2

Overview

This book is about formal reasoning, with particular emphasis on reasoning about computing systems. The dream is that we should be able to write down mathematically precise conjectures about special-purpose hardware designs, microprocessors, microcode programs, compilers, algorithms, etc., and then to prove them with a mechanized reasoning system. This dream is realizable. This book is about how to use a tool that is being used today to prove such conjectures. But more generally, the book is about how to formalize systems, how to express their properties, and how to decompose the proofs of those properties into manageable steps.

2.1 Reasoning

The dream of machines that reason has been around a long time. Leibniz expressed it this way.

> If we had some exact language ... or at least a kind of truly philosophic writing, in which the ideas were reduced to a kind of alphabet of human thought, then all that follows rationally from what is given could be found by a kind of calculus, just as arithmetical or geometrical problems are solved. — Leibniz (1646–1716)

By the end of the nineteenth century Boole, Frege, Peano, and others had clearly established the foundations of what we now call *symbolic logic*. Leibniz's dream of a calculus of human thought was apparently coming true. Bits of the mathematical world, *e.g.*, propositional calculus and elementary arithmetic, were described formally by axioms written as formulas in a fixed syntax. Rules were proposed for generating new formulas from old formulas—rules with the property that the new formulas are true if the old formulas are.

In the hands of such giants as Russell, Whitehead, Skolem, Herbrand, and Hilbert it looked as though symbolic logic could place all of mathematics on a firm foundation. Then, in 1931, Gödel showed the limitations of formal logic: given any sufficiently powerful formal logic there are truths that cannot be proved in the logic.

Gödel's theorem might be thought of as sealing the fate of Leibniz's dream. But does its inability to deduce all truths really justify the abandonment of formality? Not at all. If formality allows us to deduce some truths about new ideas, it is a useful tool. There is a far more practical obstacle to the construction of a calculus of human thought: the nature of the physical world and its role in what we think about.

Few human artifacts or endeavors, beyond mathematics itself, can be so accurately described by axioms that we would profess to believe any consequence of those axioms no matter how deep. Too many aspects of the physical world are vague, random, or governed by processes best described quantitatively with continuous mathematics like differential equations. Formal logic is good for describing abstract symbolic systems—worlds inhabited by discrete objects manipulated by clearly specified rules determining the behavior of the system. For most of the twentieth century the majority of work in formal logic was the study of formal logic itself.

Much of that changed with the invention of the digital computer. Circuits, processors, programming languages, programs, networks, protocols, formats, ... , all are abstract symbolic systems. Such systems are fruitfully described and studied with symbolic logic.

2.2 Philosophic Writing

What kind of "philosophic writing," to use Leibniz's term, is appropriate for describing computing systems? Our choice is, in essence, a fairly conventional functional programming language. This book is about that language, how to use it to describe computing systems, how to reason in the language, and how to use a certain computer program to help you.

The name of the language, and of the reasoning engine for it, is "ACL2." It stands for "A Computational Logic for Applicative Common Lisp." The ACL2 language is a variant of Common Lisp. But instead of just providing an execution engine for the language, we describe the language as a mathematical logic, with axioms and rules of inference. And we provide a reasoning engine. Thus, in addition to being able to execute your programs, you can use the reasoning engine to prove things about them.

For example, here is a Lisp program to reverse a list.[1] It is not important that you understand this definition now. But we will use it just to talk about Lisp and logic.

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

[1] ACL2 provides a function named `reverse` but it is not defined the way `reverse` is defined here.

This `defun` command defines a function named `reverse`, with one formal parameter, named `x`. The body of the definition is an expression, which is evaluated to determine the value of the function. In this case, the body is an `if`-expression. The `if` has three arguments, a test, a true branch, and a false branch, each of which is an expression. In this case, the test is `(consp x)`, the true branch is an `append` expression, and the false branch is the constant expression `'nil`. As you can see, Lisp expressions are either variables, constants, or function applications. To apply a function f to the values of some expressions, e_i, we write (f e_1 ... e_n), instead of the more traditional $f(e_1, \ldots, e_n)$.

The function defined above reverses its argument. More precisely, if applied to a linear list it returns a linear list with the elements in the opposite order. For example, to apply the function to the list `(5 6 7)` we could evaluate `(reverse '(5 6 7))`. `(5 6 7)`, by the way, is a list with three elements. The first is the number `5`, the second is the number `6`, and the third is the number `7`. The little "single quote" mark before it in the `reverse` expression is the way we write the literal constant whose value is the list `(5 6 7)`.

