Contents

Preface

This story grew in the telling. We set out to edit the proceedings of a workshop on the ACL2 theorem prover—adding a little introductory material to tie the research papers together—and ended up not with one but with two books. The subject of both books is computer-aided reasoning, from the ACL2 perspective. The first book is about *how* do it; the second book, this one, is about *what* can be done.

The creation of ACL2, by Kaufmann and Moore, was the first step in the process of writing this book. It was a step that took many years and involved many people and organizations. We only list the names of the people here, but in the Preface to the first book we give more complete acknowledgments. For help in creating ACL2 and developing many of the modeling and proof techniques used here, Kaufmann and Moore thank Ken Albin, Larry Akers, Bill Bevier, Bob Boyer, Bishop Brock, Alessandro Cimatti, Rich Cohen, George Cotter, John Cowles, Art Flatau, Noah Friedman, Ruben Gamboa, Fausto Giunchiglia, Norm Glick, David Greve, Don Good, David Hardin, Calvin Harrison, Joe Hill, Warren Hunt, Terry Ireland, Robert Krug, Laura Lawless, Bill Legato, Tom Lynch, Panagiotis (Pete) Manolios, William McCune, Robert Morris, Dave Opitz, Laurence Pierre, Dave Reed, David Russinoff, Jun Sawada, Bill Schelter, Bill Scherlis, Larry Smith, Mike Smith, Rob Sumners, Ralph Wachter, Matthew Wilding, and Bill Young. We also are indebted to those who defined Common Lisp, and the entire user communities of both ACL2 and its predecessor, the Boyer-Moore theorem prover (Nqthm). Bob Boyer deserves special recognition for his contributions to ACL2's design and implementation during the first few years of its development.

Financial and moral support during the first eight years of ACL2's creation was provided by the U.S. Department of Defense, including DARPA and the Office of Naval Research, and Computational Logic, Inc. Subsequently, ACL2's development has been supported in part by the University of Texas at Austin, the Austin Renovation Center of EDS, Inc., Advanced Micro Devices, Inc., and Rockwell Collins, Inc.

Turning from the ACL2 system to this book, we owe our greatest debt to the participants in the 1999 workshop where the idea of these books was born. Ken Albin, Warren Hunt, and Matthew Wilding were among the first to push for a workshop. We thank those participants who wrote material

for the book, including Vernon Austel and all those listed in the table of contents.

In addition to all the contributors and many of the people named above, we thank Rajeev Joshi, Yi Mao, Jennifer Maas, George Porter, and David Streckmann for proof reading drafts of various portions of the book.

We thank the series editor, Mike Hinchey, and Lance Wobus at Kluwer, who patiently tolerated and adjusted to the increasing scope of this enterprise.

For many months now, much of our "free time" has been spent writing and editing these books. Without the cooperation and understanding support of our wives we simply would not have done it. So we thank them most of all.

Matt Kaufmann
Panagiotis Manolios
J Strother Moore

Austin, Texas
February 2000

List of Contributors

Piergiorgio Bertoli
IRST - Istituto per la Ricerca
Scientifica e Tecnologica
Povo, Italy
Email: bertoli@itc.it

Dominique Borrione
TIMA-UJF
Grenoble, France
Email: Dominique.Borrione@imag.fr

John Cowles
Department of Computer Science
University of Wyoming
Laramie, Wyoming
Email: cowles@uwyo.edu

Arthur Flatau
Advanced Micro Devices, Inc.
Austin, Texas
Email: arthur.flatau@amd.com

Ruben Gamboa
Logical Information Machines, Inc.
Austin, Texas
Email: ruben@lim.com

Philippe Georgelin
TIMA-UJF
Grenoble, France
Email: Philippe.Georgelin@imag.fr

Wolfgang Goerigk
Institut für Informatik und
Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Kiel, Germany
Email: wg@informatik.uni-kiel.de

David Greve
Rockwell Collins
Advanced Technology Center
Cedar Rapids, Iowa
Email: dagreve@collins.rockwell.com

David Hardin Ajile Systems, Inc.
Oakdale, Iowa
Email: david.hardin@ajile.com

Warren A. Hunt, Jr. IBM Austin Research Laboratory
Austin, Texas
Email: WHunt@Austin.IBM.COM

Damir A. Jamsek IBM Austin Research Laboratory
Austin, Texas
Email: jamsek@us.ibm.com

Matt Kaufmann Advanced Micro Devices, Inc.
Austin, Texas
Email: matt.kaufmann@amd.com

Panagiotis Manolios Department of Computer Sciences
University of Texas at Austin
Austin, Texas
Email: pete@cs.utexas.edu

William McCune Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois
Email: mccune@mcs.anl.gov

J Strother Moore Department of Computer Sciences
University of Texas at Austin
Austin, Texas
Email: moore@cs.utexas.edu

Vanderlei Rodrigues TIMA-UJF
Grenoble, France
(on leave from UFRGS,
Porto Alegre, Brazil)
Email: vandi@inf.ufrgs.br

David M. Russinoff Advanced Micro Devices, Inc.
Austin, Texas
Email: david.russinoff@amd.com

Jun Sawada Department of Computer Sciences,
University of Texas at Austin

Austin, Texas
Email: sawada@cs.utexas.edu

Olga Shumsky
Department of Electrical and
Computer Engineering
Northwestern University
Evanston, Illinois
Email: shumsky@ece.nwu.edu

Paolo Traverso
IRST - Istituto per la Ricerca
Scientifica e Tecnologica
Povo, Italy
Email: leaf@itc.it

Matthew Wilding
Rockwell Collins
Advanced Technology Center
Cedar Rapids, Iowa
Email: mmwildin@collins.rockwell.com

1

Introduction

This book shows what can be done with computer aided reasoning. Included here are descriptions of mathematical, hardware, and software systems. These systems and their desired properties are modeled with formulas in a mathematical language. That language has an associated mechanized reasoning tool, called ACL2, which is used to prove that these properties hold. With these techniques it is possible to describe components clearly and reliably, permitting them to be combined in new ways with predictable results.

The heart of the book reports on a sequence of case studies carried out by twenty-one researchers, including the three editors. The case studies are summarized starting on page 21. These studies are self-contained technical papers. They contain exercises for the reader who wants to master the material. In addition, complete ACL2 solutions for both the exercises and the results reported in each case study are available on the Web, as described below.

The book is meant for two audiences: those looking for innovative ways to design, build, and maintain systems (especially hardware and software) faster and more reliably, and those wishing to learn how to do this. The former audience includes project managers in the hardware or software industry and students in survey-oriented software engineering courses. The latter audience includes students and professionals pursuing rigorous approaches to hardware and software engineering or formal methods, who may consider applying such methods in their work. We include in this audience fellow researchers in formal methods who are building "competing" systems and who wish to keep abreast of what is happening in the ACL2 camp.

We assume you are familiar with computer programming. We also assume you are familiar with traditional mathematical notation: for example, "$f(x,y)$" denotes the application of the function f to (the values denoted by) x and y, and "$|x|$" denotes either the absolute value of x or its cardinality, depending on the context.

We also assume that you are comfortable with the idea that mathematics can be used to describe and predict the behavior of physical artifacts. This notion is fundamental to modern engineering. It is non-controversial that mathematically assisted engineering allows the construction of reliable complex systems faster than can be built by "intuitive engineering."

A major difficulty with applying mathematical modeling and analysis to the engineering of hardware and software systems is that the mathematics traditionally taught in college—calculus—is inappropriate for application to discrete systems. An appropriate mathematical framework is symbolic logic, where it is possible to describe and analyze the properties of recursive functions on inductively constructed domains. Mechanical tools exist to assist people in reasoning about such systems, relieving them of the heavy burden of logical correctness while targeting their talents towards creative insights and the high level decomposition of the problem.

If you are a member of the audience looking for innovative ways to build systems, you need not care about the mathematical details as long as you accept that mathematical modeling and analysis are the keys to better engineering and that practitioners must be trained in appropriate mathematics and tools. The operative questions are probably "What can be done by these people with their tools?" and "How long does it take?" This book addresses these questions.

If you are a member of the other audience and wish to learn how to do such modeling and analysis, we recommend that you eventually also read the companion book, *Computer-Aided Reasoning: An Approach* [58]. But the present book is of interest because it shows you what is possible. It also provides many exercises on which you can hone the skills taught in [58].

Flipping through this book will reveal a certain uniformity: Lisp expressions appear everywhere! That is because the mathematical logic used here, ACL2, is based on Common Lisp. To be more precise, it is a functional (side-effect free) extension of a subset of Common Lisp. Such a language is ideally suited to many modeling problems, because it supports formal models—of algorithms, compilers, microprocessors, machine languages—that are both executable and analyzable. That is, the formal models do double duty: they specify the results to be delivered and they can be used as efficient simulators. For example, before proving a floating-point multiplier correct, one might test it with millions of test vectors. Or, engineers unfamiliar with formal methods might compile and run Java programs on a formal model of a microprocessor. Such applications are real and are described here. Indeed, the ACL2 system itself is written almost entirely in the ACL2 language; its size (6 megabytes of source code), reliability, efficiency, and general utility demonstrate the practicality of the language.

What is surprising to many people is the range of ideas that can be discussed with such a simple language. The heart of this book consists of fourteen chapters written by various contributors. The ACL2 language is used to model and analyze problems in the following areas: graph theory, model checking, integral calculus, microprocessor simulation, pipelined architectures, an occurrence-oriented hardware description language, VHDL, symbolic trajectory analysis, floating-point multiplication, a safety-critical compiler, Trojan horses, a proof checker for the Otter theorem prover, a mathematical challenge by Knuth, and non-standard real analysis. The breadth of this collection is more impressive once you realize that all the claims are expressed in a formal mathematical system and all the theorems cited are proved mechanically. The list of authors also shows that you do not have to be the creators of ACL2 to use the system well.

One might group these case studies into four categories: tutorial, hardware, software, and mathematics. But there is much overlap. The tutorial on model checking explains, with the precision of ACL2, an algorithm used primarily in hardware verification; but the chapter then proves the algorithm correct, an exercise in software verification. The hardware models are written in executable Lisp; hence, the theorems proved are really theorems about software systems that simulate hardware. The safety-critical compiler uses modular arithmetic and hence depends on the Chinese remainder theorem, which is normally considered a mathematical exercise.

The book is divided into two parts. In Part I we deal very briefly with certain preliminaries: the effort involved in pursing this approach, followed by an extremely brief discussion of the ACL2 logic and its mechanization that is intended to provide the necessary background for the second part. The heart of the book is Part II, where the case studies are presented.

The authors of these case studies were asked to do three things that are quite unusual. First, they were asked to provide exercises in their particular applications. Second, they were asked to provide solutions to all their exercises so that we, the editors, could post them on the Web. Third, they were asked to provide us with the entire ACL2 scripts necessary to formalize the models and prove all the properties discussed in their studies. These too are on the Web, as described below. When we say, for example, that one of the case studies formalizes a floating-point multiplier and proves it correct, we mean that not only can you read an English description of the model and how it was proved correct, but you can obtain the entire transcript of the project and replay the proofs, if you wish, on your copy of ACL2. Several industrial projects were "sanitized" for inclusion here (or were not included at all). But the resulting scripts are extremely valuable to the serious student of formal methods. Every case study can be treated as an exercise in formalizing the model and proof described, and a complete solution is available to help you through the hard parts.

Recall that we edited the book with two audiences in mind. If you are a member of the first audience, looking to survey the state of the art, we recommend that you read both parts, but not pay too much attention to the formulas in the second part. Most of the case studies paraphrase the formulas. Just remember that not only can the informal remarks be made mathematically precise but they are being made precise; not only can the arguments be checked by machine, they were checked by machine. Indeed, you can obtain the scripts if you wish. We also recommend that you read the exercises, even though we do not expect you to do them. By reading them you will learn what experienced users think are reasonable challenges for people expecting to move on to industrial applications.

If you are a member of the second audience, trying to learn how to do this, then your approach to this book depends on whether you are already familiar with ACL2. If so, we recommend that you skim Part I. Then, read the first three case studies of Part II, doing many of the exercises as you go. Once you get through that, we recommend reading the rest of Part II and doing the exercises for those studies that seem relevant to your own work.

On the other hand, if you want to learn ACL2 but have not yet begun, we recommend reading this book in the "survey style" suggested above, so you get an idea of the kind of thinking required. We then recommend that you read and work your way through the companion book [58], and then return to the exercises in this book, starting with the first three case studies.

The ACL2 system is available for free on the Web (under the terms of the Gnu General Public License). The ACL2 home page is `http://-www.cs.utexas.edu/users/moore/acl2`. There you will find the source code of the system, downloadable images for several platforms, installation instructions, two guided tours, a quick reference card, tutorials, an online User's Manual, useful email addresses (including how to join the mailing list or ask the community for help), scientific papers about applications, and much more.

The ACL2 online documentation is almost 3 megabytes of hypertext and is available in several formats. The HTML version can be inspected from the ACL2 home page with your browser. Other formats are explained in the "Documentation" section of the installation instructions accessible from the ACl2 home page.

Important: In this book, you will often see underlined strings in typewriter font in such phrases as "see `defthm`." These are references to the online documentation. To pursue them, go to the ACL2 home page, click on "The User's Manual" link, and then click on the "Index of all documented topics." You will see a list from A to Z. Click on the appropriate letter and scan the topics for the one referenced (in this case, `defthm`) and click on it.

While the online documentation is quite extensive, it is not organized linearly. There are several tutorials and fully worked problems, but the documentation is primarily a reference manual. If you are a newcomer to ACL2 and want to learn how to use it effectively, we strongly recommend that you read the companion book [58].

Solutions to all the exercises are available online. Go to the ACL2 home page, click on the link to this book and follow the directions there. The directions also explain how to obtain the ACL2 scripts for each case study.

You will note that on the Web page for this book there is a link named "Errata." As the name suggests, there you will find corrections to the printed version of the book. But more importantly, you may find differences between the version of ACL2 described in the printed book (Version 2.5) and whatever version is current when you go to the home page. The ideas discussed here are fundamental. But we do display such syntactic entities as command names, session logs, etc. These may change. Therefore, look at the online Errata when you first begin to use ACL2 in conjunction with this book.

We believe it is appropriate to use this book in graduate and upper-division undergraduate courses on Software Engineering or Formal Methods. It could be used in conjunction with other books in courses on Hardware Design, Discrete Mathematics, or Theory (especially courses stressing formalism, rigor, or mechanized support). It is also appropriate for courses on Artificial Intelligence or Automated Reasoning.

Part I

Preliminaries

2

Overview

When people talk about "theorems" and "proofs" most of us either think of the elementary results of high school geometry, *e.g.*, "If two distinct lines intersect, then they intersect in exactly one point," or famous unsolved problems, such as Goldbach's question, "Is there an even number greater than 2 that is not the sum of two primes?".

But consider the following theorems of a different sort.

- The hardware implementing floating point multiplication produces answers in accordance with the IEEE floating-point standard.
- The desired instruction set architecture is implemented by the pipelined architecture.
- The program identifies the five highest peaks in the data stream and stores the addresses of those peaks.
- The compilation algorithm produces object code that preserves the semantics of the source program.
- The processor is "secure."

These statements are informal, but they can be made formal. And once made formal, such statements can sometimes be shown to be true: they can be proved as theorems of the mathematical logic in which they are formalized. We know, because each of the statements above has been formalized and proved, using the formal logic used in this book. Indeed, the proofs were checked mechanically by the system used in this book. Furthermore, the computing systems studied were not "toys." Each system and theorem was of interest to an industrial sponsor or designer.

2.1 Some Questions

You probably have many questions, ranging from the philosophical to the practical.

- How can anyone make mathematically precise statements about physical artifacts like computers?

- What does it mean to say that such a statement is true?
- How does the computer program described in this book work? What does it do?
- Can a computer program really help a person reason?
- Who can learn to use it?
- How long does it take to learn?
- How automatic is it? How long does it take to prove a statement like those above?
- How realistic are the problems to which it has been applied?

Most of this book is devoted to the last question. It is answered by showing you the case studies and their solutions. But we will briefly address the other questions now.

2.2 Some Answers

You cannot prove theorems about physical artifacts. Theorems are proved about mathematical models of physical artifacts. More precisely, mathematical formulas can be written to describe the behavior of artifacts. We call those formulas "models" of the artifacts. Part of an engineer's job training is to learn how to create mathematical models of structures. These models address some concerns and ignore others: the shape and composition of the supporting beams are carefully modeled, but the texture and color of the paint might be ignored. These models are then used to answer questions about the behavior of the artifact before it is actually constructed.

The same basic ideas can be applied to the design of computing artifacts. But how would you describe, say, a microcode engine or a compiler? You certainly would not write differential equations. You might write the description as a computer program. That is, you might construct a program that produces the desired behavior in response to given input. Such programs are called "simulators" in hardware design and are sometimes called "prototypes" in software design. They are models of the artifact (and they often ignore important aspects, like power consumption or user interface). But generally such models are used to *test* the artifact, by executing the code for the model on concrete examples to see if it behaves as intended.

Of course, bugs could also be found by inspection of the model itself. A clever programmer might stare at the code for the model and realize that a certain input causes "unintended" behavior. What is this programmer doing? She is not executing the model. She is reasoning abstractly—symbolically—about the model. Now imagine that we could offer her some mechanical assistance in the symbolic manipulation of the model.

To offer mechanical assistance, we need to "program" the model in a language that is unambiguous, a language that is simple enough to reason about but rich enough to model a lot of computing systems. It helps if the language is executable, so we can continue to use testing as a way to evaluate and analyze our models. What language shall we use?

The answer in this book is a functional programming language based on Common Lisp. The language is called ACL2. To the newcomer, it is a variant of Lisp, without side-effects. Models of interesting systems can be coded up in this language, compiled with off-the-shelf Common Lisp compilers, and made to execute fairly efficiently on a variety of platforms. This book is full of examples, but Chapter 8 discusses microprocessor models and simulation efficiency at length.

Now suppose we want to reason about a model. For example, suppose we want to determine whether its output has a certain relationship to its input. We do this by defining the relation as another Lisp program and composing it with the model. The question we then want to answer is whether this composite expression always return the answer `t`. This is akin to annotating a conventional model with a test to determine whether the computed answer is correct.

To convince ourselves that the answer is always `t` (or, often, to discover why it is not), we might symbolically expand the model, "running" it on indeterminate data and imagining the possible execution paths. This might be called "symbolic simulation" but in a mathematical setting it is just *simplification* of symbolic expressions.

We handle loops (or recursions) in the model by thinking about "what the loop is doing" in general. That is, we posit some property that is true whenever we arrive at the top of the loop (not just the first time) and we try to show that it is true the next time we arrive at the top of the loop. If we can do that, and if the property is true the first time, then it is always true. This kind of reasoning is familiar to all programmers and is, of course, just *mathematical induction*.

To aid the "clever programmer" in the inspection and analysis of a model, we have a mechanical theorem prover. The theorem prover provides, among many other features, a powerful symbolic simplifier and a sophisticated mechanism for inventing inductive arguments.

The system is *rule driven* in the sense that its behavior is affected by rules in a database. These rules mainly tell it how to simplify expressions but also control many other aspects of the system. The rules are "programmed" into the database by the user, after considering the model in question and the kinds of problems it presents. But the ACL2 user cannot add just any rule. If that were so, the logical correctness of the system's arguments would depend on the infallibility of its human user. Instead, every rule in the database is derived from a mathematical theorem which must first be proved by ACL2. So the user writes conjectures, which, if proved by the system, are turned into rules by the system, which, in turn, determine how the system behaves in the future. The user's job is entirely strategic. Blunders by the human might prevent the system from finding a proof, but they will not make the system assent to something that does not follow logically.[1]

This design empowers the user to think creativity about how to manipulate the concepts. Along the way, the user codifies strategies that the system will apply to prove theorems in the particular application domain.

2.3 Anecdotal Evidence from Two Projects

The ACL2 system was designed by Boyer, Moore, and Kaufmann in response to the problems faced by users of the Boyer-Moore theorem prover, Nqthm [7], in applying that system to large-scale proof projects [59]. Those projects included the proof of Gödel's incompleteness theorem [102], the verification of the gate-level description of the FM9001 microprocessor [53], the KIT operating system [2], the CLI stack [3] (which consists of some verified applications written in a high-level language, a verified compiler for that language, a verified assembler/loader targeting the FM9001), and the Berkeley C string library (as compiled by `gcc` for the Motorola MC68020) [9]. For a more complete summary of Nqthm's applications, see [7]. Such projects set the standards against which we measure ACL2.

How hard is it to use ACL2 to prove theorems of commercial interest? What is involved? Here are very brief descriptions of two major ACL2 projects, with emphasis on what had to be done, who did it, and how long it took.

2.3.1 The Motorola CAP DSP

The Motorola Complex Arithmetic Processor (CAP) is a single-chip DSP (digital signal processing) co-processor optimized for communications signal processing. The CAP project started in 1992 at the Motorola Government and Systems Technology Group, Scottsdale, Arizona [38]. The project lasted several years.

The CAP is an interestingly complex microprocessor. Aspects of its design include separate program and data memories, 252 programmer-visible registers, 6 independently addressable data and parameter memories with the data memories logically partitioned (under program control) into source and destination memories, and an ALU with 4 multiplier-accumulators and a 6-adder array. The instruction word is 64 bits, which is decoded into a 317-bit low-level control word within the ALU. The instruction set includes

[1] In order to provide greater flexibility for the proof process, ACL2 allows the user to explicitly add axioms and temporarily skip proofs.

<u>**Task**</u>	<u>**Man-Months**</u>
Microarchitecture model	15
Sequential model and equivalence proof	5
Reusable books	6
Microcode proofs	2
Meetings and reports	3

Figure 2.1: CAP Tasks Breakdown

no-overhead looping constructs. As many as 10 different registers can be involved in the determination of the next program counter and a single instruction can simultaneously modify over 100 registers. The 3-stage instruction pipeline contains hazards visible to the programmer.

The motivation behind this complexity and unusual design was to allow Motorola engineers to code DSP application programs in CAP microcode and have those programs execute extremely efficiently.

One ACL2 user (Bishop Brock) was assigned the job of providing formal methods support for the design effort. Brock spent 31 months on the project, the first seven of which were in Scottsdale interacting with the design team. The rest of Brock's time was in Austin, Texas. During the project Brock described the machine at two levels: the microarchitecture level, which includes the pipeline, and the microcode programmer's level, which is a simpler sequential machine. Anecdotal evidence suggests that Brock contributed to the design in minor ways merely by recording design decisions formally and commenting on their implications.

Using ACL2, Brock proved that the two views of the machine are equivalent provided the microcode being executed does not expose any hazards. The formalization of when a hazard is present in microcode was an important contribution. Brock defined a function that recognizes whether a piece of CAP microcode is hazard-free. The equivalence theorem he proved shows that his function is adequate. Because the function is executable, Brock certified DSP application programs to be hazard-free merely by executing the function on the microcode. About 50 programs were so certified. Hazards were found and eliminated. Brock also proved several application programs correct with respect to the sequential model. See [12, 13]. In the course of the work, Brock developed several ACL2 books—collections of theorems encoding useful theorem proving strategies in certain domains—which are independent of the CAP project and have found use in other ACL2 projects. Among the books he produced is the extremely useful integer hardware specification (`ihs`) library.

A breakdown of Brock's tasks and the time taken on each is provided in Figure 2.1. Because the design was under constant evolution during the period, the formal models were also under constant evolution and "the" equivalence theorem was proved many times. This highlights the advantage of developing general proof strategies embodied in books. It also highlights the utility of having a good inference engine: minor changes in the theorem being proved do not necessarily disrupt the proof replay.

It generally takes much longer for the ACL2 user to develop the models, theorems, and libraries than it does for the theorem prover to check successful proof scripts. The CAP proofs can be reproduced in about one hour on a 200 MHz Sun Microsystems Ultra 2 with 256 MB of memory.

2.3.2 Division on the AMD-K5

A considerably simpler ACL2 project was the modeling and correctness proof for the floating point division microcode on the AMD-K5[2] microprocessor. This was carried out by ACL2 authors Moore and Kaufmann together with AMD designer Tom Lynch.

