forall ?L ?x (= (sum-list (Cons ?x ?L))
    (sum-list-compute ?L ?x)))]
```

We assert them and instantiate the correctness proof as follows:

```
(assert sum-list-axioms)

(!property 'sum-list-correctness Monoid-ops Sum-list-theory)
```

Here is the Oz code that CODEGEN then produces from these axioms.

```
fun {SumListCompute L X}
  case [L X]
  of [nil X] then X
  [] [Y|L X] then {SumListCompute L {Times X Y}}
  end

fun {SumList L}
  case L
  of nil then 1
  [] X|L then {SumListCompute L X}
  end
end
```
6.1.6.3 Joining list elements via generic sum-list

Our last example of instantiation shows the construction of a sum-list function that joins list elements (which are required to be lists) with Append and nil as the monoid operations.

The consumer starts with defining the following Monoid-ops function.

```
(define (Monoid-ops op)
  (match op
    ('Plus Append)
    ('Zero Nil)))
```

All the other steps, commands and functions being used in this example are identical with the ones we used for the previous two examples. Therefore, we will only list the result of executions without explaining the details.

```
(define sum-list@-axioms
  (map (function (name)
           (get-property name Monoid-ops Sum-list-theory)))
       ['sum-list@-empty 'sum-list@-nonempty]))
```

produces

```
(= (sum-list@ Nil) Nil)
```

```
(forall ?L ?y
  (= (sum-list@ (Cons ?y ?L))
      (Append ?y (sum-list@ ?L))))
```

Asserting them and defining the helper function:

```
(assert sum-list@-axioms)
```

```
(define sum-list-compute-axioms
  (map (function (name)
           (get-property name Monoid-ops Sum-list-theory)))
       ['sum-list-compute-empty 'sum-list-compute-nonempty]))
```

yields
List: 

\[(\forall ?x \\
\quad (= (\text{sum-list-compute} \text{ Nil} \ ?x) \ ?x))\]

\[(\forall ?L \ ?x \ ?y \\
\quad (= (\text{sum-list-compute} \text{ (Cons } \ ?y \ ?L) \ ?x) \\
\quad \quad \quad \quad \text{sum-list-compute} ?L \text{ (Append } ?x \ ?y)))\]

The consumer asserts them after the termination and consistency checks are done:

\[
\text{(assert sum-list-compute-axioms)}
\]

We again instantiate the proof of \text{sum-list-compute-relation}.

\[
\text{(!property-test 'sum-list-compute-relation} \\
\quad \text{Monoid-ops Sum-list-theory})
\]

The consumer defines \text{sum-list-axioms}, checks their termination and consistency successfully, asserts them, and proves the required correctness condition.

\[
\text{(define sum-list-axioms} \\
\text{(map (function (name}} \\
\quad \text{(get-property name Monoid-ops Sum-list-theory))} \\
\quad \quad \quad \quad \quad ['\text{sum-list-empty 'sum-list-nonempty}])
\]

\[
\text{(assert sum-list-axioms)}
\]

\[
\text{(!property 'sum-list-correctness Monoid-ops Sum-list-theory})
\]

Finally, having successfully completed these steps, the consumer uses \text{CODEGEN} to generate the following two functions:
fun {SumListCompute L X}
  case [L X]
  of [nil X] then X
  [] [Y|L X] then {SumListCompute L {Append X Y}}
end

fun {SumList L}
  case L
  of nil then nil
  [] X|L then {SumListCompute L X}
end
end

As can be seen from these three examples, the consumer did not need to send separate specifications to the producer in each case. Similarly, the producer did not have to provide separate axioms and proofs for each example. Transmission of generic definitions and proofs are done only once. This is especially critical if the definitions and proofs of complex applications are required to be transmitted.

6.1.7 Termination and consistency of generic sum-list

As in the nongeneric sum-list example, consistency and termination are required to be proved for both sum-list-compute and sum-list. After loading TC-Gen a function called termination-axiom function can be used with a list of axioms to produce the corresponding termination axioms. Similarly, assuming that CCGen is loaded, a consistency-condition function is used in order to generate the additional consistency condition for a given list of axioms.

6.1.7.1 Proofs for sum-list-compute

- Termination of sum-list-compute:

  The producer starts with declaring sum-list-compute_t to express whether sum-list-compute terminates or not:
As in defining generic specifications, the termination condition is defined with a generic property function.

\[
\text{(define (sum-list-compute-termination name ops)}
\text{ (match name)}
\text{ ('sum-list-compute-termination}
\text{ (forall ?L ?x (sum-list-compute_t ?L ?x))}))
\]

In the next step, the producer proves \text{sum-list-compute-termination} generically as follows:

\[
\text{(define (sum-list-compute-termination-proof name ops)}
\text{ (dlet ((Plus (ops 'Plus)))}
\text{ (tprop (method (name)}
\text{ (!claim (termination-axiom}
\text{ (get-property name ops Sum-list-theory)
\text{ ops))}))
\text{ (theorem (sum-list-compute-termination name ops))})
\text{(by-induction theorem (Nil (!tprop 'sum-list-compute-empty)) ((Cons y L) (dlet ((induction-hypothesis (forall ?x (sum-list-compute_t L ?x)))) (pick-any x (!mp (!uspec* (!tprop 'sum-list-compute-nonempty) [L x y]) (!uspec induction-hypothesis (Plus x y)))))))})
\]
In the declarations section of this proof, we show how to use `termination-axiom` to produce a termination axiom. Since the generic definitions and proofs are in `Sum-list-theory`, this specific theory is used for property search. The termination condition to be proved, in this case, is `theorem`, which is computed by the `sum-list-compute-termination` function. In addition, the `Plus` operator in this example is generic. A concrete operator can be either `Plus`, `Times`, `Append`, or any other binary monoid operator to be decided in an instantiation. Everything else in the body of the generic proof is almost identical to the nongeneric one.

In the last step, we add the termination condition and its proof to the `Sum-list-theory` for future property searches and instantiations.

```
(evolve Sum-list-theory [sum-list-compute-termination
    sum-list-compute-termination-proof])
```

- Consistency of `sum-list-compute`:

In computing the consistency condition of `sum-list-compute`, the producer passes both `sum-list-compute-empty` and `sum-list-compute-nonempty` to the `consistency-condition` function. The generic CC is defined as follows:

```
(define (sum-list-compute-consistency name ops)
  (match name
    ('sum-list-compute-consistency
      (consistency-condition
        (map (function (name)
               (get-property name ops Sum-list-theory))
             ['sum-list-compute-empty
              'sum-list-compute-nonempty]))))
```

The producer next defines the following generic proof procedure in which `theorem` is the additional consistency condition to be proved and the external resolution prover `SPASS` is invoked via the `prove-from` method to prove it.
(define (sum-list-compute-consistency-proof name ops)
  (dlet ((theorem (sum-list-compute-consistency name ops)))
    (!prove-from theorem (datatype-axioms "List-0f"))))

After completing the proof, the producer updates the Sum-list-theory with the proof of consistency condition for future use.

(evolve Sum-list-theory [sum-list-compute-consistency
                         sum-list-compute-consistency-proof])

6.1.7.2 Proofs of sum-list

We showed that the generic sum-list-compute axioms do not produce any inconsistencies and the function to be extracted from them will terminate. We now do the same for the generic sum-list axioms.

- Termination of sum-list: The producer first declares sum-list_t.

  (declare sum-list_t ((T) ->
                       ((List-Of T))
                       Boolean))

  and the termination condition is defined in a generic property function:

  (define (sum-list-termination name ops)
    (match name
      ('sum-list-termination
       (forall ?L (sum-list_t ?L)))))

To prove this, we need to use the termination axioms produced for sum-list-axioms. In addition, the termination proof of sum-list-compute which is executed at the end of this proof has to be provided by the time the proof is instantiated.
(define (sum-list-termination-proof name ops)
(dlet ((tprop
  (method (name)
    (!claim (termination-axiom
      (get-property name ops Sum-list-theory)
      ops))))
  (prop
    (method (name)
      (!property name ops Sum-list-theory)))
  (theorem
    (sum-list-termination name ops)))
(by-induction
  theorem
  (Nil
    (!tprop 'sum-list-empty))
  ((Cons y L)
    (!mp (!uspec* (!tprop 'sum-list-nonempty) [L y])
      (!uspec* (!prop 'sum-list-compute-termination)
        [L y])))
))

Sum-list-theory can now be updated with sum-list-termination and its generic proof.

(evolve Sum-list-theory [sum-list-termination
  sum-list-termination-proof])

• Consistency of sum-list:

The producer defines another generic property function for the consistency condition of sum-list axioms, proves it, and adds it to the Sum-list-theory.

(define (sum-list-consistency name ops)
  (match name
    ('sum-list-consistency
      (consistency-condition
        (map (function (name)))))
6.2 Case study: CCT steps for a generic memory-updating function

The reverse-range example has already been presented in its nongeneric form in Chapter 5. In this section, we present how we carry out the CCT protocol for a generic version of reverse-range.

- Step 1: Defining Requirements (Code Consumer)

The code consumer determines a correctness condition for generic reverse-range and defines it with the following function:

```
(define (reverse-range-property name ops)
  (let ((valid (ops 'valid))
        (reverse-range (ops 'reverse-range))
        (reverse (ops 'reverse))
        (access-range (ops 'access-range)))
    (match name
            ('reverse-range-property
             (forall
              ?i ?j
              (if (valid (range ?i ?j))
                  (forall ?M
                           (= (access-range (reverse-range ?M (range ?i ?j))
                                          (reverse (access-range ?M (range ?i ?j)))))))))
```
A concrete correctness condition is produced with proper instantiation of `reverse-range-property`, which serves as the specification of the generic `reverse-range` function to be expected from the producer.

**Step 2: Defining generic `reverse-range` (Code Producer)**

After receiving the specification and the correctness condition from the consumer, the code producer defines the generic property function `reverse-range-definition`:

```scheme
(define (reverse-range-definition name ops)
  (let ((* (ops '*))
        (++ (ops '++))
        (-- (ops '--))
        (reverse-range (ops 'reverse-range))
        (swap (ops 'swap)))
    (match name
      ('reverse-empty-range-axiom
       (forall ?i ?M
         (= (reverse-range ?M (range ?i ?i)) ?M)))
      ('reverse-nonempty-range-axiom1
       (forall ?i ?j ?M
         (if (and (not (= ?i ?j))
                     (= (++ ?i) ?j))
            (= (reverse-range ?M (range ?i ?j)) ?M)))
      ('reverse-nonempty-range-axiom2
       (forall ?i ?j ?M
         (if (and (valid (range ?i ?j))
                     (not (= ?i ?j)))
            (= (reverse-range ?M (range ?i ?j)) ?M))))
    ))
```
• Step 3: Proving Termination (Code Producer)

The code producer states and proves the generic termination condition:

\[
\begin{align*}
&\text{(not (= (++ ?i) ?j)))} \\
&\text{(=} \text{(reverse-range } ?M \text{ (range } ?i \text{ ?j))} \\
&\text{(reverse-range (swap } ?M \text{ (* } ?i \text{)(* (-?j)))} \\
&\text{(range (++ ?i) (-- ?j)))})\end{align*}
\]

This function produces a conditional termination condition. The \text{reverse-range} function must terminate if the iterator pair of \text{?i} and \text{?j} comprise a valid range.

• Step 4: Proving the Additional Consistency Condition (Code Producer)

The producer first implements the following function, \text{reverse-range-consistency}, in order to define the generic consistency condition with the help of CCGen.
In the next step, the generic proof is constructed as follows:

```
(define (reverse-range-consistency-proof name ops)
  (dlet ((theorem (reverse-range-consistency name ops))
         (prop (method (name)
                      (prove-from theorem
                      (join (datatype-axioms "List-Of")
                          [(!prop 'access-nonempty-range-axiom)
                           (!prop 'access-empty-range-axiom)])))))

and we add the theorem with its proof to the related theory, which is \texttt{Memory-range-theory}.

```

\texttt{Memory-range-theory} combines the theory of iterators, valid ranges, and memory. In addition, we have defined some \texttt{memory-range} operators with the following axioms, from which we also proved many other properties that will not be listed here.

```
(define (Memory-range name ops)
  (let ((* (ops '*))
         (++ (ops '++))
         (access (ops 'access)))

```
(assign (ops 'assign))
(access-range (ops 'access-range))
(assign-range (ops 'assign-range))

(match name
('access-empty-range-axiom
 (forall ?M ?i
  (= (access-range ?M (range ?i ?i))
     Nil)))
)

('access-nonempty-range-axiom
 (forall ?M ?i ?j
  (if (not (= ?i ?j))
   (= (access-range ?M (range ?i ?j))
      (Cons (access ?M (* ?i))
        (access-range ?M (range (++ ?i) ?j))
        )))
))

('assign-empty-range-axiom
 (forall ?M ?i
  (= (assign-range ?M ?i Nil)
     ?M)))

('assign-nonempty-range-axiom
 (forall ?M ?i ?L ?x
  (= (assign-range ?M ?i (Cons ?x ?L))
     ))))

(define Memory-range-theory
 (refine [Memory-theory Range-theory] [Memory-range Axiom]))

Since both the termination and the consistency conditions are proved, the axioms defining generic reverse-range can be asserted and used in the proof of the correctness condition.

• Step 5: Proving Correctness of Generic reverse-range (Code Producer)

The proof of correctness condition—reverse-range-property, which was
given at Step 1—requires the following lemma.

\[
\text{(define \ (reverse-range-has-no-effect-outside name \ ops)}
\]
\[
\text{(let \ ((valid \ (ops 'valid)))}
\text{\ (in \ (ops 'in)))}
\text{\ (reverse-range \ (ops 'reverse-range))}
\text{\ (* \ (ops '*)))}
\text{\ (access \ (ops 'access)))}
\]
\[
\text{(match name}
\text{'reverse-range-has-no-effect-outside}
\text{\ (forall \ ?i \ ?j}
\text{\ (if \ (valid \ (range ?i \ ?j))}
\text{\ \ (forall}
\text{\ \ ?k \ ?M}
\text{\ \ (if \ (not \ (in \ ?k \ (range ?i \ ?j)))}
\text{\ \ \ \ (forall}
\text{\ \ \ \ \ ?M \ (range ?i \ ?j))}
\text{\ \ \ \ \ (* \ ?k))}
\text{\ \ \ \ \ (access \ ?M \ (* \ ?k)))))})
\]

The proofs of the lemma and correctness condition are generic versions of the proofs discussed in Section 5.3 using our range2-induction method. These generic theorems and proofs are added to Memory-range-theory.

- Step 6: Termination Checking (Code Consumer)

The code consumer should first declare the reverse-range operator.

\[
\text{(declare reverse-range \ ((T) \ ->)}
\text{\ (Memory \ (Range \ (Iterator \ T))))}
\text{\ Memory))}
\]

Operator mapping is done with the following function.

\[
\text{(define rev-ops}
\text{\ (function \ (op)}
\text{\ \ (match \ op}
\text{\ \ \ \ (let}
\text{\ \ \ \ ((valid \ (ops 'valid)))}
\text{\ \ \ \ (in \ (ops 'in)))}
\text{\ \ \ \ (reverse-range \ (ops 'reverse-range))}
\text{\ \ \ \ (* \ (ops '*)))}
\text{\ \ \ \ (access \ (ops 'access)))}
\text{\ \ \ \ (match name}
\text{\ \ \ \ ('reverse-range-has-no-effect-outside}
\text{\ \ \ \ \ (forall \ ?i \ ?j}
\text{\ \ \ \ \ (if \ (valid \ (range ?i \ ?j))}
\text{\ \ \ \ \ \ (forall}
\text{\ \ \ \ \ \ \ \ ?k \ ?M}
\text{\ \ \ \ \ \ \ \ (if \ (not \ (in \ ?k \ (range ?i \ ?j)))}
\text{\ \ \ \ \ \ \ \ \ \ \ \ (forall}
\text{\ \ \ \ \ \ \ \ \ \ \ \ \ \ ?M \ (range ?i \ ?j))}
\text{\ \ \ \ \ \ \ \ \ \ \ \ \ \ (* \ ?k))}
\text{\ \ \ \ \ \ \ \ \ \ \ \ \ \ (access \ ?M \ (* \ ?k)))}})))
\]
The consumer constructs `reverse-range-axioms` by instantiating the generic axioms that define `reverse-range`.

```scheme
(define reverse-range-axioms
  (map (function (name)
             (get-property name rev-ops
                           Memory-range-theory))
       ['reverse-empty-range-axiom
        'reverse-nonempty-range-axiom1
        'reverse-nonempty-range-axiom2]))
```

The `reverse-range_t` symbol has been used to express the termination of `reverse-range`. It has to be declared by the consumer so that the termination proof can be checked.

```scheme
(declare reverse-range_t ((T) ->
                          (Memory (Range (Iterator T)))
                          Boolean))
```
The termination axioms for reverse-range are generated with TCGEN and asserted into the assumption base.

```lisp
(define reverse-range-termination-axioms
  (map (function (axiom)
           (termination-axiom axiom
                        reverse-range_t-opmap))
       reverse-range-axioms))

(assert reverse-range-termination-axioms)
```

The consumer enters the following command to instantiate the termination proof of reverse-range.

```lisp
(!property 'reverse-range-termination
             (join-maps rev-ops reverse-range_t-opmap)
             Memory-range-theory)
```

- Step 7: Additional Consistency Checking (Code Consumer)

The proof of the additional consistency condition of reverse-range is checked as follows:

```lisp
(! (conclude
          (consistency-condition reverse-range-axioms))
!!!property 'reverse-range-consistency
             rev-ops Memory-range-theory))
```

- Step 8: Checking Correctness (Code Consumer)

The termination and consistency checks having succeeded, the consumer can assert the axioms defining reverse-range function and check the proof of the correctness condition.

```lisp
(assert reverse-range-axioms)

(!property 'reverse-range-property rev-ops Memory-range-theory)
```
Note that during this proof, the proof of the lemma `reverse-range-has-no-effect-outside` will be retrieved and executed also, so success in proving it is required for success in completing the proof of `reverse-range-property`.

- Step 9: Code Extraction (Code Consumer)

`CODEGen` extracts the following function from the axioms defining `reverse-range`.

