How To Prove Theorems Formally

Matt Kaufmann[1] and J Strother Moore[2]

[1] Advanced Micro Devices, Inc., 5900 East Ben White Blvd., Austin, TX 78741,
matt.kaufmann@amd.com
[2] Department of Computer Sciences, University of Texas at Austin,
Taylor Hall 2.124, Austin, Texas 78712,
moore@cs.utexas.edu
August 5, 2005

1 Abstract

Today it is impractical to prove – formally and mechanically – the correctness of entire computing systems of commercial interest. There are many reasons for this, both technical and economic. Mechanized theorem proving is nevertheless relevant in commercial hardware and software production. But practical considerations require that we focus our attention on problems that are both technically feasible within the time limits available and of interest to system designers.

Why might designers turn to a mechanical theorem prover? Because the problems – even the little ones – are so complicated they do not have confidence that their reasoning is sound. Formal, mechanized reasoning is crucial.

In this paper we briefly will describe several such "little theorems," that is, theorems that address issues of concern to designers without trying to address the complete correctness of the system. The theorems have all been formalized and proved with the ACL2 theorem prover. "ACL2" stands for A Computational Logic for Applicative Common Lisp. It is a theorem prover in the Boyer-Moore tradition that uses rewriting, decision procedures, mathematical induction and many other proof techniques to prove theorems in a first-order mathematical theory of recursively defined functions and inductively constructed objects [6].

However, these descriptions are just motivational. The real purpose of this paper is to answer the question how does one construct and manage large mechanically checked proofs (in ACL2)? After mention of the big industrial examples, we turn our attention to truly simple formal theorems about list processing and develop some advice to the reader. Most of this advice is meant to be helpful no matter what mechanized system or mathematical logic you are using.

The paper contains exercises. To learn how to do proofs, it is crucial that you work the exercises. The ACL2 system is available without charge on the net; see the ACL2 home page [8].

However, it is not necessary to use the ACL2 system to do the exercises. They can be done with pencil and paper or with other mechanical theorem proving systems with which you might be familiar. But the only way to learn how to do proofs is to do proofs!

Answers to the exercises are available on the web. See `http://www.cs-.utexas.edu/users/moore/publications/how-to-prove-thms/index.html`.

Early exercises ask the reader to define some functions used in challenge theorems in later exercises. To ensure those challenges are understood, we have included in Appendix A our definitions of all functions mentioned in any challenge theorem.

Keywords: ACL2, hardware verification, software verification, formal verification, theorem proving, automated reasoning.

2 Background

"ACL2" is the name of a functional programming language (based on Common Lisp), a first-order mathematical logic, and a mechanical theorem prover. ACL2, which is sometimes called an "industrial strength version of the Boyer-Moore system," is the product of Kaufmann and Moore, with many early design contributions by Boyer. It has been used for a variety of important formal methods projects of industrial and commercial interest, including:

- verification that the register-transfer level description of the AMD Athlon^{TM} processor's elementary floating point arithmetic circuitry implements the IEEE floating point standard [14, 15]; similar work has been done for components of the AMD-K5 processor [13], the IBM Power 4 [16], and the AMD Opteron^{TM} processor.[1]
- verification that a microarchitectural model of a Motorola digital signal processor (DSP) implements a given microcode engine [1] and verification that specific microcode extracted from the ROM implements certain DSP algorithms [2];
- verification that microcode for the Rockwell Collins AAMP7 implements a given security policy having to do with process separation [3];
- verification that the JVM bytecode produced by the Sun compiler `javac` on certain simple Java classes implements the claimed functionality [12] and the verification of properties of importance to the Sun bytecode verifier as described in JSR-139 for J2ME JVMs [10];
- verification of the soundness and completeness of a Lisp implementation of a BDD package that has achieved runtime speeds of about 60% those of the CUDD package (however, unlike CUDD, the verified package does not support dynamic variable reordering and is thus more limited in scope) [17];
- verification of the soundness of a Lisp program that checks the proofs produced by the Ivy theorem prover from Argonne National Labs; Ivy proofs may thus be generated by unverified code but confirmed to be proofs by a verified Lisp function [11].

[1] AMD, the AMD logo, AMD Athlon, AMD Opteron, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

Other applications are described in [5] and in the papers distributed as part of the periodic ACL2 workshops, the proceedings of which may be found via the Workshops link on the ACL2 home page [8].

As these examples demonstrate, it is possible to construct mechanically checked proofs of properties of great interest in industrial hardware and software designers. The properties proved are typically not complete characterizations of the correctness of the systems studied. For example, the proofs about the AMD microprocessors – the AMD-K5 processor, the AMD Athlon processor, and the AMD Opteron processor – just deal with the IEEE compliance of certain floating point operations modeled at the register transfer level. The microprocessors contain many unverified components and the verified ones could fail due to violations of their input conditions.

Nevertheless, these theorems were proved for good reason: the designers were concerned about their designs. Aspects of these designs are quite subtle or complicated and formal specification and mechanized proof offer the most complete way to relieve the concerns that something critical to correct functionality had been overlooked in the designs.

In addition to being interesting, these theorems are hard to prove. That is a relative judgment of course. Compared to longstanding open problems, these theorems are all trivial. But by many measures each of these proofs is much more complicated than any proof ever encountered by most readers. For example, the IEEE compliance proof for the floating point division microcode for the AMD-K5 processor (in 1995) required the formal statement and proof of approximately 1,200 lemmas. Subsequent AMD floating-point proofs are harder to measure because they build on libraries of lemmas that have been accumulating since 1995. The correspondence result between the Motorola DSP microarchitecture and its microcode engine involved intermediate formulas that, when printed, consumed 25 megabytes (approximately 5000 pages of densely packed text) *per formula.* And the proof involved hundreds of such formulas. The correctness argument for one particular DSP microcode program required an extremely subtle generalization that took many days for the author to craft. The formal model of the Connected Limited Device Configuration (CLDC) JVM and bytecode verifier is almost 700 pages of densely packed text. The proof that a simple Java class, which spawns an unbounded number of threads, produces a monotonic increase in the value of a certain shared counter produces about 19,000 subgoals and requires about 84 megabytes to print.