The divide microcode is less than one page long. Its informal analysis by the AMD designers [66] was about 10 pages long and relied on common knowledge among floating-point designers, as well as on some subtle original arguments. Peer review was limited and time was short. Informal proofs of similar length for other algorithms had previously been found incorrect during testing of the "proved" part. Hence, confidence in the analysis was not commensurate with the risks involved and AMD decided to have the proof mechanically checked with ACL2.

Ten weeks elapsed from the time the project started to the time the final theorem was mechanically proved (June–August, 1995). At the beginning of that period Lynch explained the microcode to Moore. With help from Warren Hunt, Moore came to a partial understanding of the microcode and began to formalize its semantics and the specification. This involved formalizing many floating-point concepts. Until that time, ACL2 had never been used to prove anything about floating-point arithmetic. Approximately 80% of the project's time was devoted to the formal development of concepts and relationships that are common knowledge in the floating-point design community.

About one month into the project, Moore enlisted the aid of Kaufmann. Moore and Kaufmann divided the work of formalization and proof between them, "contracting" with each other to prove certain lemmas. Lynch was involved in the formal phase of the proof whenever the informal arguments "broke down" or were unclear. The key lemma (Theorem 1 of [79]) was mechanically checked approximately one month after Kaufmann joined the project. Moore then worked alone two more weeks to complete the proof.

The theorem has since been changed several times, most recently in response to the reviewers of [79]. The "social process of mathematics" was at work here on an accurate formal modeling of the algorithm and its desired properties. Each time the theorem was changed, Moore used ACL2 to prove

[2]AMD, the AMD logo and combinations thereof, AMD Athlon, and AMD-K5 are trademarks of Advanced Micro Devices, Inc.

the modified conjecture (when correct), working from the revised proof script. The new proofs were most often constructed automatically because the changes were slight (*e.g.*, narrowing by 1 the bounds of a representable exponent).

Approximately 130 functions were defined. Forty-seven of them are specific to the algorithm and its proof. The others are general-purpose floating-point concepts. Approximately 1,200 lemmas were mechanically proved. Sixty percent of them are of general interest. The other 40% are specific to the analysis of the particular algorithm. It takes approximately one hour (on a 200 MHz Sun Microsystems Ultra 2 with 256 MB of memory) to replay the entire sequence of definitions and theorems starting from scratch.

2.4 Sociology

Industrial ACL2 proof projects usually involve several people, although often only one person interacts with the theorem prover. Several people are involved because it is rare to find a single person who has the requisite skills. Some members of the group must completely understand the application in question: what informal property is supposed to be true and why. We will call these people the *implementors*. In addition, the group must include one or more people who know how to

- use the ACL2 logic to formalize informally described concepts,
- do pencil-and-paper proofs of ACL2 formulas, and
- drive the ACL2 theorem prover.

We will call these people the *formalizers*.

Often, the formalizers do not have an intuitive grasp of the problem at the beginning of the project. This is because they are often a late addition to a pre-existing team of implementors who have been studying the practicality of the proposed "product."

Usually the first step in a project is for the implementors to explain to the formalizers what the product does and how it does it. The implementors might not give clues to why they expect the product's design to be correct; and if they do, their notion of "correctness" may not seem anything like a property that can be nicely formalized. Indeed, the notion of an explicit, abstract statement of correctness is foreign to many implementors. From their perspective, correctness is best described as the absence of "bugs" that are obvious to the end-user. "Bugs" may be detected "by inspection" but are more likely to be exposed through testing.

It is not uncommon for the crucial first meeting to go badly. The different attitudes of the two sides are obvious. The implementors are anxious to construct something that "works" and are determined to "sell" the project. They feel that their reputations, if not their jobs, are on the line. Furthermore, they may have never presented the details of the project to anyone outside the management structure. They almost certainly have never tried to explain the internal workings to outsiders, much less to outsiders unable to speak the in-house language and not possessing the common knowledge of the field. To make matters worse, the implementors may be defensive: they feel that the formalizers are out to find fault in their design and may exaggerate the importance of their discoveries.

Meanwhile, the formalizers are asking for the impossible and the implementors are not able to deliver it. The formalizers want an utterly precise specification of the component in isolation, understandable to the non-expert. They too feel that their reputations are on the line. They know they must get results soon, because of the project deadlines and because of skepticism within the company of the worth of formal methods. But the whole project seems vague, poorly understood, and rushed. And they too are defensive; they have never tried anything quite like this project and do not know how vast a body of mathematics they have to formalize just to get to where the implementors start in their informal argument.

Consider the following a worst-case scenario. Both sides may leave this crucial first meeting feeling frustrated and alarmed. The "team" is full of ignorant people! When either side tries to speak the language of the other, they either reinforce the impression of ignorance or inadvertently mislead! Neither side seems to value the skills or objectives of the other. The mutual defensiveness exacerbates the problem. The implementors hide their main worries about the design and try to bluff their way around them. The formalizers hide their main worries about the inadequacy of their tools and try to blame the problem on poorly understood goals.

As you might well imagine, this would be a difficult beginning! Crucial to the success of the project is good communication and mutual respect between these two "sides." Indeed, the polarization into two sides is destructive. The implementors should regard the formalizers as friends: the bugs they find will be fixed before they get out. The formalizers should see the implementors as friends: their designs are state-of-the-art solutions to novel problems; bugs are inevitable but the implementors often have exceedingly well-honed intuitions about how to solve their problems. The skill-sets of the two sides are complementary and should be appreciated by all. The implementors have good intuitions—it is not random chance that enables them to produce complex designs that almost always work—but they do not have the formalization skills to prove their designs correct. The formalizers are masters at casting intuitive ideas into logic, but do not have the engineering experience to navigate through the tradeoffs of behavior, performance, cost, and time-to-market. Each side has to trust that the other will fill the gaps. Each side has to be honest with the other. The implementors must explain why they mistrust their design. The formalizers must confess that they do not understand the goal or the methods, that the extant formal work in the field is miniscule, and that the expressive power of whatever formalism they are using limits their ability to capture what is desired. All team members should agree that the goal is to produce a better understood product.

Often the main contributions of the formalizers are to foster communication and reduce ambiguity. As the team coalesces, the formalizers become lexicographers: they record the precise meanings of terms used by the team and look out for misunderstandings between team members. Bugs are often found by trying to make precise the statement of what the product is supposed to do and why the implementation works.

Once the basic ideas have been formalized and agreed upon, the implementors and formalizers can design clear interfaces between the modules and get on with their main tasks, of producing implementations and proofs. Often feedback between the tasks is very helpful. You know the team has come together when the implementors ask the formalizers whether it is permitted to assume that a certain module has a given property, or when the formalizers ask the implementors for help in proving some key property.

2.5 Toy Models

The formalizers will often be struggling with several issues at once: understanding the informal descriptions of the product, discovering and formalizing the relevant knowledge that the implementors take for granted, and formalizing the design and its specification. To a large extent this phenomenon is caused by the fact that formal methods is only now being injected into industry. Once a significant portion of a group's past projects has been formalized, along with the then-common knowledge, it will be much easier to keep up. But at this moment in history, keeping up during the earliest phases of a project can be quite stressful.

When working on a new project, we recommend that the formalizers start by formalizing the simplest imaginable model, *e.g.*, the instruction set with one data operation and a branch instruction, or the protocol with a simple handshake. Choosing this initial model requires some experience. The trick is to choose a model that exhibits the "new" problems—problems the formalizers have never dealt with—that will be encountered when the model is elaborated to the interesting case. Such toy models are extremely useful for developing the form of the model, the statement of its key properties, tool support (*e.g.*, simulation support), and the structure of the proofs. Since iteration is often required to get things to fit together properly, it is crucial that this initial foray into the unknown be done with a small enough model to permit complete understanding and rapid, radical revision.

The most common insight gained from this process is that intermediate abstractions are being used implicitly. Informal language is so flexible we often develop new models without realizing it. "Imagine that we did runtime error checking" calls into being an intermediate model in which certain conditions are checked at runtime and explicitly signaled. Often, subsequent arguments are couched in terms of the original model but in fact are being conducted about this new model. Without careful scrutiny, one may not be aware that two models are being used by the team and that the two models have important relationships, *e.g.*, that they produce identical outcomes when one of them satisfies some additional constraint *that cannot even be expressed about the other.*

Once such insights are made, they not only dramatically influence the formalization of the actual models but they influence the language used by the design team to discuss the evolving implementation.

A toy model is also a good sandbox in which the implementors and formalizers can learn to communicate and can come to a clear agreement as to what is to be modeled, formalized, and proved.

Finally, it is crucial that the project management understand the importance of these initial toy models. The premature introduction of realistic complexity into the model can delay the completion of the project. A well-chosen toy can provide a road map to a correct design, implementation, and proof.

2.6 Requirements on the User

The "typical" ACL2 user has a bachelor's degree in computer science or mathematics. We expect that reading [58] and working the exercises there will be sufficient training to prepare a newcomer for the exercises here.

How long does it take for a novice to become an effective ACL2 user?

Let us first answer a different question.[3] How long does it take for a novice to become an effective C programmer? (Substitute for "C" your favorite programming language.) It takes weeks or months to learn the language but months or years to become a good programmer. The long learning curve is not due to the complexity of the programming language but to the complexity of the whole enterprise of programming. Shallow issues, like syntax and basic data structures, are easy to learn and allow you to write useful programs. Deep skills—like system decomposition, proper design of the interfaces between modules, and recognizing when to let efficiency impact clarity or vice-versa—take much longer to master. Once deep skills are learned, they carry over almost intact to other languages and other projects. Learning to be a good programmer need not require using a computer to run your programs. The deep skills can be learned from disciplined reflection and analysis. But writing your programs in an

[3]The following three paragraphs are taken verbatim from our discussion of this issue in [58] because they answer the question so appropriately in the current context.

implemented language and running them is rewarding, it often highlights details or even methodological errors that might not have been noticed otherwise, and, mainly, it gives you the opportunity to practice.

We hope that you find the above comments about programming non-controversial because analogous comments can be made about learning to use ACL2 (or any other mechanized proof system).

How long does it take for a novice to become an effective ACL2 user? It takes weeks or months to learn to use the language and theorem prover, but months or years to become really good at it. The long learning curve is not due to the complexity of ACL2—the logic or the system—but to the complexity of the whole enterprise of formal mathematical proof. Shallow issues, like syntax and how to give hints to the theorem prover, are easy to learn and allow you carry out interesting proof projects. But deep skills—like the decomposition of a problem into lemmas, how to define concepts to make proofs easier, and when to strive for generality and when not to—take much longer to master. These skills, once learned, carry over to other proof systems and other projects. You can learn these deep skills without doing mechanical proofs at all—indeed, you may feel that you have learned these skills from your mathematical training. Your appraisal of your skills may be correct. But writing your theorems in a truly formal language and checking your proofs mechanically is rewarding, it often points out details and even methodological errors that you might not have noticed otherwise, and, mainly, it gives you the opportunity to practice.

3

Summaries of the Case Studies

There are fourteen case studies, organized as follows. The first three, written individually by the editors, are especially appropriate for beginners, but contain useful information for all readers. The next six chapters are related to the formalization, specification, and verification of computer hardware. The next two deal explicitly with computer software applications. The last three focus on problems in logic and mathematics. We say "explicitly" above because all the applications can be seen as illustrative of software verification: since the logic is in essence Lisp, the models being verified are in essence just software systems.

As noted in the Introduction, each case study is supported by material on the Web, including full solutions to the exercises and all definitions and theorems discussed. See page 4.

- Chapter 5, An Exercise in Graph Theory, by J Moore. This chapter formalizes the notion of a directed graph and shows how to prove the correctness of a depth-first path finding algorithm. The chapter requires no specialized knowledge of graph theory and is meant entirely as an exercise in formalization and use of ACL2.

- Chapter 6, Modular Proof: The Fundamental Theorem of Calculus, by Matt Kaufmann. This chapter presents a modular, top-down ACL2 proof methodology and then uses the methodology to outline a formalization and proof of the Fundamental Theorem of Calculus. While the example is based on the non-standard extension of ACL2 described by Gamboa in Chapter 18, non-standard analysis is not a prerequisite either for this chapter or for the proof methodology presented.

- Chapter 7, Mu-Calculus Model-Checking, by Panagiotis Manolios. The Mu-Calculus is a formal logic into which many temporal logics, including CTL, CTL^*, and LTL, can be translated. This chapter presents a formal development of the syntax and semantics for the Mu-Calculus, a model-checker for the Mu-Calculus in ACL2, and a discussion of the translation of other temporal logics into the Mu-Calculus. There are several self-contained sections in which the reader is presented with exercises whose solutions lead to books on set theory, fixpoint theory, and relation theory. These books will be of interest even to readers not interested in the Mu-Calculus.

- Chapter 8, High-Speed, Analyzable Simulators, by David Greve, Matthew Wilding, and David Hardin. High-speed simulation models are routinely developed during the design of complex hardware systems in order to predict performance, detect design flaws, and allow hardware/software co-design. Writing such an executable model in ACL2 brings the additional benefit of formal analysis; however, much care is required to construct an ACL2 model that is both fast and analyzable. In this chapter, techniques are described for the construction of high-speed formally analyzable simulators in ACL2. Their utility is demonstrated on a simple processor model.

- Chapter 9, Verification of a Simple Pipelined Machine Model, by Jun Sawada. An ACL2 model of a three-stage pipelined machine is defined, along with a model of the corresponding sequential machine. Then a proof of the equivalence between the two machines is presented. More importantly, the method of decomposing the proof applies to much more complicated pipelined architectures.

- Chapter 10, The DE Language, by Warren Hunt. The DE language is an occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language. The syntax and semantics of the language are formalized and the formalization is used to prove the correctness of a simple hardware circuit. Such formal HDLs have been used to prove properties of much more complicated designs.

- Chapter 11, Using Macros to Mimic VHDL, by Dominique Borrione, Philippe Georgelin, and Vanderlei Rodrigues. The purpose of this project was to formalize a small synthesizable behavioral subset of VHDL, preserving as much as possible the syntactic flavor of VHDL and facilitating verification by symbolic simulation and theorem proving.

- Chapter 12, Symbolic Trajectory Evaluation, by Damir Jamsek. Symbolic Trajectory Evaluation (STE) is a form of model checking fundamentally based on symbolic simulation. This chapter presents a formal treatment of STE, including ACL2 proofs of results presented in the Seger and Joyce paper [101].

- Chapter 13, RTL: A Verified Floating-Point Multiplier, by David M. Russinoff and Arthur Flatau. This chapter describes a mechanical proof system for designs represented in the RTL language of Advanced Micro Devices. The system consists of a translator to the ACL2 logical programming language and a methodology for verifying properties of the resulting programs using the ACL2 prover. The correctness of a simple floating-point multiplier is proved.

- Chapter 14, Design Verification of a Safety-Critical Embedded Verifier, by Piergiorgio Bertoli and Paolo Traverso. This case study shows the use of ACL2 for the design verification of a piece of safety-critical software, the Embedded Verifier. The Embedded Verifier checks on-line that each execution of a safety-critical translator is correct. The translator is a component of a software system used by Union Switch & Signal to build trainborne control systems.

- Chapter 15, Compiler Verification Revisited, by Wolfgang Goerigk. This study illustrates a fact observed by Ken Thompson [106] in his Turing Award Lecture: the machine code of a correct compiler can be altered to contain a Trojan Horse so that the compiler passes almost every test, including the so-called bootstrap test in which it compiles its own source code with identical results, and still be capable of generating "bad" code. The compiler, the object code machine, and the experiments are formalized in ACL2.

- Chapter 16, Ivy: A Proof Checker for First-order Logic, by William McCune and Olga Shumsky. In this case study, a proof checker for first-order logic is proved sound for finite interpretations. More generally, the study shows how non-ACL2 programs can be combined with ACL2 functions in such a way that useful properties can be proved about the composite programs. Nothing is proved about the non-ACL2 programs. Instead, the results of the non-ACL2 programs are checked at run time by ACL2 functions, and properties of these checker functions are proved.

- Chapter 17, Knuth's Generalization of McCarthy's 91 Function, by John Cowles. This project deals with a challenge by Donald Knuth [63] for a "proof by computer" of a theorem about his generalization of John McCarthy's famous "91 function." The generalization involves *real* numbers, and the case study uses ACL2 to meet Knuth's challenge by mechanically verifying results not only about the field of all real numbers, but also about every subfield of that field.

- Chapter 18, Continuity and Differentiability, by Ruben Gamboa. This chapter shows how an extended version of ACL2 can be used to reason about the real and complex numbers, using non-standard analysis. It describes some modifications to ACL2 that introduce the irrational real and complex numbers into ACL2's number system. It then shows how the modified ACL2 can prove classic theorems of analysis, such as the intermediate-value and mean-value theorems.

We close this chapter with a brief sketch of an interesting case study not included in this book.

ACL2 was used by Vernon Austel and Sean Smith at IBM Research in the formal analysis of the bootstrapping code for the IBM 4758 secure coprocessor.[1] Roughly speaking, a secure coprocessor is a small computer inside a container that should prevent an attacker from reading or modifying its data using unauthorized means (for example, by opening it up and directly reading the contents of memory using a probe); the device should detect such attempts and take defensive action, such as erasing cryptographic keys. In the case of the 4758, the container is about the size of a thick book and the device has successfully withstood all physical attacks mounted against it as of this writing. The U.S. government established Federal Information Processing Standard 140-1 (or FIPS 140-1) to impose requirements on cryptographic devices for use in government work.[2] FIPS 140-1 defines four levels of effectiveness concerning software and hardware, level four being the highest. In order for cryptographic hardware to achieve level four, it must withstand any physical attack; in order for software to achieve level four, it must be formally modeled.

The 4758 has been evaluated by an independent commercial evaluator according to the FIPS 140-1 criteria and has achieved level four in both hardware and software. A detailed description of the security-critical software in a state machine notation was required for the software evaluation, together with a careful description of the properties the state machine has that collectively justify calling the device "secure" for the purpose to which it is put. Translating this state machine into ACL2 was straightforward; translating the four properties that define the notion of "secure" was sometimes not straightforward.

The state machine (and hence the ACL2 model) is fairly low-level, in that all data structures in the software being modeled are represented in the state machine (albeit abstractly), and one state transition in the model corresponds to roughly ten lines of C code; however, no formal connection between the code and the state machine was established. The ACL2 code for the model is about 15,000 lines long, including comments, and required approximately three person months to develop.

This case study is not further discussed in this book because the ACL2 code implementing it is proprietary. Indeed, several of the case studies presented here are distillations of larger proprietary projects.

We believe it is good for prospective users of any tool or methodology to know about the existence of proprietary applications. Often the main obstacle to trying out a new technology is deciding whether it might be applicable. One aim of this book is to be a guide for those who wish to learn ACL2. As such, we felt it necessary to focus on reproducible results and case studies that can be disclosed in full detail.

---

[1] Information concerning the IBM 4758 may be obtained at `http://www.ibm.com/-security/cryptocards`.

[2] The NIST Web page concerning FIPS 140-1 is `http://csrc.nist.gov/cryptval/-#140-1`.

We believe that when you read the case studies you too will be convinced that formality is truly practical. The practical requirement is to learn how to be truly formal.

4

ACL2 Essentials

We present here a brief, and very informal, introduction to ACL2. Our purpose is to provide just enough ACL2 background to support reading the ACL2 formulas displayed in this book's case studies. The reader interested in learning more about ACL2 is invited to take a look at the companion volume, [58], and to visit the ACL2 home page (see page 4).

ACL2 is both a logic and a programming language. As a programming language it is closely related to the Common Lisp programming language [104, 21]. In fact, ACL2 is intended to be consistent with Common Lisp where the two languages overlap. We do not assume that the reader is familiar with Common Lisp, but point out that most of these ACL2 essentials apply to Common Lisp as well.

Data types are presented in Section 4.1 and expressions are presented in Section 4.2. Readers already familiar with Lisp can probably skip most of these two sections. In Section 4.3 we discuss definitions, how to state properties (theorems) about defined notions, and how to submit definitions, theorems, and other *events* during an ACL2 session.

4.1 Data Types

The universe of ACL2 objects consists of several data types.

Numbers include the integers, rationals, and complex rational numbers.

Strings such as `"abcd"` are sequences of *characters*.

Symbols such as `ABC` and `A-TYPICAL-SYMBOL` may be viewed as structures containing two fields, each of which is a string: a *package name* and a *symbol name*. The package name is beyond the scope of this introduction, except to say that it is usually implicit with one major exception: symbols printed with a leading colon (:) have a package name of `"KEYWORD"`. For example, `:HINTS` is a symbol with package name `"KEYWORD"` and symbol name `"HINTS"`. In fact ACL2 is case-insensitive, at least for our purposes here, except for strings. So for example, `:hints` and `:HINTS` are the same symbol, both with symbol name `"HINTS"`. We generally use lower-case in this book except when displaying ACL2 output.

Objects of the above types are called *atoms*. ACL2 also contains ordered pairs called *conses*. Binary trees are represented in ACL2 as conses whose two components may be either conses (*i.e.*, binary trees) or atoms. These binary trees thus have atoms at their leaves.

In the remainder of this section, we discuss a few important data structures that can be constructed from the small set of data types described above.

A *true list* (often referred to simply as a *list*) is either the special atom `nil`, which represents the empty list and is sometimes written `()`, or else a cons whose second component is a (true) list. Lists are represented by enclosing their elements in parentheses. For example, `(3 2 4)` is a list with the following *elements*, or *members*: the numbers 3, 2, and 4. More literally, it is a cons whose first component, called its *car*, is 3 and whose second component, called its *cdr*, is the list `(2 4)`.[1] The list `(2 4)` in turn has a car of 2 and a cdr of `(4)`, which in turn is the list whose car is `4` and whose cdr is `()`, the symbol `nil`. Lists can of course be nested. The list `(A 5/6 (3 2 4) "B")` has elements `A` (a symbol), `5/6` (a rational), `(3 2 4)` (a list), and `"B"` (a string).

ACL2 uses two symbols to represent the Boolean values *true* and *false*: `t` and `nil`, respectively. Notice that this is the same `nil` that is used to represent the empty list. Such overloading is not generally confusing. It comes from four decades of Lisp tradition and is analogous to the treatment of 0 as a Boolean in the C programming language.

A very common data structure is the *association list*, or *alist* for short, which is a true list of cons pairs. An alist represents a mapping, so that (roughly speaking) when a pair is an element of an alist, then the car of that pair is associated with the cdr of that pair. We say more about alists on page 32.

4.2 Expressions

ACL2 expressions (or *terms*) evaluate to the data objects described in the preceding section. Expressions are, in turn, represented by certain of the data objects. There are essentially four kinds of expressions (see <u>`term`</u> for details[2]):

- the symbols `t`, `nil`, and those whose package name is `"KEYWORD"`, which evaluate to themselves;
- all other symbols, which take their values from the environment;
- `(quote` x`)`, also written `'`x, whose value is the ACL2 object x;

[1] The names "car" and "cdr" come from Lisp.

[2] Recall our convention of underlining topics discussed in ACL2's online documentation. See page 4.

- `(`f x_1 x_2 ... x_n`)` where $n \geq 0$, f is a symbol[3] denoting an n-ary function, and each x_i is a term, whose value is the result of applying that function to the values of the x_i.

Figure 4.1 should clarify the above notion of *value*. It also serves to introduce some important primitive (built-in) functions, many of which are used frequently in the case studies. (Many others are not listed here, but we expect their meanings to be reasonably clear from context, and the ACL2 documentation can resolve ambiguity when necessary.) In addition, they introduce the notation for same-line comments: text from a semicolon (;) to the end of the line is treated by ACL2 (and Common Lisp) as a comment. The function symbols of Figure 4.1, including `<`, are all underlined to remind you of the online documentation.

We will see a few more built-in functions in Section 4.3. Note also that `(car (cdr x))` may be written as `(cadr x)`, that `(cdr (cdr x))` may be written as `(cddr x)`, and so on. Some prefer to use `nth` for zero-based access to elements of a list; so `(nth 0 x)` has the same value as `(car x)`, `(nth 1 x)` as `(cadr x)`, `(nth 2 x)` as `(caddr x)`, and `(nth 3 x)` as `(cadddr x)`.

The expression language is slightly complicated by *macros*. The following built-in macros are most easily understood as though they were operators of indeterminate arity. Let v_i be value of `x`*i* below.