```plaintext
fun {ReverseRange M R }
  case R
  of range(I I) then M
  [] range(I J) then
    if {Not (I == J)} then
      if ({++ I} == J) then M
      else {ReverseRange {Swap M {'*' I} {'*' {'--' J}}} range({'++' I} {'--' J})} end
    end
  end
end
```
A Larger Example of the CCT Approach

Although the CCT examples presented in this thesis are very small in terms of the size of the code produced, the examples that are based on STL components have a much larger significance because of their nature as generic library components. Their parameterization over different sequence representations (through iterator ranges) and other interface design characteristics borrowed from the STL make them highly versatile building blocks for composition into larger, more complex components. In this section we demonstrate this point with an example that takes three generic functions that have been subjected to the CCT approach and composes them into a new and useful function. Though still of only very modest size and complexity, this new component, called PartialPartition, performs a function that could be useful in a real-time operating system. Its arguments are a memory $M$, a range $[i,j)$, and a predicate $P$. It updates the memory so that the first group of elements of the range that satisfy $P$ occur at the beginning of the range, followed by the rest of the original range except for those moved to the beginning. In an operating system the range could be a task queue and the predicate could be “has priority higher than 10,” for example.\footnote{For an analogous situation in a real world scenario think of people waiting in line (the range) at an airline ticket counter, and a plane for Detroit is about to close boarding, so the ticket agent wants to process Detroit passengers first. The agent begins by moving the first group of Detroit passengers in the line (the predicate is “is going to Detroit”) to the head of the line.} By repeated use of this PartialPartition operation, one could achieve a complete partitioning of the range, resulting in all elements satisfying $P$ coming first, followed by all elements not satisfying $P$. While there are more efficient algorithms for complete partitioning, doing it incrementally might be advantageous in some situations; e.g., in a real-time operating system so that high priority tasks could be serviced between successive calls of PartialPartition (such as those moved to the beginning of the task queue).

This function can be programmed in Oz as follows:
fun {PartialPartition M range(I J) P}
    if I == J then M
    else local K1 = {FindIf M range(I J) P} in
        if K1 == J then M
        else K2 = {FindIf M range(K1 J) {Negate P}} in
            {Rotate M range(I K2) K1}
        end
    end
end
end

It uses two functions: Rotate, which swaps the contents of two subranges on either side of an iterator pointing within the range; and FindIf, which finds the first position within a range satisfying a given predicate. The two calls of FindIf determine iterators K1 and K2 delimiting the subrange that is brought to the beginning of the range from I to J by the Rotate call.

The CCT development of these two functions is presented in Sections 7.1 and 7.2. A third CCT-developed function that we have already seen, ReverseRange, is also involved since Rotate is programmed in terms of (in fact, three calls of) ReverseRange.

Even though PartialPartition as presented here is hand-coded in Oz and has itself not (as yet) been developed via CCT, it would be useful to have the components that a more complex function like PartialPartition is composed of be securely delivered by the producer even when the composed function is itself not done that way. For example, a more efficient version of FindIf in which loop unrolling is done might be delivered and, if accepted according to the CCT protocol, installed in place of the original version, thereby speeding up PartialPartition and any other functions that use FindIf.

7.1 A generic rotate function

The rotate function specified, correctly implemented, and tested in this section is very similar to the STL’s rotate function. It takes three parameters: a memory, a range, and an iterator pointing to a location in the range. It returns an
updated memory in which the subranges on either side of the iterator are swapped.

In this chapter, rather than stepping through the CCT protocol as before, we divide the presentation according to contents of specification, implementation, and unit test files. The relation to the CCT protocol is that the consumer could construct and send the specification file to the producer, who responds with the implementation file, upon which the consumer loads the unit test file. The latter file in turn loads the implementation file and puts it through tests to check that all of the required proofs go through. Note that if any of these proof tests fails, the error condition that results causes processing of the file to cease, so that one would not get to the code generation step at the end of the unit test file. Thus, this division into files provides a fairly realistic simulation of the CCT protocol.

7.1.1 Specification file: rotate-specification.ath

# Correctness property of rotate, a memory-updating function based
# on the STL rotate function, defined on a valid range.

(load-file "memory.ath")
(load-file "iterator.ath")
(load-file "range.ath")
(load-file "memory-range.ath")
(load-file "reverse-range.ath")
(load-file "consistency.ath")
(load-file "termination-axiom.ath")

# Correctness condition: accessing the range [i, j) in the updated
# memory is the same as appending the results of accessing the ranges
# [k, j) and [i, k) in the original memory.

(define (rotate-property name ops)
  (let ((valid (ops 'valid))
        (rotate (ops 'rotate))
        (access-range (ops 'access-range))
        (append (ops 'append))))
(match name
   ('rotate-property
     (forall ?M ?i ?j ?k
       (if (and (valid (range ?i ?k))
            (valid (range ?k ?j)))
        (= (access-range (rotate ?M (range ?i ?j) ?k)
            (range ?i ?j))
          (append (access-range ?M (range ?k ?j))
            (access-range ?M (range ?i ?k)))))))

7.1.2 Implementation file: rotate-implementation.ath
# Implementation of rotate, satisfying the specification
# given in rotate-specification.ath.

(load-file "rotate-specification.ath")

# This is the well-known "three reverses" implementation.

(define (rotate-definition name ops)
  (let ((rotate (ops 'rotate))
        (reverse-range (ops 'reverse-range)))
    (match name
      ('rotate-axiom
       (forall ?M ?i ?j ?k
        (= (rotate ?M (range ?i ?j) ?k)
          (reverse-range
           (reverse-range
            (reverse-range ?M (range ?i ?k))
             (range ?k ?j))
           (range ?i ?j))))))

(evolve Memory-range-theory [rotate-definition Axiom])
# For consistency:

(define (rotate-consistency name ops)
  (let ((rotate (ops 'rotate))
         (reverse-range (ops 'reverse-range)))
    (match name
      ('rotate-consistency
       (consistency-condition
        [(get-property 'rotate-axiom ops Memory-range-theory)]))))

(define (rotate-consistency-proof name ops)
  (dlet ((rotate (ops 'rotate))
         (reverse-range (ops 'reverse-range))
         (theorem (rotate-consistency name ops)))
    (pick-any
     M k r i j
     (dlet ((term
              (reverse-range
               (reverse-range
                (reverse-range M (range i k))
                (range k j))
               (range i j)))
             (dseq
              (assume (= r (range i j))
               (!equality term term))
              (!egen (exists ?result
                            (if (= r (range i j))
                                (= ?result term))))
              term))))))

(evolve Memory-range-theory [rotate-consistency
(define rotate-t-opdefs (cell []))
(define (rotate-t-opmap op)
  (find op (ref rotate-t-opdefs)))

(define (rotate-termination name ops)
  (let ((rotate (ops 'rotate))
        (rotate_t (ops rotate))
        (valid (ops 'valid)))
    (match name
      ('rotate-termination
       (forall ?M ?i ?j ?k
         (if (and (valid (range ?i ?j))
               (and (valid (range ?i ?k))
                   (valid (range ?k ?j)))
           (rotate_t ?M (range ?i ?j) ?k)))))))

(define (rotate-termination-proof name ops)
  (dlet ((rotate (ops 'rotate))
         (rotate_t (ops rotate))
         (reverse-range (ops 'reverse-range))
         (reverse-range_t (ops reverse-range))
         (tprop
          (method (name)
            (!claim (termination-axiom
                      (get-property name ops Memory-range-theory)
                      ops)))))
      (theorem
       (rotate-termination name ops)))
 (!(conclude theorem)
 (dlet ((reverse-range-termination
   (forall ?i ?j
     (if (valid (range ?i ?j))
       (forall ?M
         (reverse-range_t ?M (range ?i ?j)))))
   (pick-any
    M i j k
    (assume-let (assumptions
      (and (valid (range i j))
       (and (valid (range i k))
        (valid (range k j))))))
   (dseq
    (!left-and assumptions)
    (!left-and (!right-and assumptions))
    (!right-and (!right-and assumptions))
    (!mp (!uspec* (!tprop 'rotate-axiom) [M i j k])
      (!both
        (!uspec (!mp (!uspec* reverse-range-termination
          [i j])
          (valid (range i j)))
        (reverse-range
         (reverse-range M (range i k))
         (range k j)))
      (!both (!uspec
        (!mp (!uspec* reverse-range-termination
          [k j])
          (valid (range k j)))
        (reverse-range M (range i k))))
      (!uspec
        (!mp (!uspec* reverse-range-termination
          [i k])
          (valid (range i k))))
(define (rotate-property-proof name ops)
  (dlet ((valid (ops 'valid))
          (rotate (ops 'rotate))
          (access-range (ops 'access-range))
          (append (ops 'append))
          (reverse (ops 'reverse))
          (left (cell true))
          (right (cell true))
          (prop (method (name)
                         (!property name ops Memory-range-theory)))
          (theorem (rotate-property name ops)))
  (conclude theorem)
  (pick-any M i j k
    (assume-let (assumptions (and (valid (range i k))
                                   (valid (range k j))))
                (valid (range i j))))
  (dseq
   (!left-and assumptions)
   (!right-and assumptions)
   (!both (valid (range k j))
      (valid (range i k)))
   (!conclude (valid (range i j)))
   (!instance (!prop 'Valid-Range-Transitive-property)
               [i k j])
   (!setup left (access-range (rotate M (range i j) k)
                            (range i j))))
(!setup right (append (access-range M (range k j))
   (access-range M (range i k))))

(dlet ((M1 (reverse-range M (range i k)))
   (M2 (reverse-range M1 (range k j))))

(dseq
  (!reduce left (access-range
   (reverse-range M2 (range i j))
   (range i j))
   (!prop 'rotate-axiom))
  (!reduce left (reverse
   (access-range M2 (range i j)))
   (!instance (!prop 'reverse-range-property)
                [i j]))
  (!reduce left (reverse
   (append (access-range M2 (range i k))
           (access-range M2 (range k j)))
   (!instance
    (!instance
     (!prop 'combined-memory-range-property)
     [i j]
     [M2 k])))
  (!reduce left
   (reverse
    (append (access-range M1 (range i k))
           (access-range M2 (range k j)))
    (!prop 'reverse-range-has-no-effect-outside3))
  (!reduce left
   (reverse
    (append
     (reverse (access-range M (range i k)))
     (access-range M2 (range k j))))
   (!instance (!prop 'reverse-range-property)
(reduce left
  (reverse
   (append
    (reverse (access-range M (range i k)))
    (reverse (access-range M1 (range k j)))))
  (!instance (!prop 'reverse-range-property) [k j]))

(reduce left
  (append
   (reverse
    (reverse (access-range M1 (range k j))))
   (reverse
    (reverse (access-range M (range i k))))))
reverse-append-property)

(reduce left
  (append
   (access-range M1 (range k j))
   (reverse
    (reverse (access-range M (range i k))))))
reverse-reverse-property)

(reduce left
  (append (access-range M1 (range k j))
    (access-range M (range i k)))
reverse-reverse-property)
((conclude (= (access-range M1 (range k j))
  (access-range M (range k j))))

(instance
  (!prop 'reverse-range-has-no-effect-outside2) [k j M i]))
(reduce left (append (access-range M (range k j))
  (access-range M (range i k)))
  (= (access-range M1 (range k j)))
7.1.3 Unit test file: rotate-implementation_unittest.ath
### Test rotate-implementation.ath.

(load-file "test-memory.ath")
(load-file "test-iterator.ath")
(load-file "test-range.ath")
(load-file "test-memory-range.ath")
(load-file "test-reverse-range.ath")
(load-file "extractor2.ath")
(load-file "rotate-implementation.ath")

(declare rotate ((T) ->
                 (Memory (Range (Iterator T)) (Iterator T))
                 Memory))

(define (rotate-ops op)
  (match op
         ('rotate rotate)
         (_ (rev-ops op)))))

(define rotate-axioms
  [(get-property 'rotate-axiom rotate-ops Memory-range-theory)])

########################################################################
# Consistency:

(! (conclude
         (consistency-condition rotate-axioms))
     (!property 'rotate-consistency rotate-ops Memory-range-theory))
# Termination:

(declare rotate_t ((T) ->
    (Memory (Range (Iterator T)) (Iterator T))
    Boolean))

(declare reverse-range_t ((T) ->
    (Memory (Range (Iterator T)))
    Boolean))

(enter rotate_t-opdefs [reverse-range reverse-range_t])
(enter rotate_t-opdefs [rotate rotate_t])
(enter rotate_t-opdefs [range range])

(define rotate-termination-axioms
    [(termination-axiom
      (get-property 'rotate-axiom rotate-ops Memory-range-theory)
      rotate_t-opmap)])

(assert rotate-termination-axioms)

(!property 'rotate-termination
    (join-maps rotate-ops rotate_t-opmap)
    Memory-range-theory)

# Consistency and termination proved, so assert the axioms
# and test the proof of the correctness property:

(assert rotate-axioms)
(define clause
 (make-clause rotate-ops Memory-range-theory))

(enter sym-map ["rotate" "Rotate"])

(extract rotate [?M ?r ?k]
    [(!clause 'rotate-axiom [?M ?i ?j ?k])]
    true
    "rotate.oz")

7.1.4 Generated Oz code: rotate.oz
declare
fun {Rotate M R K}
case R
  of range(I J) then {ReverseRange
                        {ReverseRange
                        {ReverseRange M
                        range(I K)}
                        range(K J)}
                        range(I J)}
  end
end

declare
fun {Rotate M R K}
case R
  of range(I J) then {ReverseRange {ReverseRange {ReverseRange M range(I K)} range(K J)} range(I J)}
end
7.1.5 Unit test file for generated code: rotate_unittest.oz

declare
\insert 'basis.oz'
\insert 'testing.oz'
\insert 'reverse-range.oz'
\insert 'rotate.oz'

N = 10
M = {Array.new 0 N-1 0}

proc {Initialize}
  for I in 0..N-1 do {Array.put M I 5*I} end
end

fun {RunRotate M I J K}
  {Show {Rotate M range(I J) K}}
  {AccessRange M range(I J)}
end

% Show what we're initializing the array to.
{Initialize}
{Testing "{AccessRange M range(0 N)}"}
{Check
 {AccessRange M range(0 N)}
 expecting([0 5 10 15 20 25 30 35 40 45])}

% Rotate (left) by 3 elements.
{Testing "{RunRotate M 0 N 3}"}
7.2 A generic find-if function

The find-if function specified, correctly implemented, and tested in this section is very similar to the STL’s find-if function. It takes three parameters: a memory, a range, and a predicate. It returns the first iterator in the given range such that the value pointed to by the iterator satisfies the given predicate. If none of the values in this range satisfies the predicate, the last iterator in the range is returned.
7.2.1 Specification file: find-if-specification.ath

# Correctness properties of find-if, a pure function based
# on the STL find_if function, defined on a valid range.

(load-file "iterator.ath")
(load-file "range.ath")
(load-file "memory-range.ath")

# Correctness conditions:

## 1. If k is returned, p isn’t true at any i in [i, k).
(define (find-if-property1 name ops)
  (let ((valid (ops 'valid))
        (find-if (ops 'find-if))
        (in (ops 'in))
        (apply (ops 'apply))
        (access (ops 'access))
        (* (ops '*)))
    (match name
      ('find-if-property1
       (forall ?i ?j
        (if (valid (range ?i ?j))
          (forall ?M ?p ?k
            (if (= ?k (find-if ?M (range ?i ?j) ?p))
              (not (exists ?u
                 (and (in ?u (range ?i ?k))
                 (apply ?p
                  (access ?M (* ?u)
                   )))))))))))))

## 2. If k is returned and k != j, p is true at k.
(define (find-if-property2 name ops)
  (let ((valid (ops 'valid))
        (find-if (ops 'find-if))
        (in (ops 'in))
        (apply (ops 'apply))
        (access (ops 'access))
        (* (ops '*)))
    (match name
      ('find-if-property2
       (forall ?i ?j
        (if (valid (range ?i ?j))
          (forall ?M ?p ?k
            (if (= ?k (find-if ?M (range ?i ?j) ?p))
              (not (exists ?u
                 (and (in ?u (range ?i ?k))
                 (apply ?p
                  (access ?M (* ?u)
                   )))))))))))))
(in (ops 'in))
(app (ops 'app))
(access (ops 'acc))
(* (ops '*)))

(match name
('find-if-property2
(forall ?i ?j
  (if (valid (range ?i ?j))
    (forall ?M ?p ?k
      (if (and
        (= ?k (find-if
        (not (= ?k ?j)))
        (and (in ?k (range ?i ?j))
          (app ?p
            (access ?M (* ?k)
              )))))))))

## 3. If there is a u in [i, j) for which p is true,
## j is not returned.
(define (find-if-property3 name ops)
  (let ((valid (ops 'valid))
         (find-if (ops 'find-if))
         (in (ops 'in))
         (app (ops 'app))
         (access (ops 'acc))
         (* (ops '*)))
    (match name
      ('find-if-property3
        (forall ?i ?j
          (if (valid (range ?i ?j))
            (forall ?M ?p
              (if (exists ?u

```
7.2.2 Implementation file: find-if-implementation.ath

# Implementation of find-if, satisfying the specification
# given in find-if-specification.ath.

(load-file "find-if-specification.ath")

(define (find-if-definition name ops)
  (let ((find-if (ops 'find-if))
        (apply (ops 'apply))
        (access (ops 'access)))
    (match name
      ('find-if-axiom1
        (if (= ?i ?j)
         (= (find-if ?M (range ?i ?j) ?p) ?i)))))
      ('find-if-axiom2
        (if (and (valid (range ?i ?j)))
         (and (not (= ?i ?j))
          (apply ?p (access ?M (* ?i))))))
         (= (find-if ?M (range ?i ?j) ?p) ?i))))
      ('find-if-axiom3
        (if (and (valid (range ?i ?j)))
         (and (not (= ?i ?j))
          ))
(not (apply ?p (access ?M (* ?i)))))

(= (find-if ?M (range ?i ?j) ?p)
    (find-if ?M (range (++ ?i) ?j) ?p)))))))

(evolve Range-theory [find-if-definition Axiom])

# For consistency:

(define (find-if-consistency name ops)
  (match name
    (find-if-consistency
      (consistency-condition
        (map (function (name)
               (get-property name ops Range-theory))
              ['find-if-axiom1 'find-if-axiom2 'find-if-axiom3])))))

(define (find-if-consistency-proof name ops)
  (dlet ((theorem (find-if-consistency name ops)))
    (!prove-from theorem []))

(evolve Range-theory [find-if-consistency
                       find-if-consistency-proof])

# For termination:

(declare find-if_t ((T) ->
                      (Memory
                       (Range (Iterator T))
                       (Predicate T))
                      Boolean))
(define find-if_t-opdefs (cell []))
(define (find-if_t-opmap op)
  (find op (ref find-if_t-opdefs)))

(define (find-if-termination name ops)
  (match name
    ('find-if-termination
      (let ((valid (ops 'valid)))
          (if (valid (range ?i ?j))
            (find-if_t ?M (range ?i ?j) ?p)))))))

(define (find-if-termination-induction-cases ops)
  (dlet ((find-if (ops 'find-if))
          (find-if_t (ops find-if))
          (valid (ops 'valid))
          (++ (ops '++))
          (* (ops '*))
          (access (ops 'access))
          (apply (ops 'apply))
          (prop (method (name)
                        (!property name ops Memory-range-theory)))
          (left (cell true))
          (right (cell true))
          (tprop
            (method (name)
             (!claim (termination-axiom
                          (get-property name ops Memory-range-theory)
                          ops))))
          (P (function (i j)
               (forall ?M ?p
                (find-if_t ?M (range i j) ?p))))))

  (!both
((conclude (range-basis-case P ops))
 (pick-any
  i M p
  (\mp (\uspec* (!tprop 'find-if-axiom1) [i i M p])
    (!equality i i))))
((conclude (range-induction-step P ops))
 (pick-any
  i j
  (assume-let (facts (and (not (= i j))
                        (and (valid (range i j))
                             (P (++ i) j))))))
((conclude (P i j))
 (pick-any
  M p
  (!cases
   (assume (apply p (access M (* i))))
   (dseq
    (!both (!left-and (!right-and facts))
      (!both (!left-and facts)
        (apply p (access M (* i))))))
    (!instance (!tprop 'find-if-axiom2)
      [i j M p])))
  (assume (not (apply p (access M (* i))))
    (!\mp (dseq
      (!both (!left-and (!right-and facts))
        (!both (!left-and facts)
          (not (apply p
            (access
              M (* i))))))
      (!instance (!tprop 'find-if-axiom3)
        [i j M p]))
    (!uspec* (!right-and (!right-and facts))
      [M p]))))))
(define (find-if-termination-proof name ops)
  (!range-induction
   (!find-if-termination-induction-cases ops)
   (symbol-definition
    (add name (map ops ['find-if 'valid]))
    Memory-range-symdefs)
   ops))

(evolve Range-theory [find-if-termination
     find-if-termination-proof])

##################################################################
# Proof of find-if-property1 by range-induction:

(define (find-if-property1-induction-cases ops)
  (dlet ((++ (ops '++))
         (-- (ops '--))
         (* (ops '*))
         (I- (ops 'I-))
         (I-I (ops 'I-I))
         (find-if (ops 'find-if))
         (apply (ops 'apply))
         (access (ops 'access))
         (valid (ops 'valid))
         (P (function (a b)
              (forall
               ?M ?p ?k
               (if (= ?k (find-if ?M (range a b) ?p))
                (not
                 (exists ?u
                  (and (in ?u (range a ?k))
                    (apply ?p (access ?M (* ?u))))))))))))
(left (cell true))
(right (cell true))
(prop (method (name)
  (!property name ops Memory-range-theory))))
(!both
(!(conclude (range-basis-case P ops))
(pick-any
 i M p k
(assume-let
 (assumption (= k (find-if M (range i i) p)))))
(dseq
(!(conclude (= k i)))
(dseq
(!mp (!uspec* (!prop 'find-if-axiom1) [i i M p])
  (!equality i i))
  (!substitute-equals (find-if M (range i i) p)
    assumption i)))
(!by-contradiction
(assume-let (solution-found
  (exists ?u
    (and (in ?u (range i k))
      (apply p (access M (* ?u)))))))
(pick-witness u
 solution-found
(dlet ((witnessed
    (and (in u (range i k))
      (apply p (access M (* u))))))
(!absurd
  (!substitute-equals k (!left-and witnessed)
    i)
  (!instance (!prop 'not-in-empty-range)
    [i u]))))))
(!conclude (range-induction-step P ops))
(pick-any
  i j
(assume-let
  (inductive
    (and (not (= i j))
      (and (valid (range i j))
        (forall
          ?M ?p ?k
          (if (= ?k
              (find-if ?M (range (++ i) j)
                ?p))
            (not
              (exists ?u
                (and (in ?u (range (++ i) ?k))
                  (apply ?p
                    (access ?M
                      (* ?u)))))
            ))))))))
(pick-any
  M p k
(assume-let (assumption
    (= k (find-if M (range i j) p)))
(!by-contradiction
(assume-let
  (solution-found
    (exists ?u
      (and (in ?u (range i k))
        (apply p (access M (* ?u)))))
  (pick-witness u
  solution-found
  (dlet ((witnessed
          (and (in u (range i k))
            (apply p (access M (* u)))))))
  )))
(assume (apply p (access M (* i))))
(dseq
  (!conclude (= k i))
  (dseq
    (!setup left k)
    (!reduce left
      (find-if M (range i j) p)
      assumption)
    (!both (!left-and
      (!right-and inductive))
      (!both
       (!left-and inductive)
       (apply p
         (access M (* i)))))
    (!reduce left i
      (!prop 'find-if-axiom2)))))
(absurd (!substitute-equals
  k (!left-and witnessed) i)
  (!instance
   (!prop 'not-in-empty-range)
   [i u]))))
(assume (not (apply p (access M (* i))))
(dseq
  (!conclude
    (= k (find-if M (range (++ i) j)
      p)))
  (dseq
    (!setup left k)
    (!setup
      right (find-if M (range (++ i) j)
      p)))
    (!reduce left
(find-if M (range i j) p)
assumption)

(!both
(!left-and
(!right-and inductive))
(!both
(!left-and inductive)
(not (apply p
(access M (* i))))))
(!reduce left
(find-if M (range (++ i) j)
p)
(!prop 'find-if-axiom3)))

(!absurd
(!conclude
(exists ?u
(and (in ?u (range (++ i) k))
(apply p (access M
(* ?u))))))
(dseq
(!both
solution-found
(not (apply p
(access M (* i))))))
(!instance
(!prop 'true-in-smaller-range)
[i k M p])))
(!conclude
(not (exists ?u
(and (in ?u
(range (++ i) k))
(apply p
(access M
(* i))))))
(* ?u)
))))))

(!instance
(!right-and
  (!right-and inductive))

[0 M p k]))))))))))))))))))))))

(define (find-if-property1-proof name ops)
  (!range-induction
   (!find-if-property1-induction-cases ops)
   (symbol-definition
    (add name (map ops ['valid 'find-if 'in 'apply 'access '*]))
    Memory-range-symdefs)
   ops))

(evolve Memory-range-theory [find-if-property1
    find-if-property1-proof])

########################################################################

# Proof of find-if-property2 by range-induction:

(define (find-if-property2-induction-cases ops)
  (dlet ((++ (ops '++))
      (-- (ops '--))
      (* (ops '*))
      (I- (ops 'I-))
      (I-I (ops 'I-I))
      (find-if (ops 'find-if))
      (apply (ops 'apply))
      (access (ops 'access))
      (valid (ops 'valid))
      (P (function (i j)))
      (forall
        (P (i j)))))

########################################################################
126

?M ?p ?k
(if (and (= ?k (find-if ?M (range i j) ?p))
  (not (= ?k j)))
  (and (in ?k (range i j))
    (apply ?p (access ?M (* ?k))))))))
  (left (cell true))
  (right (cell true))
  (prop (method (name)
    (!property name ops Memory-range-theory))))
(!both
  (!!(conclude (range-basis-case P ops))
    (pick-any
      i M p k
      (assume-let
        (assumption
          (and (= k (find-if M (range i i) p))
            (not (= k i))))
        (!by-contradiction
          (assume (not (and (in k (range i i))
            (apply p (access M (* k))))))
          (!absurd (!!(conclude (= k i))
            (dseq
              (!mp (!uspec*
                (!prop 'find-if-axiom1)
                [i i M p])
                (!equality i i))
              (!substitute-equals
                (find-if M (range i i) p)
                (!left-and assumption) i)))
              (!right-and assumption))))))))
  (!!!(conclude (range-induction-step P ops))
    (pick-any
      i j
(assume-let
  (inductive
    (and (not (= i j))
      (and (valid (range i j))
        (P (++ i) j)))
  )
  (!conclude (P i j))
  (pick-any
    M p k
    (assume-let (assumption
      (and (= k (find-if M (range i j) p))
        (not (= k j)))
    )
    (!cases
      (assume (apply p (access M (* i))))
      (dseq
        (!sym
          (!conclude (= k i))
          (dseq
            (!setup left k)
            (!reduce left
              (find-if M (range i j) p)
              (!left-and assumption)
            )
            (!both (!left-and
              (!right-and inductive))
            )
            (!both (!left-and !left-and inductive)
              (!right-and inductive)
            )
            (!both
              (!left-and inductive)
              (apply p (access M (* i)))
            )
            (!reduce left i
              (!prop 'find-if-axiom2))
            )
          )
        )
      )
    )
  )
(!left-and inductive))
(!pos-substitute-equals
 i (!instance
   (!prop 'first-in-range-property)
   [i j])
   [1] k)))
(!(conclude (apply p (access M (* k))))
(!substitute-equals
 i
   (apply p (access M (* i))) k))))
(assume (not (apply p (access M (* i)))))
(dseq
  (!both
    (!(!conclude
       (= k (find-if M (range (++ i) j) p))))
    (dseq
     (!setup left k)
     (!reduce left (find-if M (range i j) p)
      (!left-and assumption))
     (!both (!left-and
       (!right-and inductive))
     (!both (!left-and inductive)
       (!both (!left-and inductive)
         (not
         (apply p (access M (* i)))))))))
    (!reduce
     left (find-if M (range (++ i) j) p)
     (!prop 'find-if-axiom3)))
    (!right-and assumption))
  (dlet ((almost-done
    (!(!conclude
       (and (in k (range (++ i) j))
       (apply p
(access M (* k)))))

(!instance
  (!right-and
    (!right-and inductive))
  [M p k])))])

(!both
  (!((conclude (in k (range i j)))
    (!by-contradiction
      (assume (not (in k (range i j)))
        (dseq
          (!both
            (!left-and
              (!right-and inductive))
            (!both (!left-and inductive)
              (not (in k (range i j))))))
          (!absurd
            (!left-and almost-done)
            (!instance
              (!prop 'not-in-smaller-range1)
              [i j k])))])
        (!right-and almost-done))))))))))))

(define (find-if-property2-proof name ops)
  (!range-induction
    (!find-if-property2-induction-cases ops)
    (symbol-definition
      (add name (map ops ['valid 'find-if 'in 'apply 'access '*])
        Memory-range-symdefs)
      ops))

  (evolve Memory-range-theory [find-if-property2
                              find-if-property2-proof])
# Proof of find-if-property3 by range-induction:

(define (find-if-property3-induction-cases ops)
  (dlet (;++ (ops '++))
    (-- (ops '--))
    (* (ops '*))
    (I- (ops 'I-))
    (I-I (ops 'I-I))
    (find-if (ops 'find-if))
    (apply (ops 'apply))
    (access (ops 'access))
    (valid (ops 'valid))
    (P (function (a b)
        (forall ?M ?p
          (if (exists ?u
              (and (in ?u (range a b))
                (apply ?p (access ?M (* ?u))))))
            (not (= (find-if ?M (range a b) ?p)
                b))))))
  (left (cell true))
  (right (cell true))
  (prop (method (name)
    (!property name ops Memory-range-theory))))
  (!both
    (! (conclude (range-basis-case P ops))
      (pick-any
        i M p
        (assume-let (assumption
          (exists ?u
            (and (in ?u (range i i))
              (apply p (access M (* ?u))))))
            (!by-contradiction
(assume (= (find-if M (range i i) p) i)

(pick-witness u assumption
(dseq
  (!((conclude (in u (range i i))))
   (!left-and (and (in u (range i i))
                (apply p (access M (* u)))))
  (pick-witness n
   (!((conclude (exists ?n
                    (and (and (not (= ?n zero))
                         (= ?n (I-I i i)))
                      (<= ?n (I-I i i)))
                    (= u (I- i ?n)))))))
   (!instance (!prop 'in-definition)
               [i i u]))
  (dlet ((witnessed
          (and (and (not (= n zero))
               (= n (I-I i i)))
               (= u (I- i n)))))))
  (dseq
   (!((conclude (<= n zero))))
   (dseq
    (!instance (!prop 'I-I-zero-property)
                [i])
    (!substitute-equals
     (I-I i i)
     (!right-and
      (!left-and witnessed))
     zero)))))

(!cd
  (!((conclude (or (< n zero) (= n zero))))
   (!instance <=-definition [n zero]))
  (assume (< n zero)
(!absurd (< n zero)
  (!instance <-not-zero-axiom
   [n])))

(assume (= n zero)
  (!absurd (= n zero)
    (!left-and
     (!left-and witnessed))))))))

((conclude (range-induction-step P ops))
  (pick-any
    i j
    (assume-let
      (inductive
        (and (not (= i j))
          (and (valid (range i j))
            (forall
              ?M ?p
              (if (exists ?u
                  (and (in ?u (range i j))
                    (apply ?p
                      (access ?M
                        (* ?u))))))
                (not
                  (= (find-if ?M (range i j) ?p)
                    j))))))))
  (pick-any
    M p
    (assume-let (assumption
      (exists ?u
        (and (in ?u (range i j))
          (apply p (access M (* ?u))))))
      (!by-contradiction
       (exists ?u
         (and (in ?u (range i j))
           (apply p (access M (* ?u))))))))
\[\text{(assume \(=\) (find-if} \text{M} \text{(range i j) p)}\]

\] j} 

(dseq  
(!cases 

(assume (apply p (access M \(*\ i))) 

(dseq 
(!both 
(!left-and (!right-and inductive))  
(!both (!left-and inductive)  
(apply p (access M \(*\ i)))) 
(!conclude \(=\) (find-if \text{M} \text{(range i j)} p) \ i)) 
(!instance (!prop 'find-if-axiom2) 
[i j M p])) 
(!absurd (!conclude \(=\) i j)) 
(!tran 
(!sym 
\(=\) (find-if \text{M} \text{(range i j)} p) \ i)) 
\(=\) (find-if \text{M} \text{(range i j)} p) \ j)))) 
(!left-and inductive)))) 

(assume (not (apply p (access M \(*\ i)))) 

(dseq 
(!both (!left-and (!right-and inductive)) 
(!both (!left-and inductive) 
(not (apply p (access M 
\(*\ i)))))) 
(!conclude 
\(=\) (find-if \text{M} \text{(range i j)} p)  
(find-if \text{M} \text{(range \((+\ i)\) j) p)\)) 
(!instance (!prop 'find-if-axiom3)
[i j M p])
(!both assumption
  (not (apply p (access M (* i)))))
(!conclude
  (exists ?u
    (and (in ?u (range (++ i) j))
      (apply p (access M (* ?u)))))
  )
(!instance
  (!prop 'true-in-smaller-range)
  [i j M p])
(!conclude
  (not (= (find-if M (range (++ i) j) p)
         j))
  )
(!instance
  (!right-and (!right-and inductive))
  [M p])
(!absurd
  (!conclude
    (= (find-if M (range (++ i) j) p)
        j))
  )
(!tran
  (!sym (= (find-if M (range i j) p)
        (find-if M (range (++ i) j) p)))
  (= (find-if M (range i j) p)
      j))
  )
  (not (= (find-if M (range (++ i) j) p)
          j))))))))))

(define (find-if-property3-proof name ops)
  (!range-induction
   (!find-if-property3-induction-cases ops)
   (symbol-definition

7.2.3 Unit test file: find-if-implementation_unittest.ath
### Test find-if-implementation.ath.

(load-file "test-memory.ath")
(load-file "test-iterator.ath")
(load-file "range-induction.ath")
(load-file "test-range.ath")
(load-file "test-memory-range.ath")
(load-file "consistency.ath")
(load-file "termination-axiom.ath")
(load-file "extractor2.ath")
(load-file "find-if-implementation.ath")
(get-property name find-if-ops Range-theory)
['find-if-axiom1 'find-if-axiom2 'find-if-axiom3])

#################################################################
# Consistency: #

(! (conclude (consistency-condition find-if-axioms))
 (!property 'find-if-consistency find-if-ops Range-theory))

#################################################################
# Termination: #

(enter find-if_t-opdefs [find-if find-if_t])
(enter-pair find-if_t-opdefs valid)
(enter-pair find-if_t-opdefs apply)
(enter-pair find-if_t-opdefs Access)
(enter-pair find-if_t-opdefs *)
(enter-pair find-if_t-opdefs ++)
(enter-pair find-if_t-opdefs range)

(assert
 (map (function (axiom)
 (termination-axiom
 axiom
 (join-maps find-if-ops find-if_t-opmap)))
 find-if-axioms)))

(define-symbol
 find-if-termination|find-if|valid
 (forall
 ?i ?j
 (iff
 (find-if-termination|find-if|valid ?i ?j)
(forall ?M ?p
   (find-if-t ?M (range ?i ?j) ?p)))

(enter Memory-range-symdefs
   [(add 'find-if-termination
      [find-if valid])
      (head (ab))])

(!property-test 'find-if-termination
   (join-maps find-if-ops find-if_t-opmap)
   Range-theory)

# Consistency and termination proved, so assert the axioms
# and test the proofs of the correctness properties:

(assert find-if-axioms)

# Property 1:
(define-symbol
   find-property1|valid|find-if|in|apply|Access|*
   (forall ?i ?j
      (iff (find-property1|valid|find-if|in|apply|Access|* ?i ?j)
         (forall ?M ?p ?k
            (if (= ?k (find-if ?M (range ?i ?j) ?p))
               (not (exists ?u
                  (and (in ?u (range ?i ?k))
                     (apply ?p
                        (Access ?M (* ?u)))))
               )))

(enter
   Memory-range-symdefs
([add 'find-if-property1
   (map find-if-ops [ 'valid 'find-if 'in 'apply 'access '])
   (head (ab))])

(!property-test 'find-if-property1 find-if-ops Memory-range-theory)

# Property 2:
(define-symbol
  find-if-property2|valid|find-if|in|apply|Access|*
  (forall ?i ?j
    (iff (find-if-property2|valid|find-if|in|apply|Access|* ?i ?j)
      (forall ?M ?p ?k
        (if (and (= ?k (find-if ?M (range ?i ?j) ?p))
          (not (= ?k ?j)))
          (and (in ?k (range ?i ?j))
            (apply ?p (Access ?M (* ?k))))))))

(enter
  Memory-range-symdefs
  [(add 'find-if-property2
   (map find-if-ops [ 'valid 'find-if 'in 'apply 'access '])
   (head (ab))])

(!property-test 'find-if-property2 find-if-ops Memory-range-theory)

# Property 3:
(define-symbol
  find-if-property3|valid|find-if|in|apply|Access|*
  (forall
    ?i ?j
    (iff (find-if-property3|valid|find-if|in|apply|Access|* ?i ?j)
      (forall ?M ?p
        (if (exists ?u
(and (in ?u (range ?i ?j))
   (apply ?p (Access ?M (* ?u))))
   (not (= (find-if ?M (range ?i ?j) ?p)
   ?j)))))

(enter
Memory-range-symdefs
[(add 'find-if-property3
   (map find-if-ops ['valid 'find-if 'in 'apply 'access '*]))
   (head (ab))])

(!property-test 'find-if-property3 find-if-ops Memory-range-theory)

########################################################################
# Correctness properties proved, so proceed with code generation:

(enter sym-map ["find-if" "FindIf"])
(enter sym-map ["apply" "Apply"])
(extract-code (find-if ?M ?r ?p) (rev find-if-axioms)
   (valid (range ?i ?j))
   "find-if.oz")

7.2.4 Generated Oz code: find-if.oz
declare
fun {FindIf M R P}
  case R
  of range(I J) then if (I == J) then I
      elseif {Apply P {Access M {'*' I}}} then I
      else {FindIf M range({'+' I} J) P}
      end
  end
end
7.2.5 Unit test file for generated code: find-if_unittest.oz

declare
\insert 'basis.oz'
\insert 'testing.oz'
\insert 'find-if.oz'

N = 10
M = {NewArray 0 N-1 0}

proc {Initialize}
    for I in 0..N-1 do {Array.put M I 3*I} end
end

fun {Greater20 X}
    X > 20
end

fun {Less20 X}
    X < 20
end

% Show what we're initializing the array to.
{Initialize}
{Testing "{AccessRange M range(0 N)}"}
{Check
    {AccessRange M range(0 N)}
    expecting([0 3 6 9 12 15 18 21 24 27])}
{Testing "{FindIf M range(0 N) Even}"}
{Check
  {FindIf M range(0 N) Even}
  expecting(0)}

{Testing "{FindIf M range(1 N) Even}"}
{Check
  {FindIf M range(1 N) Even}
  expecting(2)}

{Testing "{FindIf M range(0 N) {Negate Even}}"}
{Check
  {FindIf M range(0 N) {Negate Even}}
  expecting(1)}

{Testing "{FindIf M range(0 N) Greater20}"}
{Check
  {FindIf M range(0 N) Greater20}
  expecting(7)}

{Testing "{FindIf M range(0 N) {Negate Greater20}}"}
{Check
  {FindIf M range(0 N) {Negate Greater20}}
  expecting(0)}

{Testing "{FindIf M range(0 N) Less20}"}
{Check
  {FindIf M range(0 N) Less20}
  expecting(0)}

{Testing "{FindIf M range(0 N) {Negate Less20}}"}
{Check
  {FindIf M range(0 N) {Negate Less20}}
  expecting(7)}
7.3 Testing the PartialPartition function

We have not carried out a CCT development of PartialPartition and only have the handwritten code given at the beginning of the chapter. Here we close the example with a small set of unit tests of this function.

```
\insert 'basis.oz'
\insert 'testing.oz'
\insert 'find-if.oz'
\insert 'reverse-range.oz'
\insert 'rotate.oz'
\insert 'partial-partition.oz'

N = 10
M = {Array.new 0 N-1 0}

proc {Initialize}
    {Array.put M 0 2} {Array.put M 1 4} {Array.put M 2 5}
    {Array.put M 3 7} {Array.put M 4 4} {Array.put M 5 0}
    {Array.put M 6 8} {Array.put M 7 11} {Array.put M 8 1}
    {Array.put M 9 6}
end

fun {RunPP M I J P}
    {Show {PartialPartition M range(I J) P}}
    {AccessRange M range(I J)}
end

% Show what we're initializing the array to.
{Initialize}
```
{Testing "{AccessRange M range(0 N)}"}
{Check \{AccessRange M range(0 N)\}
  expecting([2 4 5 7 4 0 8 11 1 6])}

{Testing "{PartialPartition M range(0 N) \{Negate Even\}}"}
{Check \{RunPP M 0 N \{Negate Even\}\}
  expecting([5 7 2 4 4 0 8 11 1 6])}

{Testing "{PartialPartition M range(2 N) \{Negate Even\}}"}
{Check \{RunPP M 2 N \{Negate Even\}\}
  expecting([11 1 2 4 4 0 8 6])}

{Testing "{AccessRange M range(0 N)}"}
{Check \{AccessRange M range(0 N)\}
  expecting([5 7 11 1 2 4 4 0 8 6])}
CHAPTER 8
Conclusions

In this dissertation, we first categorized the potential problems of mobile code during delivery as security, safety, and correctness problems. With CCT, we have explored solutions for correctness and, to a limited extent, safety issues. Our main concern was to solve correctness problems, which are often harder than safety problems. CCT provides strong assurance with the help of CCGen, TCGen, and CodeGen. Both code producers and consumers have CCGen and TCGen to produce verification conditions and termination axioms. Producers do the hard work and prove the termination, consistency, and correctness properties. After the definitions and proofs are received from the producer, the consumer first generates the termination and consistency properties. The next step is to check their proofs with a fast and a simple process. If the correctness proof is valid too, CodeGen is applied to the function-defining axioms sent by the producer in order to produce code.