In these senses, the theorems in which we are interested are little (but hard) theorems about big systems, or put another way, they are valuable and non-trivial theorems about parts of very complicated systems.

How do we prove theorems like this? There is no mystery. We prove theorems like this the same way we prove simple theorems: by properly defining the concepts and carefully stating the theorem, by separating concerns, by appropriate decomposition of the proof into more general lemmas, and by the recursive application of the same methodology. But to do it on a grand scale takes more than the usual attention to detail and good taste. Minor misjudgments that are tolerable in small proofs are blown out of proportion in big ones. Unnecessarily complicated function definitions or messy, hand-guided proofs are things that can be tolerated in small projects without endangering success; but in large projects, such things can doom of the proof effort.

If you aim to produce big proofs, it pays to learn how to produce small ones well.

3 The Mechanics of Using the System

You do not need to use the ACL2 system to learn from this paper. This paper is primarily about how to state and prove theorems in a particular formal mathematical logic. Students do not often do that! Students studying mathematical logic mostly read about *meta*-theorems proved *about* formal logics, e.g., completeness, undecidability, etc. Students studying mathematics and computer science see "theorems" stated *informally* (in a mixture of English and mathematical notation) and "prove" them *informally.*

We study theorems and proofs in a rigorous setting. Expressing every claim *formally* – as a formula in a well-defined syntax – takes some getting used to! Proving them rigorously – justifying every step with formulas and appeals to rules of inference – takes even more! Our logic is supported by a mechanized theorem prover that prints out its proofs in an informal style that should be familiar to most mathematically literate readers. But when the theorem prover fails and you are called upon to help it, that help almost always takes the form of a formula or a hint to use a certain formula. Thus, to use the theorem prover effectively you must learn to think with formulas.

You may read this paper in conjunction with the ACL2 system to learn both how to express your ideas in formulas and how to use the system. We recommend reading this section even if you do not intend to use the ACL2 system; at least it will give you a sense of what is involved with producing large mechanically checked proofs.

If you want to use the system, first install it on your machine (if it is not already there), following the instructions in the Obtaining and Installing link of the ACL2 home page [8].

Next, learn to navigate The User's Manual linked to from the home page. In particular, learn to use the Index. For example, find and read the documentation under `STARTUP`. Try the SEARCH link near the top of the home page.

Throughout the rest of this document, when we say "see *name*," we mean find and read the documentation of *name* in the online documentation.

Most users use ACL2 from within Emacs. See Appendix B for a few helpful notes. Within Emacs, create a shell buffer (by typing `meta-x shell enter`). Typically, the buffer created is named `*shell*`. Lines typed into the bottom of the `*shell*` are fed as input to a Linux or other operating system "shell" process running under Emacs; output is streamed into the `*shell*` buffer. In the `*shell*` buffer invoke ACL2. Thus, the `*shell*` buffer becomes an editable and searchable log of all your transactions with ACL2.

ACL2 presents itself as a read-eval-print loop, with the prompt `ACL2 !>`. Whatever you type after this prompt is read by ACL2, evaluated, and the results are printed. Then you are prompted for another input. However, it is rare that you will type directly to the ACL2 loop. Instead, we create our definitions and proof plans in another buffer and submit commands from that buffer to ACL2.

So create another buffer, which here we call `script`, by typing `ctrl-x b script enter`. In this buffer you will write down your proposed proof scripts. It will typically consist of a sequence of definitions and lemmas, concluding with your main theorem. These scripts typically contain many Lisp comments, preceded on each line by a semi-colon. After typing the initial draft of the problem and its solution, position the Emacs cursor at the top of the `script` buffer.

We maintain the following invariant between the `script` buffer and the `*shell*` buffer: every command in the `script` preceding the cursor has been successfully executed in the `*shell*` (in the same order) and those are the only commands that have been executed. That is, the cursor of the `script` buffer defines a "barrier" between what has been done and what is left to do. The command immediately after the cursor is the "next command" to try.

You should learn how to copy that command into the `*shell*` buffer, at the bottom, and to advance the `script` buffer cursor past the command. This is how you will submit previously prepared ACL2 forms to ACL2.

Before continuing we answer two commonly asked questions. First Question: Why are we bothering to show Emacs in a discussion of how to use a mechanical theorem prover? Answer: It serves as a warning. Don't aspire to prove big theorems mechanically unless you are prepared to use a variety of sophisticated text processing tools. When you have to inspect multi-megabyte formulas, you will be happy to know of the existence of Emacs commands like `meta-1 meta-x compare-windows` (which compares two windows containing s-expressions, ignoring differences in prettyprinting style).

Second Question: Why don't we implement an Emacs/ACL2 interface instead of recommending that the user learn Emacs? Answer: We have learned that users evolve their own styles, both for ACL2 and for Emacs. Using somebody else's style is often cumbersome. We are describing our style. Use it as a starting point but understand how personal it is and how flexible Emacs is.

Having prepared the initial draft of the `script`, we submit commands successively. If a command is successful (i.e., the definition is admitted or the theorem is proved) we submit the next command. If the command is unsuccessful, we move the `script` cursor back in front of the failed command and inspect the output of ACL2 in the `*shell*` buffer.