If you calculate the value of `(reverse '(5 6 7))` using the definition of `reverse` above the result is `(7 6 5)`. To compute the reverse of `(5 6 7)`, `reverse` recursively reverses `(6 7)`, obtaining `(7 6)`, and then concatenates it, with the function `append`, to the list `(5)`, producing `(7 6 5)`.

Here is a conjecture about our reverse function.

```
(equal (reverse (reverse x)) x)
```

Informally, this conjecture says that the reverse of the reverse of `x` is `x`. More formally, the conjecture says that, for any value of the variable `x`, `(reverse (reverse x))` and `x` evaluate to `equal` things.

You could test this expression in any Lisp (where the relevant functions are defined) and you would find that the value of the expression is often `t`.

But you can test the expression only on a finite number of cases. Will the expression always evaluate to `t`? One way to try to establish that is to *prove* it, *i.e.*, give a careful argument based on the definition of `reverse` and properties of other functions. We are not prepared to give such an argument here—we have not told you the relevant properties of all the symbols we use. But an attempt to construct a proof will lead you to realize that the conjecture is not always true!

However, a closely related formula can be proved. The related formula says that the reverse of the reverse of `x` is `x`, provided `x` is a list.[2] You might object and say, "Of course we meant that `x` was a list! Why else would we speak of reversing it?" But an aspect of formality is that one must say

[2] Here by "list" we actually mean the concept formalized by the predicate `true-listp`.

what one means and leave no hidden assumptions.[3] One consequence of proving a formula is that we know that its value, under any assignment of objects to its variables, will always be true.

Not only can the new formula be proved, but the ACL2 reasoning engine, or theorem prover, can prove it automatically.

It is good that the theorem prover can prove the "`reverse reverse`" theorem automatically because this theorem is absurdly simple compared to conjectures that are likely to be of interest to you. It is a long way from the definition of `reverse` to the definition of a microprocessor or programming language.

The bad news is that the `reverse reverse` theorem is pretty complicated from the theorem prover's perspective. In fact, this theorem is a perfect example of the mileage we humans get out of informal reasoning. Calling this concept "reverse" loads it with intuitive meaning that is not explicit in the definitional equation. The obviousness of the claim that reversing a list twice produces the original list probably owes more to the name "reverse" and experience with the physical world than it does to the definitional equation of `reverse`.

Suppose we introduced the concept with

```
(defun fn123 (x)
  (fn1 (fn2 x)
       (fn122 (fn123 (fn3 x)) (m8 (fn4 x)))
       'nil)),
```

where `fn1`, `fn2`, `fn3`, etc., are all described by their own equations. How impressive is it to discover or prove that "the `fn123` of the `fn123` of `x` is `x`, when `x` is a `fn98`?"

Whether we give our functions sensible names or not, their properties are derived from the axioms we write down about them, not the names we give them. The same is true of our circuits, components, programs, etc. Just because we call a component a "divider" does not mean it computes the quotient of one number by another! We tend to give our creations sensible names and we use those names to help us reason informally. But because those names mean more to us than to any mechanized reasoning engine, it is often necessary to tell the engine "obvious" properties. Sometimes when the engine fails to prove these "obvious" properties, we come to realize they are actually false!

2.3 System Models

To prove something it must first be expressed as a statement in the formal syntax. This means you cannot prove that a circuit, a microprocessor, or

[3] Syntactic typing sometimes makes it more convenient to express such assumptions. But ACL2 does not have syntactic typing.

any other physical artifact is correct. You can only prove that some model of it has a certain property. It is up to you to use the logic to create a model of the artifact in question and then to write down conjectures that express your understanding of correctness—conjectures that you believe, or at least hope, to be true. One of the main thrusts of this book is to teach you how to use the ACL2 logic to model digital artifacts. We sketch several models in this chapter.

Once you have defined a model of the artifact, you must express, in the formal syntax, the conjecture you wish to prove. Even this can be hard. Theorems proved informally are often never expressed as formulas! For example, "theorems" that otherwise might look like formulas often contain ellipses (...). Ellipses are usually an IQ test: can you guess what the author has in mind? Such informal notation must be made precise, which often requires defining additional concepts. Another common pitfall in converting from an informal statement to a formula you can prove is identifying all the hidden or implicit assumptions.