CCT is flexible and general. Constructing specifications and requirements is a very easy task and does not require any fancy knowledge of hardware or machine instructions. One can define functions with basic mathematics and first-order logic. Consumers do not need to know the identity of the producers. With its small computing base, CCT is more trustable than PCC and it can provide solutions for small computers too.

One of the building blocks of our research was writing readable, easily maintainable, and machine-checkable proofs. In addition, we went a step further, designing and implementing a set of functions and methods to make it easier to write generic proofs. We developed a mechanism to organize theorems and proofs. Related theorems and proofs can be added dynamically to existing theories, enabling proof and theorem libraries to be easily extended. We tested CCT with thousands of lines of generic definitions and generic proofs, which also helped us to reduce proof sizes. In many cases, producers do not need to send similar definitions repeatedly since instantiation of existing generic theorems and proofs is sufficient for consumers. In
PCC, a methodology called “lemma extraction” is used for the same purpose, but to be able to do this, automatically generated proofs should be processed, requiring more computation power.
Proofs about the properties of programs are often large compared to the amount of code to be extracted. In CCT, we showed that proving generic lemmas, adding them to the theories, using them in the proofs of main theorems, and properly instantiating them at the consumer’s site can significantly reduce the amount of information that needs to be transferred to the consumer. However, it would be helpful to compare the amount of information transferred in both nongeneric and generic cases. We have already tested our ideas with many examples in CCT. For most of them, we have produced both nongeneric and generic proofs which can be used to give a ratio.

In Chapter 7, we presented one of our larger examples of the CCT approach. We also proved properties of several generic STL functions that can be used to compose complex applications. Even though the CCT approach worked successfully for these examples, it should be applied to other programs, including ones that are larger and more complex.

The current version of CODEGEN does not extract generic code. If the producer sends generic definitions and proofs that satisfy the requirements, CODEGEN is applied to an instantiated set of generic axioms or theorems. Instead, CODEGEN could be updated to process more generic axioms and theorems and produce generic Oz code by using higher-order functions. While not having this capability is not a problem in CCT, improving CODEGEN along this line would save consumers from having to extract similar code many times. Once the generic function is available, the same code can be used with different instantiations.

While executing CODEGEN, we provide a precondition in addition to the function-defining axioms to eliminate some checks from the extracted code. An intruder could change the precondition during transmission. Even if the other definitions and proofs satisfy the requirements, there is currently no way that the consumer can detect this problem, which may cause CODEGEN to extract in-
efficient or even non-executable code. For example, if we pass \texttt{true} instead of \texttt{(valid (range ?i ?j))} to the code extractor with the \texttt{reverse-range} axioms, it will not eliminate the valid-range check from the resultant code. But in most contexts there is no practical way to implement the valid-range check (the obvious way would be nonterminating when the range is invalid).

CCT can provide assurance for both correctness and safety of programs. Nevertheless, we have mainly considered correctness requirements and concentrated on proving correctness of functions defined by the code producers. Even though it is harder to solve correctness issues, we presented many examples where CCT was successful. A direction for future research is to define various safety policies and prove safety of programs delivered to consumers.

One of Tony Hoare’s grand challenge problems is a verifying compiler [18], which can check the correctness of the programs that it compiles. Current compilers cannot return any helpful error messages if, for example, a sorting program compiles but does not sort its input due to violation of semantic requirements. Adding proof checking to compilers for semantic analysis could be a small step to solve this problem. If we could define algorithms by axioms as in CCT, we could prove termination and consistency properties, and show that the program extracted from these axioms works correctly by providing and checking correctness proofs. CCT already provides tools and a working mechanism for doing this, which could be used in a compiler to give more meaningful error messages about the correctness of a given code segment. In that case, \texttt{CCGen} and \texttt{TGen} can be used as part of the compiler for consistency and termination checking, though, of course, not in all cases because of undecidability. If such checks succeed, then the proof checker can be used to check the proof of correctness conditions. If the axioms pass this last check too, a tool like \texttt{CodeGen} could be used within the compiler to extract correct code.
LITERATURE CITED


citeseer.nj.nec.com/caldwell02representing.html. 1


APPENDIX A

Athena

The Athena language and proof system developed at MIT by K. Arkoudas provides a way to present and work with mathematical proofs. It provides a language both for ordinary computation and for logical deduction. As a functional programming language, it provides higher-order functions and is lexically scoped in the tradition of Scheme and ML. As a deductive programming language, Athena makes it possible to formulate and write proofs as function-like constructions called methods whose executions either result in theorems if they properly apply rules of inference, or stop with an error condition if they do not. It is an example of a denotational proof language (DPL) [2] and has been proved sound and complete. A proof can be expressed at a high-level of abstraction, allowing the user to call the functions and methods defining it in many different ways. Therefore the same proof does not have to be redeveloped for different parameters. A generic library of proofs can be constructed with this ability.

Athena has an abstraction called the assumption base, an associative memory maintained by the system as it applies deductive methods; it is basically a record of what propositions have been asserted or proved since the beginning of the current session (or since the most recent clear-assumption-base directive which removes all of the propositions in the assumption base). All of Athena’s primitive methods refer to the assumption base and most extend it with new propositions (none can remove a proposition from it).

Athena has primitive functions for unification, matching, substitution and mapping. ML-like pattern matching is provided for defining both functions and methods. Athena also offers support for both forward and backward inference steps, proofs by induction, proof by contradiction, and equational reasoning.

A cell is a value that contains another value. It acts as memory location into which other values can be placed. When a cell is created it is given an initial value. Cells (and only cells) can be updated (by set!). Note that this is the same as in
ML but different from Scheme, which allows for unrestricted updating of the value associated with any symbol. A cell can contain another cell, since cells are values.

A.1 Primitive deduction methods

ATHENA provides fifteen primitive deduction methods, which will be described in this section.

\texttt{claim }P\

succeeds if proposition \(P\) is in the assumption base, and fails (results in an error) otherwise. If succeeds, it returns \(P\) as a theorem.

\texttt{mp (if }P Q\text{) }P\

adds the proposition \(Q\) to the assumption base if both (if \(P Q\)) and \(P\) are in the assumption base.

\texttt{both }P Q\

adds the proposition (and \(P Q\)) to the assumption base when \(P\) and \(Q\) are both in the assumption base. Note that the order in which the arguments are given to the both method is important because (and \(P Q\)) and (and \(Q P\)) are considered to be distinct propositions — commutativity of \texttt{and} is not automatically taken into account.

\texttt{left-and (and }P Q\text{)}\

adds the proposition \(P\) to the assumption base when (and \(P Q\)) is in the assumption base.

\texttt{right-and (and }P Q\text{)}
adds the proposition $Q$ to the assumption base when (and $P \land Q$) is in the assumption base.

One can use a combination of left-and, right-and, and both to reorder the arguments of and (which, as previously noted, may be occasionally necessary because commutativity of and is not automatically taken into account)

| equiv (if $P \land Q$) (if $Q \land P$) |
| adds the proposition (iff $P \land Q$) to the assumption base when both (iff $P \land Q$) and (iff $Q \land P$) are in the assumption base. |

| left-iff (iff $P \land Q$) |
| adds the proposition (if $P \land Q$) to the assumption base when (iff $P \land Q$) is in the assumption base. |

| right-iff (iff $P \land Q$) |
| adds the proposition (if $Q \land P$) to the assumption base when (iff $P \land Q$) is in the assumption base. |

Just as with left-and, right-and, and both, one can use a combination of left-iff, right-iff, and equiv to reverse the order the arguments of an iff expression.

| either $P \land Q$ |
| adds the proposition (or $P \lor Q$) to the assumption base when either $P$ or $Q$ is in the assumption base. |

| cd (or $P \land Q$) (if $P \land R$) (if $Q \land R$) |
| This method, called “constructive dilemma,” |
adds the proposition \( R \) to the assumption base when all three of the argument propositions are in the assumption base:

\[
\text{cases (if } P R \text{) (if (not } P \text{) } R)\]

adds the proposition \( R \) to the assumption base when both of the argument propositions are in the assumption base.

\[
\text{absurd } P \text{ (not } P)\]

adds the proposition \textit{false} to the assumption base when both \( P \) and \( \text{not } P \) are in the assumption base. It is meant to be used within a by-contradiction deduction, as the last step of a \textit{proof by contradiction}.

\[
\text{dn (not (not } P)\]

This method, called “double negation,” adds the proposition \( P \) to the assumption base when \( \text{not (not } P) \) is in the assumption base. This can be useful, for example, when a proof by contradiction is done using by-contradiction with a proposition of form \( \text{not } P \), producing, if it succeeds, one of the form \( \text{not (not } P) \).

The following methods, unlike all the previously described primitive methods, perform deductions that depend on their argument proposition having a (top-level) \textit{forall} or \textit{exists} quantifier. For a proposition \( P \), variable \(?x\), and term \( t \), let \( P\{t/?x}\) denote the proposition that results from replacing all free occurrences of \(?x\) in \( P \) with \( t \). The type of \( t \) must be compatible with the type of \(?x\).

\[
\text{uspec (forall } ?x \text{ } P \text{) } t\]

universal specialization adds the proposition \( Pt/?x \) to the assumption base when (forall \(?x \ text{ } P\) is in the assumption base and \( t \) is a term whose type is compatible with the type of \(?x\).
existential generalization adds the proposition (exists \( ?x \ P \)) to the assumption base when the proposition \( Pt/?x \) is in the assumption base.

There is also a useful generalization of the \texttt{uspec} method:

\texttt{uspec* \( P [t_1 \ldots t_n] \)}

replaces the first \( n \) quantified variables in \( P \) with the terms \( t_1, \ldots, t_n \) and adds the resulting proposition to the assumption base, provided \( P \) is in the assumption base.

\section*{A.2 Some example proofs}

In this section we illustrate how to write simple proofs in \textsc{Athena} with formalization of some properties of an order relation.

We first declare the arity of the \( < \) operator in terms of an arbitrary domain we call \( D \):

\begin{verbatim}
(domain D)

(declare < (-> (D D) Boolean))
\end{verbatim}

which means that \( < \) is a function that takes two arguments from the domain \( D \) and produces an element of the domain \texttt{Boolean}, which is a predefined domain in \textsc{Athena}.

We now assert as axioms the irreflexive and transitive laws for \( < \):

\begin{verbatim}
egen (exists ?x P) t
\end{verbatim}
(define Order-Irreflexive
   (forall ?x
      (not (< ?x ?x))))

(define Order-Transitive
   (forall ?x ?y ?z
      (if (and (< ?x ?y) (< ?y ?z))
         (< ?x ?z))))

(assert Order-Irreflexive Order-Transitive)

The **assert** command is used for inserting propositions into the ATHENA’s assumption base (without proof; it is up to the user to avoid introducing contradictory assertions).

These axioms characterize < as a **strict partial order** relation. From these axioms we can prove another law, asymmetry:

(define Order-Asymmetry
   (forall ?x ?y
      (forall ?x ?y
         (if (< ?x ?y)
             (not (< ?y ?x))))))

Before getting into the details, we first describe some of the ATHENA special deductive forms to be used in the proof.

**assume** P D

first adds the proposition P to the assumption base temporarily. Then deduction D is evaluated. If this evaluation fails ATHENA the whole deduction fails. Otherwise the proposition (if P R) is added to the assumption base, where R is the proposition proved by D. In either case, the proposition P is removed from the assumption base after this evaluation.

**P BY** D
adds the proposition $P$ to the assumption base if it can be deduced from the steps given in $D$. It is an error if deduction $D$ fails or proves a proposition different from $P$. (Deductions of this form could always be shortened just to $D$, but including “$P$ BY” serves to clarify for the human reader what is the goal of the deduction.)

We have implemented a function called `conclude` that is a different form of `BY`.

```
(!(conclude P) D)
```

adds the proposition $P$ to the assumption base if it can be deduced from the steps given in $D$. It is an error if deduction $D$ fails or proves a proposition different from $P$.

Sometimes debugging of proofs is necessary. We have defined a variable called `tracing` whose default value is `false`. It is mainly used as a flag for tracing proof steps. If one wants to see what is produced after each deduction step, tracing should be turned on with the following command:

```
(set! tracing true)
```

If the `conclude` function is used after the `tracing` is set to `true`, the results of proof steps or an error message will be displayed with proper indentations and numbers showing each level.

```
pick-any I D
```

provides variables via the identifier $I$. $D$ is then evaluated by using these variables.

In addition to these special deductive forms we define the following method:

```
by-contradiction (assume P D)
```
temporarily adds to the assumption base proposition $P$ by using an `assume` and performs deduction $D$ to show that `false` follows from $P$. If this evaluation fails `ATHENA` the whole deduction fails. Otherwise the proposition (not $P$) is added to the assumption base. In either case, the proposition $P$ is removed from the assumption base after this evaluation.

Now let’s return to the proof of Order-Asymmetry. The proof is by contradiction. We use the `assume` special deductive form to put ($< x y$) is in the assumption base temporarily, because the goal we are trying to prove is a conditional proposition. To be able to prove (not ($< y x$)) we use `by-contradiction` and add ($< y x$) to the assumption base temporarily with an `assume`. Then we proceed to derive a contradiction.

(Order-Asymmetry

BY

(pick-any $x y$

(assume ($< x y$)

((not ($< y x$))

BY (!by-contradiction

(assume ($< y x$)

(!absurd

($< x x$

BY (!mp (!uspec* Order-Transitive [x y x])

(!both ($< x y$

($< y x$)))

((not ($< x x$))

BY (!uspec Order-Irreflexive x))))))))

The contradiction we get, expressed in the arguments to `absurd`, is ($< x x$) and (not ($< x x$)).

Thus, we’ve shown that the asymmetry law is a theorem when one has a partial order relation; *one does not have to assert it as a separate axiom*.

Now suppose we define a binary relation $E$ as follows:
\begin{quote}
(declare E (-> (D D) Boolean))

(define E-Definition
  (forall ?x ?y
    (iff (E ?x ?y)
      (and (not (< ?x ?y))
       (not (< ?y ?x)))))

The name E for this relation is motivated by fact that we can show that if E is
assumed to be transitive then in combination with the partial order axioms for <
we can prove that E is in fact an equivalence relation; i.e., that it also obeys the
other two axioms of an equivalence relation besides the transitive law, namely the
reflexive and symmetric laws.

(define E-Transitive
  (forall ?x ?y ?z
    (if (and (E ?x ?y) (E ?y ?z))
      (E ?x ?z))))

(assert E-Definition E-Transitive)

(define E-Reflexive
  (forall ?x
    (E ?x ?x)))

(E-Reflexive
  BY (pick-any x
    (dseq
      ((not (< x x))
        BY (!uspec Order-Irreflexive x))
      ((E x x)
        BY (!mp (!right-iff (!uspec* E-Definition [x x]))
          (!both (not (< x x))
            (not (< x x)))))})

\end{quote}
(define E-Symmetric
 (forall ?x ?y
     (if (E ?x ?y)
         (E ?y ?x))))

(E-Symmetric
 BY (pick-any x y
     (assume (E x y)
         (dlet ((both-not
             ((and (not (< x y))
                (not (< y x))))
             BY (!mp (!left-iff (!uspec* E-Definition [x y])
                 (E x y)))))))

       ((E y x)
        BY (!mp (!right-iff (!uspec* E-Definition [y x])
              (!both (!right-and both-not)
              (!left-and both-not)))))))))

(define E-Equivalent
 (and E-Reflexive
     (and E-Symmetric
         E-Transitive)))

(E-Equivalent
 BY (!both E-Reflexive
     (!!both E-Symmetric
     E-Transitive)))

A.3 Inductive definitions of new types

In ATHENA one way of introducing a new type is to use a datatype declaration.

(datatype Nat zero (succ Nat))
This declaration introduces two constructors that produce values of type \texttt{Nat}. The first constructor, \texttt{zero}, is a nullary, or constant, constructor. The second constructor, \texttt{succ}, is a unary constructor which takes a natural number as an input and returns another natural number. In addition to saying that these constructors produce values of type \texttt{Nat}, it also implicitly states that the only values of type \texttt{Nat} are those that can be constructed by some finite number of applications of these constructors; there are no other \texttt{Nat} values. That is, the \texttt{Nat} type is exactly the (infinite) set
\[
\{ \texttt{zero}, \texttt{succ(}\texttt{zero})\}, \texttt{succ(succ(}\texttt{zero})\}, \ldots \}
\]

This inductive definition principle is the basis of the mathematical induction proof method, implemented for each \texttt{datatype}-defined type in \texttt{ATHENA} with its \texttt{by-induction} construct, as discussed below. First, however, we discuss another important kind of proof method, term rewriting methods for proving equations. The kind of equations we will be most concerned with are ones defining a new function symbol on a type such as \texttt{Nat}.

Suppose, for example, we define a \texttt{Plus} function. We do so by specifying its behavior axiomatically in terms of equations. This function takes two \texttt{Nat} values as parameters and return their sum as a \texttt{Nat} value.

\[
(\text{declare Plus} \hspace{1em} (\texttt{->} \hspace{1em} (\texttt{Nat} \hspace{1em} \texttt{Nat}) \hspace{1em} \texttt{Nat}))
\]

Here are the propositions we intend to use to define the meaning of \texttt{Plus} axiomatically.

\[
(\text{define Plus-zero-axiom} \\
(\text{forall} \ ?n \ (\texttt{=} \ (\text{Plus} \ ?n \ \texttt{zero}) \ ?n)))
\]

\[
(\text{define Plus-succ-axiom} \\
(\text{forall} \ ?n \ ?m \\
\hspace{1em} (\texttt{=} \ (\text{Plus} \ ?n \ (\text{succ} \ ?m)) \\
\hspace{2em} (\text{succ} \ (\text{Plus} \ ?n \ ?m)))))
\]

After defining these propositions, we add them as axioms to \texttt{ATHENA}’s assumption base.

\[
(\text{assert Plus-zero-axiom} \hspace{1em} \text{Plus-succ-axiom})
\]
In the next two sections we will use these axioms first for a purely equational proof and then in proofs involving both the equations and the principle of mathematical induction.