Typically, one of two things must be done. Either the command is faulty (e.g., syntactically ill-formed) and must be edited, or additional lemmas must be proved before it can be successfully submitted. Learning how to read ACL2 output and determine what to do is the biggest task facing the new user because it is usually tantamount to the theorem proving problem: how can this theorem be decomposed into provable lemmas? Ultimately, that is what this paper is about.

But suppose we have determined, somehow, that the appropriate response to the failed command, γ, is to prove two new lemmas, say α and β. Then we type them into the `script` buffer, one after the other, immediately in front of γ. Then we position the cursor in front of α and resume our iterative submission process.

Note that if α fails, the process just described will lead us to begin working on the subtask of how to prove α, by inserting additional lemmas in front of it. When we ultimately succeed in proving them and α, the cursor will have moved just past α and be in front of β. Only after β is proved will we confront γ again, and by then our proof plan for it, namely α and β, will be in place. Had we not written α and β down when we analyzed the failure of γ and used the `script` buffer in a disciplined way, we might well have forgotten about β and had to re-analyze the failure of γ to re-discover the need for β.

There is no guarantee here that our plan for proving γ will succeed. If it doesn't, this method will cause us to insert additional lemmas for γ just after β. A modular top-down proof development methodology, in which we check and guarantee that α and β permit the proof of γ before descending to prove them, is described in [4].

We regularly save our work, by saving the `script` to a file, say `script.lisp`. This allows us to recreate our state, should we wish to quit for the time being and resume later. We typically put the command `(i-am-here)` into the `script` buffer at the cursor, just to mark the current location of the barrier. When `(i-am-here)` is executed it causes an error.

When we are ready to resume our work, we start a new ACL2 in the `*shell*` buffer (if the old one has been lost) and type:

```
(ld "script.lisp" :ld-pre-eval-print t)
```

which will execute all of the commands in the file until the first error. This recreates the ACL2 state we had when we saved the file, re-establishing our invariant between `script` and `*shell*`. If the proofs take a long time we might do

```
(ld "script.lisp" :ld-pre-eval-print t :ld-skip-proofs t)
```

or, more briefly,

```
(rebuild "script.lisp" t)
```

which *just assumes* the proof obligations of each command. Given that we have successfully executed them in a prior session, this is a reasonable way to re-establish the invariant and leaves us in exactly the same state.

When we finally execute the last command in the `script`, we have succeeded and `script.lisp` is a re-playable proof script for our main theorem. We usually try to certify it as a *book* so that it can be easily referenced in future proofs. See `certify-book`.

So much for the mechanics of using ACL2. We now get on the with task of explaining what it all means, by describing the ACL2 functional programming language, the logic, and the theorem prover.

4 Programming in ACL2

ACL2 is Lisp. A typical term or expression is `(cons (car x) (len a))`. In this expression, `x` and `a` are *variable symbols*, and `cons`, `car`, and `len` are *function symbols*. In more traditional mathematical syntax, this expression would be written $cons(car(x), len(a))$. For the purposes of this paper, it is sufficient to understand only the expressions of Figure 4. ACL2 is much richer than these few primitives would suggest, but throughout this paper we limit ourselves to a tiny subset so we can discuss in detail how to develop proofs.

Full ACL2 describes five kinds of data objects in detail – numbers, characters, strings, symbols, and (ordered) pairs – and each can be written as a constant and used in expressions. The most commonly used numeric constants are integers; rationals and complex constants are allowed, but we will not have occasion to use them in early exercises. We will not use character constants here. Strings are enclosed in single-character double-quotation marks, `"Hello world!"`. The special symbol constants `nil` and `t` – which have the apparent syntax of variable symbols – are written as shown, but all other symbol constants are preceded by a single quotation mark, `'ok` and `'quick-sort`. The constant `nil` is used both as the "false" truth value and the "empty list" (or, more accurately, as the standard terminator of a nest of pairs used as a list). Pairs, or "conses," are written in list notation, e.g., ⟨1,⟨2,⟨3,`nil`⟩⟩⟩ is written `'(1 2 3)` and nests not right-terminated with `nil` are written using "dot notation," e.g., ⟨1,⟨2,3⟩⟩ is written `'(1 2 . 3)`. Indeed, `'(1 2 3)` may also be written `'(1 2 3 . nil)`.

Newcomers are often confused by when to use the single-quote mark. `'(1 2 3)` is a term that evaluates to the list constant `(1 2 3)`. Why not write `(1 2 3)`? Well, consider the two terms `(car x)` and `'(car x)`. The first is how we write the application of the function symbol `car` to the variable symbol `x`. The second is a term that evaluates to the list constant `(car x)`, i.e., list whose first element is the symbol `car`. If α is a parenthesized expression or a symbol, like `car` or `x`, and you are writing a term, write `'`α if you mean the term that evaluates to α, and write α if you mean the term (function application or variable symbol) α.

Each of the five data types can be created and decomposed by various functions. But in this paper we omit all mention of the structural properties of numbers, characters, strings and symbols and deal only with pairs. By limiting ourselves to pairs, we can quickly dispense with their basic properties and get on with the task of learning how to define recursive functions and prove theorems.

`(cons` x y`)`	construct the ordered pair $\langle x, y\rangle$
`(car` x`)`	left component of x, if x is a pair; `nil` otherwise
`(cdr` x`)`	right component of x, if x is a pair; `nil` otherwise
`(consp` x`)`	`t` if x is a pair; `nil` otherwise
`(if` x y z`)`	z if x is `nil`; y otherwise
`(equal` x y`)`	`t` if x is y; `nil` otherwise

Fig. 1. The Primitives for This Paper

To define a function, we use the form (`defun` f $(v_1 \ldots v_n)$ β) where f is the function symbol being defined, the v_i are the distinct formal variables, and β is the body of the function.