`(+ x1 x2 ... xn)`	*; the sum of* $v_1, v_2, \ldots, v_n$
`(* x1 x2 ... xn)`	*; the product of* $v_1, v_2, \ldots, v_n$
`(list x1 x2 ... xn)`	*; the list* $(v_1, v_2, \ldots, v_n)$
`(and x1 x2 ... xn)`	*;* v_n *if each* v_i *is not* `nil`*, else* `nil`
`(or x1 x2 ... xn)`	*;* v_i *for the least* i *such that* v_i
	; is not `nil`*, if any; else,* `nil`

Two more built-in macros extend the function `if` defined in Figure 4.1. The macro `cond` is applied to 2-element lists each of the form (*test form*) and returns the value of the first *form* for which the *test* has a value other than `nil`.

```
(cond (test1 form1)
      (test2 form2)
      ...
      (testn-1 formn-1)
      (t formn))
 =
(if test1 form1
    (if test2 form2
    ...
      (if testn-1 formn-1
          formn) ... ))
```

[3] f can also be something called a *lambda expression*, but it is safe to ignore this point for purposes of the case studies.

Term	**Value**
`3`	*; The number 3*
`-3/4`	*; The number -3/4*
`x`	*; Depends on the environment*
`(car x)`	*; If the value of* `x` *is a cons,* *; its first component, else* `nil`
`(cdr x)`	*; If the value of* `x` *is a cons,* *; its second component, else* `nil`
`(consp x)`	*;* `T` *if the value of* `x` *is a cons, else* `nil`
`(acl2-numberp x)`	*;* `T` *if the value of* `x` *is a number, else* `nil`
`(integerp x)`	*;* `T` *if the value of* `x` *is an integer, else* `nil`
`(rationalp x)`	*;* `T` *if the value of* `x` *is a rational number,* *; else* `nil`
`(zp x)`	*;* `T` *if the value of* `x` *is 0 or is not a* *; natural number, else* `nil`
`(nfix x)`	*; The value of* `x` *if it is a natural* *; number, else 0*
`(<= x y)`	*;* `T` *if the value of* `x` *is less than the* *; value of* `y`*, else* `nil`
`(1- x)`	*; One less than the value of* `x`
`(1+ x)`	*; One more than the value of* `x`
`(equal x y)`	*;* `T` *if the values of* `x` *and* `y` *are the* *; same, else* `nil`
`(iff x y)`	*;* `T` *if the values of* `x` *and* `y` *are either both* *;* `nil` *or both non-*`nil`*, else* `nil`
`(if x y z)`	*; The value of* `z` *if the value of* `x` *is* `nil`*,* *; else the value of* `y`
`(implies x y)`	*;* `T` *if the value of* `x` *is* `nil` *or the* *; value of* `y` *is not* `nil`*, else* `nil`
`(not x)`	*;* `T` *if the value of* `x` *is* `nil`*, else* `nil`

Figure 4.1: Some Terms and Built-in Function Symbols

Related macros case and case-match are used occasionally. Their meanings may be clear from context; if not, we recommend consulting the ACL2 documentation. We say a bit more about macros in the next section.

Finally, we introduce the constructs let and let*. Each of these forms takes two "arguments": a list of *bindings* (2-element lists) followed by a form to be evaluated relative to those bindings. Consider first this example.

```
(let ((x (+ 5 3))
      (y (- 5 3)))
  (* x y))
```

How is this form evaluated? First, x and y are *bound* in parallel to the values of their respective forms, *i.e.*, to 8 and 2, respectively. Then (* x y) is evaluated in the resulting environment by multiplying 8 by 2 to get the final value of 16. The construct let* is similar, but evaluates the bindings sequentially. So, the following expression has the same value as the one above.

```
(let* ((a 3)
       (x (+ 5 a))
       (y (- 5 a)))
  (* x y))
```

Let us do a bit of expression evaluation involving lists. Two important built-in list manipulation functions are append, which concatenates two lists, and assoc, which finds the first cons pair having a given car in a given alist. This time we illustrate evaluation using the ACL2 *read-eval-print loop*. First we illustrate append. User input is on lines following the prompt, "ACL2 !>"; the rest is printed by ACL2.

```
ACL2 !>(append '(1 2) '(a b))
(1 2 A B)
ACL2 !>
```

Before turning to assoc, we introduce so-called *dot notation*. The cons pair with car a and cdr b is sometimes written (a . b). The following examples illustrate this notation. Notice that (x_1 ... x_n . nil) represents the same value as (x_1 ... x_n).

```
ACL2 !>(cons 3 4)
(3 . 4)
ACL2 !>(cons 'a 'b)
(A . B)
ACL2 !>(cons 3 nil)
(3)
ACL2 !>'(3 . nil) ; same as (quote (3 . nil))
(3)
ACL2 !>'(3)
(3)
```

```
ACL2 !>(cons 3 (cons 4 5))
(3 4 . 5)
ACL2 !>(cons 3 (cons 4 nil))
(3 4)
ACL2 !>(list (cons 'a 2) (cons 'b 4) (cons 'b 6))
((A . 2) (B . 4) (B . 6))
ACL2 !>
```

Now we are ready to give examples illustrating assoc. We use the alist shown just above. Notice that `assoc` returns either a cons or `nil`.

```
ACL2 !>(assoc 'b '((a . 2) (b . 4) (b . 6)))
(B . 4)
ACL2 !>(assoc 'e '((a . 2) (b . 4) (b . 6)))
NIL
ACL2 !>
```

For a more complete discussion of expressions see the ACL2 documentation for term. The description above should generally be sufficient for reading the case studies. Most of the built-in functions used in this book are either mentioned above or in the next section.

4.3 Events

The ACL2 user strives to define functions and to lead ACL2 to proofs of theorems about the built-in and defined functions. Function definitions and theorems are the most important types of events. In this section we give the syntax of these events and a couple of others.

4.3.1 Function Definitions

The basic form for defining a function is defun. Most of the built-in functions are defined (in the ACL2 source code) using defun. For example, the built-in function not is defined as follows.

```
(defun not (p)
  (if p nil t))
```

This definition says: "Define function `not`, with formal parameter list `(p)`, so that the application of `not` to such a list is equal to `(if p nil t)`."

Definitions may be used to build up hierarchies of concepts. For example, the built-in function `consp` that recognizes cons data objects may be used, together with the function `not` defined above, to define the following function atom, which recognizes non-conses, *i.e.*, atoms.

```
(defun atom (x)
  (not (consp x)))
```

We can take this a step further. In many cases we want a function that distinguishes non-empty lists from `nil`. Although `atom` does this, Common Lisp provides another function `endp` that may be slightly more efficient than `atom`. ACL2 builds in the following definition for `endp`.[4]

```
(defun endp (x)
  (declare (xargs :guard (or (consp x) (equal x nil))))
  (atom x))
```

The list above that starts with `declare`, inserted between the formal parameter list `(x)` and the body `(atom x)`, is called a *declare form*, which unlike the body, is not a function application. It is generally harmless for the casual reader to ignore declare forms, and it is beyond the scope of this introduction to discuss them in much detail. The subform `(xargs ... )` is used to provide ACL2-specific information. In this case, ACL2 is being informed that the formal parameter `x` is intended to have a "type" specified by the indicated term, *i.e.*, `x` should either be a cons or `nil`. Through this mechanism we can allow ACL2 to use the underlying Common Lisp execution engine in a direct manner to achieve greater execution speed; see `guard` for details.

In the same spirit as with `endp`, ACL2 has definitions for fast equality tests: `eq` can be used when at least one argument is a symbol and `eql` can be used when at least one argument is a number, symbol, or character. *Logically*, however, `endp` is the same function as `atom`, and all of `equal`, `eq`, and `eql` are the same function. For example, it is a theorem that `(endp lst)` is equal to `(atom lst)`. Some built-in functions use `eql` and have built-in analogues that use `equal` and `eq`, for example: `assoc`, `assoc-equal`, and `assoc-eq`. As with the {`eql,equal,eq`} family, these functions are all semantically the same.

ACL2 supports recursive definition, *i.e.*, definition where the body of the function mentions the function being defined. Here is a definition of the built-in function `member`, which checks for membership in a list. Notice that when `(member e x)` is not equal to `nil`, it is the first tail of list `x` that starts with `e`. We omit the guard below, which allows the use of `eql` below.

```
(defun member (e x)
  (cond ((endp x) nil)
        ((eql e (car x)) x)
        (t (member e (cdr x)))))
```

Recall that `cond` picks out the form corresponding to the first test that is true (non-`nil`). So for example, we can see that `(member 3 '(4 5))` is equal to `(member 3 '(5))` using this definition: Suppose the value of `e` is 3 and the value of `x` is the value of `'(4 5)`, *i.e.*, the list `(4 5)`. Then the value of `(endp x)` is `nil`, so we move past the first pair in the `cond`. Next,

[4] Certain details not germane to the discussion are omitted from some of these definitions.

the value of `(car x)` is 4 since `car` takes the first element of a list; so, the value of `(eql e (car x))` is `nil` since 3 does not equal 4. Hence the second test in the `cond` does not apply. Since `t` is not `nil`, the final pair in the `cond` does apply, showing that `(member 3 '(4 5))` has the same value as `(member 3 '(5))`. Similar reasoning shows that `(member 3 '(5))` has the same value as `(member 3 '())`, which in turn has the value `nil`. This chain of reasoning suggests how to compute with recursive definitions, in this case computing the value of `(member 3 '(4 5))` to be `nil`.

It is possible to introduce inconsistency with recursive definitions, for example by defining $f(x)$ to equal $1 + f(x)$ (from which we obtain $0 = 1$ by arithmetic). ACL2's definitional principle (see defun) avoids this problem by requiring, in essence, a proof of termination. Consider for example the definition of `member`, above. The recursive call of `member` is made only under the conditions that the first two tests of the `cond` are false (`nil`), *i.e.*, `x` is not an atom and `e` is not the first element of `x`. The first of these two conditions guarantees that `(cdr x)` is a "smaller" list than `x`.

In the above example ACL2 uses a built-in *measure* (see defun) on the size of `x`. However, there are cases where one needs to help ACL2 by specifying a particular measure. For example, one can add the following `declare` form just before the body of `member`, above, to specify the length `(len x)` of the list `x` as the measure.

```
(declare (xargs :measure (len x)))
```

4.3.2 Theorems

We have seen how to define functions using `defun`. The analogous construct for submitting theorems to ACL2 is defthm. Here is an example, a theorem with name `member-append`, which says that `a` is an element of the concatenation of (lists) `x` and `y` if and only if `a` is an element of at least one of them.

```
(defthm member-append
  (iff (member a (append x y))
       (or (member a x) (member a y))))
```

ACL2 can prove the above theorem without any help. More complex theorems generally require user assistance. Probably the most common form of help is to prove lemmas to be used automatically in subsequent proofs. In fact it is easy to imagine that one would prove the lemma above in order to cause the theorem prover, in proofs of subsequent theorems, to simplify terms of the form `(member a (append x y))` to corresponding terms `(or (member a x) (member a y))`. Such simplification is an example of *rewriting*, a further description of which is beyond the scope of these Essentials. But sometimes one needs to give explicit hints for a proof, or to give explicit directions for how to store a theorem, which is otherwise stored by default as a rewrite rule (see rewrite). Here is an example called `equal-char-code`, from ACL2 source file `axioms.lisp`, which states that two characters are equal if they have the same char-code. The `:rule-classes` value `nil` says that this theorem is not to be stored as a rule, and the `:hints` direct ACL2 to use two explicit instances of a previously-proved theorem called `code-char-char-code-is-identity`.

```
(defthm equal-char-code
  (implies (and (characterp x)
                (characterp y))
           (implies (equal (char-code x) (char-code y))
                    (equal x y)))
  :rule-classes nil
  :hints (("Goal" :use
           ((:instance
             code-char-char-code-is-identity
             (c x))
            (:instance
             code-char-char-code-is-identity
             (c y))))))
```

In the case studies, the reader is often spared the trouble of looking at these details by way of "...". An example follows.

```
(defthm equal-char-code
  (implies (and (characterp x)
                (characterp y))
           (implies (equal (char-code x) (char-code y))
                    (equal x y)))
  :rule-classes nil
  :hints ...)
```

4.3.3 Macros and Backquote

Users can define macros to extend the language. The symbol cadr is defined as a macro in the ACL2 source code. Here is its definition.

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

The body of the macro constructs an expression that is used in place of calls to `cadr`.

This macro could be written using "backquote" notation.

```
(defmacro cadr (x)
  `(car (cdr ,x)))
```

Backquote allows the exhibition of what might be called "near-constants." It is like the normal single quote (') except that any expression after a comma is evaluated and the value used as the next element. If the comma is immediately followed by @, the value of the expression is spliced in.

Macros can be defined to take a varying number of arguments and can use arbitrary processing to construct the new expression. Here is the definition of `list`.

```
(defmacro list (&rest args)
  (list-macro args))
```

The symbol `list-macro` is just a defined function.

```
(defun list-macro (lst)
  (if (consp lst)
      (cons 'cons
            (cons (car lst)
                  (cons (list-macro (cdr lst)) nil)))
      nil))
```

Thus, `(list a b c)` expands to `(cons a (cons b (cons c nil)))`.

4.3.4 Single-Threaded Objects

ACL2 provides *single-threaded objects*; they are sometimes called "stobjs" (pronounced "stob-jays"). Logically, such objects are just lists containing several components. Logically, they are "changed" in the usual way, by constructing new lists of the appropriate shape from the parts of the old one. Stobjs are introduced with `defstobj`, together with functions for accessing and "changing" their components.

But syntactic restrictions insure that once a stobj is modified, it is impossible for any function to obtain the "old" list structure. The ACL2 system takes advantage of this fact, by modifying stobjs destructively while preserving the applicative semantics. See `stobj` or [8].

4.3.5 Structuring Mechanisms

Files composed of events are called `books`. Ultimately, the goal of the ACL2 user is to get ACL2 to accept all the events in a book, discharging all proof obligations. Once a book is thus *certified* (see `certify-book`), its events may be included into an ACL2 session using `include-book`.

Some events in a book may be wrapped with `local`, as in the following.

```
(local
  (defthm my-lemma
    ...))
```

Local events are proved when the book is certified by ACL2, but they are omitted by `include-book`.

ACL2 provides another structuring mechanism, `encapsulate`. This mechanism provides both a scoping mechanism and a way to introduce functions that are constrained by axioms, rather than defined. (When there are no constraints on a function then the shorthand `defstub` may be used.) Here is an example that constrains a function `fn` of one argument to return a cons.

```
(encapsulate
  ;; signatures:
 ((fn (x) t))
  ;; local definition:
 (local (defun fn (x) (cons x x)))
  ;; exported theorem:
 (defthm consp-fn
   (consp (fn x))))
```

4.4 Concluding Remarks

The discussion in this chapter has been very informal and highly incomplete. It was designed to provide sufficient information to enable the reader to make sense of the case studies that follow. The companion volume, [58], gives a more thorough and precise introduction geared towards potential ACL2 users. We also strongly encourage anyone interested in using ACL2 to visit the ACL2 documentation and tours on the ACL2 home page, `http://-www.cs.utexas.edu/users/moore/acl2`.

Part II

Case Studies

5

An Exercise in Graph Theory

J Strother Moore
Department of Computer Sciences, University of Texas at Austin, Texas
Email: moore@cs.utexas.edu

Abstract

We define a function that finds a path between two given nodes of a given directed graph, if such a path exists. We prove the function terminates and we prove that it is correct. Aside from illustrating one way to formalize directed graphs in ACL2, this chapter illustrates the importance of the user's decomposition of a problem into mathematically tractable parts and the importance of defining new concepts to formalize those parts. Our proof involves such auxiliary notions as that of a simple (loop-free) path, the process for obtaining a simple path from an arbitrary path, and an algorithm for collecting all simple paths. The algorithm we analyze is a naive one that executes in time exponential in the number of edges. This chapter requires no specialized knowledge of graph theory; indeed, the main thrusts of the chapter have nothing to do with graph theory. They are: to develop your formalization skills and to refine your expectations of what a formalization will involve. Appropriate expectations of a project are often the key to success.

Introduction

Consider the obvious depth-first algorithm for looking for a path from `a` to `b` in a given directed graph, `g`: if `a` is `b`, we are done; otherwise, consider each neighbor of `a` in turn and try to find a path from it to `b`. If such a path is found, then extend it via the edge from `a` to the neighbor. Care must be taken to avoid loops in the graph; a simple solution is never to consider any path that visits the same node twice. In addition, some convention must be made to indicate whether the search succeeded or failed (and, in the former case, to indicate the path found). We will define `(find-path a b g)` in accordance with the sketch above.

The correctness of `find-path` may be informally stated as "if there is a path, `p`, from node `a` to node `b` in graph `g`, then `(find-path a b g)` finds such a path." This may be formalized as shown below.

```
(implies (and (graphp g)
              (nodep a g)
              (nodep b g)
              (path-from-to p a b g))
         (path-from-to (find-path a b g) a b g))
```

Now think about the theorem above. How does the existence of path `p` insure that `find-path` will succeed? In particular, `find-path` will not necessarily find `p`! Indeed, `p` may be a "non-simple" path (*i.e.*, one with loops). Here is the informal correctness argument.

> **Informal Proof Sketch:** It is fairly obvious from the definition of `find-path` that it returns a path from `a` to `b` *unless* it signals failure. So the problem is to show that `find-path` does not signal failure. Now given a path `p` from `a` to `b`, we can obtain a simple path, `p'`, from `a` to `b`. Furthermore, `p'` is a member of the set, `S`, of all simple paths from `a` to `b`, showing that `S` is non-empty. But `find-path` signals failure only if `S` is empty. □

ACL2 certainly cannot discover this proof! However, it can be led to it. To formalize this argument we must formalize more than just the concepts used to state the goal. Some concepts, *e.g.*, that of "simple path," are introduced during the proof. Many relationships between these concepts are left implicit. The key observations made in the sketch are not themselves proved. All of this may leave the false impression that the theorem has been proved and that the proof is quite short.

Learning how to formalize ideas that are informally understood is crucial to using ACL2. An under-appreciated but essential skill, long known to mathematicians, is recognizing when new concepts are needed, especially if those concepts are not identified in the informal arguments. Being alert to situations that may call for new lemmas of certain forms makes it easier to apply The Method (see the companion book [58] or see <u>`the-method`</u>), because you know what to look for as you inspect failed proofs.

And now for some psychological remarks: developing your expectations of the amount of work involved can make the difference between success and failure. Your experience of using The Method will be much more positive if you correctly anticipate that a step will require certain kinds of lemmas, even if you do not know what form they will take. The failure of the system to complete the step validates your intuitions and provides you with the information you need. In this situation, the theorem prover is often regarded as a partner or a useful tool in the creation of the proof. On the other hand, if you are unrealistically expecting the system to complete the step for you, its failure disappoints you. Repeated disappointment leads to disillusionment; the theorem prover becomes an adversary. It is easy in this situation to abandon the project as unworkable.

In this chapter we formalize and prove the theorem above. But much of the chapter is aimed at helping you develop your expectations.

For the record, it took me about 8 hours to develop the script described here, from my first contemplation of the problem. About 20 definitions and 70 theorems are involved. It takes ACL2 less than 30 seconds to process the script. I have worked about two weeks writing up the proof, not to make it more elegant but to explain the original intuitions and the process.

The supporting material contains four books of interest. The book `find-path1` contains my original script, developed more or less by following The Method. That script contains many "general purpose" lemmas about list processing. So next I segregated these into a book named `helpers`, and I created the book `find-path2`, which is concerned only with the graph-theory part of this problem. But `find-path2` is still linearly structured, reflecting basically the post-order traversal of my originally imagined proof tree. So in `find-path3` I defined the macro named `top-down`.

```
(defmacro top-down (main &rest others)
  `(progn ,@others ,main))
```

Thus, (`top-down` *main* $e_1 \ldots e_n$) expands into a command that first processes the e_i and then processes *main*. I use `top-down` to structure the proof as it is described here. Kaufmann's article, Chapter 6, presents much more flexible proof-structuring devices. After reading this chapter you are encouraged to look at the three `find-path`*i* books to see how proof scripts can be made more perspicuous. If you wish to use The Method to do this entire chapter as an exercise, look to `find-path1` for my solutions.

5.1 Getting Started

We use two books to make this proof simpler to describe.

```
(include-book "/projects/acl2/v2-5/books/arithmetic/top")
(include-book "helpers")
```

The supporting book `helpers` contains a definition of `rev`, the list reverse function defined in terms of `append` rather than the tail recursive `reverse` native to ACL2. In addition, it contains nineteen theorems about `rev` and/or several native ACL2 functions including `append`, `member` (which checks membership in a list), `subsetp` (which checks the subset relation between two lists), `no-duplicatesp` (which checks that a list has no duplicate elements), and `last` (which returns the last `consp` cdr of a list). We use these results largely without noting them below.

Exercise 5.1 *Is this a theorem? Why not? Prove the theorem it suggests.*

```
(defthm member-append
  (equal (member x (append a b))
         (or (member x a)
             (member x b))))
```

Exercise 5.2 *Given the definition of* `rev`*, prove* `car-last-rev` *below.*

```
(defun rev (x)
  (if (endp x)
      nil
    (append (rev (cdr x)) (list (car x)))))
(defthm car-last-rev
  (equal (car (last (rev x)))
         (car x)))
```

Exercise 5.3 *How would you rewrite*

```
(no-duplicatesp (append a (cons e b)))
```

to simplify it? Prove the theorem.

5.2 The Primitives of Directed Graphs

The most basic notion we must define is that of a "directed graph." In this work, we use the word "graph" always to mean directed graph. We represent a graph as an alist.

Formally then a graph is a list of pairs; each pair consists of a key and a value. Each key is the name of a node of the graph. The value associated with a key is the list of neighbor nodes immediately accessible from the given node. For sanity, we insist that each node name appear only once as a key in the alist, and that the list of neighbors of a node contain no duplications. These restrictions play no part in the proof and could be dropped. Note the use of `top-down` to structure the presentation.

```
(top-down
 (defun graphp (g)
   (and (alistp g)
        (no-duplicatesp (all-nodes g))
        (graph1p g (all-nodes g))))
 ; where
 (defun all-nodes (g)
   (cond ((endp g) nil)
         (t (cons (car (car g))
                  (all-nodes (cdr g))))))
 ; and
```

```
 (defun graph1p (g nodes)
   (cond ((endp g) t)
         (t (and (consp (car g))
                 (true-listp (cdr (car g)))
                 (subsetp (cdr (car g)) nodes)
                 (no-duplicatesp (cdr (car g)))
                 (graph1p (cdr g) nodes))))))
```

The functions for recognizing nodes in a graph and for computing the neighbors of a node are defined below.

```
(defun nodep (x g)
  (member x (all-nodes g)))
(defun neighbors (node g)
  (cond ((endp g) nil)
        ((equal node (car (car g))) (cdr (car g)))
        (t (neighbors node (cdr g)))))
```

The formal definition of a path in a graph `g` is given by `pathp`. A path is a non-empty list with the property that each element except the first is a neighbor of the preceding element.

```
(defun pathp (p g)
  (cond ((endp p) nil)
        ((endp (cdr p))
         (equal (cdr p) nil))
        (t (and (member (cadr p)
                        (neighbors (car p) g))
                (pathp (cdr p) g)))))
```

Having defined a path, it is convenient to define the notion of `path-from--to`, which checks that `p` is a path in `g`, with initial element `a` and final element `b`.

```
(defun path-from-to (p a b g)
  (and (pathp p g)
       (equal (car p) a)
       (equal (car (last p)) b)))
```

We test and illustrate these concepts by observing the theorem named `Example1` below. The theorem is proved by evaluation! But by making it a theorem and including it in a certified book we can help document how our functions behave and provide tests should we wish to change them in the future. Figure 5.1 shows a picture of the graph `g` used in `Example1`.

```
(defthm Example1
  (let ((g '((A B)
             (B B C)
             (C A C D)
             (D A B C))))
```

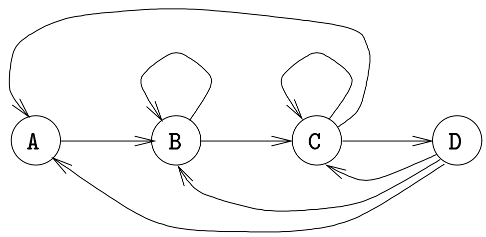

Figure 5.1: The graph in Example1

```
    (and (graphp g)
         (not (graphp (cdr g)))
         (nodep 'A g)
         (not (nodep 'E g))
         (pathp '(A B C D C A B B) g)
         (path-from-to '(A B B C) 'A 'C g)
         (not (pathp '(A B D) g))))
  :rule-classes nil)
```

5.3 The Specification of Find-Path

We desire a function `find-path` with the following property.

```
(defthm Spec-of-Find-Path
  (implies (and (graphp g)
                (nodep a g)
                (nodep b g)
                (path-from-to p a b g))
           (path-from-to (find-path a b g) a b g)))
```

After we define `find-path` we will actually prove a stronger theorem about it.

```
(defthm Main
  (implies (path-from-to p a b g)
           (path-from-to (find-path a b g) a b g)))
```

`Spec-of-Find-Path` follows easily from `Main`: the first three hypotheses of `Spec-of-Find-Path` are irrelevant!