A.4 Term rewriting methods

In many proofs, equations such as those defining $\text{Plus}$ play an important role. When working with equations, one often structures a proof as sequence of rewriting steps, in which a term is shown to be equal to another term by means of a substitution that is justified by an equational axiom (or other equational proposition already proved as a theorem). ATHENA does not currently have such rewriting methods built in, so we have created the following rewriting methods:

\begin{itemize}
\item \texttt{setup c t}\newline
    initializes cell c to hold term t. Actually it internally holds the equation $(= t t)$, and the reduce and expand methods transform the right hand side of this equation. There are two predefined cells named left and right.
\item \texttt{reduce c u E}\newline
    attempts to transform the term t in cell c to be identical with the given term u by using theorem E (or an appropriate specialization of it, if it contains quantified variables) as a left-to-right rewriting rule. E can be either an unconditional or a conditional equation, in which case, the condition of E has to be in the assumption base.
\item \texttt{expand c u E}\newline
    attempts to transform the term u to the term t in cell c by using theorem E (or an appropriate specialization of it, if it contains quantified variables) as a right-to-left rewriting rule. E can be either an unconditional
\end{itemize}
or a conditional equation, in which case, the condition of $E$ has to be in the assumption base.

**combine left right**

attempts to combine the equation internally stored in cell left, say ($= t t'$), with the equation internally stored in cell right, say ($= u u'$), to deduce ($= t u$), succeeding if and only if $t'$ and $u'$ are identical terms).

All these rewriting methods are provided in a file called `utilities.ath`.

As a simple example consider the following deduction in which we prove an equality by reducing both its left and its right hand side term to the same term, $(\text{succ (succ (succ zero))})$.

```
((= (Plus zero (succ (succ (succ zero))))
 (succ (succ (succ (Plus zero zero)))))
 BY (dseq
  (!setup left (Plus zero (succ (succ (succ zero))))))
  (!setup right (succ (succ (succ (Plus zero zero))))))
  (!reduce left (succ (Plus zero (succ (succ zero))))
    Plus-succ-axiom)
  (!reduce left (succ (succ (Plus zero (succ zero))))
    Plus-succ-axiom)
  (!reduce left (succ (succ (succ (Plus zero zero))))
    Plus-succ-axiom)
  (!reduce left (succ (succ (succ zero )))
    Plus-zero-axiom)
  (!reduce right (succ (succ (succ zero)))
    Plus-zero-axiom)
  (!combine left right)))
```

It should be noted that these rewriting methods, although user-defined, are guaranteed to be logically sound because they are programmed in terms of ATHENA’s primitive methods (assuming ATHENA’s methods are sound).

### A.5 Proof by induction

Consider the following proposition about the `Plus` function:
(define Plus-zero-property
 (forall ?n (= (Plus zero ?n) ?n)))

Whereas the Plus-zero-axiom states that zero serves as a right-identity element for the Plus operator, this proposition states that it is also a left-identity element. The validity of this property is, however, not a consequence simply of the equational axioms we’ve stated about Plus and zero; it depends on the fact that Nat is inductively defined by the structure declaration (datatype Nat zero (succ Nat)).

There are two main steps in a proof by induction of a property of natural numbers, as represented by type Nat:

- The basis case. This is a special case in which the proposition to be proved is instantiated with zero.

- The induction step. We instantiate the proposition with (succ n) for a Nat type. After the instantiation we attempt to prove it, assuming the proposition is true for n.

If these steps are successful, we can combine their results using ATHENA’s by-induction construct to conclude the proposition is true for all n.

An overview of the structure of the proof is as follows:

(Plus-property
 BY (by-induction
   ?n
   (= (Plus zero ?n) ?n)
   (zero
     ( Basis-case for Plus-zero property
     BY
     ( Proof of basis-case for Plus-zero property ))
   ((succ n)
     (( Induction-step for Plus-zero property
     BY
     ( Proof of induction-step for Plus-zero property ))))

In general,
by-induction \( V \ P \ (V_1 \ P_1) \ldots (V_n \ P_n) \)

adds the proposition \( P \) to the assumption base if for the induction variable \( V \) whose sort is some structure \( S \) and for each individual pattern \( V_i, P_i \) is the proof of \( P(V_i) \) for all possible cases that could be defined with the constructors of \( S \). Each \( V_i \) is built from one of the constructors of the structure \( S \). It is an error if any of the cases does not have a successful proof or not all of the possible cases are considered.

A.5.1 Basis case proof

We first need to prove the basis case, which is:

<table>
<thead>
<tr>
<th>Basis-case for Plus-zero property</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (= \ (\text{Plus zero zero}) \text{ zero}) )</td>
</tr>
</tbody>
</table>

The proof for this case is simply

<table>
<thead>
<tr>
<th>Proof of basis-case for Plus-zero property</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{dseq} )</td>
</tr>
<tr>
<td>( (!\text{setup left (Plus zero zero)}) )</td>
</tr>
<tr>
<td>( (!\text{reduce left zero Plus-zero-axiom}) )</td>
</tr>
</tbody>
</table>

In the proof itself, the with applications set up left and right to hold the left and right hand sides of the equation to be proved. In this simple case only one application of reduce, applied to left, is sufficient to produce the same term, zero, as we have placed in right. Note that reduce has to specialize the quantified variable \(?n\) in Plus-zero-axiom to zero in order to use the result \( (= \ (\text{Plus zero zero}) \text{ zero}) \) to substitute zero for \( (\text{Plus zero zero}) \).

A.5.2 Induction step proof

The induction step proof requires a bit more machinery to set up, but again the heart of the proof is reduction using equations. This time, however, the equations we have available to use include not only the axioms but also the induction hypothesis.
We have:

**Induction-step for Plus-zero property**

\[ (= \text{(Plus zero (succ n))} \text{(succ n)}) \]

**Proof of induction-step for Plus-zero property**

\[
\text{(dlet} ((\text{induction-hypothesis})
\quad (= \text{(Plus zero n)} \text{n}))
\quad \text{(dseq)}
\quad (!\text{setup} \text{left} \text{(Plus zero (succ n))})
\quad (!\text{setup} \text{right} \text{(succ n)})
\quad (!\text{reduce} \text{left} \text{(succ (Plus zero n)) Plus-succ-axiom})
\quad (!\text{reduce} \text{left} \text{(succ n) induction-hypothesis})
\quad (!\text{combine} \text{left right}))
\]

Note that **by-induction** automatically adds the induction hypothesis to the assumption base according to the inductive structure of the type.
APPENDIX B
Natural Numbers and Lists in Athena

According to the built-in semantics of `datatype`, the only values of type `Nat` are those that can be expressed with syntactically-correct and type-correct combinations of `zero` and `succ` terms:

zero
(succ zero)
(succ (succ zero))
(succ (succ (succ zero)))
...

For such inductively-defined types, we can prove propositions using the method of proof by induction.

The next type we frequently use is `List-Of` which is a built-in structure in Athena. The following usage of `datatype` directive introduces an additional property. Athena’s type system allows polymorphism: we can declare polymorphic lists as a structure:

```
(datatype (List-Of T) Nil (Cons T (List-Of T)))
```

Here `T` is a type variable that can be instantiated with any type, so that, for example,

- `(Cons zero (Cons (succ zero) Nil))` is a term of type `(List-Of Nat)`.
- `(Cons true Nil)` is a term of type `(List-Of Boolean)`;
- however, `(Cons zero (Cons true Nil))` is an error (“Ill-sorted term”) since all elements of a `List-Of` must have the same type.

We introduce another unary constructor here. `Cons` is used for inserting a new element in front of a given list. The `List-Of` type has an inductive structure which will be used for doing proofs by induction on lists.
After defining the structure Nat and List-Of they are ready to be used in function declarations. For example, the following declaration of insert function tells us that it takes a natural number and a list of natural numbers as an input and returns another list of natural numbers as an output.

(declare insert (-> (Nat (List-Of Nat)) (List-Of Nat)))

Since we frequently use the natural numbers and lists in our implementations and proofs we have already defined them. The file naturals.ath contains the declarations of Nat and functions using natural numbers as well as some related ATHENA theorems and their proofs. We will refer to this file as “natural-numbers-library.” Similarly the file lists.ath provides functions using lists with related theorems and their proofs. This file is called “lists-library” in this document. ATHENA’s load directive can be used to load both of these files as shown below:

(load-file "lists.ath")
(load-file "naturals.ath")

Here is a list of theorems about natural numbers:

(define succ-not-zero-axiom
  (forall ?n
    (not (= (succ ?n) zero))))

(define Plus-zero-axiom
  (forall ?n
    (= (Plus ?n zero) ?n)))

(define Plus-succ-axiom
  (forall ?n ?m
    (= (Plus ?n (succ ?m))
      (succ (Plus ?n ?m)))))

(define Plus-zero-property
  (forall ?n

(define Plus-succ-property
  (forall ?n ?m
    (= (Plus (succ ?m) ?n)
      (succ (Plus ?m ?n)))))

(define Plus-Associativity
  (forall ?n ?m ?p
    (= (Plus (Plus ?m ?p) ?n)
      (Plus ?m (Plus ?p ?n))))
)

For lists, we define nongeneric append and reverse functions axiomatically as follows:

(declare append
  ((T) ->
    ((List-Of T) (List-Of T))
    (List-Of T)))

(define append-Nil-axiom
  (forall ?q (= (append Nil ?q) ?q)))

(define append-Cons-axiom
  (forall ?x ?r ?q
    (= (append (Cons ?x ?r) ?q)
      (Cons ?x (append ?r ?q)))))

(declare reverse ((T) -> ((List-Of T)) (List-Of T)))

(define reverse-Nil-axiom
  (= (reverse Nil) Nil))

(define reverse-Cons-axiom
  (forall ?x ?r
    (= (reverse (Cons ?x ?r))
      (Cons ?x (append ?r ?q)))))
We can prove the following property of reverse-append combinations by using the axioms given above.

\[
\text{(define reverse-append-property}
  \quad (\text{forall } ?p \ ?q
  \quad \quad (= (\text{reverse} \ (\text{append} \ ?p \ ?q))
  \quad \quad \quad \quad (\text{append} \ (\text{reverse} \ ?q) \ (\text{reverse} \ ?p))))
\]

We will now show an example proof about lists, where \text{reverse} reverses itself. This proof is nongeneric and is done by using induction on lists.

\[
\text{(define reverse-reverse-property}
  \quad (\text{forall } ?p
  \quad \quad (= (\text{reverse} \ (\text{reverse} \ ?p))
  \quad \quad \quad \quad ?p)))
\]

\[
\text{(by-induction}
  \quad \text{reverse-reverse-property}
  \quad \text{(Nil}
  \quad \quad (!\text{(conclude} (= (\text{reverse} \ (\text{reverse} \ \text{Nil})))
  \quad \quad \quad \quad \text{Nil}))
  \quad \quad \text{(dseq}
  \quad \quad \quad \quad !\text{(setup left} \ (\text{reverse} \ (\text{reverse} \ \text{Nil}))))
  \quad \quad \quad \quad \text{(dseq}
  \quad \quad \quad \quad \quad \quad \text{(dlet} (\text{***induction-hypothesis***}
  \quad \quad \quad \quad \quad \quad \quad (\quad = \ (\text{reverse} \ (\text{reverse} \ ?p)))
  \quad \quad \quad \quad \quad \quad \quad (\text{reverse-Nil-axiom})
  \quad \quad \quad \quad \quad \quad \quad \text{reverse-Nil-axiom))))
  \quad \quad \quad \quad \text{(Cons x p)}
  \quad \quad \quad \quad \quad \quad \text{(dseq}
  \quad \quad \quad \quad \quad \quad \quad (\text{dlet} \ (\text{***induction-hypothesis***}
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad = \ (\text{reverse} \ (\text{reverse} \ ?p)))
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{reverse-Nil-axiom}))
  \quad \quad \quad \quad \text{(dseq}
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad !\text{(setup left} \ (\text{reverse} \ (\text{reverse} \ \text{Cons x p}))))
  \quad \quad \quad \quad)\text{))}}
\]
(!reduce left (reverse (append (reverse p)
           (Cons x Nil))))
reverse-Cons-axiom)
(!reduce left (append (reverse (Cons x Nil))
           (reverse (reverse p)))
reverse-append-property)
(!reduce left (append (reverse (Cons x Nil))
           p)
***induction-hypothesis***)
(!reduce left (append (append (reverse Nil)
           (Cons x Nil))
           p)
reverse-Cons-axiom)
(!reduce left (append (append Nil
           (Cons x Nil))
           p)
reverse-Nil-axiom)
(!reduce left (append (Cons x Nil) p)
append-Nil-axiom)
(!reduce left (Cons x (append Nil p))
append-Cons-axiom)
(!reduce left (Cons x p)
append-Nil-axiom)))))))
APPENDIX C
Range Induction

The range-induction method, which transforms induction over iterator ranges into induction over the natural numbers (and back-translates the resulting natural number theorem into one directly about iterator ranges) is presented here.

Defining the cases and setting up the range induction:

(load-file "range.ath")

(define (range-basis-case Q ops)
  (forall ?i
    (Q ?i ?i)))

(define (range-induction-step Q ops)
  (let ((++ (ops '++))
         (valid (ops 'valid)))
    (forall ?i ?j
      (if (and (not (= ?i ?j))
          (and (valid (range ?i ?j))
               (Q (++ ?i) ?j)))
       (Q ?i ?j))))

## Following is for internal use only

(define (setup-range-induction Q ops)
  (dlet ((I- (ops 'I-)) (++ (ops '++)) (valid (ops 'valid))
         (prop (method (name) (!property name ops Range-theory)))
         (left (cell true)) (right (cell true))
         (P (function (n) (forall ?j (Q (I- ?j n) ?j))))
    (dseq
      (by-induction
        (forall ?n (P ?n))))
(zero
(!conclude (P zero))
(pick-any j
dseq
(!conclude (Q j j))
(!uspec (range-basis-case Q ops) j))
(!sym (!conclude (= (I- j zero) j))
dseq
(!setup left (I- j zero))
(!reduce left j
 (!prop 'iminus-zero-axiom))))
(!conclude (Q (I- j zero) j))
(!pos-substitute-equals j (Q j j) [1] (I- j zero))))))
((succ n)
(!conclude (P (succ n)))
(pick-any j
(!conclude (Q (I- j (succ n)) j))
dseq
(!equality (I- j (succ n)) (I- j (succ n)))
(!egen (exists ?k
 (= (I- j (succ n))
 (I- j ?k)))
(succ n))
(!right-instance (!prop 'Valid-Range-definition)
 [(I- j (succ n)) j])
(!both (!conclude (not (= (I- j (succ n)) j))))
(!by-contradiction
 (assume (= (I- j (succ n)) j))
dseq
(!conclude (= j (I- j zero)))
(!sym
 (!uspec (!prop 'iminus-zero-axiom) j))
(!conclude (= (I- j (succ n)) (I- j zero)))
The following is the user-level method for performing range induction for a specific predicate $R$. $Q$-definition is a proposition in the assumption base defining some predicate symbol $Q$: it is is an equivalence $(\forall i \forall j (iff (Q i j) (R i j)))$ for some predicate $R$. induction-cases is the conjunction of $(\text{range-basis-case } R \text{ ops})$ and $(\text{range-induction-step } R \text{ ops})$; it must also be in the assumption base. The result of the proof carried out by the method is the theorem $(\forall i \forall j (if (valid (range ?i ?j)) (R ?i ?j)))$.

```
(define (range-induction induction-cases Q-definition ops)
  (dlet ((valid (ops 'valid)) (++ (ops '++))
      (Q (match (urep Q-definition [?i ?j])
        ((iff (Q0 a b) R) Q0)))
      (R (function (i j)
        (match (urep Q-definition [i j])
          ((iff Q R1) R1)))))
  (dseq
    (! (conclude (forall ?i ?j (if (valid (range ?i ?j)) (R ?i ?j))))
    (Q ?i ?j))))
```
(dseq
  (!conclude (forall ?i (Q ?i ?i)))
  (pick-any i
    (!mp (!right-iff (!uspec Q-definition [i i]))
      (!uspec (!left-and induction-cases) i)))
  (!conclude (forall ?i ?j
    (if (and (not (= ?i ?j))
      (and (valid (range ?i ?j))
        (Q (++ ?i) ?j)))
      (Q ?i ?j)))
  (pick-any
    i j
    (assume-let (assumptions
      (and (not (= i j))
        (and (valid (range i j))
          (Q (++ i) j)))))
    (dseq
      (!left-and (!right-and assumptions))
      (!left-and assumptions)
      (!right-and (!right-and assumptions))
      (!conclude (R (++ i) j))
      (!instance Q-definition [(++ i) j])
      (!both (not (= i j))
        (!both (valid (range i j))
          (R (++ i) j)))
      (!conclude (R i j))
      (!instance (!right-and induction-cases) [i j])
      (!right-instance Q-definition [i j])))
    (!setup-range-induction Q ops)))
  (!conclude (forall ?i ?j (if (valid (range ?i ?j))
    (R ?i ?j)))
  (pick-any
    i j)
C.1 A proof by range induction

(define (I-I-++-property name ops)
  (let ((valid (ops 'valid))
         (++ (ops '++))
         (I-I (ops 'I-I)))
    (match name
      ('I-I-++-property
       (forall ?i ?j
        (if (valid (range ?i ?j))
            (= (I-I (++ ?j) ?i)
                (succ (I-I ?j ?i))))))))

# Proof by range-induction:

(define (I-I-++-property-induction-cases ops)
  (dlet ((++ (ops '++))
          (-- (ops '--))
          (I- (ops 'I-))
          (I-I (ops 'I-I))
          (valid (ops 'valid))
          (prop (method (name) (!property name ops Range-theory)))
          (left (cell true))
          (right (cell true))
          (left1 (cell true))
          (right1 (cell true))
          (P (function (i j)
               (= (I-I (++ j) i)))))
(succ (I-I j i)))

(!both

(!((conclude (range-basis-case P ops))
  (pick-any i
    (dseq
      (!setup left (I-I (++ i) i))
      (!setup right (succ (I-I i i)))
      (!((conclude (= i (I- (++ i) (succ zero))))))
      (dseq
        (!setup left1 i)
        (!setup right1 (I- (++ i) (succ zero)))
        (!reduce right1 (-- (I- (++ i) zero))
          (!prop 'iminus-succ-axiom))
        (!reduce right1 (-- (++ i))
          (!prop 'iminus-zero-axiom))
        (!reduce right1 i (!prop 'dec-inc-axiom))
        (!combine left1 right1)))
      (!((conclude (valid (range i (++ i))))))
      (dseq
        (!egen (exists ?n (= i (I- (++ i) ?n)))
          (succ zero))
        (!spec-right
          (!prop 'Valid-Range-definition) [i (++ i)])))))
    (!uspec (!prop 'iplus-not-equal-property) i)
    (!both (valid (range i (++ i)))
      (not (= i (++ i)))))
    (!reduce left (succ (I-I (++ i) (++ i)))
      (!spec (!prop 'Valid-Range-I-I-axiom) [i (++ i)]))
    (!reduce left (succ zero) (!prop 'I-I-zero-property))
    (!reduce right (succ zero) (!prop 'I-I-zero-property))
    (!combine left right))))
  (!((conclude (range-induction-step P ops))
    (pick-any
i j
(assume-let
  (assumptions
   (and (not (= i j))
    (and (valid (range i j))
     (= (I- (I- j) (I- i))
        (succ (I- (I- j) (I- i)))))))
(dseq
  (!left-and (!right-and assumptions))
  (!left-and assumptions)
  (!right-and (!right-and assumptions))
  (!setup left (I- (I- j) i))
  (!setup right (succ (I- (I- j) i)))
  (!(!conclude (exists ?n (= i (I- j (succ ?n))))))
  (dseq
   (!both (valid (range i j))
    (not (= i j)))
   (!spec (!prop 'Valid-Range-length-atleast-one-property) [i j])))
(pick-witness k
  (exists ?n (= i (I- j (succ ?n))))
  (dseq
   (!(!conclude (= i (I- (I- j) (succ (succ k))))))
   (dseq
    (!setup left1 i)
    (!setup right1 (I- (I- j) (succ (succ k))))
    (!reduce left1 (I- j (succ k))
     (= i (I- j (succ k))))
    (!reduce right1 (-- (I- (I- j) (succ k)))
     (!prop 'iminus-succ-axiom))
    (!reduce right1 (-- (I- j (succ k)))
     (!prop 'iminus-++-property))
    (!reduce right1 (I- j (succ k))
     (!prop 'dec-inc-axiom))}
(!combine left1 right1))
(!conclude (not (= i (++ j))))
(!by-contradiction
(assume (= i (++ j)))
(!absurd
(dseq
(!conclude (= (I- (++ j) (succ (succ k)))
(I- (++ j) zero)))
(dseq
(!setup left1 (I- (++ j)
(succ (succ k))))
(!setup right1 (I- (++ j) zero))
(!expand left1 i
(= i (I- (++ j)
(succ (succ k))))))
(!reduce right1 (++ j)
(!prop 'iminus-zero-axiom))
(!expand right1 i (= i (++ j)))
(!combine left1 right1)))
(!spec (!prop 'iminus-equal-axiom)
[([+ j) (succ (succ k)) zero]])
(!conclude (not (= (succ (succ k)) zero)))
(!uspec succ-not-zero (succ k)))))))
(!egen (exists ?m (= i (I- (++ j) ?m)))
(succ (succ k)))
(!conclude (valid (range i (++ j))))
(!spec-right (!prop 'Valid-Range-definition)
[i (++ j)])
(!both (valid (range i (++ j))
(not (= i (++ j))))
(!reduce left (succ (I-I (++ j) (++ i)))
(!spec (!prop 'Valid-Range-I-I-axiom)
[i (++ j)]))
(!both (valid (range i j))
  (not (= i j)))
(!reduce right (succ (succ (I-I j (++ i))))
  (!spec (!prop 'Valid-Range-I-I-axiom)
    [i j]))
(!expand right (succ (I-I (++ j) (++ i)))
  (= (I-I (++ j) (++ i))
      (succ (I-I j (++ i)))))
(!combine left right))))))))

# A predicate symbol definition will be needed for the final
# range-induction application. It will be entered on the following
# list, with the name of the proposition as the key for lookup.