Here are the Lisp definitions of the standard propositional logic connectives:

```
(defun not (p) (if p nil t))
(defun and (p q) (if p q nil))
(defun or (p q) (if p p q))
(defun implies (p q) (if p (if q t nil) t))
(defun iff (p q) (and (implies p q) (implies q p)))
```

Note that in Lisp, `and` and `or` are not Boolean valued. E.g., `(and t 3)` and `(or nil 3)` both return 3. This is unimportant if they are only used propositionally, e.g., `(and t 3)` $\leftrightarrow$ `(and 3 t)` $\leftrightarrow$ `t`, if "$\leftrightarrow$" means `iff`. By convention, these two functions are allowed to take more than two arguments and when so used abbreviate right-associated nests, e.g., `(and p q r s)` is an abbreviation for `(and p (and q (and r s)))`. Technically, they are defined as "macros."

Most often we make definitions that are recursive, because ACL2 has no iterative control structures or higher-order functions, and has only primitive reasoning support for quantifiers. Here is a function that "copies" a list replacing each element occurrence by two adjacent occurrences.

```
(defun dup (x)
  (if (consp x)
      (cons (car x)
            (cons (car x)
                  (dup (cdr x))))
      nil))
```

For example, the term `(dup '(1 2 3))` has value `(1 1 2 2 3 3)` and the term `(dup '(hello))` evaluates to `(hello hello)`.

Here is a function that concatenates two lists.

```
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))
```

For example, `(app '(1 2 3) '(4 5 6))` has value `(1 2 3 4 5 6)` and `(app '(A B C . D) '(E F))` has value `(A B C E F)`.

Here is a function that determines whether `e` is an element of list `x`.

```
(defun memp (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (memp e (cdr x)))
    nil))
```

For example, `(memp 1 '(0 1 2 3))` is `t` and `(memp 5 '(0 1 2 3))` is `nil`.

Here is a function that reverses a list, e.g., `(rev '(1 2 3))` is `(3 2 1)`.

```
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
    nil))
```

Note that `(rev '(1 2 3 . ABC))` is `(3 2 1)`, i.e., the terminal marker of the input is not preserved (unless it happened to be `nil`), given the way we defined `rev`.

Here is a "tail-recursive" version of rev that uses its second argument as an "accumulator" to construct the answer more efficiently.

```
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
    a))
```

For example, `(rev1 '(1 2 3) nil)` is `(3 2 1)`.

Exercises

You may wish to define auxiliary functions to solve some of the exercises below. If you are using the ACL2 system to experiment with your answers and you try to re-define an existing ACL2 function you will get an error (unless your definition is syntactically the same as ours). To see how to inspect the pre-existing definition, see `pe` ("print event") and `pf` ("print formula"). When using the ACL2 system, be aware that it insists that all functions terminate. Thus, recursion on the list structure `x` should be controlled by a `(consp x)` test, not `(equal x nil)`.

Problem 4.1 Define the function `properp` to determine whether a list "ends in `nil`," i.e., whether the `cdr` of the right-most `cons` is `nil`. (In Lisp, functions that are used as predicates are commonly given names that conclude with the letter "`p`". Lists satisfying `properp` are sometimes called "proper lists" or "true lists.")

Problem 4.2 Define `mapnil` to "copy" a list, replacing each element by `nil`.

Problem 4.3 The result of "swapping" the pair $\langle x, y\rangle$ is the pair $\langle y, x\rangle$. Define `swaptree` to swap every cons in the binary tree `x`.

Problem 4.4 Define `ziplists` to take two lists and return a list as long as the first whose successive elements are the pairs of corresponding elements from the two lists. If the second list is too short, extend it with `nil`s.

Problem 4.5 A proper list of pairs is called an "association list" or "alist". The standard function `alistp` recognizes them. Association lists are frequently used as tables. The value associated with the key *key* in alist *a* is the `cdr` of the first pair in a whose `car` is *key*. Define `lookup` to take a key and an alist and to return the value of the key in the alist or else `nil` if no pair is found.

Problem 4.6 Define `foundp` to determine whether a given key is found in a given alist.

Problem 4.7 Define the list analogue of subset, i.e., `(subp` x y`)` returns `t` or `nil` according to whether every element of x is an element of y.

Problem 4.8 Define `int` to take two proper lists and to return the proper list of elements that appear in both.

Problem 4.9 Consider the leaves of a binary tree. We say a leaf is "lonesome" if it occurs only once. Define `lonesomes` to take a tree and return its lonesome leaves.

5 Elementary Proofs in the ACL2 Logic

Some axioms corresponding to the six primitives in Figure 4 are shown in Figure 5. The actual axioms used by ACL2 are somewhat different because they include axioms for all the data types. For example, Axiom 1 of the figure can be proved from ACL2's axioms concerning the structure of symbols. Axiom 8 of the figure, stating that `nil` is not a `cons` pair, can be inferred from ACL2's axioms stating that `nil` is a symbol and that symbols are disjoint from pairs. For our purposes, Axiom 8 is just an example of an infinite number of axioms stating `consp` is `nil` on each symbol, on each number, etc., `(consp nil)` = `(consp t)` = `(consp 'ok)` = `(consp 0)` = `(consp 1)` = `(consp 2)` = ... = `nil`.