Sometimes the informal statement uses familiar mathematical concepts that are not yet defined in the logic you are using. The public distribution of ACL2 includes many online "books" that contain definitions and theorems about integer and rational arithmetic, finite sets, tables, and vectors. But we have only scratched the surface in formalizing the many operations and relations on these concepts. Often you will find yourself defining familiar concepts just to state your goal.

Sometimes you may be unable to define the concepts of interest. The ACL2 logic is in some ways quite restrictive, lacking for example infinite sets and higher order functions. However, more often than you might think—especially if you have heretofore used higher order logic or set theory—this problem can be overcome without undue violence to the intuitions you are trying to capture. Perhaps some formalizable concept can be substituted without weakening the meaning of the statement. But if not, you may need to choose a different mechanically supported formalism such as HOL [16] or PVS [34, 13].

The activity of casting your problem into the formal syntax of some logic is called *formalization*. One of the things you will learn in this book is how to use ACL2 to model systems and formalize their properties.

Consider the following ACL2 formula.

```
(implies (and (floating-point-numberp p 15 64)
              (floating-point-numberp d 15 64)
              (not (equal d 0))
              (rounding-modep mode))
         (equal (divide p d mode)
                (round (/ p d) mode)))
```

We do not give the definitions of the concepts here, but, intuitively, the formula may be read as follows. If `p` and `d` are floating point numbers with 15 bits of exponent and 64 bits of significand, `d` is not 0, and `mode` is a rounding mode, then the output of the function `divide` on `p`, `d`, and `mode` is equal to the infinitely precise quotient of `p` divided by `d`, rounded according to `mode`.

The notions of `implies`, `and`, `not`, and `equal` are built into ACL2 and are fairly standard formalizations of implication, conjunction, logical negation, and equality. The function symbol '`/`' denotes the arithmetic operation of division of one rational number by another. It too is built into ACL2 and is axiomatized to satisfy the familiar properties of division in the rational field.

The other function symbols used in this formula were defined expressly for the purpose of stating this theorem. The question of whether they accurately capture the alleged intuitive notions is of great interest and can be difficult to settle. A careful reading of the IEEE floating point standard and its comparison with the definitions of `floating-point-numberp`, `rounding-modep`, and `round` gives some confidence that these functions capture the intended concepts.

The definition of the function `divide` was constructed, by hand, by studying the microcode implementing division on the AMD-K5 microprocessor of Advanced Micro Devices, Inc. (AMD[4]). The ACL2 function `divide` computes its final answer using a certain sequence of fixed format floating point additions and multiplications (and a few other operations). By talking to the designers of the microcode and of the AMD-K5 floating-point unit, we convinced ourselves and them that the definition of `divide` accurately captured the algorithm used by their microcode [33].

Using ACL2 we proved that the formula above, with all the definitions mentioned, is a theorem.

We therefore think of the theorem as establishing the correctness of the floating point division algorithm on the AMD-K5 microprocessor. But, of course, that characterization of the theorem is an informal one.

It is possible to state a theorem about the microcode itself—a certain fixed sequence of numbers representing instructions interpreted by the microcode engine. To state that theorem in ACL2 we would have to model the microcode engine as an ACL2 function that interprets microcode instructions and changes some state components accordingly. Such "microcode engine models" have been defined in ACL2 for other microprocessors, *e.g.*, the Motorola CAP digital signal processor [10], and "code proofs" have been done with ACL2. But that approach was not taken in the AMD-K5 division work, even though it arguably would have provided a more convincing model to study.

The reason is instructive. At the time the theorem above was formulated and proved, AMD was more concerned about the division algorithm than its expression in microcode. Furthermore, time was short because the AMD-

[4]AMD, the AMD logo and combinations thereof, AMD-K5, AMD-K7, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

K5 fabrication date was looming. The moral is: economic realities have a significant impact on formal modeling. Models must address the issues of concern and allow the analysis to be carried out in a timely fashion.

At the time of this writing, the AMD-K5 is two generations old. AMD has since released the AMD-K7^{TM} (or AMD Athlon^{TM}) microprocessor. What of ACL2's role in its development? ACL2 was used by David Russinoff to verify the AMD Athlon processor implementations of floating-point addition, subtraction, multiplication, division, and square root [35]. The implementations were done in hardware, not microcode, so no microcode engine was formalized. But the AMD hardware was described in a Verilog-like hardware description language referred to simply as "the RTL language." In the ACL2 work on the AMD Athlon processor, the RTL source was mechanically translated into ACL2 (using a translator written by Art Flatau). The correctness of the translation was not proved, because no formal semantics for the RTL was available. But the translation was tested: over 80 million floating-point test vectors were run through the ACL2 models and the results were compared to those computed by AMD's RTL simulator. The answers were identical. This highlights the value of ACL2's executability. It also highlights the inadequacy of testing: when correctness proofs were undertaken, bugs were found in the ACL2 models (and, more importantly, in the original RTL). These bugs were not exposed by the test suite. The bugs were fixed by changing the RTL, new models were mechanically produced, and those models were mechanically verified using ACL2. Russinoff and Flatau discuss their work on this in the companion book [22].