(define Range-symdefs (cell []))

(define (I-I-++-property-proof name ops)
  (!range-induction (!I-I-++-property-induction-cases ops)
    (symbol-definition
      (add name (map ops ['valid '++ 'I-I])
        Range-symdefs)
      ops))

(evolve
  Range-theory
  [I-I-++-property
    I-I-++-property-proof])
APPENDIX D
Formalization of Memory

D.1 Memory theory
D.1.1 Definition of Memory Concept in CCT

(load-file "rewriting.ath")
(load-file "property-management.ath")
(domain Memory)
(domain (Loc T))

# Memory axioms

(define (Memory name ops)
  (let ((access (ops 'access))
         (assign (ops 'assign))
         (swap (ops 'swap)))
    (match name
    ('assign-axiom1
     (forall ?M ?a ?x
      (= (access (assign ?M ?a ?x) ?a)
          ?x)))
    ('assign-axiom2
     (forall ?M ?a ?b ?x
      (if (not (= ?a ?b))
          (= (access (assign ?M ?a ?x) ?b)
              (access ?M ?b)))))
    ('swap-axiom1
     (forall ?M ?a ?b

183
(\(= (access (swap ?M ?a ?b) ?a) \\
(\(access ?M ?b))))

(\textit{\textquoteleft swap-axiom2} \\text\(\text{forall} \ ?M \ ?a \ ?b \\
(\(= (access (swap ?M ?a ?b) ?b) \\
(\(access ?M ?a))))

(\textit{\textquoteleft swap-axiom3} \\text\(\text{forall} \ ?M \ ?a \ ?b \ ?c \\
(if (and (\(\text{not} \ (\(= \ ?c \ ?a))) \\
(\(\text{not} \ (\(= \ ?c \ ?b))) \\
(\(= (access (swap ?M ?a ?b) ?c) \\
(\(access ?M ?c)))))))

(define Memory-theory
  (theory \[Memory Axiom\]))

D.1.2 CCT theorems about memory and some selected proofs

(define (Double-assign name ops)
  (let ((access (ops \'access))
    (assign (ops \'assign)))
    (match name
      (\textit{\textquoteleft Double-assign} \\
      (\text\(forall \ ?b \ ?M \ ?a \ ?x \ ?y \\
      (\(= (access (assign (assign ?M ?a ?x) ?a ?y) ?b) \\
      (\(access (assign ?M ?a ?y) ?b))))))))

Another axiom:

(define (Memory-equality name ops)
  (let ((access (ops \'access)))
    (match name
('Memory-equality
(forall ?M1 ?M2
  (iff (= ?M1 ?M2)
    (forall ?a
      (= (access ?M1 ?a)
        (access ?M2 ?a)))))))

(define (Direct-double-assign name ops)
  (let ((access (ops 'access))
        (assign (ops 'assign)))
    (match name
      ('Direct-double-assign
        (forall ?M ?a ?x ?y
            (assign ?M ?a ?y)))))))

(define (Self-assign name ops)
  (let ((access (ops 'access))
        (assign (ops 'assign)))
    (match name
      ('Self-assign
        (forall ?M ?a ?b
          (= (access (assign ?M ?a (access ?M ?a)) ?b)
            (access ?M ?b)))))))

(define (Self-assign-proof name ops)
  (dlet ((theorem (Self-assign name ops)))
    (define (Direct-double-assign name ops)
      (let ((access (ops 'access))
            (assign (ops 'assign)))
        (match name
          ('Direct-double-assign
            (forall ?M ?a ?x ?y
                (assign ?M ?a ?y)))))))

(define (Self-assign name ops)
  (let ((access (ops 'access))
        (assign (ops 'assign)))
    (match name
      ('Self-assign
        (forall ?M ?a ?b
          (= (access (assign ?M ?a (access ?M ?a)) ?b)
            (access ?M ?b)))))))

(define (Self-assign-proof name ops)
  (dlet ((theorem (Self-assign name ops)))
    (define (Direct-double-assign name ops)
      (let ((access (ops 'access))
            (assign (ops 'assign)))
        (match name
          ('Direct-double-assign
            (forall ?M ?a ?x ?y
                (assign ?M ?a ?y)))))))
(assign (ops 'assign))
(access (ops 'access))
(prop (method (name) (!property name ops Memory-theory)))

(! (conclude theorem)
(pick-any
 M a b
 (dlet ((left (cell true))
            (right (cell true)))
 (dseq
  (! cases
   (assume (= a b)
    (dseq
     (! setup left (access (assign M a (access M a)) b))
     (! setup right (access M b))
     (! expand left (access (assign M a (access M a)) a)
      (= a b))
     (! reduce left (access M a)
      (! prop 'assign-axiom1))
     (! expand right (access M a)
      (= a b))
     (! combine left right)))
   (assume (not (= a b))
    (dseq
     (! setup left (access (assign M a (access M a)) b))
     (! setup right (access M b))
     (! reduce left (access M b)
      (! spec (! prop 'assign-axiom2)
       [M a b (access M a)])
      (! combine left right)))))))

(evolve
 Memory-theory
 [Self-assign
(define (Direct-self-assign name ops)
  (let ((access (ops 'access))
        (assign (ops 'assign)))
    (match name
      ('Direct-self-assign
       (forall ?M ?a
        (= (assign ?M ?a (access ?M ?a))
        ?M))))))

(define (Direct-self-assign-proof name ops)
  (dlet ((theorem (Direct-self-assign name ops))
        (assign (ops 'assign))
        (access (ops 'access))
        (prop (method (name) (!property name ops Memory-theory))))
    (!(!conclude theorem)
     (pick-any
      M i
      (dseq
       (!(!conclude (forall ?b
                        (= (access (assign M i (access M i)) ?b)
                        (access M ?b))))
        (pick-any b
          (!spec (!prop 'Self-assign) [M i b])))
        (!spec-right (!prop 'Memory-equality)
          (!!spec-right (!prop 'Memory-equality)
            [[(assign M i (access M i)) M]])))))
    (evolve
      Memory-theory
      [Direct-self-assign Direct-self-assign-proof]))

Self-assign-proof])

#################################################################
(define (Double-swap name ops)
  (let ((access (ops 'access))
        (swap (ops 'swap)))
    (match name
      ('Double-swap
       (forall ?c ?M ?a ?b
        (= (access (swap (swap ?M ?a ?b) ?b ?a) ?c)
           (access ?M ?c)))))))

(define (Double-swap-proof name ops)
  (dlet ((theorem (Double-swap name ops))
         (assign (ops 'assign))
         (access (ops 'access))
         (swap (ops 'swap))
         (prop (method (name) (!property name ops Memory-theory))))
    (!conclude theorem)
    (pick-any
     c M a b
     (dlet ((left (cell true))
            (right (cell true)))
      (dseq
       (!three-cases
        (assume (= c a)
         (dseq
          (!setup left (access (swap (swap M a b) b a) c))
          (!setup right (access M c))
          (!reduce left (access (swap (swap M a b) b a) a)
                       (= c a))
          (!reduce left (access (swap M a b) b)
                       (!prop 'swap-axiom2))
          (!reduce left (access M a)
                       (!prop 'swap-axiom2))
          (!reduce left (access (swap M a b) a)
                       (= c a))
          (!reduce left (access (swap M a b) c)
                       (!prop 'swap-axiom2))
          (!reduce right (access (swap M a b) a)
                       (= c a))
          (!reduce right (access (swap M a b) b)
                       (!prop 'swap-axiom2))
          (!reduce right (access M c)
                       (!prop 'swap-axiom2)))))
    (!conclude theorem))

    (dlet ((left (cell true))
            (right (cell true)))
      (dseq
       (!three-cases
        (assume (= c a)
         (dseq
          (!setup left (access (swap (swap M a b) b a) c))
          (!setup right (access M c))
          (!reduce left (access (swap (swap M a b) b a) a)
                       (= c a))
          (!reduce left (access (swap M a b) b)
                       (!prop 'swap-axiom2))
          (!reduce left (access M a)
                       (!prop 'swap-axiom2))
          (!reduce left (access (swap M a b) a)
                       (= c a))
          (!reduce left (access (swap M a b) c)
                       (!prop 'swap-axiom2))
          (!reduce right (access (swap M a b) a)
                       (= c a))
          (!reduce right (access (swap M a b) b)
                       (!prop 'swap-axiom2))
          (!reduce right (access M c)
                       (!prop 'swap-axiom2)))))
    (!conclude theorem)))
(!reduce right (access M a)
  (= c a))
(!combine left right)))
(assume (= c b)
(dseq
  (!setup left (access (swap (swap M a b) b a) c))
  (!setup right (access M c))
  (!reduce left (access (swap (swap M a b) b a) b)
    (= c b))
  (!reduce left (access (swap M a b) a)
    (!prop 'swap-axiom1))
  (!reduce left (access M b)
    (!prop 'swap-axiom1))
  (!reduce right (access M b)
    (= c b))
  (!combine left right)))
(assume (and (not (= c a))
  (not (= c b)))
(dseq
  (!setup left (access (swap (swap M a b) b a) c))
  (!setup right (access M c))
  (!reorder (and (not (= c a))
    (not (= c b))))
  (!reduce left (access (swap M a b) c)
    (!spec (!prop 'swap-axiom3)
      [(swap M a b) b a c])
  (!reduce left (access M c)
    (!prop 'swap-axiom3))
  (!prop 'swap-axiom3))
  (!combine left right)))))))))

(evolve
Memory-theory
[Double-swap Double-swap-proof])
(define (Direct-double-swap name ops)
  (let ((swap (ops 'swap)))
    (match name
      ('Direct-double-swap
       (forall ?M ?a ?b

(define (Direct-double-swap-proof name ops)
  (dlet ((theorem (Direct-double-swap name ops))
         (access (ops 'access))
         (swap (ops 'swap))
         (prop (method (name) (!property name ops Memory-theory))))
    (!conclude theorem)
    (pick-any
     M a b
     (dseq
      (!conclude (forall ?c
                    (= (access (swap (swap M a b) b a) ?c)
                    (access M ?c)))
      (pick-any c
        (!spec (!prop 'Double-swap)
          [c M a b]))
      (!spec-right (!prop 'Memory-equality)
        [[(swap (swap M a b) b a) M]])))))

(evolve Memory-theory
  [Direct-double-swap Direct-double-swap-proof])

******************************************************************************
D.2 Memory-range Theory

D.2.1 Definition of memory range concept in CCT

(load-file "range-induction.ath")

(load-file "lists.ath")

(load-file "memory.ath")

(define (Memory-range name ops)
  (let ((* (ops '*))
         (++ (ops '++))
         (access (ops 'access))
         (assign (ops 'assign))
         (access-range (ops 'access-range))
         (assign-range (ops 'assign-range)))
    (match name
      ("access-empty-range-axiom"
       (forall ?M ?i
           (= (access-range ?M (range ?i ?i))
              Nil)))
      ("access-nonempty-range-axiom"
       (forall ?M ?i ?j
           (if (not (= ?i ?j))
               (= (access-range ?M (range ?i ?j))
                  (Cons (access ?M (* ?i))
                         (access-range ?M (range (++ ?i) ?j))))))
      ("assign-empty-range-axiom"
       (forall ?M ?i
           (= (assign-range ?M ?i Nil)
              ?M)))
      ("assign-nonempty-range-axiom"
       (forall ?M ?i ?L ?x..."
(= (assign-range ?M ?i (Cons ?x ?L))

(define Memory-range-theory
  (refine
    [Memory-theory Range-theory]
    [Memory-range Axiom]))

########################################################################

D.2.2 CCT theorems about memory range and some selected proofs

(define (Self-assign-range-property name ops)
  (let ((valid (ops 'valid))
         (access-range (ops 'access-range))
         (assign-range (ops 'assign-range)))
    (match name
      ('Self-assign-range-property
       (forall ?i ?j
          (if (valid (range ?i ?j))
            (forall ?M
              (= (assign-range ?M ?i
                (access-range ?M (range ?i ?j)))
                ?M))))))))

# Proof by range-induction:

(define (Self-assign-range-property-induction-cases ops)
  (dlet (((++ (ops ‘++))
           (* (ops ‘*)))
         (valid (ops ‘valid))
         (access (ops ‘access))
         (assign (ops ‘assign))
         (access-range (ops ‘access-range))
         (assign-range (ops ‘assign-range))
         (assign-range (ops ‘assign-range))))
(prop (method (name)
  (!property name ops Memory-range-theory)))
(left (cell true))
(right (cell true))
(P (function (i j)
  (forall ?M
    (= (assign-range ?M i
        (access-range ?M (range i j)))
     ?M)))))

(!both
(!conclude (range-basis-case P ops))
(pick-any
  i M
  (dseq
    (!setup left (assign-range M i (access-range M (range i i))))
    (!setup right M)
    (!reduce left (assign-range M i Nil)
      (!prop 'access-empty-range-axiom))
    (!reduce left M
      (!prop 'assign-empty-range-axiom))
    (!combine left right))))

(!conclude (range-induction-step P ops))
(pick-any
  i j
  (assume-let
    (assumptions
      (and (not (= i j))
      (and (valid (range i j))
        (forall ?M
          (= (assign-range ?M (++ i)
              (access-range ?M (range (++ i) j)))
           ?M))))))

(pick-any M
(dseq
  (!left-and (!right-and assumptions))
  (!left-and assumptions)
  (!uspec (!right-and (!right-and assumptions)) M)
  (!setup left (assign-range M i (access-range M (range i j))))
  (!setup right M)
  (!reduce left (assign-range M i
    (Cons (access M (* i))
      (access-range M (range (++ i) j))))
    (!prop 'access-nonempty-range-axiom))
  (!reduce left (assign-range
      (assign M (* i) (access M (* i)))
      (++ i)
      (access-range M (range (++ i) j)))
    (!prop 'assign-nonempty-range-axiom))
  (!reduce left (assign-range M (++ i)
      (access-range M (range (++ i) j)))
    (!prop 'Direct-self-assign))
  (!reduce left M
    (= (assign-range M (++ i)
      (access-range M (range (++ i) j)))
      M))
    (!combine left right)))))))))

# A predicate symbol definition will be needed for the final
# range-induction application. It will be entered on the following
# list, with the name of the proposition as the key for lookup.

(define Memory-range-symdefs (cell []))

# The proof is completed by an application of range-induction:

(define (Self-assign-range-property-proof name ops)
(define (memory-range-post-cons-property name ops)
  (let ((valid (ops 'valid))
        (++ (ops '+))
        (-- (ops '-))
        (* (ops '*))
        (access (ops 'access))
        (access-range (ops 'access-range)))
    (match name
      ('memory-range-post-cons-property
       (forall ?a ?b
         (if (valid (range ?a ?b))
           (forall ?M
             (if (and (not (= ?a ?b))
                     (not (= (++ ?a) ?b)))
               (= (access-range ?M (range ?a ?b))
                 (append (access-range ?M (range ?a (-- ?b)))
                         (Cons (access ?M (* (-- ?b)) Nil)))))))))))))

#################################################################

(define (memory-range-property name ops)
  (evolve Memory-range-theory
    [Self-assign-range-property
     Self-assign-range-property-proof])
  #################################################################

#################################################################

(define (memory-range-post-cons-property name ops)
(let ((++ (ops '++))
     (-- (ops '--))
     (* (ops '*))
     (valid (ops 'valid))
     (access (ops 'access))
     (access-range (ops 'access-range)))
  (match name
       ('memory-range-property
        (forall
         ?a ?b
         (if (valid (range ?a ?b))
          (forall ?M
           (if (and (not (= ?a ?b))
              (not (= (++ ?a) ?b)))
            (= (access-range ?M (range ?a ?b))
              (append
               (Cons (access ?M (* ?a))
                   (access-range
                    ?M
                    (range (++ ?a) (-- ?b))))
               (Cons (access ?M (* (-- ?b))
                                Nil)))))))))

#################################################################

(define (memory-range-equal-property name ops)
  (let ((in (ops 'in))
     (* (ops '*))
     (access (ops 'access))
     (access-range (ops 'access-range)))
  (match name
       ('memory-range-equal-property
         (iff (= (access-range ?M1 (range ?a ?b))
              (= (access-range ?M2 (range ?a ?b))))))))
(access-range ?M2 (range ?a ?b)))
(forall ?i
  (if (in ?i (range ?a ?b))
    (= (access ?M1 (* ?i))
      (access ?M2 (* ?i)))))

(evolve
  Memory-range-theory
  [memory-range-equal-property Axiom])

(define (swap-outside-has-no-effect-in-range name ops)
  (let ((valid (ops 'valid))
        (swap (ops 'swap))
        (in (ops 'in))
        (* (ops '*))
        (++ (ops '++))
        (-- (ops '--))
        (access-range (ops 'access-range)))
    (match name
      ('swap-outside-has-no-effect-in-range
        (forall
          ?a ?b ?M
          (if (and (valid (range ?a ?b))
                  (and (not (= ?a ?b))
                      (not (= (++ ?a) ?b))))
            (= (access-range (swap ?M (* ?a) (* (-- ?b)))
                            (range (++ ?a) (-- ?b)))
                (access-range ?M (range (++ ?a) (-- ?b))))))))))

(define (assign-outside-has-no-effect-in-range name ops)
  (let ((valid (ops 'valid)))
      (forall
        ?a ?b ?M
        (if (and (valid (range ?a ?b))
                  (not (= ?a ?b))
                      (not (= (++ ?a) ?b))))
            (= (access-range (assign ?M (* ?a) (* (-- ?b)))
                            (range (++ ?a) (-- ?b)))
                (access-range ?M (range (++ ?a) (-- ?b)))))))

#################################################################
(assign (ops 'assign))
(in (ops 'in))
(* (ops '*))
(access-range (ops 'access-range)))

(match name
('assign-outside-has-no-effect-in-range
(forall
 (if (and (valid (range ?a ?b))
         (not (in ?r (range ?a ?b))))
         (= (access-range (assign ?M (* ?r) ?x)
                         (range ?a ?b))
            (access-range ?M (range ?a ?b)))))))))

#################################################################

(define (occurs-in-smaller-range-property name ops)
 (let ((++ (ops '++))
       (in (ops 'in))
       (access (ops 'access))
       (* (ops '*)))
 (match name
 (‘occurs-in-smaller-range-property
 (forall ?i ?j ?M ?x
 (if (and (exists ?u
             (and (in ?u (range ?i ?j))
                  (= (access ?M (* ?u)) ?x))
             (not (= (access ?M (* ?i)) ?x))
             (exists ?u (and (in ?u (range ++ ?i) ?j))
                         (= (access ?M (* ?u)) ?x)))))))

#################################################################

(define (combined-memory-range-property name ops)
 (let ((valid (ops 'valid)))

(definitions)

(match name
 (‘combined-memory-range-property
 (forall ?a ?b ?M ?x
 (if (and (valid (range ?a ?b))
             (not (assign ?M (* ?a) ?x)))
             (= (access-range ?M (range ?a ?b))
                (access-range ?M (range ?a ?b)))))))

#################################################################
(access-range (ops 'access-range)))

(match name
  ('combined-memory-range-property
   (forall ?i ?j
     (if (valid (range ?i ?j))
       (forall ?M ?k
         (if (and (valid (range ?i ?k))
            (valid (range ?k ?j)))
           (= (access-range ?M (range ?i ?j))
             (append (access-range ?M (range ?i ?k))
                   (access-range ?M (range ?k ?j))))))))))

#################################################################

(define (true-in-smaller-range name ops)
  (let ((++ (ops '++))
     (in (ops 'in))
     (apply (ops 'apply))
     (access (ops 'access))
     (* (ops '*)))
    (match name
      ('true-in-smaller-range
        (if (and (exists ?u
                   (and (in ?u (range ?i ?j))
                        (apply ?p (access ?M (* ?u))))
                    (not (apply ?p (access ?M (* ?i))))
                    (exists ?u (and (in ?u (range (++ ?i) ?j))
                                   (apply ?p (access ?M (* ?u))))))))))))

#################################################################
APPENDIX E

Formalization of Iterators and Valid Range

The Standard Template Library (STL) [29, 22] is a C++ library of generic algorithms, containers, and iterators which make it possible to write efficient generic code. Some of the STL containers are vector, list, set, and map. The data in the containers are accessed by C++ pointer-like objects called iterators. STL provides standard algorithms to solve many common problems like sorting, searching, merging, copying, etc. The algorithms are adaptable and easy to use.