This is an aspect of ACL2's typeless language. `(Find-path a b g)` is defined to return something, whether `g` is a graph or not and whether `a` and `b` are nodes in it or not. `Main` establishes that if `(path-from-to p a b g)` is true then so is `(path-from-to (find-path a b g) a b g)`, regardless of the "types" of the variables.

Many new users struggle with an apparent dilemma: which of these two theorems is to be preferred? The answer is: both! The two theorems have different purposes. The weaker theorem, `Spec-of-Find-Path`, is desirable as a specification of a not-yet-defined function: it makes it easier for the implementor because it does not overconstrain the function on irrelevant inputs. The stronger theorem, `Main`, is desirable as a theorem about an already-defined function: if a defined function actually has such an elegant property, use it! In particular, the stronger theorem is easier to use (*e.g.*, as a rewrite rule), because it has fewer hypotheses. In addition, such strong theorems are often easier to prove by induction, though that consideration is irrelevant here because we will not prove `Main` by induction. Thus, we prove both theorems. The stronger one is for "internal use" (*i.e.*, it will be stored as a rewrite rule) while the weaker one is for "external use" (*i.e.*, it will be stored with `:rule-classes nil`).

Because such theorems as `Main` are easier to use and often easier to prove, it is advantageous to define our functions so as to insure such strong properties. A side-effect is that, quite often, the definitions are simpler than they would be if we included a lot of checks for type correctness.[1] How is it possible to define `find-path` so that it "works" even for non-graphs? Before we answer that, let us reconsider our representation of graphs.

Our representation is non-unique: many different objects can represent the same graph. For example, the lists `'((A A B) (B A B))` and `'((B A B) (A B A))` are different objects that represent the same graph. Our functions are not sensitive to the order in which the keys presented. `Path-from-to` is not sensitive to the order in which the neighbors are presented. `Find-path` will be sensitive to the order of the neighbors only in the sense that the order affects which path it finds, not whether it succeeds or fails.

Therefore, we could choose to strengthen the notion of `graphp` to require that graphs be presented in some canonical order. But even if we strengthened `graphp`, the `path-from-to` and `find-path` defined here would still enjoy the relationship above, even for non-canonical "graphs."

In a similar vein, our functions are not sensitive to other aspects of the concrete representation of graphs. For example, `graphp` requires that every neighbor listed for a node also appear as a key, with some associated list of neighbors. Suppose some "graph" fails to list some neighbor as a key. All our functions behave just as though the neighbor were listed and had the empty list of neighbors. Similarly, `graphp` requires that a graph list every node only once as a key. But if some "graph" has two entries for the same key, all our functions use the first entry and they all ignore the second. Thus, our notion of `graphp` could be weakened without affecting the relationship between `path-from-to` and `find-path`. In fact, it could be weakened to accept any object!

[1]This discussion ignores guards. Our functions would have to be coded somewhat differently if we wished to verify their guards. See guard.

In the sense illustrated above, our functions "coerce" any object into some graph. By insuring that all the functions—in particular both `path-from-to` and `find-path`—coerce non-graphs to graphs in exactly the same way, we can prove theorems like `Main` without making restrictions on the form of graphs. This would not be the case, for example, if `path-from-to` used the first occurrence of a key to determine the neighbors and `find-path` used the last! If we defined the functions that way, we would need to include a `graphp` hypothesis to insure that there is only one such occurrence.

Defining all the functions in a family to coerce "unexpected" inputs in the same way is a useful skill. It is generally done—as here—by defining the primitive functions for a representation—*e.g.*, `neighbors`— and then using them consistently without checking for irrelevant conditions first. This is a good discipline in part because it allows one to state and prove theorems without a lot of cluttering hypotheses.

5.4 The Definitions of Find-Path and Find-Next-Step

In this section we define `find-path`, as follows.

```
(defun find-path (a b g)
  (cond ((equal a b) (list a))
        (t (find-next-step (neighbors a g)
                           (list a)
                           b g))))
```

`Find-path` finds a path from `a` to `b`, if possible. If `a` and `b` are equal, the path is the singleton list containing `a`. Otherwise, `find-path` calls `find-next-step` to search for a path. That function takes four arguments and, in this call, those arguments are the neighbors of `a`, a stack of nodes, the target node `b`, and the graph `g`. The stack is represented as a list and initially the stack has one node on it, namely `a`. In general, the topmost node on the stack can be thought of as the current node and the stack itself (in reverse order) can be thought of as a path from the original source node to the current node. Roughly speaking, `find-next-step` does a depth-first search through the neighbors, looking for the first one from which it can build a path to `b` without visiting any node already on the stack. If and when `find-next-step` finds the target `b` among the neighbors, it builds the appropriate path from the stack. If it fails to find a path, it returns the symbol `'failure`.

Here is the definition of `find-next-step`, ignoring the necessary measure hint.

```
(defun find-next-step (c stack b g)
  (cond
   ((endp c) 'failure)                           ; (1)
   ((member (car c) stack)                       ; (2)
```

```
    (find-next-step (cdr c) stack b g))
   ((equal (car c) b)                          ; (3)
    (rev (cons b stack)))
   (t (let ((temp (find-next-step              ; (4)
                   (neighbors (car c) g)
                   (cons (car c) stack)
                   b g)))
        (if (equal temp 'failure)              ; (4a)
            (find-next-step (cdr c) stack b g) ; (4b)
          temp)))))                            ; (4c)
```

Reading it clause-by-clause: `(1)` If there are no neighbors left to consider, return `'failure`. `(2)` If the next neighbor is a member of the stack, continue to the neighbor after that. `(3)` If the next neighbor is the target, construct a suitable path using the stack. Finally, `(4)` call `find-next-step` recursively, letting the result be called `temp`. In the recursive call, try to find a path to the target through the neighbors of the next neighbor, after pushing that neighbor onto the stack. Then `(4a)` if `temp` is `'failure`, there is no path to `b` through the neighbor just tried and so `(4b)` try the rest of the neighbors, using the input stack. Otherwise, `(4c)` return the path found.

Some readers may have preferred mutually-recursive definitions of functions `find-path` and `find-next-step`. However, mutually-recursive functions are somewhat more awkward to reason about formally than simple recursive functions.

Few programmers would code this algorithm. Its run time is exponential in the number of edges in the graph: we repeat the work done for a node upon each simple arrival at the node, even though a previous arrival at the node via a different path would have fully explored the simple paths from it and found that none reaches the target. By coloring the nodes we can produce a linear time algorithm. It is tempting to improve `find-path` in this way. We resist the temptation and explain why later.

Instead, let us consider the admission of `find-path` and `find-next-step` as defined above. The former is easily admitted after the latter. But how do we admit `find-next-step`? We must exhibit a measure of the arguments of `find-next-step` that gets smaller in each recursive call.

Exercise 5.4 *Before reading on, think about* `find-next-step`. *Why does it terminate?*

The measure we have in mind is a lexicographic combination of two measures. The first is the number of nodes of `g` not yet on the stack. Naively speaking, that number decreases every time we add a new neighbor to the stack. But not every recursive call adds an item to the stack. In particular, as we scan the neighbors of a node, the stack is fixed. Thus, the second component of our lexicographic measure is the number of neighbors left to be explored.

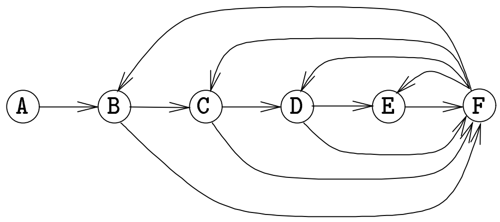

Figure 5.2: The graph in Example2

Exercise 5.5 *Before reading on, formalize what was just said. Define* `(measure c stack g)` *to decrease in each recursion in* `find-next-step`.

Here is our definition.

```
(defun measure (c stack g)
  (cons (+ 1 (count-non-members (all-nodes g) stack))
        (len c)))
```

Exercise 5.6 *Define* `count-non-members`.

Exercise 5.7 *The termination proof is more subtle than you might expect. The recursive call on line* `(4)` *generates the measure conjecture shown below.*

```
(implies (and (not (endp c))
              (not (member (car c) stack))
              (not (equal (car c) b)))
         (e0-ord-< (measure (neighbors (car c) g)
                            (cons (car c) stack)
                            g)
                   (measure c stack g)))
```

ACL2 proves this automatically. But is it really a theorem? The naive explanation sketched above can be elaborated as follows. "In the recursion, the stack grows by one item, `(car c)`*, which previously was not a member of the stack. Thus, the number of unvisited nodes in* `g` *decreases." But this naive argument assumes that the new item is a node in* `g`*. What if it is not? The subtlety of this argument is the price we pay for not checking that* `g` *is a graph and that* `c` *is a list of nodes in it. The reward is a much simpler definition and main theorem. But why does* `measure` *decrease?*

Here is an example of `find-path` on the graph of Figure 5.2.

```
(defthm Example2
  (let ((g '((A B)
             (B C F)
```

```
            (C D F)
            (D E F)
            (E F)
            (F B C D E))))
    (and (graphp g)
         (equal (find-path 'A 'E g) '(A B C D E))
         (path-from-to '(A B C D E) 'A 'E g)
         (path-from-to '(A B F E)   'A 'E g)
         (equal (find-path 'F 'A g) 'failure)))
  :rule-classes nil)
```

The conjuncts inform us that `g` is a graph, `find-path` succeeds from `A` to `E`, the alleged path found is indeed a suitable path, there is a shorter path from `A` to `E`, and that `find-path` fails to find a path from `F` to `A`.

5.5 Sketch of the Proof of Our Main Theorem

Recall the strong conjecture about `find-path`.

```
(defthm Main
  (implies (path-from-to p a b g)
           (path-from-to (find-path a b g) a b g))
  :hints ...)
```

Here we begin to formalize the proof sketched on page 42, by stating the four observations as formulas. In subsequent sections, we define the necessary concepts and prove each of the observations. We will eventually prove `Main` by filling in the hints above to use our observations.

Proof Sketch

If `find-path` does not signal failure, then it actually does return a path from the indicated source, `a`, to the indicated target, `b`.

```
(defthm Observation-0
  (implies (not (equal (find-path a b g) 'failure))
           (path-from-to (find-path a b g) a b g))
  :rule-classes nil)
```

(One might call this "weak correctness" because it does not preclude the possibility that `find-path` fails unnecessarily often.) The proof of `Main` is thus reduced to showing that `find-path` does not fail unnecessarily.

If there is a path `p` from `a` to `b`, then there is a simple path. Rather than use <u>`defun-sk`</u> to introduce an existential quantifier, we will define a function, `simplify-path`, to construct a simple path from an arbitrary path, and use it in the statement of our observation.

```
(defthm Observation-1
  (implies (path-from-to p a b g)
```

```
          (and (simple-pathp (simplify-path p) g)
               (path-from-to (simplify-path p) a b g)))
  :rule-classes nil)
```

Furthermore, if there is a simple path from a to b, then it is a member of the set of all simple paths from a to b, as constructed by the (soon to be defined) function find-all-simple-paths.

```
(defthm Observation-2
  (implies (and (simple-pathp p g)
               (path-from-to p a b g))
          (member p (find-all-simple-paths a b g)))
  :rule-classes nil
  :hints ...)
```

From Observations 1 and 2, together with the hypothesis of our Main theorem, we know that the set of all simple paths from a to b is non-empty.

We therefore conclude the proof by observing that find-path succeeds precisely when the set of all simple paths is non-empty.

```
(defthm Observation-3
  (iff (find-all-simple-paths a b g)
       (not (equal (find-path a b g) 'failure)))
  :rule-classes nil)
```

□

5.6 Observation 0

```
(defthm Observation-0
  (implies (not (equal (find-path a b g) 'failure))
          (path-from-to (find-path a b g) a b g))
  :rule-classes nil)
```

We have to prove that if find-path does not report failure, then it returns a path, the path starts at a, and the path ends at b. Roughly speaking, these theorems follow immediately from three analogous theorems about the function find-next-step, in terms of which find-path is defined. However, to prove things about find-next-step one must use induction, because it is a recursive function. And to prove inductive theorems one must find suitably strong statements of those theorems. So below you will see three inductively provable theorems about find-next-step. The first says it returns a path and the next two say that the source and target of that path are appropriate. ACL2 proves these without assistance. But the real contribution of the user here was to find suitably strong statements of these properties. Basically, when looking for theorems like these, one wants to ask the question: what is the most general situation in which the function can be called? More formally, contemplate a call of the function with distinct variables in every argument position.

We anticipate the lemmas noted above from our experience, not from a detailed a look at the proof of `Observation-0`. In particular, our expectations are that we should seek lemmas about `find-next-step`, that they will be proved inductively, and that they must be general. We find them by applying The Method to `Observation-0`.

The first lemma is that `find-next-step` returns a path in the graph, when it does not report failure. For this to be true, the initial stack must be a non-empty list which, when reversed, is a path in the graph. Furthermore, the list `c` of neighbors to consider for extending that path must be a subset of the neighbors of the top of the stack.

```
(defthm pathp-find-next-step
  (implies (and (true-listp stack)
                (consp stack)
                (pathp (rev stack) g)
                (subsetp c (neighbors (car stack) g))
                (not (equal (find-next-step c stack b g)
                            'failure)))
           (pathp (find-next-step c stack b g) g)))
```

ACL2 proves this inductively, but finding its proof via The Method requires that we find several other lemmas. In our decomposition of the proof, we identified eleven such lemmas. All but one were list processing lemmas now in the support book `helpers`. They were suggested by the occurrence in failed proofs of such terms as `(append (append x y) z)`, `(last (append a b))` and `(subsetp x x)`.

Exercise 5.8 *The theorem above requires one lemma that is not just list processing. It was suggested by a failed proof of* `pathp-find-next-step` *in which the following term arose.*

```
(pathp (append (rev stack)
               (list (car c)))
       g)
```

What is it about `find-next-step` *that led us to expect the need for such a lemma? What is the suggested lemma?*

Exercise 5.9 *As an exercise in using The Method, get ACL2 to prove the theorem above, without all of the necessary list processing helpers in the database. Start by doing*

```
(rebuild "find-path1.lisp"
         'pathp-append)
```

to initialize the database. The second argument to `rebuild` *tells ACL2 to to stop loading the file after processing the event named* `pathp-append`. *Then, prove* `pathp-find-next-step`.

The second lemma states that `find-next-step` returns a list whose first element is the last (bottom-most) element of its initial stack.

```
(defthm car-find-next-step
  (implies (and (true-listp stack)
                (consp stack)
                (not (equal (find-next-step c stack b g)
                            'failure)))
           (equal (car (find-next-step c stack b g))
                  (car (last stack)))))
```

The third is that `find-next-step` returns a list whose last element is the target.

```
(defthm car-last-find-next-step
  (implies (and (true-listp stack)
                (consp stack)
                (not (equal (find-next-step c stack b g)
                            'failure)))
           (equal (car (last (find-next-step c stack b g)))
                  b)))
```

Note that all three of these lemmas are made rewrite rules. These three results give us `Observation-0` directly.

Exercise 5.10 *Starting with the database constructed by the successful completion of Exercise 5.9, prove* `car-find-next-step` *and* `car-last--find-next-step` *above.*

5.7 Observation 1

The next observation constructively establishes that if `p` is a path then there exists a simple path with the same end points.

```
(defun simple-pathp (p g)
  (and (no-duplicatesp p)
       (pathp p g)))
```

Here is the definition of `simplify-path`.

```
(top-down
 (defun simplify-path (p)
   (cond ((endp p) nil)
         ((member (car p) (cdr p))
          (simplify-path (chop (car p) (cdr p))))
         (t (cons (car p) (simplify-path (cdr p))))))
 ; where
 (defun chop (e p)
   (cond ((endp p) nil)
```

```
        ((equal e (car p)) p)
        (t (chop e (cdr p))))))
```

When `simplify-path` finds that some element of the path occurs later in the path, it uses the function `chop` to delete from the path the first occurrence of that element and all subsequent elements up to the second occurrence, thus removing one loop from the path. The admission of `simplify-path` is mildly interesting. ACL2 has to prove that `chop` returns a list that is shorter than `p`. This is an inductive proof, but ACL2 manages it without help.[2]

Since the result of `simplify-path` is obtained by repeatedly applying `chop` to its input under certain conditions, we can expect that every inductive lemma we need about `simplify-path` will require proving an inductive analogue about `chop`. Developing this sense of expectation is important. With it, you can easily sketch a proof. Without it, you may be worn down by the continued need to state unexpected lemmas!

Here is a typical pair of lemmas. The first shown is a fact about `simplify-path` that is used repeatedly in the proof of `Observation-1`. The second is the corresponding fact about `chop`.

```
(top-down
 (defthm simplify-path-iff-consp
   (iff (consp (simplify-path p)) (consp p)))
 ; Proof:  An easy induction, using:
 (defthm chop-iff-consp
   (implies (member e p)
            (consp (chop e p)))))
```

We now turn our attention to `Observation-1`. Its conclusion is a conjunction of two requirements on `(simplify-path p)`. The first is a `simple-path` condition and the second is a `path-from-to` condition. But each of these concepts is defined as a conjunction of others. The `simple-path` condition breaks down to a `pathp` condition and a `no-duplicatesp` condition. The `path-from-to` condition breaks down to the same `pathp` condition, and conditions on both the `car` of the path and the `car` of the `last` of the path. We prove these four conjuncts about `simplify-path` separately. Each requires a corresponding property about `chop`.

With the foregoing as prelude, the details, which were finalized by applying The Method, should be more or less obvious to you. After following The Method to develop the proof, we structured it using `top-down`.

```
(top-down
 (defthm Observation-1
   (implies (path-from-to p a b g)
            (and (simple-pathp (simplify-path p) g)
```

[2]The astute reader might notice that `chop` is equivalent to `member`. We keep the name `chop` for pedagogical reasons.

```
               (path-from-to (simplify-path p) a b g)))
  :rule-classes nil)
;; Proof of Observation-1
(top-down
 (defthm car-simplify-path
   (equal (car (simplify-path p)) (car p)))
 ;; by an easy induction using the analogous
 (defthm car-chop
   (implies (member e p)
            (equal (car (chop e p)) e))))
(top-down
 (defthm car-last-simplify-path
   (equal (car (last (simplify-path p))) (car (last p))))
 ;; by an easy induction using the analogous
 (defthm car-last-chop
   (implies (member e p)
            (equal (car (last (chop e p)))
                   (car (last p))))))
(top-down
 (defthm pathp-simplify-path
   (implies (pathp p g)
            (pathp (simplify-path p) g)))
 ;; by an easy induction using the analogous
 (defthm pathp-chop
   (implies (and (member e p)
                 (pathp p g))
            (pathp (chop e p) g))))
(top-down
 (defthm no-duplicatesp-simplify-path
   (no-duplicatesp (simplify-path p)))
 ;; Proof
 (top-down
  (defthm not-member-simplify-path
    (implies (not (member x p))
             (not (member x (simplify-path p)))))
  ;; by an easy induction using the analogous
  (defthm not-member-chop
    (implies (not (member x p))
             (not (member x (chop e p)))))))
;; Q.E.D. Observation-1
)
```

5.8 Observation 2

The next observation is the key to the whole exercise. We wish to prove that if `p` is a simple path from `a` to `b`, then it is among the paths collected by `find-all-simple-paths`. We must define `find-all-simple--paths` first. The definition is below. The reader will note that this function is analogous to `find-path`, and `find-all-next-steps` is analogous to `find-next-step`. But these new functions do not signal failure when no path is found. Nor do they quit when the first path is found. Instead, they find and collect all simple paths. The measure justifying the admission of `find-all-next-steps` is exactly the same as before.

```
(top-down
 (defun find-all-simple-paths (a b g)
   (if (equal a b)
       (list (list a))
     (find-all-next-steps (neighbors a g)
                          (list a)
                          b g)))
 ; where
 (defun find-all-next-steps (c stack b g)
   (declare (xargs :measure (measure c stack g)))
   (cond
    ((endp c) nil)
    ((member (car c) stack)
     (find-all-next-steps (cdr c) stack b g))
    ((equal (car c) b)
     (cons (rev (cons b stack))
           (find-all-next-steps (cdr c) stack b g)))
    (t (append (find-all-next-steps (neighbors (car c) g)
                                    (cons (car c) stack)
                                    b g)
               (find-all-next-steps (cdr c) stack b g))))))
```

The similarity between these two functions and `find-path` and `find-next--step` will be important when we come to prove Observation 3, the relation between them. It is a good idea when defining auxiliary functions such as these to define them so as to make the proof obligations as easy as possible.

Observation 2 requires that we show that every simple path `p` between `a` and `b` is in the list of paths collected by `(find-all-simple-paths a b g)`.

Before proceeding, let us do a shallow analysis, to refine our expectations and structure our search for a proof. How will the proof go? Clearly, since `find-all-simple-paths` is defined in terms of `find-all-next-steps`, we must prove a property about that function that is analogous to Observation 2. But because `find-all-next-steps` is recursive, the proof will be inductive and the property we seek must be very general. So think about (find-all-next-steps c stack b g). When is something a member of the result?

We may assume that stack is non-empty and contains no duplications. Obviously, every path in the final list will be some extension of the reverse of stack, so the typical element will be of the form (append (rev stack) p). What is a sufficient condition on the extension, p, to insure membership? P must start at some element of c and will end at b and will be a path. Furthermore, p must share no elements with stack.

Below is a formal rendering of this analysis. You will note that there is no mention of b in the formula; instead, the last element of p is used in its place.

```
(defthm Crux
  (implies (and (true-listp stack)
                (consp stack)
                (pathp p g)
                (member (car p) c)
                (no-duplicatesp (append stack p)))
           (member (append (rev stack) p)
                   (find-all-next-steps c stack
                                        (car (last p)) g)))
  :rule-classes nil
  :hints ...)
```

This theorem is named "Crux" because it is the hardest part of the entire proof. Perhaps the single most difficult aspect of Crux was stating it! However the inductive proof is somewhat subtle.

We are trying to prove that a certain object, namely, the one constructed by (append (rev stack) p), is a member of the list returned by find-all-next-steps. Call this "certain object" α. How is α put into the answer returned by find-all-next-steps? Find-all-next-steps explores all the nodes in c and extends the stack appropriately for each one. At the bottom, it reverses the stack and adds that path to its answer; at all other levels it just concatenates the answers together.

Most subtle formal proofs start with an informal idea. Here is what came to me: imagine that find-all-next-steps were miraculously able to extend the stack only by successive elements of p. Then it would obviously build up α in the stack and put it in the answer.

A little more formally, think of tracing the computation of find-all--next-steps and watching it choose just the successive elements of path p from among the neighbors at each successive level. That is, when find--all-next-steps is pursuing the paths from a neighbor that is not the next node in p, we are not too interested in the result— α will be generated by a call further along in the cdr of c and will survive in the answer produced by subsequent concatenations—so we just want an inductive hypothesis about `(cdr c)`. On the other hand, we are quite interested when the neighbor is `(car p)`. The addition of that neighbor, *i.e.*, `(car p)`, to the stack builds up part of α and the inductive hypothesis for that stack and `(cdr p)` tells us that α is in the list returned.