The point of this discussion is to drive home the fact that models of artifacts of commercial interest are not constructed in an ivory tower. They are constructed in the midst of a rapidly evolving design and implementation project. Such formal models may look *ad hoc* or incomplete. Instead of iterating on a given project and producing a *tour de force* model that answers every objection, a new project is undertaken where the issues are often completely different.

2.4 Truth and Proofs

Why prove formulas? The reason is that proof is a way to establish truth. A *theorem* is either an axiom or a formula produced by applying some rule of inference to other theorems. A *proof* is a finite structure showing the derivation of a theorem from the axioms.

But (most) ACL2 formulas can be executed. If you select concrete values for the variables and then evaluate a formula, some concrete value will be computed. (We here imagine an unbounded amount of memory.) Theorems always evaluate to true.[5] The basic argument for this fundamental property of theorems is that (a) the axioms always evaluate to true and (b) the rules of inference preserve this property. Since a theorem is derived by applying a finite number of rules of inference to axioms, theorems must always evaluate to true.

But there are an infinite number of possible assignments to the variables of any formula containing a variable. So something wonderful has happened: We can determine that a formula will evaluate to true on the infinite number of possible cases by constructing a finite proof of it! That is why we prove formulas.

To use formal proofs to check your reasoning you must somehow translate your informal argument into a formal one. Informal proofs often involve important labor-saving metatheorems or derived rules of inference. Typical phrases suggesting the use of such rules are "without loss of generality we restrict our attention to," "by symmetry it suffices to prove," and "the other cases are analogous."

Suppose then that you have an informal proof of some formal statement. Imagine that you begin to present the proof to a colleague. Some of the steps in your proof are sufficiently small that your colleague follows them. But other steps are challenging and he or she interrupts and asks how you got from one formula to the next. When this happens you expand your explanation, perhaps discovering details that you had previously overlooked. A formal proof would be created if your colleague simply asked "how?" on every step requiring more than the most basic logical transformation.

2.5 The Reasoning Engine

The ACL2 theorem prover was designed for the kind of interaction sketched above. The theorem prover attempts to fill in every gap in your alleged proof of a formula. When successful, the computation done by the theorem prover is intended to guarantee that a formal proof of the formula exists, although you never actually see this formal proof. (Nevertheless, we often refer to ACL2's computation as its "proof" or "proof attempt.") When a gap is too big for the theorem prover to follow, you must break that step into smaller pieces.

As depicted in Figure 2.1, the theorem prover takes input from both you and a data base, called the *logical world* or simply <u>`world`</u>[6]. One view of the world is that it contains axioms, definitions, and previously proved theorems. But a more effective view of the world is that it embodies a theorem proving strategy, developed by you and codified into *rules* that direct certain aspects of the theorem prover's behavior. When trying to

[5] To be more precise, instances of theorems evaluate to non-`nil`.

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

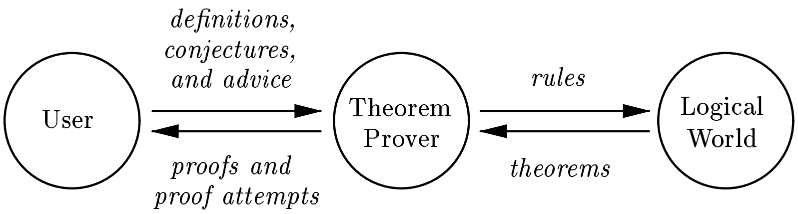

Figure 2.1: Data Flow in the Theorem Prover

prove a theorem, the theorem prover applies that strategy and prints its proof attempt. You can interrupt and abort or let the attempt continue. The theorem prover may eventually stop, in which case it will have either succeeded in proving the formula or failed to prove the formula. When it succeeds, your theorem is converted into a rule, using advice from you, and this new rule changes the theorem prover's future strategy. When it fails, you must either change the alleged theorem or give additional guidance to the theorem prover, often by inspecting its proof attempt to see what went wrong.