1.	`t` ≠ `nil`
2.	`x` ≠ `nil` → `(if x y z)` = `y`
3.	`x` = `nil` → `(if x y z)` = `z`
4.	`(equal x y)` = `nil` ∨ `(equal x y)` = `t`
5.	`x` = `y` ↔ `(equal x y)` = `t`
6.	`(consp x)` = `nil` ∨ `(consp x)` = `t`
7.	`(consp (cons x y))` = `t`
8.	`(consp nil)` = `nil`
9.	`(car (cons x y))` = `x`
10.	`(cdr (cons x y))` = `y`
11.	`(consp x)` = `t` → `(cons (car x) (cdr x))` = `x`

Fig. 2. The Primitive Axioms for This Paper

Implicit in this axiomatization is the logical infrastructure to do propositional calculus and equality. That is, we take for granted the axioms and rules of inference allowing us to prove propositional tautologies, perform substitution of equals for equals, etc.

We also give ourselves the ability to do induction on well-founded orderings. This involves some additional logical infrastructure, including an Induction Principle, the introduction of the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$, a well-founded relation o< on such ordinals, and axioms defining the size (measured with the function `acl2-count`) of ACL2 objects. The most common use of `o<` is on natural numbers, where it reduces to the ordinary < relation.

The inductive arguments required in this paper all use structural inductions on lists and binary trees. With the infrastructure described above it can be shown that the sizes of `(car x)` and `(cdr x)` are each smaller than the size of `x` when `(consp x)` is true. We can then use the Induction Principle to prove an arbitrary formula, (ψ `x y`), where `x` and `y` are variable symbols, by proving a

Base Case:

```
(implies (not (consp x)) (ψ x y))
```

and an

Induction Step:

```
(implies (and (consp x)           ; test
              (ψ (car x) α1)       ; induction hypothesis 1
              (ψ (cdr x) α2))      ; induction hypothesis 2
         (ψ x y))                  ; induction conclusion
```

where the α_i are arbitrary terms replacing the non-induction variable `y`. Of course, we often need only one of the two induction hypotheses; we can provide as many different "copies" of a hypothesis we wish, using different choices of α_i for `y`; and we can use nests of `cars` and `cdrs` in the `x` position. Indeed, we may provide as an induction hypothesis any ($\psi\ \delta\ \alpha$) such that we can prove `(implies (consp x) (o< (acl2-count` δ`) (acl2-count x)))`.

It must be emphasized that ACL2's Induction Principle is much more general than the scheme above suggests. Below we state the Induction Principle carefully, for those readers who are curious about it. In general in this document we take the position that you can learn to do much with ACL2 by example and by elaboration, and in that spirit we shy away from the precise details. They may be found, however, in [7].

The Induction Principle allows one to derive an arbitrary formula, ψ, from

- *Base Case*:

  `(implies (and (not` q_1`) ... (not` q_k`))` ψ`)`, and

- *Induction Step(s)*: For each $1 \le i \le k$,

  `(implies (and` q_i $\psi/\sigma_{i,1}$ ... $\psi/\sigma_{i,h_i}$`)`
  ψ`)` ,

provided that for terms m, $q_1, ...q_k$, and variable substitutions $\sigma_{i,j}$ ($1 \le i \le k, 1 \le j \le h_i$), the following are theorems:

- *Ordinal Condition*:

  `(o-p` m`)` , and

- *Measure Condition(s)*: For each $1 \le i \le k$, and $1 \le j \le h_i$,

  `(implies` q_i `(o<` $m/\sigma_{i,j}$ m`))` .

In the above, "τ/σ" represents the term or formula obtained by applying the variable substitution σ to the term or formula τ, uniformly replacing all free occurrences of the variables as indicated by the substitution.

In other words, to prove ψ by induction, you may assume as many arbitrary instances of ψ as you want, as long as they make some ordinal-valued measure of the variables in ψ decrease. The key to induction is well-foundedness and the key to well-foundedness in ACL2 is the notion of the ordinals. The ordinals (up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$) in ACL2 are recognized by the function `o-p` and compared with the relation `o<`. To learn more, see `ordinals`.

ACL2 also has a Definitional Principle, implemented by `defun`. When a function definition is submitted, ACL2 must prove "measure conjectures" establishing that some measure of the arguments is decreasing in a well-founded way under the tests governing the recursion. Operationally, the validity of the measure conjectures ensures that the recursion terminates; logically, it ensures that there exists a function satisfying the definitional equation. Only after these conjectures are proved is the definitional equation "admitted" as a new axiom. We do not deal with termination further in this paper.

So let's prove some theorems! Here is a function that "copies" a tree. Prove it is the identity function.

```
(defun treecopy (x)
  (if (consp x)
      (cons (treecopy (car x))
            (treecopy (cdr x)))
    x))
```

Theorem `(equal (treecopy x) x)`.
Proof.

Name the formula above *1.

We prove *1 by induction. One induction scheme is suggested by this conjecture – namely the one that unwinds the recursion in `treecopy`.

If we let `(`ψ` x)` denote *1 above then the induction scheme we'll use is

```
(and (implies (not (consp x)) (ψ x))
     (implies (and (consp x)
                   (ψ (car x))
                   (ψ (cdr x)))
              (ψ x))).
```

This induction is justified by the same argument used to "admit" `treecopy`, namely, the size of `x` is decreasing according a certain well-founded relation. When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.

Subgoal *1/2
```
(implies (not (consp x))
         (equal (treecopy x) x)).
```

But simplification reduces this to `t`, using the definition of `treecopy` and the primitive axioms.

Subgoal *1/1
```
(implies (and (consp x)
              (equal (treecopy (car x)) (car x))
              (equal (treecopy (cdr x)) (cdr x)))
         (equal (treecopy x) x)).
```

But simplification reduces this to `t`, using the definition of `treecopy` and the primitive axioms.

That completes the proof of *1.

Q.E.D.