So much for intuitions. The devil is in the details. Below, we present a proof.

Proof Sketch

Denote the `Crux` formula above by `(`ϕ `c stack p g)`. We induct according to the following scheme.

```
(and (implies (endp c)                          ; Base
              (φ c stack p g))
     (implies (and (not (endp c))               ; Ind Step 1
                   (member (car c) stack)
                   (φ (cdr c) stack p g))
              (φ c stack p g))
     (implies (and (not (endp c))               ; Ind Step 2
                   (not (member (car c) stack))
                   (equal (car c) (car p))
                   (φ (neighbors (car c) g)
                      (cons (car c) stack)
                      (cdr p)
                      g))
              (φ c stack p g))
     (implies (and (not (endp c))               ; Ind Step 3
                   (not (member (car c) stack))
                   (not (equal (car c) (car p)))
                   (φ (cdr c) stack p g))
              (φ c stack p g)))
```

The first conjunct is the Base Case. The other three are Induction Steps. The Induction Step 2 is the interesting one.

Base Case

```
(implies
 (endp c)                                          ; (0)
 (implies (and (true-listp stack)                  ; (1)
               (consp stack)                       ; (2)
               (pathp p g)                         ; (3)
               (member (car p) c)                  ; (4)
               (no-duplicatesp (append stack p)))  ; (5)
          (member (append (rev stack) p)           ; (6)
                  (find-all-next-steps c stack
                                       (car (last p))
                                       g))))
```

From `(0)` we see that `c` is empty. But then hypothesis `(4)` is contradicted.

Induction Step 1

```
(implies
 (and
  (not (endp c))                                      ; (0)
  (member (car c) stack)                              ; (1)
  (implies
    (and (true-listp stack)                           ; (2')
         (consp stack)                                ; (3')
         (pathp p g)                                  ; (4')
         (member (car p) (cdr c))                     ; (5')
         (no-duplicatesp (append stack p)))           ; (6')
    (member (append (rev stack) p)                    ; (7')
            (find-all-next-steps (cdr c)              ; (8')
                                 stack                ; (9')
                                 (car (last p))       ; (10')
                                 g))))                ; (11')
 (implies (and (true-listp stack)                     ; (2)
               (consp stack)                          ; (3)
               (pathp p g)                            ; (4)
               (member (car p) c)                     ; (5)
               (no-duplicatesp (append stack p)))     ; (6)
          (member (append (rev stack) p)              ; (7)
                  (find-all-next-steps c              ; (8)
                                       stack          ; (9)
                                       (car (last p)); (10)
                                       g))))          ; (11)
```

Note that lines `(2')`–`(11')` constitute the induction hypothesis. Lines `(2)`–`(11)` constitute the induction conclusion. We get to assume `(0)` and `(1)`, the induction hypothesis `(2')`–`(11')`, and the hypotheses `(2)`–`(6)` of the induction conclusion. We have to prove the conclusion of the conclusion, namely the `member` expression that starts on line `(7)`. We know that the `member` expression that starts on line `(7')` is true provided we can show that the hypotheses `(2')`–`(6')` of the induction hypothesis are true.

By `(0)` and `(1)` and the definition of `find-all-next-steps`, the `find-all-next-steps` expression at `(8)` is equal to the `find-all-next-steps` expression at `(8')`. The proof would be finished if we could relieve the hypotheses `(2')`–`(6')` of the induction hypothesis. Note that `(2')`, `(3')`, `(4')`, and `(6')` are all identical to their counterparts `(2)`, `(3)`, `(4)`, and `(6)`, which are given. We are left to show that `(5)` implies `(5')`, that is, that when `(car p)` is in `c` it is in `(cdr c)`. By the definition of `member`, if `(car p)` is in `c`, then it is either equal to `(car c)` or it is a member of `(cdr c)`. If the latter, we are done. So assume `(car p)` is `(car c)`. Then by `(1)` and equality, `(car p)` is in `stack`. But if `(car p)` is in `stack`, then there are duplications in `(append stack p)`, contradicting `(6)`. Recall Exercise 5.3.

ACL2 does this proof without help, given the theorems in the supporting book `helpers`.

Induction Step 2

```
(implies
 (and
  (not (endp c))                                  ; (0)
  (not (member (car c) stack))                    ; (1)
  (equal (car c) (car p))                         ; (2)
  (implies
    (and (true-listp (cons (car c) stack))        ; (3')
         (consp (cons (car c) stack))             ; (4')
         (pathp (cdr p) g)                        ; (5')
         (member (cadr p) (neighbors (car c) g))  ; (6')
         (no-duplicatesp (append (cons (car c) stack); (7')
                                 (cdr p))))
    (member (append (rev (cons (car c) stack))    ; (8')
                    (cdr p))
            (find-all-next-steps                  ; (9')
               (neighbors (car c) g)
               (cons (car c) stack)
               (car (last (cdr p)))
               g))))
 (implies (and (true-listp stack)                 ; (3)
               (consp stack)                      ; (4)
               (pathp p g)                        ; (5)
               (member (car p) c)                 ; (6)
               (no-duplicatesp (append stack p))) ; (7)
          (member (append (rev stack) p)          ; (8)
                  (find-all-next-steps c          ; (9)
                                       stack
                                       (car (last p))
                                       g))))
```

Elementary list processing lemmas in the book `helpers`, together with `(2)`, tell us that the `append` expression at `(8')` is equal to the `append` expression at `(8)`. By the definition of `find-all-next-steps`, the `find-all-next-steps` at `(9)` is equal to the following.

```
(if (equal (car c) (car (last p)))                ; (9a)
    (cons (rev (cons (car (last p)) stack))       ; (9b)
          (find-all-next-steps (cdr c)
                               stack
                               (car (last p))
                               g))
```

```
  (append (find-all-next-steps (neighbors (car c) g) ; (9c)
                               (cons (car c) stack)
                               (car (last p))
                               g)
          (find-all-next-steps (cdr c)
                               stack
                               (car (last p))
                               g)))
```

If `(9a)` is false, that is `(car c)` is different from `(car (last p))`, `(9)` is `(9c)`, and the induction hypothesis is sufficient to finish, if we can relieve the hypotheses of the induction hypothesis. The only problematic hypothesis is `(6')`, and given `(5)`, which tells us that `p` is a path in `g`, and `(2)`, which says that `(car c)` is `(car p)`, it is pretty easy to see that `(cadr p)` is a member of the neighbors of `(car p)`. But this follows from the `pathp` hypothesis only if `(cadr p)` is an element of `p`. What if `p` is a singleton? If `p` were a singleton, then `(car p)` would be `(car (last p))`, contrary to our assumption that `(9a)` is false.

So suppose `(9a)` is true, *i.e.*, `(car c)` is `(car (last p))`. Then `(9)` is `(9b)`. We must show that the `append` call in `(8)` is a member of `(9b)`. From the fact that `(car c)` is both `(car p)` and `(car (last p))`, and `(7)`, we conclude that `p` has length `1`. That being the case, the first element of `(9b)`, namely `(rev (cons (car (last p)) stack))`, is equal to `(append (rev stack) p)`, which is the `append` term in `(8)`.

Given the `helpers` book, ACL2 needs no additional help with this argument either.

Induction Step 3

```
(implies
 (and
  (not (endp c))                                       ; (0)
  (not (member (car c) stack))                         ; (1)
  (not (equal (car c) (car p)))                        ; (2)
  (implies
    (and (true-listp stack)                            ; (3')
         (consp stack)                                 ; (4')
         (pathp p g)                                   ; (5')
         (member (car p) (cdr c))                      ; (6')
         (no-duplicatesp (append stack p)))            ; (7')
    (member (append (rev stack) p)                     ; (8')
            (find-all-next-steps (cdr c)               ; (9')
                                 stack
                                 (car (last p))
                                 g))))
 (implies (and (true-listp stack)                      ; (3)
               (consp stack)                           ; (4)
```

```
          (pathp p g)                       ; (5)
          (member (car p) c)                ; (6)
          (no-duplicatesp (append stack p)))    ; (7)
     (member (append (rev stack) p)         ; (8)
             (find-all-next-steps c         ; (9)
                                  stack
                                  (car (last p))
                                  g))))
```

Observe that (3)–(5) and (7) imply (3')–(5') and (7'), because corresponding hypotheses are identical. Furthermore, (6) implies (6') because of (2). Thus, we have relieved the hypotheses of the induction hypothesis. That leaves us with proving that the `member` expression at (8') implies the `member` expression at (8). That is handled by the lemma `Crux-cdr`, below. ACL2 cannot discover this lemma by itself.

That completes the inductive proof of `Crux`.

□

As noted earlier, it is good practice, when searching for a proof, to *write down* the theorem you are proving and the inductive argument used. Writing down the induction hypotheses is especially important. As shown above, this can be quite verbose. But with practice you will learn to recognize certain patterns of formula formation and proof obligations, such as the effect of the inductive substitutions and the need to relieve the hypotheses of the induction hypothesis. Once these patterns are well established, you will find that the disciplined use of well-chosen notational conventions will allow you to explore a proof without writing so much. After some practice you will probably find yourself just writing down the theorem, the inductive case analysis and the substitutions, and then "reading off" and jotting down the lemmas you need. But be aware that the mental process is more like the detailed proof above.

The proof patterns manifest themselves in ACL2's output. Our tedious consideration of how to relieve the hypotheses of the induction hypothesis (*e.g.*, , showing that "(5) implies (5')" in Induction Step 1) appears as case analysis in ACL2 output. When ACL2 simplifies

$$
\begin{array}{l}
\texttt{(implies (and } c \\
\qquad\qquad\texttt{(implies (and } \ldots p_5' \ldots \texttt{)} \\
\qquad\qquad\qquad\qquad q'\texttt{))} \\
\qquad\texttt{(implies (and } \ldots p_5 \ldots \texttt{)} \\
\qquad\qquad\quad q\texttt{))}
\end{array}
$$

it will generate such subgoals as `(implies (and` c `(NOT` p_5'`)` $\ldots p_5 \ldots$`)` q`)`, except that the subterms will have been simplified. By recognizing the origin of such subgoals and understanding patterns of proof, you will be able to use ACL2 as an assistant. Rather than write the proof on paper you will let ACL2 do the induction and simplifications and then read its output. That is why The Method works. Expert users of ACL2 rarely turn to paper to work out low-level technical details. For more on this topic, see [60]. But we encourage beginners to develop the necessary skills by sketching proofs on paper first, until the patterns become clear.

Exercise 5.11 *To get ACL2 to do the induction above, it is necessary to supply an induction hint. Such hints are given by exhibiting a function call that suggests the desired induction. Our hint is*

```
(defthm Crux
  ...
  :rule-classes nil
  :hints (("Goal" :induct
                  (induction-hint-function p c stack g))))
```

Thus, `induction-hint-function` *must be defined to suggest the induction done above. See the companion book [58] and* hints. *Define and admit* `induction-hint-function`. *Note: Until we prove* `Crux-cdr`, *ACL2 will be unable to complete the proof of* `Crux`, *even if you code the induction hint correctly.*

We now turn to the proof of the just-mentioned lemma for going from (8') to (8).

```
(defthm Crux-cdr
  (implies
   (and (consp c)
        (member p (find-all-next-steps (cdr c) stack b g)))
   (member p (find-all-next-steps c stack b g)))
  :hints ...)
```

This is a pretty obvious property of `find-all-next-steps` because the set of paths found for `(cdr c)` is a subset of that found for `c`, when `c` is non-empty. However, as stated, it cannot be proved by induction because it is too weak. Instead, we prove the following stronger property.

```
(defthm subsetp-find-all-next-steps
  (implies (subsetp c d)
           (subsetp (find-all-next-steps c stack b g)
                    (find-all-next-steps d stack b g))))
```

Exercise 5.12 *The stronger property, above, is a fundamental and "obvious" property of* `find-all-next-steps`. *Prove it, both by hand and with ACL2. To do the ACL2 proof, first execute the following.*

```
(rebuild "find-path2.lisp"
         'induction-hint-function)
```

Then use The Method.

Exercise 5.13 *The exercise above is made easier by the fact that the necessary list processing lemmas are in the supporting book* `helpers`*, which is included above. But these lemmas were originally discovered by using The Method. So, back up to the initial state, with* `:ubt! 1`*, and then execute the following.*

```
(rebuild "find-path1.lisp"
         'induction-hint-function)
```

Then prove `subsetp-find-all-next-steps` *again, using The Method. Obviously, you should place the lemmas formulated in the previous exercise above* `subsetp-find-all-next-steps` *in your script—unless you wish to "rediscover" them. But to prove them you will need to discover the helper lemmas yourself.*

Given `subsetp-find-all-next-steps`, our proof of `Crux-cdr` is as shown by the hint below.

```
(defthm Crux-cdr
  (implies
   (and (consp c)
        (member p (find-all-next-steps (cdr c) stack b g)))
   (member p (find-all-next-steps c stack b g)))
  :hints
  (("Goal"
    :use (:instance subset-member-member
          (a (find-all-next-steps (cdr c) stack b g))
          (b (find-all-next-steps c stack b g))
          (e p))
    :in-theory (disable subsetp-member-member))))
```

Here is the lemma mentioned in the `:use` hint.

```
(defthm subsetp-member-member
  (implies (and (subsetp a b)
                (member e a))
           (member e b)))
```

Recall how `:use` hints are implemented: the indicated theorem is instantiated and added as a hypothesis. So with the hint above we instruct ACL2 to instantiate `subsetp-member-member` with the substitution [`a` ◁ `(find-all-next-steps (cdr c) stack b g)`; `b` ◁ `(find-all-next--steps c stack b g)`; `e` ◁ `p`]. The instance thus created is added as a hypothesis, producing the goal shown below.

```
(implies
 (implies
  (and (subsetp (find-all-next-steps (cdr c) stack b g)
                (find-all-next-steps c stack b g))
       (member p (find-all-next-steps (cdr c) stack b g)))
```

```
   (member p (find-all-next-steps c stack b g)))
 (implies
   (and (consp c)
        (member p (find-all-next-steps (cdr c) stack b g)))
   (member p (find-all-next-steps c stack b g))))
```

This goal is proved by rewriting the `subsetp` hypothesis to true, by backchaining through `subsetp-find-all-next-steps`. To relieve the hypothesis of that lemma, namely, `(subsetp (cdr c) c)` in this case, ACL2 uses `(consp c)` and the definition of `subsetp`. Note that our hint above also included an `:in-theory` disabling the lemma used. We discuss the reason for this on page 67.

So where are we? We have just finished explaining the proof of `Crux`.

```
(defthm Crux
  (implies (and (true-listp stack)
                (consp stack)
                (pathp p g)
                (member (car p) c)
                (no-duplicatesp (append stack p)))
           (member (append (rev stack) p)
                   (find-all-next-steps c stack
                                        (car (last p)) g)))
  :rule-classes nil
  :hints ...)
```

Observation 2 follows from `Crux`.

```
(defthm Observation-2
  (implies (and (simple-pathp p g)
                (path-from-to p a b g))
           (member p (find-all-simple-paths a b g)))
  :rule-classes nil
  :hints (("Goal"
           :use ((:instance Crux ...)))))
```

Exercise 5.14 *Fill in the dots above with a substitution so that* `Observation-2` *is proved from the resulting instance of* `Crux` *by simplification.*

5.9 Observation 3

```
(defthm Observation-3
    (iff (find-all-simple-paths a b g)
         (not (equal (find-path a b g) 'failure)))
    :rule-classes nil)
```

Exercise 5.15 *Prove* `Observation-3`.

5.10 The Main Theorem

Recall our sketch of the proof of `Main` on page 51. We have now completed the proofs of the four observations, each of which was made `:rule-classes nil`. Hence, in our proof of `Main` we must give explicit hints to use these observations.

```
(defthm Main
  (implies (path-from-to p a b g)
           (path-from-to (find-path a b g) a b g))
  :hints (("Goal"
           :use (Observation-0
                 Observation-1
                 (:instance Observation-2
                            (p (simplify-path p)))
                 Observation-3)
           :in-theory (disable find-path
                               find-all-simple-paths))))
```

Note that we disable two functions. This illustrates a common and bothersome aspect of the implementation of `:use` hints. Once the observations are instantiated and added as hypotheses, the new goal, which might be written as

(implies (and *observation*$_0$
observation$_1$
observation$_2$
observation$_3$)
main)

is simplified. But if the simplifier can prove *observation*$_0$, say, then it is removed from the hypotheses and we are in the same state we would have been had we not provided *observation*$_0$ as a hint. That is why we chose `:rule-classes nil` for our four observations: to keep the stored version of `Observation-`i from rewriting the *observation*$_i$ away. But things are more subtle than that. If `find-path` is expanded, then the simplifier can prove *observation*$_0$ by appealing to the same lemmas we used to prove `Observation-0` in the first place. To prevent that, we disable the two non-recursive function symbols that occur in the observations.

5.11 The Specification of Find-Path

Finally, we wish to prove that `find-path` satisfies its specification.

```
(defthm Spec-of-Find-Path
  (implies (and (graphp g)
                (nodep a g)
                (nodep b g)
                (path-from-to p a b g))
           (path-from-to (find-path a b g) a b g))
  :hints (("Goal" ...)))
```

Exercise 5.16 *Fill in the dots above to make the proof go through. Note that* `Main` *is proved as a rewrite rule.*

Exercise 5.17 *The introduction of* `find-all-simple-paths` *is not strictly necessary. It is possible to prove directly that the existence of a simple path implies that* `find-path` *will not fail. Formalize and prove this lemma.*

5.12 Reflexive Definitions

As noted, our `find-path` runs in exponential time. We can reduce it to linear time by "coloring" or "marking" the nodes as we visit them. We can formalize this by adding a new argument to `find-next-step`, called `mt` (for "mark table"). The table is just a list of all the nodes visited so far, by any recursive call. The new function should fail immediately if it arrives at a marked node. Otherwise, in addition to returning the winning path (or failure signal), it should return an extended mark table. The returned table should contain all the nodes in the old table plus any new nodes visited. The measure used for admission is lexicographic, as before, but the first component is the number of unmarked nodes of the graph (rather than the number of "unstacked" nodes). Linearity is assured by the fact that the edges from a given node are explored at most once.

But the termination argument for this new function is much more subtle. Consider exploring the neighbors, $n_1, n_2, \ldots, n_k$ of some node. Suppose n_1 is not in the mark table, mt. Then the new function will explore the neighbors of n_1, using `(cons` n_1 mt`)` as the mark table. It will obtain a path or failure message and a new mark table, mt'. Suppose no path through n_1 is found. Then the function will recursively consider $n_2, \ldots, n_k$, using the mark table mt' obtained from the first recursive call. Note carefully: the mark table given to the second recursive call is one of the results returned by the first recursive call. The measure depends on properties of the returned table, in particular, that it contain all the entries of the input table. (Contemplate termination if some recursive call "unmarks" a node.) But the measure conjectures must be proved *before* the function is admitted.

Such function definitions are said to be *reflexive*. Here is a sequence of exercises to teach you how to admit reflexive definitions.

Exercise 5.18 *Does the following function terminate?*

```
(defun f (x)
  (if (zp x)
      0
      (+ 1 (f (f (- x 1))))))
```

What are the measure conjectures to be proved? Is there any measure for which the measure conjectures can be proved before `f` *is defined? Can you admit this definition under the ACL2 definitional principle?*

Exercise 5.19 *Consider the following function.*

```
(defun f (x)
  (declare (xargs :measure (m x)))
  (if (zp x)
      0
      (if (e0-ord-< (m (f (- x 1))) (m x))
          (+ 1 (f (f (- x 1))))
          'impossible)))
```

What are the measure conjectures? Can you define `m` *so that this is admissible? Admit the function.*

Exercise 5.20 *Prove that the admitted* `f` *satisfies the originally desired equation.*

```
(defthm f-satisfies-original-equation
  (equal (f x)
         (if (zp x)
             0
           (+ 1 (f (f (- x 1)))))))
```

Hint: Prove that the test on `e0-ord-<` *always succeeds. Note that this exercise does not establish that the original equation defines a function, only that the original equation is satisfiable.*

Exercise 5.21 *Prove that* any *function* `g` *satisfying the original equation is equal to* `f`*. That is, use* `encapsulate` *to constrain* `g` *to satisfy* `(equal (g x) (if (zp x) 0 (+ 1 (g (g (- x 1))))))`*. What witness can you use to show at least one such* `g` *exists? Once you have introduced this constrained* `g`*, prove* `(equal (g x) (f x))`*.*

Exercise 5.22 *Repeat what you have learned from our toy reflexive function* `f` *to admit the linear time* `find-next-step`*. Call the function* `linear-find-next-step`*. Define* `linear-find-path` *appropriately.*

Exercise 5.23 *Prove that* `linear-find-path` *is correct. Hint: Can you prove that it is equal to* `find-path`*? If so, the formal proof of the linear time algorithm exploits 100% of the work done for the "toy" problem.*

The following exercise, due to Matt Wilding, is for advanced users. Our solution to Exercise 5.23 marked nodes already seen by adding them to the mark table list. A node is considered marked if it is a member of this list. While our algorithm can be thought of as modeling a linear-time algorithm, our definition of `linear-find-path` does not execute in linear time because the mark table is searched to determine if a node is marked.

Exercise 5.24 *Use single-threaded objects (see* `stobj`*) to implement an algorithm for* `find-path` *in ACL2 that executes in linear time. Prove that your implementation is "equivalent" to* `linear-find-path`*. (*Hint: *The single-threaded object used in Wilding's solution is given below.*

```
(defstobj st
  (g      :type (array list (100000)) :initially nil)
  (marks  :type (array (integer 0 1) (100000)) :initially 0)
  (stack  :type (satisfies true-listp))
  (status :type (integer 0 1) :initially 0))
```

The new algorithm takes and returns this single-threaded object, which encodes the graph, the mark table, the stack, and the success/failure answer.)

It is tempting to think that reflexive functions arise only in academic settings. This is not true. Consider the termination of a garbage collector, or any program that explores and marks a global state and depends on that marking for its termination. How can you reason about such a program except to tie together its effect and its termination? That is what reflexive functions do and this section shows a method for dealing with them in ACL2.

5.13 Less Elementary Further Work

This chapter is not about graph theory. It is about how to formalize and prove things in ACL2. Indeed, one reason we chose such a naive algorithm is so that the chapter is accessible to anyone interested in this book.

But graph theory is an extremely important and well developed subject. An excellent ACL2 project would be to formalize and prove correct some of the classic algorithms of graph theory.

Here are a few pointers. We do not make these *Exercises* simply because we have not provided solutions.

Formalize and prove the correctness of Dijkstra's shortest path algorithm [25].

The Bellman-Ford algorithm [1, 35] is more general than Dijkstra's because edge weights can be negative. Formalize and prove the correctness of the Bellman-Ford algorithm.

Savitch's Theorem [97] establishes that reachability can be done in $log^2 n$ space, where n is the number of nodes. The algorithm used is similar to our `find-path`, but instead of looking at the neighbors of a node it looks at the "midpoint" between nodes. Formalize the algorithm and prove it correct.

Aside from the correctness of the algorithms, users are often interested in their performance. Dijkstra's algorithm has complexity $O(n^2)$, and the Bellman-Ford algorithm has complexity $O(e \times n)$, where e is the number of edges. Formalize and prove such results. You might start by formalizing the performance of our `find-path`. Just as we have used recursive functions to characterize the answers produced by an algorithm, it is possible to use recursive functions to characterize other aspects of the algorithm, such as the number of comparisons it makes, the number of conses created (as opposed to the length of the answer), or some other measure of the performance. McCarthy [private communication] called these *derived functions* in the 1960's.

Any modern book on computational complexity or algorithms contains a wealth of examples of interesting graph algorithms. Two good references are [84, 22].

5.14 General Lessons

We now return to the particular problem solved here and two general lessons we can draw from it.

The first lesson is the importance of the proper choice of problem. We could have attacked a published algorithm or, at least, the linear `find-path`. We chose a far simpler algorithm. But look at all the work involved! However, in this simple setting we got to explore fundamental representational issues and discover the importance of certain concepts and techniques in this formal setting. Furthermore, with these issues behind us it is relatively easy to contemplate incremental elaborations, such as the introduction of the linear `find-path`.