Accessing sequence elements through a pair of iterators is another important operation used in STL-style generic code. A pair of iterators that points to the beginning and end of a sequence is called range. For example, a range \([i, j]\) refers to the elements in the container starting with the one pointed to by \(i\) and up to but not including the one pointed to by \(j\). When accessing the elements, not just any pair of iterators can be used, however; the pair must comprise a valid range. A pair of iterators constitute valid range if and only if the first iterator will become equal to the second after a (finite, possibly empty) sequence of applications of \texttt{operator++}.

This section contains some definitions, theorems, axioms, and proofs about iterators, iterator ranges, and valid ranges. This is a preliminary attempt at formalizing STL iterator concepts.

E.1 Iterator theory: formalization of iterators in Athena

E.1.1 Definition of iterator theory in CCT

\[
\begin{align*}
\text{domain (Iterator T))} \\
\text{define (Iterator name ops)} \\
\text{let ((+ (ops '++))} \\
\text{(- (ops '--))} \\
\text{(* (ops '*))} \\
\text{(I- (ops 'I-))}
\end{align*}
\]
(I+ (ops 'I+))
(I-I (ops 'I-I))

(match name
  ('iminus-zero-axiom
  (forall ?i
    (= (I- ?i zero) ?i)))
  ('iminus-succ-axiom
  (forall ?n ?i
    (= (I- ?i (succ ?n))
      (-- (I- ?i ?n)))))
  ('iminus-equal-axiom
  (forall ?i ?n ?m
    (if (= (I- ?i ?n)
        (I- ?i ?m))
      (= ?n ?m))))
  ('iplus-zero-axiom
  (forall ?i
    (= (I+ ?i zero) ?i)))
  ('iplus-succ-axiom
  (forall ?n ?i
    (= (I+ ?i (succ ?n))
      (++ (I+ ?i ?n)))))
  ('dec-equal-axiom
  (forall ?i ?j
    (if (= (-- ?i) (-- ?j))
      (= ?i ?j))))
  ('dec-inc-axiom
  (forall ?i
    (= (-- (++ ?i))
      ?i)))
  ('iterator-diff-axiom
  (forall ?i ?j ?n
(iff (= (I-I ?j ?i) ?n)
   (= ?i (I- ?j ?n))))

('memory-loc-property
  (forall ?i ?j
   (if (not (= ?i ?j))
       (not (= (* ?i) (* ?j)))))))

(define Iterator-theory
  (theory [Iterator Axiom]))

E.1.2 CCT theorems about iterators and some selected proofs

(define (iminus&--commute name ops)
  (let ((-- (ops '--))
         (I- (ops 'I-)))
    (match name
      ('iminus&--commute
       (forall ?n ?i
        (= (I- (-- ?i) ?n)
           (-- (I- ?i ?n)))))))

################################################################

(define (iminus-succ-property name ops)
  (let ((-- (ops '--))
         (I- (ops 'I-)))
    (match name
      ('iminus-succ-property
       (forall ?m ?n ?i
        (= (I- (I- ?i (succ ?m)) ?n)
           (I- (I- ?i ?m) (succ ?n)))))))

################################################################

(define (iminus-addition-property name ops)
  (let ((I- (ops 'I-)))
    (match name
('iminus-addition-property
 (forall ?m ?n ?i
   (= (I- (I- ?i ?m) ?n)
       (I- ?i (Plus ?m ?n)))))

(define (inc-property name ops)
  (let ((++ (ops '++))
         (I+ (ops 'I+)))
    (match name
      ('inc-property
       (forall ?i
         (= (++ ?i)
             (I+ ?i (succ zero))))))))

(define (dec-property name ops)
  (let ((-- (ops '--))
         (I- (ops 'I-)))
    (match name
      ('dec-property
       (forall ?i
         (= (-- ?i)
             (I- ?i (succ zero))))))))

(define (iminus-iplus-cancellation name ops)
  (let ((I+ (ops 'I+))
        (I- (ops 'I-)))
    (match name
      ('iminus-iplus-cancellation
       (forall ?n ?i
         (= (I- (I+ ?i ?n) ?n) ?n))))
(define (iminus-equal-property name ops)
  (let ((I- (ops 'I-)))
    (match name
      ('iminus-equal-property
        (forall ?n ?i ?j
          (if (= (I- ?i ?n)
              (I- ?j ?n))
            (= ?i ?j)))))))

(define (iplus-iminus-property name ops)
  (let ((I+ (ops 'I+))
        (I- (ops 'I-)))
    (match name
      ('iplus-iminus-property
        (forall ?n ?i
          (= (I+ (I- ?i ?n) ?n)
             ?i))))))

(define (inc-dec-property name ops)
  (let ((++ (ops '++))
        (-- (ops '--)))
    (match name
      ('inc-dec-property
        (forall ?i
          (= (++ (-- ?i))
             ?i))))))
(define (inc-dec-property-proof name ops)
  (dlet ((theorem (inc-dec-property name ops))
    (I+ (ops 'I+))
    (I- (ops 'I-))
    (++ (ops '++))
    (-- (ops '--))
    (prop (method (name)
        (!property name ops Iterator-theory)))
    (left (cell true))
    (right (cell true)))
  (!conclude theorem)
  (pick-any i
    (dseq
      (!setup left (++ (-- i)))
      (!setup right i)
      (!reduce left (I+ (-- i) (succ zero))
        (!prop 'inc-property))
      (!reduce left (I+ (I- i (succ zero)) (succ zero))
        (!prop 'dec-property))
      (!reduce left i
        (!prop 'iplus-iminus-property)))))))

(evolve
  Iterator-theory
  [inc-dec-property
    inc-dec-property-proof])

(define (inc-iminus-property name ops)
  (let ((++ (ops '++))
    (I- (ops 'I-)))
    (match name
      ('inc-iminus-property
        (prop 'iplus-iminus-property))))
(forall ?i ?n
   (= (++ (I- ?i (succ ?n)))
      (I- ?i ?n))))))

################################################################
(define (iterator-not-equal-property name ops)
  (let ((++ (ops '++))
         (-- (ops '--)))
    (match name
      ('iterator-not-equal-property
       (forall ?i ?j
         (if (and (not (= ?i ?j))
                (not (= (++ ?i) ?j)))
            (not (= ?i (-- ?j))))))))))

################################################################
(define (nonempty-range-property name ops)
  (let ((I- (ops 'I-)))
    (match name
      ('nonempty-range-property
       (forall ?n ?j
         (not (= (I- ?j (succ ?n)) ?j))))))

(define (nonempty-range-property-proof name ops)
  (dlet ((theorem (nonempty-range-property name ops))
         (I- (ops 'I-))
         (prop (method (name)
                      (!property name ops Iterator-theory)))
         (left (cell true))
         (right (cell true)))
    (! (conclude theorem)
      (pick-any
       n j)
(suppose-absurd
  (= (I- j (succ n)) j)
)(dseq
  ((= (I- j (succ n)) (I- j zero))
   BY (dseq
     (!setup left (I- j (succ n)))
     (!setup right (I- j zero))
     (!reduce left j (= (I- j (succ n)) j))
     (!reduce right j (!property 'iminus-zero-axiom
                        ops Iterator-theory))
     (!combine left right))))
(!absurd
  ((= (succ n) zero)
   BY (!spec (!property 'iminus-equal-axiom
                ops Iterator-theory) [j (succ n) zero]))
  ((not (= (succ n) zero))
   BY (!spec succ-not-zero [n]))))))))

(evolve
  Iterator-theory
  [nonempty-range-property
   nonempty-range-property-proof])

(define (iplus-not-equal-property name ops)
  (let (((+ (ops '++)))
        (match name
          ('iplus-not-equal-property
           (forall ?i
                    (not (= ?i (++ ?i)))))
  (define (iplus-not-equal-property-proof name ops)
    (dlet ((theorem (iplus-not-equal-property name ops)))
(++ (ops '++))
(I- (ops 'I-))
(I+ (ops 'I+))
(prop (method (name)
    (!property name ops Iterator-theory)))
(left (cell true))
(right (cell true))

(! (conclude theorem)
  (pick-any i
    (!by-contradiction
      (assume (= i (++ i))
        (!absurd
          (! (conclude (= (I- i (succ zero)) i))
            (dseq
              (! setup left (I- i (succ zero)))
              (! reduce left (I- (++ i) (succ zero))
                (= i (++ i)))
              (! reduce left (I- (I+ i (succ zero)) (succ zero))
                (! prop 'inc-property))
              (! reduce left i (! prop 'iminus-iplus-cancellation)))
            (! conclude (not (= (I- i (succ zero)) i)))
            (! uspec* (! prop 'nonempty-range-property) [zero i])
          ))))
  )))

(evolve
  Iterator-theory
  [iplus-not-equal-property
    iplus-not-equal-property-proof])

******************************************************************************

(define (I-I-zero-property name ops)
  (let ((I-I (ops 'I-I))
     (match name
       ('I-I-zero-property
         (!property name ops Iterator-theory))
       (...))))
(forall ?i
  (= (I-I ?i ?i) zero)))

(define (I-I-zero-property-proof name ops)
  (dlet ((theorem (I-I-zero-property name ops))
    (I-I (ops 'I-I))
    (I- (ops 'I-))
    (prop (method (name)
      (!property name ops Iterator-theory)))
    (left (cell true))
    (right (cell true)))
  (!conclude theorem)
  (pick-any i
    (dseq
      (!setup left i)
      (!setup right (I- i zero))
      (!reduce right i (!prop 'iminus-zero-axiom))
      (!combine left right)
      (!right-instance (!prop 'iterator-diff-axiom) [i i zero])))
)

(evolve
  Iterator-theory
  [I-I-zero-property
    I-I-zero-property-proof])

(define (iplus-left-cancellation name ops)
  (let ((I+ (ops 'I+)))
    (match name
      ('iplus-left-cancellation
        (forall ?i ?n
          (forall ?i ?n
            (if (= ?i (I+ ?i ?n))
              (= ?n zero)))))))
(define (iminus-to-iplus-property name ops)
  (let ((I+ (ops 'I+))
         (I- (ops 'I-)))
    (match name
      ('iminus-to-iplus-property
        (forall ?i ?j ?n
          (if (= ?i (I- ?j ?n))
            (= (I+ ?i ?n) ?j)))))))

(define (iminus-to-iplus-property-proof name ops)
  (dlet ((theorem (iminus-to-iplus-property name ops))
          (I- (ops 'I-))
          (I+ (ops 'I+))
          (prop (method (name)
               (!property name ops Iterator-theory)))
          (left (cell true))
          (right (cell true)))
    (pick-any
      i j n
      (assume (= i (I- j n)))
      (dseq
        (!setup left (I+ i n))
        (!setup right j)
        (!reduce left (I+ (I- j n) n) (= i (I- j n)))
        (!reduce left j (!prop 'iplus-iminus-property))
        (!combine left right)))))))

(evolve
  Iterator-theory
  [iminus-to-iplus-property
   iminus-to-iplus-property-proof])
(define (iplus-dec-property name ops)
  (let ((-- (ops ?--))
        (I+ (ops 'I+)))
    (match name
      ('iplus-dec-property
        (forall ?n ?i
          (= (I+ (-- ?i) ?n)
              (-- (I+ ?i ?n)))))))))

(define (iplus-iminus-succ-property name ops)
  (let ((I- (ops 'I-))
        (I+ (ops 'I+)))
    (match name
      ('iplus-iminus-succ-property
        (forall ?i ?n ?m
          (= (I+ (I- ?i (succ ?n)) (succ ?m))
              (I+ (I- ?i ?n) ?m)))))))))

(define (iminus-one-iplus-property name ops)
  (let ((I- (ops 'I-))
        (I+ (ops 'I+)))
    (match name
      ('iminus-one-iplus-property
        (forall ?n ?i
          (= (I- (I+ ?i (succ ?n)) (succ zero))
              (I+ ?i ?n)))))))

(define (iminus-++-property name ops)
(let ((++ (ops '++))
     (I- (ops 'I-)))
  (match name
    ('iminus-++-property
     (forall ?n ?i
      (= (I- (++ ?i) ?n)
       (++ (I- ?i ?n))))))))

################################################################
(define (not-inc-dec-property name ops)
  (let ((++ (ops '++))
        (-- (ops '--)))
    (match name
      ('not-inc-dec-property
       (forall ?i ?j
        (if (not (= (++ ?i) ?j))
          (not (= ?i (-- ?j)))))))))

################################################################
(define (iminus-not-equal-property name ops)
  (let ((-- (ops '--)))
    (match name
      ('iminus-not-equal-property
       (forall ?i
        (not (= ?i (-- ?i)))))))

(define (iminus-not-equal-property-proof name ops)
  (dlet ((theorem (iminus-not-equal-property name ops))
         (++ (ops '++))
         (-- (ops '--))
         (prop (method (name)
                       (!property name ops Iterator-theory)))
         (left (cell true)))
(right (cell true)))
(! (conclude theorem)
(pick-any
   i
(! by-contradiction
   (assume (= i (-- i)))
   (! absurd
     (dseq
      (! setup left i)
      (! setup right (++ i))
      (! reduce right (++ (-- i)) (= i (-- i)))
      (! reduce right i (! prop 'inc-dec-property))
      (! combine left right)
      (! uspec (! prop 'iplus-not-equal-property) i)))))))

(evolve
Iterator-theory
[iminus-not-equal-property
  iminus-not-equal-property-proof])

E.2 Range theory: formalization of valid range property in Athena

E.2.1 Definition of range theory in CCT

## Range definitions and a simple theorem.

(load-file "rewriting.ath")

(load-file "property-management.ath")

(load-file "iterator.ath")
(load-file "naturals-ordering.ath")

(datatype (Range T) (range T T))

(define (Range name ops)
  (let ((I- (ops 'I-))
        (I-I (ops 'I-I))
        (++ (ops '++))
        (valid (ops 'valid))
        (in (ops 'in)))
    (match name
      ('Valid-Range-definition
       (forall ?i ?j
        (iff (valid (range ?i ?j))
            (exists ?n (= ?i (I- ?j ?n))))))
      ('Valid-Range-I-I-axiom
       (forall ?i ?j
        (if (and (valid (range ?i ?j))
                  (not (= ?i ?j)))
            (= (I-I ?j ?i)
               (succ (I-I ?j (++ ?i))))))
      ('in-definition
       (forall ?a ?b ?i
        (iff (in ?i (range ?a ?b))
            (exists ?n (and (and (not (= ?n zero))
                              (<= ?n (I-I ?b ?a)))
                             (= ?i (I- ?b ?n))))))
      )
    ))

(define Range-theory
  (refine
    [Iterator-theory]
    [Range Axiom])))
E.2.2 CCT theorems about valid ranges and some selected proofs

(define (Valid-Range-property name ops)
  (let ((I- (ops 'I-))
        (valid (ops 'valid)))
    (match name
      ('Valid-Range-property
        (forall ?n ?j
          (valid (range (I- ?j ?n) ?j)))))))

(define (Valid-Range-Reflexive-property name ops)
  (let ((valid (ops 'valid)))
    (match name
      ('Valid-Range-Reflexive-property
        (forall ?i
          (valid (range ?i ?i)))))))

(define (Valid-Range-Reflexive-property-proof name ops)
  (dlet ((theorem (Valid-Range-Reflexive-property name ops))
        (I- (ops 'I-))
        (prop (method (name)
                       (!property name ops Range-theory)))
        (left (cell true))
        (right (cell true))
        (valid (ops 'valid)))
    (!conclude theorem)
    (pick-any i
      (dseq
        ((exists ?n (= i (I- i ?n)))
         (forall (range ?i ?i)))
      (/setup left i))
(!setup right (I- i zero))
(!reduce right i (!prop 'iminus-zero-axiom))
(!combine left right))
(!egen (exists ?n (= i (I- i ?n))) zero)))
(!spec-right (!prop 'Valid-Range-definition)
 [i i]))))

(evolve
 Range-theory
 [Valid-Range-Reflexive-property
  Valid-Range-Reflexive-property-proof])

(define (Valid-Range-Transitive-property name ops)
 (let ((valid (ops 'valid)))
  (match name
    ('Valid-Range-Transitive-property
      (forall ?i ?k ?j
        (if (and (valid (range ?i ?k))
            (valid (range ?k ?j)))
            (valid (range ?i ?j))))))))

(define (Valid-Range-Transitive-property-proof name ops)
 (dlet ((valid (ops 'valid))
    (I- (ops 'I-))
    (left (cell true))
    (prop (method (name)
      (!property name ops Range-theory)))
    (theorem (Valid-Range-Transitive-property name ops)))
  (!conclude theorem)
  (pick-any
    i j k)
(assume-let (assumptions (and (valid (range i j)))
               (valid (range j k))))

(! (conclude (valid (range i k))))

(pick-witness m

(! (conclude (exists ?m (= i (I- j ?m)))))

(dseq
  (! left-and assumptions)
  (! instance (! prop 'Valid-Range-definition
                [i j])))

(pick-witness p

(! (conclude (exists ?p (= j (I- k ?p)))))

(dseq
  (! right-and assumptions)
  (! instance (! prop 'Valid-Range-definition
                [j k])))