Let us look more closely at the reduction of Subgoal *1/1. Consider the left-hand side of the concluding equality. Here is how it reduces to the right-hand side under the hypotheses.

```
(treecopy x)
=                       {def treecopy}
(if (consp x)
    (cons (treecopy (car x))
          (treecopy (cdr x)))
    x)
=                       {hypothesis 1 and Axiom 6}
(if t
    (cons (treecopy (car x))
          (treecopy (cdr x)))
    x)
=                       {Axioms 2 and 1}
(cons (treecopy (car x))
      (treecopy (cdr x)))
=                       {hypothesis 2}
(cons (car x)
      (treecopy (cdr x)))
=                       {hypothesis 3}
(cons (car x)
      (cdr x))
=                       {Axiom 11 and hypothesis 1}
x
```

This proof is of a very routine nature: induct so as to unwind some particular function appearing in the conjecture and then use the axioms and definitions to simplify each case to `t`.

Exercises

Prove each of the formulas below. Don't use the ACL2 system! Work out the proofs by hand. We want you to learn two things from these exercises: the importance of choosing the right variable to induct upon and what it means to simplify a formula using definitions and axioms.

Problem 5.1 `(equal (app (app a b) c) (app a (app b c)))`.

Problem 5.2 `(equal (dup (app a b)) (app (dup a) (dup b)))`.

Problem 5.3 `(equal (dup (mapnil a)) (mapnil (dup a)))`.

Problem 5.4 `(properp (app a nil))`.

Problem 5.5 `(equal (swaptree (swaptree x)) x)`.

Problem 5.6 `(equal (memp e (app a b)) (or (memp e a) (memp e b)))`.

6 Three Basic Proof Techniques

The proofs above are very routine – if the right induction argument is chosen. Each proof has the "induct and simplify" structure mentioned earlier. The ACL2 theorem prover uses an elaboration of that same strategy. In this section we briefly discuss three important techniques used by ACL2: induction, rewriting, and inequality chaining (linear arithmetic). The last two are the key parts of the ACL2 simplifier. The reader uninterested in the ACL2 system should read this section anyway! We explain why at the end.

6.1 Induction

The induction heuristic chooses an induction scheme based on the recursively defined functions used in the conjecture. Sometimes the system synthesizes a scheme by combining two or more recursive schemes used in the formula.

Consider Problem 5.6 above,

```
(equal (memp e (app a b))
       (or (memp e a)
           (memp e b))).
```

The successful proof will be by induction on `a`, i.e., by unwinding the recursion in `(memp e a)`. Why not induction on `b`? If you did the proof by hand you might have discovered that induction on `b` doesn't work.

The basic idea is that if we induct on `a`, we get our induction hypothesis by replacing each `a` by `(cdr a)`. So where the induction conclusion has an `a`, the induction hypothesis will have a `(cdr a)`. In particular, the terms `(memp e a)` and `(memp e (app a b))`, in the conclusion, correspond to `(memp e (cdr a))` and `(memp e (app (cdr a) b))` in the hypothesis. But when we expand the definitions of `memp` and `app` in the conclusion, those expressions reduce to their correspondents in the hypothesis!

Now try the same induction on `b`. Where the conclusion has `(memp e (app a b))` the hypothesis will have `(memp e (app a (cdr b)))`. And there is no way we can use the definitions of `memp` and `app` to reduce the conclusion term to the hypothesis term. The key observation is that the second argument of `app` is held constant in the recursion of `app`, so inducting on it is probably a bad idea. We say the induction on `b` here is "flawed."

ACL2 uses a variety of heuristics to select an induction argument. If the system's chosen scheme is inappropriate, the user can specify a scheme with a hint; see **`hints`**.

Choosing the right induction is crucial to a successful proof. But there is an earlier, much more subtle step: choosing the right theorem to try to prove by induction!

Exercise

Problem Try to prove the following special case of Problem 5.6 directly by induction: `(equal (memp e (app a a)) (mem e a))`.

If your goal is the above Problem, you will at some point have to prove something much more general, e.g., Problem 5.6 first.

This is a particularly trivial example of a phenomenon familiar to people trying to do inductive proofs. In induction, your main tool is the induction hypothesis, which is an instance of the conjecture you're trying to prove. If the conjecture you're trying to prove is not strong enough, your hypothesis will be useless.

Hint on How To Prove Things: *Induction must be applied to strong theorems, not weak ones! Always try to invent the strongest theorems you can think of!*

By understanding the link between recursion and induction you can learn to anticipate many problems. The fact that `(app x y)` is defined to take the `cdr` of `x` while holding `y` constant in the recursion is a sure sign that if `y` is the induction variable the proof will either fail or you will need a lemma that "moves a `cdr` out of the second argument of `app`", i.e., a lemma that transforms `(app x (cdr y))` to something involving `(app x y)` or vice versa.

6.2 Simplification via Rewriting

As important as induction is, the key to any successful proof is simplification. *Simplification* means the reduction of the formula to some *preferred form* by the use of *rules.* In ACL2, these rules are derived from axioms, definitions and previously proved theorems.

The previous paragraph is incredibly important if you are going to learn to use the ACL2 system! You essentially *program* the ACL2 simplifier by getting the system to prove theorems *which are then turned into rules.* The preferred form enforced by the system is largely determined by your rules. All the rules ever produced in a session are available to ACL2's simplifier, so once you have added a rule it may participate in any subsequent proof unless you take active steps to disable it. To use ACL2, you must understand (a) how theorems are turned into rules, (b) what those rules make the simplifier do, and (c) how to disable and enable rules.