Many newcomers to formality make the mistake of assuming that results or algorithms they take for granted will be easy to prove. They fail to appreciate how much informal knowledge they are using. They often do not understand the trade-offs between alternatives ways to formalize that knowledge. They have not yet internalized the formal patterns corresponding to intuitive leaps in the subject area. In short, newcomers to formality often forget that they must explore formality itself before trying to apply it.

The discussion of the linear `find-path` provides a succinct example. A major issue with that definition is its reflexive character, an issue that arises explicitly only by virtue of our using a formal principle of definition. Arguing the termination of that function is technically subtle since its recursive equation is not known to define a function until termination has been proved and the termination depends on the value computed by the function. This issue was easily explored with our toy reflexive function `f` in the exercises. The lessons of that toy carry over directly to the linear `find-path`, but would have been far harder to explore in that relatively complicated function.

So the first lesson is: Choose your problems carefully. Resist the temptation to elaborate early! Do not be afraid to start with a ridiculously simple "toy problem" that you think you understand perfectly. Develop a sense of what constitutes a useful "toy" and then trust your intuition that understanding the "toy" will make your actual problem easier to solve. This point is illustrated dramatically in [77].

The other general lesson is the importance of learning how to turn convincing informal arguments into formal ones. One aspect of this is correctly anticipating where a given formalization will lead.

To state the theorem we had to define `graphp`, `nodep`, `path-from-to`, and `find-path`. Only the last two are necessary to state the `Main` property. Now recall the original sketch of the proof.

> **Informal Proof Sketch:** It is fairly obvious from the definition of `find-path` that it returns a path from `a` to `b` *unless* it signals failure. So the problem is to show that `find-path` does not signal failure. Now given a path `p` from `a` to `b`, we can obtain a simple path, `p'`, from `a` to `b`. Furthermore, `p'` is a member of the set, `S`, of all simple paths from `a` to `b`, showing that `S` is non-empty. But `find-path` signals failure only if `S` is empty. □

To formalize this we had to define `simple-pathp`, `simplify-path`, and `find-all-simple-paths`. The first is pretty obviously necessary because the sketch mentions "simple path." The second is only implicit in the sketch, at the mention of "`p'`". The third is perhaps the hardest to see in the sketch: it corresponds to `S`, the set of "all simple paths." That our `find-all-simple-paths` returns the list of *all* simple paths is our `Observation-2`. In the "naive set theory" in which we are often taught to express informal proofs, this concept is not defined algorithmically. The termination of our function implies the finiteness of the "set" constructed.

As this discussion suggests, formalizing an informally given proof requires careful attention to what is being said. Short sentences can apparently lead to large "detours." But they are not detours. They are often the essence of the proof. The function `find-all-simple-paths` is far more useful and general than `find-path` and if our proof had one key idea in it, it is to shift our attention from finding one path to finding them all. Of course, this implies collections of paths and the concomitant ability to reason about them.

Such subtle shifts in attention happen all the time in informal proofs. By noting them, you can save yourself a lot of grief. Many times we have seen users rail at the theorem prover's inability to discover an "obvious" proof when, in fact, the user has utterly failed to formalize the key notions in the proof he or she had in mind.

For example, one might overlook the observation that the existence of an arbitrary path implies the existence of a simple path. That is, one might introduce `simple-pathp` but never introduce `simplify-path`.

Our informal proof says nothing about *how* one obtains a simple path from an arbitrary one (which is to say, it does not give a proof of existence). But on page 55, we finally explain that `simplify-path` iteratively uses `chop` to remove an initial segment from a path. But `chop` is itself an iterative (recursive) process. Hence, we should expect that each inductively proved theorem about `simplify-path` will require an analogous inductively proved theorem about `chop`. This compositionality cuts both ways: it "doubles" the number of lemmas we have to prove, but it "halves" the difficulty of each lemma. Reducing the difficulty is clearly beneficial, so you must learn to anticipate it so you can manage the increase in the number of lemmas.

On page 58 we said "α will be generated by a call further along in the `cdr` of `c` and will survive in the answer produced by subsequent concatenations." This is a clear indication that we will need to prove that `(append a b)` contains all the elements of `a` and all the elements of `b`, *i.e.*, our `member-append`. Some anticipation of the likely lemma development makes it much easier to follow The Method.

In discussing `Crux-cdr` on page 64, we said "This is a pretty obvious property of `find-all-next-steps` because the set of paths found for `(cdr c)` is a subset of that found for `c`, when `c` is non-empty." This sentence foreshadowed two formal developments.

- The explicit identification of `subsetp-member-member`

  ```
  (implies (and (subsetp a b)
                (member e a))
           (member e b))
  ```

  for our hint in `Crux-cdr`. This shifts our attention from a `member` question to a `subsetp` question.

- The articulation of the relevant fundamental and obvious property of `find-all-next-steps`.

  ```
  (defthm subsetp-find-all-next-steps
    (implies (subsetp c d)
             (subsetp
              (find-all-next-steps c stack b g)
              (find-all-next-steps d stack b g))))
  ```

By reading or listening carefully, you can often see the seeds of a formal proof in an informal one. You should strive to develop this skill. It makes it easier for you to fill in the gaps in an informal proof. In addition, it gives you a more realistic appraisal of the length of the journey, which allows you to manage it more comfortably and successfully.

6

Modular Proof: The Fundamental Theorem of Calculus

Matt Kaufmann
Advanced Micro Devices, Inc., Austin, Texas[1]
Email: matt.kaufmann@amd.com

Abstract

This chapter presents a modular, top-down proof methodology for the effective use of ACL2. This methodology is intended both to ease the proof development process and to assist in proof presentation. An application is presented: a formalization and proof of the Fundamental Theorem of Calculus. An unusual characteristic of this application is the use of non-standard analysis, which however is not a prerequisite either for this chapter or for the utility of the proof methodology presented herein.

Introduction

ACL2 users sometimes lose their way in the middle of substantial proof development efforts. Moreover, once the proof is complete, it can be quite difficult to comprehend the overall structure. Such comprehension is important for presenting the proof to others, and is also useful for modifying the proof, either in order to clean it up or in order to prove a related theorem.

This chapter suggests a solution to development and comprehension problems by introducing a simple modular proof development methodology, which we have used to develop a proof of the Fundamental Theorem of Calculus (FTOC). This case study also illustrates how an *outline tool* exploits the resulting proof structure by presenting a top-down view of the ACL2 proof input.

The proof of the FTOC uses a modification of ACL2, developed by Ruben Gamboa [36] to support reasoning about the real numbers using non-standard analysis. However, non-standard analysis is neither necessary for the modular proof methodology nor a prerequisite for reading this chapter. Furthermore, the exercises are designed for standard ACL2.

---

[1] The work described here was performed while the author was at EDS, Inc.

We are aware of an earlier formalization and mechanically-checked proof of the FTOC, performed by John Harrison in his doctoral dissertation [47]. The present mechanized proof may be the first employing non-standard analysis.

The first section below presents this modular proof methodology. The second section shows how this methodology has been applied to prove the Fundamental Theorem of Calculus. We conclude with some observations. Exercises appear below in several places.

6.1 A Modular Proof Methodology

The modular, top-down methodology presented below reflects common proof development practice in the mathematical community. This methodology assists both in proof *development* and in proof *presentation*, as explained in the two subsections below. See also Chapter 5 for a more primitive approach to structuring proofs, using a macro `top-down`.

A makefile provided in the supporting material automates proof replay as well as outline creation.[2]

6.1.1 Proof Development

Many Nqthm and ACL2 users have experienced the following phenomenon. One desires to prove a certain lemma, but requires a lemma in support of that proof, which then leads to another lemma to be proved in support of *that* one, and so on. At some point the effort seems misguided, but by then the evolving proof structure is far from clear and it is difficult to decide how to back up. Even if a decision is made to back up to a particular lemma, is it clear which lemmas already proved may be discarded?

Even if the above process is successful for awhile, it can ultimately be problematic in a frustrating way. Suppose one attempts to prove some goal theorem, and from the failed proof attempt one identifies rewrite rules that appear to be helpful, say, `L1` and `L2`. Suppose further that additional rewrite rules are proved on the way to proving `L1` and `L2`. When one again attempts the original goal theorem, those additional rules can send its proof attempt in a new direction and prevent `L1` and `L2` from being used.

Below we develop a top-down methodology that has the following properties.

- The methodology facilitates organization.
- The methodology can eliminate proof replay problems.

[2] We thank Bishop Brock for providing an earlier makefile that we extended for our purposes.

Introduction: A Typical High-Level Proof Outline

Here is an outline describing many proofs, both mechanically checked ones and others.

1. *Goal*
   To prove theorem THM.

2. *Main Reduction*
   It should suffice to prove lemmas L1, L2, ...

3. *Support*
   Include results from at least one "library" book, lib.

4. *Proof Hacking*
   Prove additional lemmas as needed in order to derive THM.

The outline above may be reflected in the following structure of a top-level book, which (at least initially) can use skip-proofs as shown in order to defer the proofs of the main lemmas.

```
; 3. Support
(include-book "lib")
; 2. Main Reduction
(skip-proofs (defthm L1 ...))
(skip-proofs (defthm L2 ...))
; 4. Proof Hacking
<Miscellaneous lemmas>
; 1. Goal
(defthm THM ...)
```

A common approach to completing this ACL2 book includes the removal of "skip-proofs" by supplying necessary sub-lemmas (and occasionally, definitions) in front of these lemmas. In our modular methodology we instead place the proofs of those lemmas in subsidiary books, using encapsulate in the parent book as follows. Notice that each such sub-book has the same name as the lemma it is intended to prove.

```
(include-book "lib")
(encapsulate ()
   (local (include-book "L1"))
   (defthm L1 ...))
(encapsulate ()
   (local (include-book "L2"))
   (defthm L2 ...))
; Miscellaneous lemmas (for the "proof hacking") go here.
(defthm THM ...)
```

Each sub-book initially contains the corresponding `skip-proofs` form shown earlier above, preceded by all `include-book` forms needed for definitions of function symbols used in that form. If the proof of `THM` requires a change in the statement of, say, `L1`, ACL2 will refuse to accept the `include-book` of `"L1"` unless the statement of `L1` in that book agrees with the one in the `encapsulate` form above, so that ACL2 can recognize the latter `L1` as redundant. In such a circumstance one of course changes the sub-book `L1`, but can probably leave sub-book `L2` unchanged.

Lemma and Library Books

The proposed proof methodology relies on notions of *lemma book* and *library book*. A *lemma book* is an ACL2 book whose last event is a theorem that has the same name as the name of the book. We call this last event the *goal theorem* of the book. (The outline tool described in Section 6.1.2 exploits this naming convention.) For example, in the book skeleton presented just above, books `L1` and `L2` are lemma books provided their last events are `defthm` events named, respectively, `L1` and `L2`. A *library book* is simply any ACL2 book other than a lemma book. Such books typically contain either lemmas of general utility or definitions (or occasionally, both).

Using the Methodology

Our top-down approach suggests a focus on developing reasonably short lemma books. The trick to keeping their lengths under control is first to identify the main lemmas in support of the book's goal theorem, then pushing their proofs into subsidiary lemma sub-books, especially when supporting sub-lemmas may be required. Each main lemma is handled as illustrated above with sub-books `L1` and `L2`: an `encapsulate` event contains first, a `local` `include-book` of the corresponding lemma sub-book, and second, the main lemma.

An important aspect of this approach is that the way in which a lemma is proved in such a sub-book will not affect the certification of the parent book. That is, the use of `local` around the `include-book` of a lemma sub-book allows the sub-book to be changed, other than changing its goal theorem, without affecting the certification of the parent book. This modularity can prevent replay problems often encountered when using less structured approaches to mechanically-assisted proof.

Although our focus is on lemma books, there is still a role for library books. During the course of developing a book, it will often seem most convenient to prove some of the simpler lemmas without going through the effort to create lemma books for them. It can be useful from time to time to browse through one's current collection of books and, after backing up the whole set of them in case the process becomes awkward, to pull out the most general of these lemmas and put them into one or more library books. For each lemma book that has had at least one such lemma removed, at least one `include-book` event will be necessary in order to account for the removed lemmas. Of course, the resulting collection of books might not be accepted by ACL2 because of the rearrangement, but often one can find a way to make appropriate modifications. (At any rate, this process of moving lemmas to library books is optional.) The resulting library books can then be used to advantage during the rest of the development effort.

There are of course reasonable variations of this methodology. One variation is to include a subsidiary book at the top level, rather than locally to an `encapsulate`. For example, a comment in book `riemann-sum--approximates-integral-1.lisp` explains that its lemmas `car-common--refinement` and `car-last-common-refinement` are needed in a parent book. Hence, that parent book includes the former book at the top level, not locally to an `encapsulate`. Perhaps a better approach would have been to move these two lemmas into a lemma book, but the modular methodology accommodates such variation. Specifically, the tool we now describe is based simply on the notions of lemma books and library books.

6.1.2 Proof Presentation

We describe here an *outline tool* that assists in proof presentation by creating a top-down outline of the proof. This tool is available on the Web pages for this book, [67].

The outline tool takes a parameter that specifies the maximum depth displayed in the outline. Each entry in the outline gives the following summary of one lemma book:

- the lemma proved in the book, *i.e.*, the last `defthm` in the book;
- the lemma books included (via `include-book`) in the book; and
- the library books included (via `include-book`) in the book.

Each lemma book generates a sub-entry, but only to the specified depth limit. Library books do not generate entries.

For example, the Appendix at the end of this chapter shows the depth-3 outline that was generated mechanically from the books for the Fundamental Theorem of Calculus. The first entry is, as always, labeled `Main`; it corresponds to the top-level lemma book, `fundamental-theorem-of-calculus`. The outline tool generates an entry for each lemma book included within that top-level book: entry `Main.1` for lemma book `split-integral-by--subintervals`, and entry `Main.2` for lemma book `ftoc-lemma`. These each generate entries as well, but those entries generate no further entries because of the depth limit of 3.

One exception to this format is the case of a book `B` included in more than one parent book. After the entry for `B` is printed, subsequent entries for `B` will be abbreviated as shown in the following example from an outline of depth at least 5.

```
Main.1.1.2.1.  riemann-sum-approximates-integral-1.
 <See Main.1.1.1 for details.>
```

6.2 Case Study: The Fundamental Theorem of Calculus

We assume that the reader has seen the Fundamental Theorem of Calculus (FTOC) but has perhaps forgotten it. Hence, below we give an informal review of that theorem, from which we build its formalization. We then give a high-level description of our mechanized proof.

Before all that, we discuss briefly an unusual aspect of this effort: the use of *non-standard analysis*, which introduces a notion of *infinitesimal* numbers as described below. In spite of this twist, this chapter assumes no prior familiarity with non-standard analysis and brings it only minimally into the discussion. Our primary goal is to illustrate the modular proof methodology described above. Our secondary goal is to give an overview of our formalization and mechanized proof of the FTOC.

6.2.1 A Very Short Introduction to Non-Standard Analysis and Its Acl2-Based Mechanization

This case study uses *non-standard analysis* together with a corresponding modification ACL2(r) of ACL2 implemented by Ruben Gamboa [36]. Non-standard analysis was introduced by Abraham Robinson in about 1960 (see [92]) in order to make rigorous Leibniz's approach to calculus from the 17^{th} century. Its main idea is to extend the real number line by adding non-zero *infinitesimals*, numbers that are less in absolute value than every positive real number. Thus, the real numbers are embedded in a larger number system that contains these infinitesimals. See also the freshman calculus book [62] for a development along such lines. A different way of looking at non-standard analysis is given by Nelson [81, 82], where there is still one number line embedded into another, but this time the larger one is considered to be the real line and the smaller one is considered to be the set of *standard* real numbers, which includes all definable real numbers (*e.g.*, 3 and π).

Nelson's view guided Gamboa's development of ACL2(r). There, the predicate `realp` is a recognizer for points on the larger number line, while `standard-numberp` is true of only the standard real numbers. In ACL2(r) there is a constant `(i-large-integer)` to be interpreted as an integer greater than every standard integer.[3] Thus its reciprocal is a non-zero

[3] By the comment on definability at the end of the preceding paragraph, `i-large-integer` is not definable in the real number field.

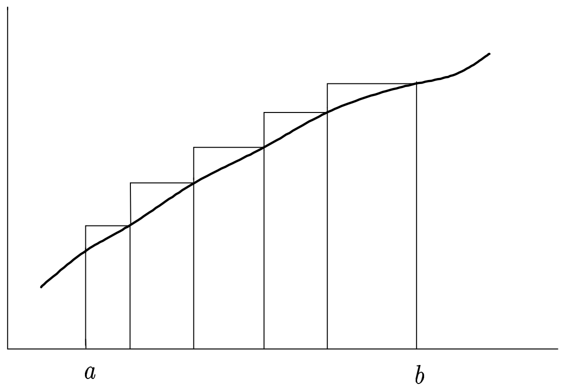

Figure 6.1: Area under the curve $y = f(x)$

infinitesimal, *i.e.*, a non-zero `realp` that is smaller in absolute value than every positive real `standard-numberp`.

Chapter 18 in this book gives the basics of non-standard analysis in an ACL2 setting. More extensive background may be found in Gamboa's doctoral dissertation [36] and also in [37]. Here we attempt to keep the discussion reasonably self-contained, introducing concepts from non-standard analysis only as needed. The full proof script can of course be found in the Web pages for this book [67].

6.2.2 Brief Informal Review of the FTOC

This section provides a quick review of the FTOC, which is often characterized by saying "the derivative of the integral of a function is the function itself." We refer implicitly to ideas presented here when discussing the formalization and proof that follow.

Consider real numbers a and b with $a < b$, together with a corresponding *partition* P, *i.e.*, a finite increasing sequence of numbers starting with a and ending with b. The *mesh* of P is defined to be the maximum difference between two successive numbers in P. Figure 6.1 shows rectangles formed by connecting successive numbers in P to form the bases and using a curve $y = f(x)$ to form the heights.

These rectangles can be used to approximate the area under this curve between a and b. The *definite integral of f from a to b* is approximated by the *Riemann sum* of f over this partition, obtained by adding up the areas of all these rectangles. These Riemann sums provide arbitrarily good approximations to the definite integral, by using partitions of arbitrarily small mesh.

If we imagine moving b a little bit to one side, then the area changes by an amount roughly equal to the height of the curve at b times the change in b. This observation is made precise in calculus, using the following idea. Given a function $f(x)$ on the real numbers, define $I(x)$ to be the definite integral from a fixed value a to x. Let $I'(x)$ be the *derivative* of $I(x)$, that is, the rate of change of $I(x)$ with respect to x. In this chapter we consider the following form of the FTOC: $I'(b) = f(b)$. Informally, the rate of change at b of the area accumulated from a under the curve $y = f(x)$ is $f(b)$.

Exercise 6.1 *Define a function* `(partitionp p)` *in ACL2, which is true exactly when* `p` *is a non-empty, strictly increasing sequence of rational numbers. Test your function.*

6.2.3 Formalizing the Fundamental Theorem of Calculus

The approach outlined above is formalized in the following theorem, which we have proved using ACL2(r). It reflects the discussion above: the derivative of the definite integral is the value of the function.

```
(implies (and (realp a) (realp x))
         (equal (integral-rcfn-prime a x)
                (rcfn x)))
```

In order to understand this theorem, we need to understand the functions in it. The function `rcfn` is constrained to be a real-valued continuous function on the reals, using the `encapsulate` event on page 304. The function `integral-rcfn-prime` is intended to formalize the notion, described informally above, of the derivative of the integral `integral-rcfn` (defined below) of `rcfn`. The following expression represents the rate of change of the integral over the interval from `x` to `(+ x eps)`; thus, `eps` is the change in `x`.

```
(/ (- (integral-rcfn a (+ x eps))
      (integral-rcfn a x))
   eps)
```

The derivative of the integral is then obtained by choosing an infinitesimal value of `eps`. But that so-called *difference quotient* is then merely *infinitely close* to the actual derivative; that is, it differs from the derivative by an infinitesimal. In order to obtain equality, we can use the `standard-part` function, which produces the unique real number that is infinitely close to its argument. Here then is the definition of the derivative of the integral.

```
(defun-std integral-rcfn-prime (a x)
   (if (and (realp a) (realp x))
       (let ((eps (/ (i-large-integer))))
```

```
         (standard-part
          (/ (- (integral-rcfn a (+ x eps))
                (integral-rcfn a x))
             eps)))
     0))  ; default
```

The reader may notice the use of `defun-std` rather than defun in this definition. The corresponding axiom equates `(integral-rcfn-prime a x)` with the definition body, under the hypotheses that the arguments `a` and `x` are standard. Further discussion of `defun-std` is beyond the scope of this paper; see Chapter 18 and see [36, 37].

To be fair, the usual definition of the derivative in non-standard analysis requires the result to be independent of the change in `x`; see for example [62]. This property is guaranteed by the following theorem.

```
(defthm integral-rcfn-prime-exists
  (implies (and (realp eps)
                (not (equal eps 0))
                (i-small eps)  ; eps is infinitesimal
                (realp a) (standard-numberp a)
                (realp x) (standard-numberp x))
           (equal (standard-part
                   (/ (- (integral-rcfn a (+ x eps))
                         (integral-rcfn a x))
                      eps))
                  (integral-rcfn-prime a x)))
  :hints ...)
```

The function `integral-rcfn` represents the definite integral of `rcfn` between its two arguments. For standard reals `a` and `b`, its definition uses a specific partition P into a non-standard number of equal-sized subintervals, given by `(make-partition a b (i-large-integer))`, to form a Riemann sum `(riemann-rcfn` P`)`. `Standard-part` is applied to this sum in order to obtain a standard real number.

```
(defun-std integral-rcfn (a b)
  (cond ((or (not (realp a)) (not (realp b)))
         0)  ; default
         ((< a b)
          (standard-part
           (riemann-rcfn
            (make-partition a b (i-large-integer)))))
         ((< b a)  ; then reverse the sign as well as a and b
          (- (standard-part
              (riemann-rcfn
               (make-partition b a (i-large-integer))))))
         (t 0)))
```

The following exercises elaborate on our FTOC formalization. They are to be done using ACL2 (although ACL2(r) should also work).

Exercise 6.2 *Define a function* `(make-partition a b n)` *which, for rational numbers* `a` *and* `b` *and positive integer* `n`*, splits the interval from* `a` *to* `b` *into* `n` *equal-sized subintervals by returning an appropriate sequence of numbers from* `a` *to* `b`*. Test your function, for example as follows.*

```
ACL2 !>(make-partition 3 7 8)
(3 7/2 4 9/2 5 11/2 6 13/2 7)
```

Exercise 6.3 *Define a function* `(deltas p)`*, which returns the ordered list of successive intervals represented by* `p` *as in the following example.*

```
ACL2 !>(deltas '(12 13 15 24))
(1 2 9)
```

Exercise 6.4 *Define the function* `(mesh p)`*, the* mesh *of partition* `p` *as introduced informally in Section 6.2.2.*

Create a book `partition-defuns` containing solutions to the preceding exercises, to use (via `include-book`) in the remaining exercises. Use the modular methodology where appropriate. The first two exercises below have a similar flavor, so if the first seems difficult, then it will be instructive to try the second after studying the solution to the first.