(dseq
  (! (conclude (= i (I- k (Plus p m)))))

(dseq
  (! setup left i)
  (! reduce left (I- j m)
               (= i (I- j m)))
  (! reduce left (I- (I- k p) m)
               (= j (I- k p)))
  (! reduce left (I- k (Plus p m))
               (! prop 'iminus-addition-property))
  (! egen (exists ?n (= i (I- k ?n)))
          (Plus p m))
  (! right-instance (! prop 'Valid-Range-definition
                     [i k])))
)

(evolve
 Range-theory
 [Valid-Range-Transitive-property Valid-Range-Transitive-property-proof])
(define (Valid-Range-length-atleast-one-property name ops)
  (let ((I- (ops 'I-))
        (valid (ops 'valid)))
    (match name
      ('Valid-Range-length-atleast-one-property
        (forall ?i ?j
          (if (and (valid (range ?i ?j))
                  (not (= ?i ?j)))
            (exists ?n (= ?i (I- ?j (succ ?n)))))
        ))))

(define (Valid-Range-smaller1-property name ops)
  (let ((valid (ops 'valid))
        (++ (ops '++))
        (match name
          ('Valid-Range-smaller1-property
            (forall ?i ?j
              (if (and (valid (range ?i ?j))
                      (not (= ?i ?j)))
                (valid (range (++ ?i) ?j))))
          ))))

(define (Valid-Range-smaller1-property-proof name ops)
  (dlet ((theorem (Valid-Range-smaller1-property name ops))
        (++ (ops '++))
        (I- (ops 'I-))
        (left (cell true))
        (right (cell true))
        (valid (ops 'valid)))
    (! (conclude theorem)
    (pick-any
i j
(assume (and (valid (range i j))
            (not (= i j)))
(!!spec (!property 'Valid-Range-length-atleast-one-property
            ops Range-theory) [i j]))
(pick-witness n (exists ?n (= i (I- j (succ ?n))))
   (!spec (!property 'Valid-Range-definition
            ops Range-theory) [(++ i) j])))))

(evolve Range-theory
[Valid-Range-smaller1-property
   Valid-Range-smaller1-property-proof ])

(define (Valid-Range-smaller2-property name ops)
(let ((valid (ops 'valid))
     (++ (ops '++))
     (-- (ops '--)))

(match name

  ('Valid-Range-smaller2-property
    (forall ?a ?b
      (if (and (valid (range ?a ?b))
              (and (not (= ?a ?b))
                   (not (= (++ ?a) ?b))))
       (valid (range (++ ?a) (-- ?b)))))))

(define (Valid-Range-smaller2-property-proof name ops)
  (dlet ((theorem (Valid-Range-smaller2-property name ops))
         (++ (ops '++))
         (-- (ops '--))
         (I- (ops 'I-))
         (left (cell true))
         (right (cell true))
         (valid (ops 'valid)))

  (! (conclude theorem)
    (pick-any
     a b
     (assume-let (assumptions (and (valid (range a b))
                                 (and (not (= a b))
                                      (not (= (++ a) b))))))

     (dseq
      (! left-and assumptions)
      (! left-and (! right-and assumptions))
      (! right-and (! right-and assumptions))
      (! both (valid (range a b))
        (not (= a b)))
      (! (conclude (valid (range (++ a) b))))
      (! spec (! property 'Valid-Range-smaller1-property

ops Range-theory) [a b])

(!(conclude (exists ?n (= (++ a) (I- b ?n)))))

(!spec (!property 'Valid-Range-definition
  ops Range-theory) [((+ a) b)])

(!both (valid (range (++ a) b))
  (not (= (++ a) b)))

(!(conclude (exists ?n (= (++ a) (I- b (succ ?n))))))

(!spec (!property 'Valid-Range-length-atleast-one-property
  ops Range-theory) [((+ a) b)])

(!(conclude (exists ?m (= (++ a) (I- (- b) ?m))))

(pick-witness n
  (exists ?n (= (++ a) (I- b (succ ?n))))
  (dseq
    ((= (++ a) (I- (- b) n))
      BY (dseq
        (!setup left (++ a))
        (!setup right (I- (- b) n))
        (!reduce left (I- b (succ n))
          (= (++ a) (I- b (succ n))))
        (!expand left (I- (- b zero) (succ n))
          (!property 'iminus-zero-axiom
            ops Iterator-theory))
        (!reduce right (I- (- b (succ zero)) n)
          (!property 'dec-property
            ops Iterator-theory))
        (!reduce right (I- (- b zero) (succ n))
          (!property 'iminus-succ-property
            ops Iterator-theory))
        (!combine left right)))))

(!egen (exists ?m (= (++ a) (I- (- b) ?m)))
  n))

(!spec-right (!property 'Valid-Range-definition
  ops Range-theory) [(+ a) (- b)])

)}
(define (Valid-Range-diff1-property name ops)
  (let ((valid (ops 'valid))
         (I-I (ops 'I-I)))
    (match name
      ('Valid-Range-diff1-property
        (forall ?i ?j
          (if (valid (range ?i ?j))
            (exists ?n
              (<= ?n (I-I ?j ?i)))))))))

(define (Valid-in-property name ops)
  (let ( (++ (ops '++))
          (-- (ops '--))
          (in (ops 'in)))
    (match name
      ('Valid-in-property
        (forall ?a ?b ?i
          (if (in ?i (range (++ ?a) (-- ?b)))
            (and (not (= ?i ?a))
             (not (= ?i (-- ?b))))))))))

(define (Valid-in-property-proof name ops)
  (dlet ((theorems (Valid-in-property name ops))
          (++ (ops '++)))
(--- (ops '---))
(I- (ops 'I-))
(I-I (ops 'I-I))
(in (ops 'in))
(valid (ops 'valid))
(left (cell true))
(right (cell true))
(prop (method (name)
    (!property name ops Range-theory))))

(! (conclude theorem)

(pick-any
  a b i

  (assume (in i (range (++ a) (-- b))))

  (dseq
    (! (conclude
        (exists ?nn (and (and (not (= ?nn zero))
                          (<= ?nn (I-I (-- b) (++ a))))
                          (= i (I- (-- b) ?nn)))))))

    (!spec (!prop 'in-definition) [(++ a) (-- b) i]))

(pick-witness n

  (exists ?nn (and (and (not (= ?nn zero))
                        (<= ?nn (I-I (-- b) (++ a))))
                        (= i (I- (-- b) ?nn)))))

  (dlet ((n-props (and (and (not (= n zero))
                          (<= n (I-I (-- b) (++ a))))
                          (= i (I- (-- b) n))))))

  (dseq
    (! (conclude (not (= n zero))))

    (!left-and (!left-and n-props)))

    (! (conclude (exists ?m (= n (succ ?m))))

    (! mp (!uspec nat-property n) (not (= n zero))))

    (!right-and (!left-and n-props))

    (! (conclude (not (= i a)))))
(pick-witness k
(exists ?m (= n (succ ?m)))
(!by-contradiction
(assume (= i a)
  (!absurd
    (dseq
      (!conclude (= i (I- (-- b) n)))
      (right-and n-props))
      (!conclude (= a (I- (-- b) n)))
      (!tran (!sym (= i a)) (= i (I- (-- b) n)))
      (!conclude (= (I-I (-- b) a) n))
      (!right-instance (!prop 'iterator-diff-axiom)
        [a (-- b) n]))
      (!conclude (= (I-I (-- b) a) (succ k)))
    (dseq
      (!setup left (I-I (-- b) a))
      (!setup right (succ k))
      (!reduce left n (= (I-I (-- b) a) n))
      (!expand right n (= n (succ k)))
      (!combine left right)))
  (!egen (exists ?h (= a (I- (-- b) ?h)))
    n)
  (!conclude (valid (range a (-- b)))))
(!spec-right (!prop 'Valid-Range-definition) [a (-- b)])
(!conclude (not (= a (-- b)))))
(!by-contradiction
(assume (= a (-- b))
  (!absurd
    (!conclude (= (succ k) zero))
    (dseq
      (!setup left (succ k))
      (!expand left (I-I (-- b) a)
        (= (I-I (-- b) a) (succ k))))))
(!expand left
   (I-I a a) (= a (-- b)))
(!reduce left zero
   (!prop 'I-I-zero-property)))
(!uspec succ-not-zero k))))))
(!both (valid (range a (-- b)))
   (not (= a (-- b))))
(!conclude (= (I-I (-- b) a)
   (succ (I-I (-- b) (++ a)))))
(!spec (!prop 'Valid-Range-I-I-axiom) [a (-- b)])
(!conclude (= (succ k)
   (succ (I-I (-- b) (++ a)))))
(!tran (!sym (= (I-I (-- b) a) (succ k))
   (= (I-I (-- b) a)
   (succ (I-I (-- b) (++ a))))))
(!conclude (= k (I-I (-- b) (++ a)))
(!mp (!uspec* succ-succ-property
   [k (I-I (-- b) (++ a)])
   (= (succ k)
   (succ (I-I (-- b) (++ a))))))
(dseq
   (!equality (I-I (-- b) (++ a)) (I-I (-- b) (++ a)))
   (!conclude (<= (succ k) (I-I (-- b) (++ a)))))
   (!rel-cong-2 (<= n (I-I (-- b) (++ a)))
   [(succ k) (I-I (-- b) (++ a))])
   (!conclude (not (= k (I-I (-- b) (++ a)))))
   (!spec <=-a-property
   [k (I-I (-- b) (++ a))])))
(!conclude (not (= i (-- b))))
(by-contradiction
   (assume (= i (-- b)))
   (!absurd
   (!conclude (= n zero)))
(dseq
  (!conclude (= i (I- (-- b) n)))
  (!right-and n-props)
  (!conclude (= (-- b) (I- (-- b) n)))
  (!tran (!sym (= i (-- b)))
    (= i (I- (-- b) n)))
  (!conclude (= (I-I (-- b) (-- b)) n))
  (!right-instance (!prop 'iterator-diff-axiom)
    [(-- b) (-- b) n])
  (!setup left n)
  (!setup right zero)
  (!expand left (I-I (-- b) (-- b))
    (= (I-I (-- b) (-- b)) n))
  (!expand right (I-I (-- b) (-- b))
    (!uspec (!prop 'I-I-zero-property) (-- b)))
  (!combine left right)))
  (not (= n zero)))
  (!conclude (and (not (= i a))
    (not (= i (-- b))))))
  (!both (not (= i a)) (not (= i (-- b)))))
)

(evolve
  Range-theory
  [Valid-in-property
    Valid-in-property-proof])

(define (Valid-not-in-range-property name ops)
  (let ((valid (ops 'valid))
    (-- (ops '--))
    (in (ops 'in)))
    (match name
      ('Valid-not-in-range-property
(forall ?i ?j ?k
    (if (and (valid (range ?i ?j))
        (and (not (= ?i ?j))
            (not (in ?k (range ?i ?j))))
        (and (not (= ?k ?i))
            (not (= ?k (-- ?j)))))))))

################################################################
(define (not-in-property name ops)
    (let ((++ (ops '++))
        (-- (ops '--))
        (in (ops 'in)))
        (match name
            ('not-in-property
                (forall ?i ?j
                    (not (in ?i (range (++ ?i) (-- ?j))))))))))

(define (not-in-property-proof name ops)
    (dlet ((++ (ops '++))
        (I+ (ops 'I+))
        (theorem (not-in-property name ops))
        (-- (ops '--))
        (in (ops 'in))
        (prop (method (name)
            (!property name ops Range-theory))))
    (!conclude theorem)
    (pick-any
        i j
        (!by-contradiction
            (assume (in i (range (++ i) (-- j))))
            (!absurd
                (!equality i i)
                (!conclude (not (= i i)))))
    )
(dseq
  (conclude (and (not (= i i))
    (not (= i (-- j))))))
  (spec (!prop 'Valid-in-property) [i j i])
  (left-and (and (not (= i i))
    (not (= i (-- j)))))))
)

(evolve
  Range-theory
  [not-in-property
    not-in-property-proof])

(define (not-in-property2 name ops)
  (let ((++ (ops '++))
    (-- (ops '--))
    (in (ops 'in)))
    (match name
      ('not-in-property2
        (forall ?i ?j
          (not (in (-- ?j) (range (++ ?i) (-- ?j))))))))))

(define (not-in-smaller-range name ops)
  (let ((++ (ops '++))
    (in (ops 'in))
    (valid (ops 'valid)))
    (match name
      ('not-in-smaller-range
        (forall ?i ?j ?r
          (if (and (valid (range ?i ?j))
            (and (not (= ?i ?j))
              (not (in ?r (range ?i ?j))))))))))

(define (not-in-smaller-range1 name ops)
  (let ((++ (ops '++))
        (in (ops 'in))
        (valid (ops 'valid)))
    (match name
      ('not-in-smaller-range1
       (forall ?i ?j ?r
        (if (and (valid (range ?i ?j))
               (and (not (= ?i ?j))
                (not (in ?r (range ++ ?i ?j)))))))))
      (not (in ?r (range ++ ?i ?j)))))))

(define (Valid-Range-Converse-property name ops)
  (let ((valid (ops 'valid))
        (++ (ops '++)))
    (match name
      ('Valid-Range-Converse-property
       (forall ?i ?j
        (if (valid (range ++ ?i ?j))
            (and (valid (range ?i ?j))
                 (not (= ?i ?j)))))))))

(define (first-in-range-property name ops)
  (let ((++ (ops '++))
        (in (ops 'in))
        (valid (ops 'valid)))
    (match name
      ('first-in-range-property
(forall ?i ?j
  (if (and (valid (range ?i ?j))
    (not (= ?i ?j))
    (in ?i (range ?i ?j)))))))


(define (first-not-in-rest-property name ops)
  (let ((++ (ops '++))
    (in (ops 'in)))
    (match name
      ('first-not-in-rest-property
        (forall ?a ?b ?i
          (if (in ?i (range (++ ?a) ?b))
            (not (= ?i ?a))))))))


(define (not-in-property0 name ops)
  (let ((++ (ops '++))
    (in (ops 'in)))
    (match name
      ('not-in-property0
        (forall ?i ?j
          (not (in ?i (range (++ ?i) ?j)))))))


(define (not-in-property0-proof name ops)
  (dlet ((++ (ops '++))
    (I+ (ops 'I+))
    (in (ops 'in))
    (theorem (not-in-property0 name ops))
    (prop (method (name)
      (!property name ops Range-theory)))
    (! (conclude theorem)
      (pick-any
i j

(!by-contradiction

(assume (in i (range (++ i) j))

(!absurd

(!equality i i)

(!conclude (not (= i i)))

(!instance (!prop 'first-not-in-rest-property)

[i j i])))))))))))

(evolve

Range-theory

[not-in-property0

not-in-property0-proof])

#################################################################

(define (in-smaller-range-property name ops)

(let ((in (ops 'in))

  (++ (ops '++))

  (match name

    ('in-smaller-range-property

      (forall ?i ?j ?u

        (if (and (in ?u (range ?i ?j))

          (not (= ?u ?i)))

          (in ?u (range (++ ?i) ?j))))))))

#################################################################

(define (not-in-empty-range name ops)

(let ((in (ops 'in))

  (match name

    ('not-in-empty-range

      (forall ?i ?u

        (not (in ?u (range ?i ?i))))))))
(define (not-in-empty-range-proof name ops)
  (dlet ((in (ops 'in))
          (I- (ops 'I-))
          (I-I (ops 'I-I))
          (prop (method (name)
                        (!property name ops Range-theory))))
    (pick-any i u
      (!((conclude (not (in u (range i i))))
         (!by-contradiction
          (assume (in u (range i i))
            (pick-witness n
              (!((conclude (exists ?n (and (and (not (= ?n zero))
                                             (<= ?n (I-I i i)))
                                         (= u (I- i ?n))))
                 (!instance (!prop 'in-definition)
                           [i i u]))
              (dlet ((witnessed
                       (and (and (not (= n zero))
                               (<= n (I-I i i)))
                               (= u (I- i n))))
                 (dseq
                   (!((conclude (<= n zero))
                      (dseq
                        (!instance (!prop 'I-I-zero-property)
                        [i])
                        (!substitute-equals (I-I i i)
                        (!right-and
                         (!left-and witnessed)
                         zero)))
                   (!cd
                    (!((conclude (or (< n zero) (= n zero))))
                      )
(instance <=-definition [n zero])
(assume (< n zero)
  (!absurd (< n zero)
    ((instance <=-not-zero-axiom [n]))))
(assume (= n zero)
  (!absurd (= n zero)
    (!left-and
     (!left-and witnessed)))))

(evolve
  Range-theory
  [not-in-empty-range
   not-in-empty-range-proof])

(define (range-separation-property name ops)
  (let ((valid (ops 'valid))
        (in (ops 'in)))
    (match name
      ('range-separation-property
        (forall ?i ?j ?p ?k
          (if (valid (range ?i ?j))
              (if (in ?k (range ?p ?i))
                  (not (in ?k (range ?i ?j))))))))))

(define (range-separation-property2 name ops)
  (let ((valid (ops 'valid))
        (in (ops 'in)))
    (match name
      ('range-separation-property2
        (forall ?i ?j ?p ?k
          (forall ?i ?j ?p ?k
            ...))))
(if (valid (range ?p ?i))
    (if (in ?k (range ?i ?j))
        (not (in ?k (range ?p ?i)))))))

#############################################
APPENDIX F
Proof of sum-list-compute-relation

(define (sum-list-compute-relation name ops)
  (match name
    ('sum-list-compute-relation
      (forall ?L ?x
        (= (sum-list@ (Cons ?x ?L))
          (sum-list-compute ?L ?x))))))

(define (sum-list-compute-relation-proof name ops)
  (dlet ((Plus (ops 'Plus))
          (Zero (ops 'Zero))
          (theorem (sum-list-compute-relation name ops))
          (prop (method (name)
                         (!property name ops Sum-list-theory)))
          (left (cell true))
          (right (cell true)))
  (by-induction
   theorem
   (Nil
    (!conclude (forall ?x
                     (= (sum-list@ (Cons ?x Nil))
                        (sum-list-compute Nil ?x))))
    (pick-any x
      (dseq
       (!setup left (sum-list@ (Cons x Nil)))
       (!setup right (sum-list-compute Nil x))
       (!reduce left (Plus x (sum-list@ Nil))
                  (!prop 'sum-list@-nonempty))
       (!reduce left (Plus x Zero)
                  (!prop 'sum-list@-empty))
(!reduce left x
   (!prop 'Right-Identity))

(!reduce right x
   (!prop 'sum-list-compute-empty))

(!combine left right))))))

((Cons y L)
 (!conclude (forall ?x
   (= (sum-list@ (Cons ?x (Cons y L)))
       (sum-list-compute (Cons y L) ?x))))

(dlet ((induction-hypothesis
          (forall ?x
            (= (sum-list@ (Cons ?x L))
               (sum-list-compute L ?x))))))

(pick-any x

(dseq
   (!setup left (sum-list@ (Cons x (Cons y L))))
   (!setup right (sum-list-compute (Cons y L) x))
   (!reduce left (Plus x (sum-list@ (Cons y L)))
                (!prop 'sum-list@-nonempty))
   (!reduce left (Plus x (Plus y (sum-list@ L)))
                (!prop 'sum-list@-nonempty))
   (!reduce right (sum-list-compute L (Plus x y))
                (!prop 'sum-list-compute-nonempty))
   (!expand right (sum-list@ (Cons (Plus x y) L))
               induction-hypothesis)
   (!reduce right (Plus (Plus x y) (sum-list@ L))
                (!prop 'sum-list@-nonempty))
   (!reduce right (Plus x (Plus y (sum-list@ L)))
                (!prop 'Associative))
   (!combine left right)))))))))

(evolve

   Sum-list-theory
[sum-list-compute-relation
sum-list-compute-relation-proof])