There are about a dozen kinds of rules in ACL2 and when a theorem is posed, the user specifies the kind of rule to be produced from the theorem. See `rule-classes`. In this document we see only three specifications: make a rewrite rule, make a linear arithmetic rule, or make no rule at all. In practice, these three specifications often suffice. The last is used when we have a theorem that cannot generate a useful rule – the only way such a theorem can participate in a subsequent proof is by a user-specified hint; see hints.

By far the most common form of rule is the rewrite rule. It causes the simplifier to replace one term by another, if certain hypotheses can be established by rewriting. Rewrite rules are the most direct way to program the simplifier.

The rewrite rule derived from a formula of the form

`(implies (and` hyp_1 `...` hyp_n`) (equal` *lhs rhs*`))`

makes the simplifier replace instances of *lhs* (the **l**eft-**h**and **s**ide) by the corresponding instance of *rhs*, provided the corresponding instances of each of the hyp_i rewrites to true.

Equivalent logical forms may give rise to radically different rules and hence radically different programmed behaviors! Consider the effect of the (rule generated from the) equivalent formula `(implies (and` hyp_1 `...`hyp_n`) (equal` *rhs lhs*`))`.

Hint on How To Prove Things: *Give careful thought to the "preferred" forms you use in your proofs and provide yourself with lemmas that allow you, insofar as possible, to canonicalize terms.*

Hint on How To Prove Things: *When using the ACL2 system,* never *prove a named theorem without understanding its effect as a rule!*

Some conventions make it possible to derive rewrite rules from a wide variety of formulas. The conclusion can be `(equal` *lhs rhs*`)` or `(iff` *lhs rhs*`)`. The latter kind of rule is used to replace *lhs* by *rhs* in "propositional" settings. ACL2 allows user-defined equivalence relations in rewrite rules, but we do not discuss them; see `equivalence` and `congruence`. If the conclusion, say, *concl*, is not an equivalence, it is treated as though it were `(iff` *concl* `t)`. If there are no hypotheses, it is as though there were just one: `t`. See rewrite.

6.3 Simplification via Inequality Chaining

We have not discussed arithmetic – and we will not in this document, except for a few important observations in this section.

Hint on How To Prove Things: *Realize that when you are dealing with arithmetic, your sense of what is "straightforward" has been honed by many years of drill-and-practice with manipulating algebraic properties of numbers. Be prepared to "explain" formally why some arithmetic relations hold!*

Most of that drill-and-practice is, technically speaking, the application of a large set of rewrite rules to put arithmetic expressions into a preferred form. ACL2 does not come pre-configured with those rules. But they are available in several different collections. In ACL2 a collection of rules in a file is called a "book." The ACL2 distribution comes with several arithmetic books and the community is constantly working on improving them. That is one of the reasons we have several such books now. Another is that different books are designed for different kinds of problems: elementary algebraic properties of numbers, modulo arithmetic, floating point arithmetic. See the `README.html` file in the `books` subdirectory of your ACL2 source directory, or else visit the Mathematical Tools link on the ACL2 home page.

Hint on How To Prove Things: *If you are doing arithmetic proofs with ACL2, start by including one of the arithmetic books into your script. The most commonly used book is included by adding the command* `(include-book "arithmetic/top-with-meta" :dir :system)`.

Finally, you should be aware that often in arithmetic reasoning you do a kind a inequality chaining that "feels" like rewriting but is not.

Consider a theorem that concludes with an arithmetic inequality, such as `(<= 0 (* x x))`. This says "x squared is nonnegative." [2] If it is used to generate a rewrite rule, the rule will replace certain instances of `(<= 0 (* x x))` by `t`. This does not help us much if we are trying to prove that `(+ a (* b b))` is positive when `a` is positive. But ACL2 maintains a graph of terms involved in the current conjecture and relates the terms in this graph with inequalities. It contains a decision procedure for answering questions about linear arithmetic – the fragment of arithmetic consisting of inequalities, addition, and multiplication by constants – based on the property that inequalities can be added. People sometimes call this "inequality chaining." Such chaining can be used to derive `(<= 0 (+ a (* b b)))` from `(<= 0 a)` and `(<= 0 (* b b))`. But if we are trying to prove

```
(implies (and (<= 0 a) (rationalp b)) (<= 0 (+ a (* b b))))
```

the graph contains no node for `(* b b)` – because that term is not compared to any other term – and no chaining is possible. However, if the rule above about `(<= 0 (* x x))` is available as a linear arithmetic rule instead of as a

[2] This is not always true: `x` may be complex. But we're imagining this inequality as the conclusion of a suitable implication.

rewrite rule it causes the following behavior: whenever an instance of `(*` *x x*`)` enters the problem, the inequality graph is extended with a node for the corresponding instance and it is linked to other nodes as described by the conclusion of the linear rule. This extends range of the chaining decision procedure. See linear-arithmetic.

Hint on How To Prove Things: *Realize that when you are dealing with arithmetic you may be doing inequality chaining, not replacement of equals by equals, and make that form of reasoning explicit in your notation.*

This discussion of rewrite versus linear rules is of general interest because *most mathematical facts have implicit operational import.* This discussion illustrates that. When you discover a new fact, what do you do with it? Do you store it, unanalyzed, in a long list of things you know and revisit them all every time you are asked to prove something? Or do you see ways you can use and remember the new fact? This is a difficult introspective question to answer – and the answer is probably "some of each" – but it is important to remember that many years of secondary school and college mathematics have taught you how to use certain forms of facts, e.g., associativity, commutativity, identities, idempotence, inequality chaining, cancellation, etc.

Hint on How To Prove Things: *Every time you encounter a new theorem you should give thought to how it is to be used in subsequent proofs.*

6.4 Some ACL2-Specific Details

The rest of this section is mainly of interest to potential ACL2 users, but contains a few useful hints of more general interest.