Exercise 6.5 *Prove the following theorem.*

```
(defthm partitionp-make-partition
  (implies (and (rationalp a)
                (rationalp b)
                (< a b)
                (not (zp n)))
           (partitionp (make-partition a b n))))
```

Exercise 6.6 *Prove the following theorem.*

```
(defthm mesh-make-partition
  (implies (and (rationalp a) (rationalp b) (< a b)
                (integerp n) (< 0 n))
           (equal (mesh (make-partition a b n))
                  (/ (- b a) n))))
```

Exercise 6.7 *Prove the following theorem with the hints shown.*

```
(defthm mesh-append
  (implies (and (partitionp p1)
                (partitionp p2)
                (equal (car (last p1)) (car p2)))
```

```
           (equal (mesh (append p1 (cdr p2)))
                  (max (mesh p1) (mesh p2))))
  :hints (("Goal" :do-not-induct t
           :do-not '(eliminate-destructors))))
```

Note. We have seen ACL2 prove this theorem automatically without the :hints *in about 7 minutes on a 1999-fast Sparc (an E10000). But with the appropriate lemmas, the proof can take less than a second.*

Exercise 6.8 *Define the dot product of two lists so that the following holds.*

```
(dotprod (list a1 a2 ... an) (list b1 b2 ... bn))
  =
(+ (* a1 b1) (* a2 b2) ... (* an bn))
```

Next, declare function `(rcfn x)` *using* defstub*. Then define the pointwise application of* `rcfn` *to a list, so that the following holds.*

```
(map-rcfn (list a1 a2 ... an))
  =
(list (rcfn a1) (rcfn a2) ... (rcfn an)).
```

The immediately preceding exercise allows us to define the Riemann sum as described informally in Section 6.2.2. Each rectangle's height is determined by the endpoint at the right of the corresponding subinterval.[4]

```
(defun riemann-rcfn (p)
  (dotprod (deltas p) (map-rcfn (cdr p))))
```

6.2.4 Proving the Fundamental Theorem of Calculus

Our hope is that the outline in the Appendix, at the end of this chapter, provides a contribution to the exposition below, thus justifying our claim that the outline tool is useful for proof presentation.

Recall that our formalization of the FTOC says that `(integral-rcfn--prime a x)` is equal to `(rcfn x)` for real `a` and `x`. The main structure of the proof consists of two parts. First, recall the definition of `integral-rcfn-prime` given in Section 6.2.3 in terms of a particular infinitesimal, `eps`.

```
(standard-part (/ (- (integral-rcfn a (+ x eps))
                     (integral-rcfn a x))
                  eps))
```

[4] One typically defines the Riemann sum independently of the choices of points in the subintervals. Moreover, one considers all partitions, not just the partition returned by `make-partition`. Lemma `riemann-sum-approximates-integral`, shown as `Main.1.1` in the Appendix, captures the essence of the latter issue. We have not proved results that deal with the former issue; this problem would make a nice exercise.

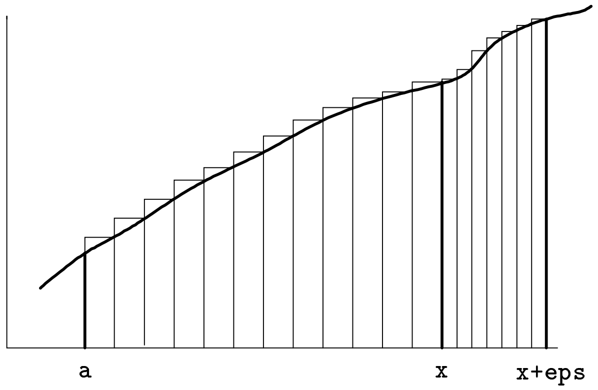

Figure 6.2: Area from a to b under $y = \texttt{rcfn}(x)$, split in two

Figure 6.2 suggests that the difference above is equal to the definite integral from `x` to `x+eps`. The first main part of the proof is a corresponding lemma, which reduces the form displayed above to the following.

```
(standard-part (/ (integral-rcfn x (+ x eps))
                  eps))
```

The second part of the proof is to show that this new form is equal to `(rcfn x)`.

These two main steps correspond to the two lemmas just under the FTOC in the outline produced by our outline tool. In the Appendix those lemmas are shown respectively as `split-integral-by-subintervals` (labeled `Main.1`) and `ftoc-lemma` (labeled `Main.2`). Actually, the first of these lemmas expresses the decomposition in terms of a sum rather than a difference; the difference version is proved directly in the top-level book, since it is a simple consequence of the sum version. We discuss the proofs of these two lemmas below.

Lemma `ftoc-lemma` actually assumes that `x` is standard, and in fact we initially derive a version of the FTOC subject to that restriction. However, the so-called *transfer principle* of non-standard analysis, in particular as implemented by event `defthm-std` in ACL2(r), allows this restriction to be removed. Further discussion of the transfer principle is beyond the scope of this chapter but may be found in Chapter 18; see also [36, 37].

Overview of Proof of `Split-integral-by-subintervals`

Lemma `split-integral-by-subintervals` is labeled `Main.1` in the outline appearing in the Appendix. It has the following statement, which can be viewed as a formalization of Figure 6.2.

```
(implies (and (realp a) (realp b) (realp c))
       (equal (integral-rcfn a c)
              (+ (integral-rcfn a b)
                 (integral-rcfn b c))))
```

Each of the two integrals added together above is the standard part of the total area of a corresponding set of rectangles, much as suggested in Figure 6.2. The right-hand side of the equality above is hence infinitely close to the total area of all these rectangles, which is a Riemann sum for the interval from `a` to `c`. Thus, a main subtask is to prove the following lemma, which is labeled `Main.1.1` in the outline. It says that every Riemann sum over a partition of infinitesimal mesh is infinitely close (`i-close`) to the exact integral, *i.e.*, their difference is infinitesimal.

```
(implies (and (partitionp p)
              (equal a (car p)) (equal b (car (last p)))
              (< a b)
              (standard-numberp a) (standard-numberp b)
              (i-small (mesh p)))  ; the mesh is infinitesimal
         (i-close (riemann-rcfn p) (integral-rcfn a b)))
```

We are now ready to sketch the proof of `split-integral-by-subintervals`. Its conclusion is equal to the following, by definition of `integral-rcfn` and for two particular terms representing partitions, which we abbreviate here as p_1 and p_2.

```
(equal (integral-rcfn a c)
       (+ (standard-part (riemann-rcfn p1))
          (standard-part (riemann-rcfn p2))))
```

Lemma `Main.1.2` from the Appendix reduces the term above to the following.

```
(i-close (integral-rcfn a c)
         (+ (riemann-rcfn p1) (riemann-rcfn p2)))
```

If we can write the sum above as an appropriate single application of `riemann-rcfn`, then (by symmetry of `i-close`) the application of Lemma `Main.1.1` should be able to complete the proof.

The following theorem serves that purpose. It appears in the lemma book `split-integral-by-subintervals`. It may have been desirable to push its proof into a lemma sub-book instead; at least, that way it would have shown up in the outline. However, its proof is fully automatic, so it was natural not to open up a sub-book for it.

```
(defthm split-riemann-rcfn-by-subintervals
  (implies
   (and (partitionp p1)
        (partitionp p2)
        (equal (car (last p1)) (car p2)))
```

```
   (equal (+ (riemann-rcfn p1)
             (riemann-rcfn p2))
          (riemann-rcfn (append p1 (cdr p2))))))
```

Overview of Proof of `Ftoc-lemma`

Lemma `ftoc-lemma` is shown in the Appendix as `Main.2`. Its conclusion is as follows.

```
(equal (standard-part
        (/ (integral-rcfn x (+ x eps)) eps))
       (rcfn x))
```

This equality can be rewritten to the following expression, using basic non-standard reasoning.

```
(i-close (/ (integral-rcfn x (+ x eps)) eps)
         (rcfn x))
```

Lemma `Main.2.2` in the Appendix reduces this problem to finding two values that are both `i-close` to `(/ (integral-rcfn x (+ x eps)) eps)` such that `(rcfn x)` is between them. Natural candidates for these values are the minimum and maximum values of `rcfn` on the interval from `x` to `(+ x eps)`, provided respectively by the application of functions `min-x` and `max-x` to these interval endpoints. This observation is captured in the conclusion of Lemma `Main.2.1` in the outline.

```
(between (/ (integral-rcfn a b) (- b a))
         (rcfn (min-x a b))
         (rcfn (max-x a b))).
```

Of course, many details are omitted here, *e.g.*, how non-standard analysis is used to define functions such as `max-x` and `min-x`. (See Chapter 18.) Nevertheless, the reader can see the structure of the proof of `ftoc-lemma` by looking at the outline in the Appendix, or by looking at an outline for depth greater than 3 in the supporting materials (see [67]).

6.3 Conclusion

I found it surprisingly pleasant to carry out the proof of the Fundamental Theorem of Calculus using the top-down, modular methodology presented here. It was very satisfying, when putting aside the unfinished proof for the day, to know that I had a self-contained collection of ACL2 books from which I could easily identify the remaining proof obligations. Especially comforting was the knowledge that the completion of those proof obligations would not interfere with the replayability of the completed parts of the proof.

Proof construction using ACL2 is a combination of high-level strategy and low-level tactics, a point stressed (for Nqthm and Pc-Nqthm) using an extensive example in the tutorial [60]. The top-down nature of the methodology presented here supports the user's development of a high-level strategy. The modularity inherent in the methodology supports the carrying out of lower-level tactics without interfering with the higher-level structure (strategy) of the proof.

Acknowledgments

This work was performed while the author was employed at the Austin Renovation Center of EDS, Inc., with wonderful encouragement from its manager, Joe Hill. I am also grateful to Dave Reed of Advanced Micro Devices, Inc. for his support of final preparations of this article and of ACL2 development.

Appendix: The Depth-3 Outline for FTOC

Some whitespace created by the outline tool has been manually modified for display purposes.

```
Main.  fundamental-theorem-of-calculus.
 (IMPLIES (AND (REALP A) (REALP X))
          (EQUAL (INTEGRAL-RCFN-PRIME A X) (RCFN X)))
using lemmas:
  ("split-integral-by-subintervals" "ftoc-lemma")
and library books:
  ("integral-rcfn" "nsa-lemmas" "integral-rcfn-lemmas"
   "riemann-lemmas" "make-partition" "riemann-defuns")
==============================
Main.1.  split-integral-by-subintervals.
 (IMPLIES (AND (REALP A) (REALP B) (REALP C))
          (EQUAL (INTEGRAL-RCFN A C)
                 (+ (INTEGRAL-RCFN A B)
                    (INTEGRAL-RCFN B C))))
using lemmas:
  ("riemann-sum-approximates-integral"
   "integral-rcfn-equal-if-i-close")
and library books:
  ("integral-rcfn-lemmas" "nsa-lemmas" "integral-rcfn"
   "riemann-lemmas" "riemann-defuns")
==============================
```

```
Main.1.1.  riemann-sum-approximates-integral.
 (IMPLIES (AND (PARTITIONP P)
               (EQUAL A (CAR P)) (EQUAL B (CAR (LAST P)))
               (< A B)
               (STANDARD-NUMBERP A) (STANDARD-NUMBERP B)
               (I-SMALL (MESH P)))
          (I-CLOSE (RIEMANN-RCFN P) (INTEGRAL-RCFN A B)))
using lemmas (NOT shown just below:  depth limit reached):
  ("riemann-sum-approximates-integral-1"
   "riemann-sum-approximates-integral-2")
and library books:
  ("integral-rcfn" "riemann-defuns")
==============================
Main.1.2.  integral-rcfn-equal-if-i-close.
 (IMPLIES (AND (REALP A) (STANDARD-NUMBERP A)
               (REALP B) (STANDARD-NUMBERP B)
               (< A B)
               (REALP Y) (REALP Z))
          (EQUAL (EQUAL (INTEGRAL-RCFN A B)
                        (+ (STANDARD-PART Y)
                           (STANDARD-PART Z)))
                 (I-CLOSE (INTEGRAL-RCFN A B) (+ Y Z))))
using lemmas (NOT shown just below:  depth limit reached):
  ("standard-part-equal-if-i-close")
and library books:
  ("integral-rcfn-lemmas" "riemann-lemmas" "integral-rcfn"
   "riemann-defuns")
==============================
Main.2.  ftoc-lemma.
 (IMPLIES (AND (REALP EPS) (NOT (EQUAL EPS 0))
               (I-SMALL EPS)
               (REALP X) (STANDARD-NUMBERP X))
          (EQUAL (STANDARD-PART
                  (/ (INTEGRAL-RCFN X (+ X EPS)) EPS))
                 (RCFN X)))
using lemmas:
  ("integral-rcfn-quotient-between-non-classical"
   "between-i-close-implies-i-close")
and library books:
  ("min-x-and-max-x-lemmas" "../nsa/realp" "defaxioms"
   "integral-rcfn" "max-and-min-attained" "nsa-lemmas"
   "integral-rcfn-lemmas" "riemann-lemmas"
   "make-partition" "riemann-defuns")
==============================
```

```
Main.2.1.  integral-rcfn-quotient-between-non-classical.
 (IMPLIES (AND (STANDARD-NUMBERP A) (REALP A)
               (STANDARD-NUMBERP B) (REALP B)
               (< A B))
          (BETWEEN (/ (INTEGRAL-RCFN A B) (- B A))
                   (RCFN (MIN-X A B))
                   (RCFN (MAX-X A B))))
using lemmas (NOT shown just below:  depth limit reached):
  ("riemann-rcfn-between" "between-limited-implies-limited"
   "standard-part-preserves-between" "rcfn-standard-part"
   "i-limited-rcfn")
and library books:
  ("../nsa/realp" "defaxioms" "integral-rcfn"
   "max-and-min-attained" "nsa-lemmas"
   "integral-rcfn-lemmas" "riemann-lemmas"
   "make-partition" "riemann-defuns")
==============================
Main.2.2.  between-i-close-implies-i-close.
 (IMPLIES (AND (REALP Z) (REALP X) (REALP Y) (REALP R)
               (BETWEEN Z X Y)
               (I-CLOSE X R) (I-CLOSE Y R))
          (I-CLOSE Z R))
using library books:
  ("../nsa/realp" "defaxioms" "../arithmetic/top-with-meta"
   "nsa-lemmas")
==============================
```

7

Mu-Calculus Model-Checking

Panagiotis Manolios
Department of Computer Sciences, University of Texas at Austin, Texas
Email: pete@cs.utexas.edu

Abstract

Temporal logic model-checking has received substantial academic interest and has enjoyed wide industrial acceptance. Temporal logics are used to describe the behavior (over time) of systems which continuously interact with their environment. Model-checking algorithms are used to decide if a finite-state system satisfies a temporal logic formula. Many temporal logics, *e.g.*, *CTL*, *LTL*, and CTL^* can be translated into the Mu-Calculus. In addition, the algorithm that decides the Mu-Calculus is used for symbolic (BDD-based) model-checking, a technique that has greatly extended the applicability of model-checking. In this case study we define a model-checker for the Mu-Calculus in ACL2 and show how to translate *CTL* into the Mu-Calculus.

In the process of defining the Mu-Calculus, we develop (ACL2) books on set theory, fixpoint theory, and relation theory. The development of these books is given as a sequence of exercises. These exercises make use of varied ACL2 features; therefore, the first few sections may be of interest to readers who want more practice in proving theorems in ACL2.

Introduction

Machine-checked proofs are increasingly being used to cope with the complexity of current hardware and software designs: such designs are too complicated to be checked by hand and machine-checked proofs are a reliable way to ensure correctness. *Reactive systems* are systems with nonterminating or concurrent behavior. Such systems are especially difficult to design and verify. Temporal logic was proposed as a formalism for specifying the correctness of reactive systems in [87]. Algorithms that decide if a finite-state system satisfies its specification are known as *model-checking* algorithms [20, 27, 89]. Model-checking has been successfully applied to automatically verify many reactive systems and is now being used by hardware companies as part of their verification process. In this chapter, we develop a model-checker for the propositional Mu-Calculus [64, 30, 32, 31, 29, 85]—a calculus that subsumes the temporal logics *CTL*, *LTL*, and CTL^*—in ACL2.

This chapter is intended as a bridge between the companion book, *Computer-Aided Reasoning: An Approach* [58], and the other case studies. There are several self-contained sections in which the reader is presented with exercises whose solutions lead to books on set theory, fixpoint theory, and relation theory. We expect that the exercises in these sections are at the right level of difficulty for readers who have read the companion book. These exercises make use of diverse, less elementary features of ACL2 such as congruence-based reasoning, refinements, packages, the use of macros, guard verification, encapsulation, mutual recursion, and functional instantiation. We also discuss *compositional reasoning*; specifically we show how to reason about efficient implementations of functions by using rewrite rules that transform the efficient functions into other functions that are easier to reason about. Therefore, we expect—at least the first part of—this chapter to be of general interest.

If you are not interested in developing the required set theoretic results, but are interested in formalizing the Mu-Calculus in ACL2, then, instead of solving the exercises on your own, download the appropriate books from the supporting material for this chapter.

This chapter is organized as follows: the next three sections develop the set theory, fixpoint theory, and relation theory discussed above. In the three sections after that, we present the notion of a model, the syntax and semantics of the Mu-Calculus, and proofs that the fixpoint operators of the Mu-Calculus actually compute fixpoints. A section on the temporal logic *CTL* and its relation to the Mu-Calculus follows. We conclude with some directions for further exploration.

Conventions on Exercises

Whenever we introduce a function or ask you to define one, admit it and add and verify guards; this is an implicit exercise. Many exercises consist solely of a term or an event; interpret this as a command to prove that the term is a theorem or to admit the event. The supporting material includes a macro that you may find useful for dealing with guards. The file `solutions/defung-intro.txt` describes the macro and contains exercises.

7.1 Set Theory

In this section, we develop some set theory. We represent sets as lists and define an equivalence relation on lists that corresponds to set equality. It turns out that we do not have to develop a "general" theory of sets; a theory of *flat* sets, *i.e.*, sets whose elements are compared by `equal`, will do. For example, in our theory of sets, `'(1 2)` is set equal to `'(2 1)`, but `'((1 2))` is not set equal to `'((2 1))`.

We develop some of the set theory in the package `SETS` (see defpkg) and the rest in the package `FAST-SETS`, in subsections labeled by the package names. When using packages, we define constants that contain all of the symbols to be imported into the package. We start by guessing which symbols will be useful. For example, we import `len` because we need it to define the cardinality of a set and we import the symbols `x` and `x-equiv`; otherwise, when using defcong, `x-equiv` prints as `ACL2::x-equiv`, which strains the eye. As we develop the book, we notice that we forgot a few symbols and add them.[1]

7.1.1 SETS

Here is how we define the package `SETS`.

```
(defconst *export-symbols*
  (union-eq *acl2-exports*
       (union-eq
        '(len  ...  *export-symbols*)
        *common-lisp-symbols-from-main-lisp-package*)))

(defconst *sets-symbols* (union-eq *export-symbols*  ...  ))

(defpkg "SETS" *sets-symbols*)
```

We use the simplest definitions that we can think of so that it is easy to prove theorems. Later, we define functions that are more efficient and prove the rewrite rules that allow us to rewrite the efficient functions into the simpler ones. In this way, once rewritten, all the theorem proving is about the simple functions, but the execution uses the efficient versions.

The definitions of `in` (set membership), `=<` (subset), and `==` (set equality) follow.

```
(defun in (a X)
  (cond ((endp X) nil)
        ((equal a (car X)) t)
        (t (in a (cdr X)))))
```

[1] Due to some technical issues (see package-reincarnation-import-restrictions), this unfortunately means that we have to start a new ACL2 session.

```
(defun =< (X Y)
  (cond ((endp X) t)
        (t (and (in (car X) Y)
                (=< (cdr X) Y)))))

(defun == (X Y)
  (and (=< X Y)
       (=< Y X)))
```

Notice that == is an *equivalence relation*: it is reflexive, symmetric, and transitive. The macro `defequiv` can be used to show that a relation is an equivalence relation. Use `:trans1` to print out the translation of the `defequiv` form in the exercise below before you do it.

Exercise 7.1 `(defequiv ==)`

We make heavy use of congruence-based reasoning and will therefore discuss the topic briefly. For a full explanation consult the companion book [58] and the documentation on `equivalence`, `defequiv`, and `congruence`. Congruence-based reasoning can be seen as an extension of the substitution of equals for equals, where arbitrary equivalence relations can be used instead of equality. We motivate the need for congruence-based reasoning with an example using the equivalence relation ==.

Consider the function `set-union` which computes the union of two sets. This function is defined below and is equivalent to `append`. We might want to prove

```
(implies (== X Z)
         (equal (set-union X Y) (set-union Z Y)))
```

so that ACL2 can replace x by z in `(set-union` x y`)`, if it can establish `(==` x z`)`. Letting x be `(1 1)` and z be `(1)`, it is easy to see that this is not a theorem. However, the following is a theorem.

```
(implies (== X Z)
         (== (set-union X Y) (set-union Z Y)))
```

If stored as a congruence rule (see `congruence` and `rule-classes`), ACL2 can use this theorem to substitute z for (a set equal) x in `(set-union` x y`)`, in a context where it is enough to preserve ==. More generally, a theorem of the form:

```
(implies (eq1 X Z)
         (eq2 (foo ... X ...)
              (foo ... Z ...)))
```

where `eq1` and `eq2` are known equivalence relations can be made a congruence rule. Such a rule allows us to replace x by z in `(foo ...` x `...)` if x and z are `eq1`-equal and we are in a context where it is enough to preserve `eq2`. This should make it clear why congruence-based reasoning is a generalization of the substitution of equals for equals.

The macro `defcong` can be used to prove congruence rules. Use `:trans1` to print out the translation of the `defcong` forms in the exercise below before you do it.

Exercise 7.2

1. `(defcong == equal (in a X) 2)`
2. `(defcong == equal (=< X Y) 1)`
3. `(defcong == equal (=< X Y) 2)`
4. `(defcong == == (cons a X) 2)`

We now give the definition of `set-union`.

```
(defun set-union (X Y)
  (if (endp X)
      Y
    (cons (car X) (set-union (cdr X) Y))))
```

Exercise 7.3

1. `(equal (in a (set-union X Y)) (or (in a X) (in a Y)))`
2. `(=< X (set-union Y X))`
3. `(== (set-union X Y) (set-union Y X))`
4. `(equal (== (set-union X Y) Y) (=< X Y))`
5. `(defcong == == (set-union X Y) 1)`
6. `(equal (=< (set-union Y Z) X) (and (=< Y X) (=< Z X)))`

The definition of `intersect`, a function which computes the intersection of two sets, follows.

```
(defun intersect (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (cons (car X) (intersect (cdr X) Y)))
        (t (intersect (cdr X) Y))))
```

Exercise 7.4

1. `(equal (in a (intersect X Y)) (and (in a X) (in a Y)))`
2. `(== (intersect X Y) (intersect Y X))`

3. `(implies (=< X Y) (== (intersect X Y) X))`

4.
```
(implies (or (=< Y X) (=< Z X))
         (=< (intersect Y Z) X))
```

The definition of `minus`, a function which computes the set difference of two sets, follows.

```
(defun minus (X Y)
  (cond ((endp X) nil)
        ((in (car X) Y)
         (minus (cdr X) Y))
        (t (cons (car X) (minus (cdr X) Y)))))
```

Exercise 7.5

1. `(implies (=< X Y) (equal (minus X Y) nil))`

2. `(implies (=< X Y) (=< (minus X Z) Y))`

The functions `set-complement`, `remove-dups`, `cardinality`, and `s<` (strict subset) are defined below.

```
(defun set-complement (X U) (minus U X))

(defun remove-dups (X)
  (cond ((endp X) nil)
        ((in (car X) (cdr X))
         (remove-dups (cdr X)))
        (t (cons (car X)
                 (remove-dups (cdr X))))))

(defun cardinality (X) (len (remove-dups X)))

(defun s< (X Y) (and (=< X Y) (not (=< Y X))))
```

Exercise 7.6 *Define* `perm`*, a function of two arguments that returns* `t` *if its arguments are permutations and* `nil` *otherwise. Prove* `(defequiv perm)` *and* `(defrefinement perm ==)`*. (*`Perm` *is defined in the companion book [58].)*

Exercise 7.7
```
(implies (s< X Y)
         (< (len (remove-dups X)) (len (remove-dups Y))))
```

7.1.2 FAST-SETS

Although the definitions of the basic set operations defined above are good for reasoning about sets, some are not appropriate for execution. For example, `set-union` is not tail-recursive[2], hence, even if compiled, we can easily get stack overflows. In this section, we define functions that are more appropriate for execution and prove rewrite rules that transform the new, efficient versions to the old, simpler versions in the appropriate context (specifically, when it is enough to preserve ==). This approach is *compositional*, *i.e.*, it allows us to decompose proof obligations of a system into proof obligations of the components of the system. Compositional reasoning is routinely used by ACL2 experts and is essential to the success of large verification efforts.