Note that the rules that control the theorem prover's behavior come from the logical world.

- ♦ Except by adding axioms, *you cannot directly add rules to the logical world.*

- ♦ *It is important to think about how your theorems will be interpreted as rules, if you want them used automatically.*

This design may at first frustrate you: you will feel powerless. But it relieves you of the burden of getting the rules logically correct! The theorem prover takes that responsibility.

With some experience you will feel empowered by this design. Why? Because you will see the theorem prover apply your strategy to prove theorems other than the ones for which you first designed it. However, you will also feel challenged to codify good strategies.

2.6 Effort Involved

How hard is it to use ACL2 to prove theorems of commercial interest? Modeling "the product" and capturing one or more important properties characterizing its "correctness" are the first and often hardest steps. These steps usually involve talking to the implementors, reading design documents, and experimenting with "toy models." You will often find that the implementors approach the project very differently than you do. They may not have in mind a correctness criterion that can be expressed as a property you can formalize. Their notion of correctness is often simply the absence of "bugs." Bugs, like good art, can evidently be recognized when seen. But remember: the implementors are probably very good at their jobs, and are grappling with other problems, like power consumption, timing, production costs, time-to-market, etc. Furthermore, it is not by dumb luck that they have built things that you buy and use and rely on. We discuss the "sociology" of typical projects in Part I of [22].

As you get more familiar with the product you will find yourself thinking of it more abstractly. General properties will occur to you. In conversation with the team members you will learn whether these are intended to hold—or, more likely, under what conditions and at what level of abstraction they are intended to hold. Identifying and peeling apart co-mingled layers of abstraction may well become a big part of your job. Just being able to provide labels for the abstractions and a vocabulary to discuss the concepts unambiguously is of great benefit to the team.

Your experimentation with "toy models"—models that intentionally omit many features—is very important. The idea is to have a model that is so small you can try one modeling approach today, abandon it as impractical on your way home, and have a new approach up and running the next day. Such models allow you to develop the formalization of the basic ideas and properties, establish clear communication with the implementors, experiment with proof methods, and develop strategies for elaborating the models to interesting levels of complexity while only incrementally affecting proof complexity. See Moore's case study in [22]. Indeed, there are several more elaborate "toys" among the case studies in [22] (*e.g.*, the pipelined machine of Sawada, the floating point multiplier of Russinoff and Flatau, and the microprocessor of Greve, Wilding, and Hardin). These "toys" present the basic problems seen in applications of much more complexity.

In successful projects, you will eventually form a productive working relationship with the implementors, one based on mutual respect for the talents of "both sides." It is likely that you will produce a sequence of ever more elaborate models and the conjectured properties of these models and relations between them will become the goal theorems. Your models will evolve with the product design and theorems proved about one model will have to be "proved again" after modifying the model. "Proof maintenance" is an important aspect of ACL2's design. The ACL2 data base contains *strategies* for finding proofs, not individual proofs. Often, your old strategies will succeed on closely related new theorems. ACL2's ability to execute your evolving models, *i.e.*, use them as simulators, is also valuable. We describe some typical models and properties in Part I of [22].

You might start the project thinking that you are going to prove a particular theorem and be done. More likely, the "proof phase" lasts as long as the project and provides a continuing clarification and confirmation of some properties. The immediate impact is a better product. The legacy, however, is a way of thinking, together with a set of concepts made precise and embodied in a collection of ACL2 books that can be used on the next project.

2.7 Requirements on the User

To use ACL2 to model a system and prove properties of the model, you must understand the system being modeled. In addition, you must understand

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

We discuss these issues in this book. The book contains many exercises to help you develop these skills.

ACL2 will help construct the proof but its primary role is to prevent logical mistakes. The creative burden—the mathematical insight into why the model has the desired property—is your responsibility.

The "typical" ACL2 user

- has a bachelor's degree in computer science or mathematics,
- has some experience with formal methods,
- has had some exposure to Lisp programming and is comfortable with Lisp notation (reading some part of this book should suffice),
- is familiar with and has unlimited access to a Common Lisp host processor, operating system, and text editor (we use Gnu Emacs on Sun workstations and PCs running Unix and Linux),
- is willing to do some of the exercises in this book, and
- is willing to consult the ACL2 online documentation.

How long does it take for a novice to become an effective ACL2 user? Let us first answer a different question. 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 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 noncontroversial 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.