The functions we define below have the same names as their analogues, but are in the package `FAST-SETS`. `FAST-SETS` imports symbols from `SETS`, *e.g.*, == (we expect this to be clear from the context, but one can consult the supporting material for the package definition, if required). The definition of `set-union`, in the package `FAST-SETS`, follows.

```
(defun set-union (X Y)
  (cond ((endp X) Y)
        ((in (car X) Y)
         (set-union (cdr X) Y))
        (t (set-union (cdr X) (cons (car X) Y)))))
```

Exercise 7.8 `(== (set-union X Y) (sets::set-union X Y))`

Recall that the above rule allows ACL2 to replace occurrences of `set-union` by `sets::set-union` in a context where it is enough to preserve ==.

The definition of `intersect` follows. Note that its auxiliary function is tail recursive.

```
(defun intersect-aux (X Y Z)
  (cond ((endp X) Z)
        ((in (car X) Y)
         (intersect-aux (cdr X) Y (cons (car X) Z)))
        (t (intersect-aux (cdr X) Y Z))))

(defun intersect (X Y) (intersect-aux X Y nil))
```

Exercise 7.9 `(== (intersect X Y) (sets::intersect X Y))`

Exercise 7.10 *Define* `minus`*, a tail-recursive version of* `sets::minus`*, and prove* `(== (minus X Y) (sets::minus X Y))`*.*

[2] See the companion book [58] for a discussion of tail recursion and for example proofs.

Alternate definitions of `remove-dups` and `cardinality` are given below.

```
(defun remove-dups (X) (set-union X nil))

(defun cardinality (X) (len (remove-dups X)))
```

Exercise 7.11 `(equal (cardinality X) (sets::cardinality X))`

7.2 Fixpoint Theory

In this section, we develop a book in the package `SETS` on the theory of fixpoints. We do this in a very general setting, by using encapsulation to reason about a constrained function, `f`, of one argument. Later, we show that certain functions compute fixpoints by using functional instantiation. An advantage of this approach is that we can ignore irrelevant issues, *e.g.*, in a later section we show that certain functions compute fixpoints; these functions have many arguments, but `f` has only one.

We say that x is a *fixpoint* of f iff $f(x) = x$. If f is a monotonic function on the powerset of a set, then by the following version of the Tarski-Knaster theorem [105], it has a least and greatest fixpoint, denoted by μf and νf, respectively.

Theorem 7.1 *Let* $f : 2^S \rightarrow 2^S$ *such that* $a \subseteq b \quad \Rightarrow \quad f(a) \subseteq f(b)$. *Then*

1. $\mu f \;=\; \cap\{b : b \subseteq S \;\wedge\; f(b) \subseteq b\} \;=\; \cup_{\alpha \in On} f^{\alpha}(\emptyset)$, *and*
2. $\nu f \;=\; \cup\{b : b \subseteq S \;\wedge\; b \subseteq f(b)\} \;=\; \cap_{\alpha \in On} f^{\alpha}(S)$,

where 2^S *is the powerset of* S, f^{α} *is the* α*-fold composition (iteration) of* f, *and* On *is the class of ordinals.*

We say that x is a *pre-fixpoint* of f iff $x \subseteq f(x)$; x is a *post-fixpoint* iff $f(x) \subseteq x$. The Tarski-Knaster theorem tells us that μf is below all post-fixpoints and that νf is above all pre-fixpoints.

We can replace On by the set of ordinals of cardinality at most $|S|$; since we are only interested in *finite* sets, this gives us an algorithm for computing least and greatest fixpoints. Notice that by the monotonicity of f, $\alpha \leq \beta \quad \Rightarrow \quad f^{\alpha}(\emptyset) \subseteq f^{\beta}(\emptyset) \quad \wedge \quad f^{\beta}(S) \subseteq f^{\alpha}(S)$. Therefore, we can compute μf by applying f to $\emptyset$ until we reach a fixpoint; similarly, we can compute νf by applying f to S until we reach a fixpoint.

We start by constraining functions `f` and `S` so that `f` is monotonic and when `f` is applied to a subset of `S`, it returns a subset of `S`. Since functions defined in ACL2 are total, we cannot say that `f` is a function whose domain is the powerset of `S`. We could add hypotheses stating that all arguments to `f` are of the right type to the theorems that constrain `f`, but this generality is not needed and will make it slightly more cumbersome to prove theorems about `f`. The issue of what to do when a function is applied outside its intended domain is one that comes up quite a bit in ACL2. The definitions of the constrained functions follow.

```
(encapsulate
 ((f (X) t)
  (S () t))
 (local (defun f(X) (declare (ignore X)) nil))
 (local (defun S() nil))
 (defthm f-is-monotonic
   (implies (=< X Y)
            (=< (f X) (f Y))))
 (defthm S-is-top
   (=< (f X) (set-union X (S))))).
```

We now define `applyf`, a function that applies `f` a given number of times.

```
(defun applyf (X n)
  (if (zp n)
      X
    (if (== X (f X))
        X
      (applyf (f X) (1- n)))))
```

From the Tarski-Knaster theorem, we expect that `lfpf` and `gfpf`, defined below, are the least and greatest fixpoints, respectively.

```
(defabbrev lfpf () (applyf nil (cardinality (S))))

(defabbrev gfpf () (applyf (S) (cardinality (S))))
```

Now all that is left is to prove the Tarski-Knaster theorem, which is given as the following two exercises.

Exercise 7.12 *Prove that* `lfpf` *is the least fixpoint:*

1. `(== (f (lfpf)) (lfpf))`
2. `(implies (=< (f X) X) (=< (lfpf) X))`

Exercise 7.13 *Prove that* `gfpf` *is the greatest fixpoint:*

1. `(== (f (gfpf)) (gfpf))`
2. ```
   (implies (and (=< X (S)) (=< X (f X)))
            (=< X (gfpf)))
   ```
   ```

7.3 Relation Theory

In this section we develop a book, in the package `RELATIONS`, on the theory of relations. We represent relations as alists which map an element to the set of elements it is related to. A recognizer for relations is the following.

```
(defun relationp (r)
  (cond ((atom r) (eq r nil))
        (t (and (consp (car r))
                (true-listp (cdar r))
                (relationp (cdr r))))))
```

The definition of `image`, a tail-recursive function that computes the image of a set under a relation, follows.

```
(defun value-of (x alist) (cdr (assoc-equal x alist)))

(defun image-aux (X r tmp)
  (if (endp X)
      tmp
    (image-aux (cdr X) r
               (set-union (value-of (car X) r) tmp))))
(defun image (X r) (image-aux X r nil))
```

Exercise 7.14 *Define* `range`*, a function that determines the range of a relation.*

Exercise 7.15 *Define* `inverse` *so that it is tail recursive and computes the inverse of a relation.*

The following function checks if the range of its first argument (a relation) is a subset of its second argument.

```
(defun rel-range-subset (r X)
  (cond ((endp r) t)
        (t (and (=< (cdar r) X)
                (rel-range-subset (cdr r) X)))))
```

Exercise 7.16

1. `(implies (rel-range-subset r X) (=< (image Y r) X))`
2. `(implies (and (rel-range-subset r X) (=< X Y))`
   `(rel-range-subset r Y))`

7.4 Models

In this section we introduce the notion of a model. A *model*, sometimes called a Kripke structure or a transition system, is a four-tuple consisting of a set of states, a transition relation, a set of atomic propositions, and a labeling relation. The transition relation relates a pair of states if the second state can be reached from the first in a single step. The atomic propositions can be thought of as Boolean variables that are either true or false at a state. The labeling relation relates states to the atomic propositions true at those states. A program can be thought of as a model: there is a state for every combination of legal assignments to the program's variables—which can be recovered from the labeling of the state—and the transition relation relates a pair of states if, in one step, the program can transition from the first state to the second. There are some technical details to consider, *e.g.*, a program can have variables of varying types, but atomic propositions are Boolean, hence, program variables are represented by a set of atomic propositions (this set can be infinite if the domain of the variable is infinite). We restrict our attention to finite models because we want to check them algorithmically.

We define the notion of a model in ACL2. The functions defined in this section, as well as the next two sections, are in the package `MODEL-CHECK`. An ACL2 model is a seven-tuple because it is useful to precompute the inverse relations of the transition relation and the labeling relation as well as the cardinality of the set of states. The inverse transition relation relates a pair of states if, in one step, the first state can be reached from the second. The inverse labeling relation relates atomic propositions to the states at which they hold. A function that creates a model is defined below.

```
(defun make-model (s r ap l)
  (list s r ap l (inverse r) (inverse l) (cardinality s)))
```

Exercise 7.17 *Define* `modelp`*, a recognizer for models. Define the accessor functions:* `states`*,* `relation`*,* `atomic-props`*,* `s-labeling`*,* `inverse--relation`*,* `a-labeling`*, and* `size` *to access the: states, transition relation, atomic propositions, (state) labeling relation, inverse transition relation, (atomic proposition) labeling relation, and cardinality of the states, respectively.*

7.5 Mu-Calculus Syntax

We are now ready to look at the Mu-Calculus. Informally, a formula of the Mu-Calculus is either an atomic proposition, a variable, a Boolean combination of formulae, $\mathtt{EX}f$, where f is a formula, or $\mu Y f$ or $\nu Y f$, where f is a formula and Y is a variable (as we will see when we discuss semantics,

```
(defun mu-symbolp (s)
  (and (symbolp s)
       (not (in s '(+ & MU NU true false)))))

(defun basic-mu-calc-formulap (f ap v)
  (cond ((symbolp f)
         (or (in f '(true false))
             (and (mu-symbolp f)
                  (or (in f ap) (in f v)))))
        ((equal (len f) 2)
         (and (in (first f) '(~ EX))
              (basic-mu-calc-formulap (second f) ap v)))
        ((equal (len f) 3)
         (let ((first (first f))
               (second (second f))
               (third (third f)))
           (or (and (in second '(& +))
                    (basic-mu-calc-formulap first ap v)
                    (basic-mu-calc-formulap third ap v))
               (and (or (in first '(MU NU)))
                    (mu-symbolp second)
                    (not (in second ap))
                    (basic-mu-calc-formulap
                     third ap (cons second v))))))))
```

Figure 7.1: The Syntax of the Mu-Calculus

f and Y define the function whose fixpoint is computed). Usually there is a further restriction that f be monotone in Y; we do not require this. We will return to the issue of monotonicity in the next section.

In Figure 7.1, we define the syntax of the Mu-Calculus (`ap` and `v` correspond to the set of atomic propositions and the set of variables, respectively). `Mu-symbolp` is used because we do not want to decide the meaning of formulae such as `'(mu + f)`.

Exercise 7.18 *Define* `translate-f`*, a function that allows us to write formulae in an extended language, by translating its input into the Mu-Calculus. The extended syntax contains* `AX` *(*`'(AX f)` *is an abbreviation for* `'(~ (EX (~ f)))`*) and the infix operators* `|` *(which abbreviates* `+`*),* `=>` *and* `->` *(both denote implication), and* `=`*,* `<->`*, and* `<=>` *(all of which denote equality).*

Exercise 7.19 `(Mu-calc-sentencep f ap)` *recognizes sentences (formulae with no free variables) in the extended syntax; define it.*

7.6 Mu-Calculus Semantics

The semantics of a Mu-Calculus formula is given with respect to a model and a valuation assigning a subset of the states to variables. The semantics of an atomic proposition is the set of states that satisfy the proposition. The semantics of a variable is its value under the valuation. Conjunctions, disjunctions, and negations correspond to intersections, unions, and complements, respectively. EXf is true at a state if the state has some successor that satisfies f. Finally, μ's and ν's correspond to least and greatest fixpoints, respectively. Note that the semantics of a sentence (a formula with no free variables) does not depend on the initial valuation. The formal definition is given in Figure 7.2; some auxiliary functions and abbreviations used in the figure follow.

```
(defabbrev semantics-EX (m f val)
  (image (mu-semantics m (second f) val)
         (inverse-relation m)))

(defabbrev semantics-NOT (m f val)
  (set-complement (mu-semantics m (second f) val)
                  (states m)))

(defabbrev semantics-AND (m f val)
  (intersect (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-OR (m f val)
  (set-union (mu-semantics m (first f) val)
             (mu-semantics m (third f) val)))

(defabbrev semantics-fix (m f val s)
  (compute-fix-point
   m (third f) (put-assoc-equal (second f) s val)
   (second f) (size m)))

(defabbrev semantics-MU (m f val)
  (semantics-fix m f val nil))

(defabbrev semantics-NU (m f val)
  (semantics-fix m f val (states m)))
```

Now, we are ready to define the main function:

```
(defun semantics (m f)
  (if (mu-calc-sentencep f (atomic-props m))
      (mu-semantics m (translate-f f) nil)
    "not a valid mu-calculus formula"))
```

```
(mutual-recursion
(defun mu-semantics (m f val)
  (cond ((eq f 'true) (states m))
        ((eq f 'false) nil)
        ((mu-symbolp f)
         (cond ((in f (atomic-props m))
                (value-of f (a-labeling m)))
               (t (value-of f val))))
        ((equal (len f) 2)
         (cond ((equal (first f) 'EX)
                (semantics-EX m f val))
               ((equal (first f) '~)
                (semantics-NOT m f val))))
        ((equal (len f) 3)
         (cond ((equal (second f) '&)
                (semantics-AND m f val))
               ((equal (second f) '+)
                (semantics-OR m f val))
               ((equal (first f) 'MU)
                (semantics-MU m f val))
               ((equal (first f) 'NU)
                (semantics-NU m f val))))))

(defun compute-fix-point (m f val y n)
  (if (zp n)
      (value-of y val)
    (let ((x (value-of y val))
          (new-x (mu-semantics m f val)))
      (if (== x new-x)
          x
        (compute-fix-point
         m f (put-assoc-equal y new-x val) y (- n 1))))))
          ; note that the valuation is updated
)
```

Figure 7.2: The Semantics of the Mu-Calculus

`Semantics` returns the set of states in `m` satisfying `f`, if `f` is a valid Mu-Calculus formula, otherwise, it returns an error string.

How would you write a Mu-Calculus formula that holds exactly in those states where it is possible to reach a p-state (*i.e.*, a state labeled by the atomic proposition p)? The idea is to start with p-states, then add states that can reach a p-state in one step, two steps, and so on. When you are adding states, this corresponds to a least fixpoint computation. A solution is $\mu Y(p \ \vee \ \texttt{EX}Y)$; it may help to think about "unrolling" the fixpoint.

How would you write a Mu-Calculus formula that holds exactly in those states where every reachable state is a p-state? The idea is to start with p-states, then remove states that can reach a non p-state in one step, two steps, and so on. When you are removing states, this corresponds to a greatest fixpoint computation. A solution is $\nu Y(p \ \wedge \ \neg\texttt{EX}\neg Y)$; as before it may help to think about unrolling the fixpoint. Similar exercises follow so that you can gain some experience with the Mu-Calculus.

Exercise 7.20 *For each case below, define a Mu-Calculus formula that holds exactly in states that satisfy the description. A* path *is a sequence of states such that adjacent states are related by the transition relation. A* fullpath *is a maximal path,* i.e., *a path that cannot be extended.*

1. *There is a fullpath whose every state is a p-state.*
2. *Along every fullpath, it is possible to reach a p-state.*
3. *There is a fullpath with an infinite number of p-states.*

The model-checking algorithm we presented is *global*, meaning that it returns the set of states satisfying a Mu-Calculus formula. Another approach is to use a *local* model-checking algorithm. The difference is that the local algorithm is also given as input a state and checks whether that particular state satisfies the formula; in some cases this can be done without exploring the entire structure, as is required with the global approach.

The model-checking algorithm we presented is *extensional*, meaning that it represents both the model and the sets of states it computes explicitly. If any of these structures gets too big—since a model is exponential in the size of the program text, *state explosion* is common—resource constraints will make the problem practically unsolvable. Symbolic model-checking [74, 16, 86] is a technique that has greatly extended the applicability of model-checking. The idea is to use compact representations of the model and of sets of states. This is done by using BDDs[3] (binary decision diagrams), which on many examples have been shown to represent states and

[3] BDDs can be thought of as deterministic finite state automata (see any book covering Automata Theory, *e.g.*, [50]). A Boolean function, f, of n variables can be thought of as a set of n-length strings over the alphabet $\{0,1\}$. We start by ordering the variables; in this way an n-length string over $\{0,1\}$ corresponds to an assignment of values to the

models very compactly [14]. Symbolic model-checking algorithms, even for temporal logics such as CTL whose expressive power compared with the Mu-Calculus is quite limited, are based on the algorithm we presented (except that BDDs are used to represent sets of states and models).

Now that we have written down the semantics of the Mu-Calculus in ACL2, we can decide to stop and declare success, because we have an executable model-checker. In many cases this is an appropriate response, because deciding if you wrote what you meant is not a formal question. However, in our case, we expect that `MU` formulae are least fixpoints (if the formulae are monotonic in the variable of the `MU` and certain "type" conditions hold), and similarly `NU` formulae are greatest fixpoints. We will check this. We start by defining what it means to be a fixpoint.

```
(defun fixpointp (m f val x s)
  (== (mu-semantics m f (put-assoc-equal x s val)) s))

(defun post-fixpointp (m f val x s)
  (=< (mu-semantics m f (put-assoc-equal x s val)) s))

(defun pre-fixpointp (m f val x s)
  (=< s (mu-semantics m f (put-assoc-equal x s val))))
```

Read the rest of the exercises in this section before trying to solve any of them.

Exercise 7.21 *Use encapsulation to constrain the functions* `sem-mon-f`*,* `good-model`*,* `good-val`*, and* `good-var` *so that* `sem-mon-f` *is monotone in* `good-var`*,* `good-model` *is a "reasonable" model,* `good-val` *is a "reasonable" valuation, and* `good-var` *is a "reasonable" variable.*

We prove the fixpoint theorems by functionally instantiating the main theorems in the supporting book `fixpoints`. (See <u>`lemma-instance`</u>; an example of functional instantiation can be found in the companion book [58].)

Exercise 7.22 *Prove that* `MU` *formulae are least fixpoints and that* `NU` *formulae are greatest fixpoints. As a hint, we include the statement of one of the four required theorems.*

---

variables. We can represent f by an automaton whose language is the set of strings that make f true. We can now use the results of automata theory, *e.g.*, deterministic automata can be minimized in $O(n \log n)$ time (the reason why nondeterministic automata are not used is that minimizing them is a PSPACE-complete problem), hence, we have a canonical representation of Boolean functions. Automata that correspond to Boolean functions have a simpler structure than general automata (*e.g.*, they do not have cycles); BDDs are a data structure that takes advantage of this structure. Sets of states as well as transition relations can be thought of as Boolean functions, so they too can be represented using BDDs. Finally, note that the order of the variables can make a big (exponential) difference in the size of the BDD corresponding to a Boolean function.

```
(defmu semmu-is-a-fixpoint
  (fixpointp (good-model) (sem-mon-f) (good-val) (good-var)
             (mu-semantics
              (good-model)
              (list 'mu (good-var) (sem-mon-f))
              (good-val)))
  sets::lfix-is-a-fixpoint)
```

Exercise 7.23 *The hint in the previous example is a macro call. This saves us from having to type the appropriate functional instantiation several times. Define the macro. Our solution is of the following form.*

```
(defmacro defmu (name thm fn-inst &rest args)
  `(defthm ,name ,thm
     :hints
     (("goal"
       :use (:functional-instance
             ,fn-inst
             (sets::S (lambda() (states (good-model))))
             (sets::f (lambda(y) (mu-semantics  ...  )))
             (sets::applyf
              (lambda(y n) (compute-fix-point  ...  )))
             (sets::cardinality cardinality)))
      ,@args)))
```

You will notice that reasoning about mutually recursive functions (which is required for the exercises above) can be tricky, *e.g.*, even admitting the mutually recursive functions and verifying their guards (as mentioned in the introduction, this is an implicit exercise for every function we introduce) can be a challenge. Read the documentation for mutual-recursion and package-reincarnation-import-restrictions. There are several approaches to dealing with mutually recursive functions in ACL2. One is to remove the mutual recursion by defining a recursive function that has an extra argument which is used as a flag to indicate which of the functions in the nest to execute. Another approach is to identify a sufficiently powerful induction scheme for the functions, add it as an induction rule (see induction) so that this induction is suggested where appropriate, and prove theorems by simultaneous induction, *i.e.*, prove theorems that are about all the functions in the mutual recursion nest. We suggest that you try both approaches.

7.7 Temporal Logic

Temporal logics can be classified as either *linear-time* or *branching-time* (see [28]). In linear-time logics the semantics of a program is the set of its possible executions, whereas in branching-time, the semantics of a program is its computation tree; therefore, branching time logics can distinguish between programs that linear-time logics consider identical. A branching time logic of interest is CTL: many model-checkers are written for it because of algorithmic considerations. We present the syntax and semantics of CTL. It turns out that CTL, as well as the propositional linear time logic LTL, and the branching time logic CTL^* can be translated to the Mu-Calculus.

The syntax of CTL is defined inductively by the following rules:

1. p, where p is an atomic proposition, and

2. $\neg f, f \vee g$, where f is a CTL formula, and

3. $\mathtt{EX}f, \mathtt{E}(f\mathtt{U}g), \mathtt{E}\neg(f\mathtt{U}g)$, where f and g are CTL formulae.

Although we presented the syntax of CTL, it turns out to be just as easy to present the semantics of what is essentially CTL^*. The semantics are given with respect to a *fullpath*, *i.e.*, an infinite path through the model. If x is a fullpath, then by x_i we denote the i^{th} element of x and by x^i we denote the suffix $\langle x_i, \ldots \rangle$. Henceforth, we assume that the transition relation of models is *left total*, *i.e.*, every state has a successor. Note that CTL formulae are *state formulae*, *i.e.*, formulae whose semantics depends only on the first state of the fullpath. $M, x \models f$ means that fullpath x of model M satisfies formula f.

1. $M, x \models p$ iff x_0 is labeled with p;

2. $M, x \models \neg f$ iff not $M, x \models f$;
   $M, x \models f \vee g$ iff $M, x \models f$ or $M, x \models g$;

3. $M, x \models \mathtt{E}f$ iff there is a fullpath $y = \langle x_0, \ldots \rangle$ in M s.t. $M, y \models f$;
   $M, x \models \mathtt{X}f$ iff $M, x^1 \models f$; and
   $M, x \models f\mathtt{U}g$ iff there exists $i \in \mathbb{N}$ s.t. $M, x^i \models g$ and for all $j < i,\ M, x^j \models f$.

The first two items above correspond to Boolean formulae built out of atomic propositions. $\mathtt{E}f$ is true at a state if there exists a fullpath from the state that satisfies f. A fullpath satisfies $\mathtt{X}f$ if in one step (next time), the fullpath satisfies f. A fullpath satisfies $f\mathtt{U}g$ if g holds at some point on the fullpath and f holds until then.

The following abbreviations are useful:

$\mathtt{A}f = \neg\mathtt{E}\neg f,\ \ \mathtt{F}g = \mathsf{true}\,\mathtt{U}g,\ \ \mathtt{G}f = \neg\mathtt{F}\neg f$

$\mathtt{A}f$ is true at a state if every fullpath from the state satisfies f. A fullpath satisfies $\mathtt{F}g$ if eventually g holds on the path. A fullpath satisfies $\mathtt{G}f$ if f holds everywhere on the path.

Exercise 7.24 *Translate the following state formulae into Mu-Calculus formulae (the penultimate formula is not a CTL formula, but is a CTL* formula which you can think of as saying "there exists a path such that infinitely often p"):* $\mathtt{EF}p, \mathtt{AF}p, \mathtt{AG}p, \mathtt{EG}p, \mathtt{EGF}p$*, and* $\mathtt{EGEF}p$.

Exercise 7.25 *Define a translator that translates CTL formulae (where the abbreviations above, as well as* true *and* false *are allowed) into the Mu-Calculus.*

7.8 Conclusions

We gave a formal introduction to model-checking via the Mu-Calculus, but only scratched the surface. We conclude by listing some of the many interesting directions one can explore from here. One can define a programming language so that models can be described in a more convenient way. One can make the algorithm symbolic, by using BDDs instead of our explicit representation. One can define the semantics of a temporal logic (*e.g.*, CTL^*) in ACL2 and prove the correctness of the translation from the temporal logic to the Mu-Calculus. One can use monotonicity arguments and memoization to make the model-checking algorithms faster. Finally, one can verify that the optimizations suggested above preserve the semantics of the Mu-Calculus.