To prove a named theorem with ACL2, use `(defthm` *name term*`)` if you want *term* used as a rewrite rule. Use `(defthm` *name term* `:rule-classes :linear)` if you want *term* used to extend the linear arithmetic inequality graph. This is possible only if *term* is an arithmetic inequality, i.e., `(<` *lhs rhs*`)`, `(<=` *lhs rhs*`)`, `(>=` *lhs rhs*`)`, `(>` *lhs rhs*`)`, `(equal` *lhs rhs*`)`, or `(not (equal` *lhs rhs*`))`, where, for the last two, *lhs* and *rhs* are numerically valued expressions. If you want no rule generated from *term*, use `(defthm` *name term* `:rule-classes nil)`.

Note carefully: if you just use the simple form of `defthm` to prove a named theorem, you are telling ACL2 to use it as a rewrite rule!

If you need to supply hints to the theorem prover, use the optional `:hints` "keyword argument," e.g., write

```
(defthm name term :hints hints)
```

or

```
(defthm name term
   :hints hints
   :rule-classes classes).
```

. See **hints**.

Finally, if you have proved a rule named *name* and want to disable it so that ACL2 no longer considers applying it, use `(in-theory (disable` *name*`))`. To undo that, use `(in-theory (enable` *name*`))`. It is possible to collect groups of names together so as to enable and disable them in concert. Each such group represents a strategy. See **theories**.

The command

```
(defthm app-right-identity
  (implies (properp x) (equal (app x nil) x)))
```

commands ACL2 to prove the formula named `app-right-identity` and store it as a rewrite rule if the proof is successful. The rule generated rewrites instances of `(app x nil)`. After `app-right-identity` has been proved, if the simplifier encounters a target term like `(app (rev (app a b)) nil)` it will try the rule, because the target matches (is an instance of) `(app x nil)`. The substitution produced by the matching process binds the variable symbol `x`, from the rule, to `(rev (app a b))`, from the target. To apply the rule, the simplifier considers the "corresponding instance" of `(properp x)`, namely `(properp (rev (app a b)))`. It tries to rewrite this to true. If it can, it will replace the target by the "corresponding instance" of the right-hand side from the rule. Thus, it will replace the target by `(rev (app a b))`.

Consider what would happen if you proved a rule that rewrites *lhs* to *rhs* and another rule that rewrites *rhs* to *lhs*. More complicated cycles are more likely, of course. ACL2 has special heuristics for handling commutativity and similarly simple permutative rules. But in general ACL2's simplifier can be made to loop forever by programming it with circular rules. Such behavior will generally be reported by the simplifier together with instructions for how to debug the failure.

Hint on How To Prove Things: *Ensure that your rules do not loop! One way to do this is to keep in mind some ordering on your preferred terms and be sure that the right-hand side of each rule is lower in this ordering than the left-hand side.*

You may sometimes wish to interrupt the theorem prover, e.g., because of a "runaway proof." To interrupt the ACL2 prover while using it under Emacs, type `ctrl-c ctrl-c`. This will leave you in a Common Lisp (not ACL2!) read-eval-print break. To this break you should type the command that aborts an interrupted Common Lisp computation. That command varies according to your host Lisp. If you are running GCL, type `:q` followed by `enter`; if you are running Allegro, type `:reset` followed by `enter`; if you are running CMU CL, type `q`, followed by `enter`; if you are running MCL, type `:pop`.

Hint on How To Prove Things: *Definitions are (generally) used as expansion rules, i.e., function calls are replaced by their instantiated bodies. This imposes a restriction on your choice of preferred forms: function bodies are preferable* *to function calls. If you want to override that built-in preference in ACL2, you should disable the functions after proving the rules you need about them.*

You might wonder how ACL2 can replace function calls by their bodies and not loop indefinitely on recursive definitions. The answer is that ACL2 contains heuristics for controlling the expansion of recursive definitions. These heuristics generally do a good job and most users find it better to arrange their rewrite rules to be compatible with these heuristics than to fight the heuristics.

To simplify a formula, the ACL2 simplifier rewrites the formula from left-to-right and inside-out. Thus, a rule with the left-hand side `(foo (car (cons x y)))` will *never* be applied! Why? The only possible target term for this problematic rule is of the form `(foo (car (cons` α β`)))`. But the ACL2 simplifier sweeps inside-out, so the problematic rule is not tried until the interior terms of the target have been rewritten. Given the rule that reduces `(car (cons x y))` to `x` (which is derived from primitive Axiom 9 of Figure 5), the target will have been transformed to `(foo` α`)` before the problematic rule is tried. It will therefore fail to match.

Hint on How To Prove Things: *When designing your rewrite rules, be sure the left-hand sides are in your preferred form!*

7 ACL2's Proof Strategy

Rather than "induct and simplify," ACL2's strategy is "simplify (and some other things), induct and repeat." The reason ACL2's strategy looks like "induct and simplify" is that, often, the initial simplification does not change the goal formula so it looks like ACL2 immediately went to induction. Some theorems are proved by the initial simplification and no induction is used. It is possible to program ACL2's simplifier so that almost every proof that ACL2 can do can be put into the "simplify, induct and simplify" form, by proving appropriate lemmas first. We recommend that new users concentrate on producing proofs in that form.

Hint on How To Prove Things: *Keep your proofs in the "simplify, induct, simplify" form. That is, identify each inductively proved lemma you need in a proof, write it down, and give it a name. Do not get into the habit of letting the ACL2 prover invent and inductively prove lemmas "on the fly" in the middle of other proofs. It is better that you understand and control the lemma decomposition of your theorems.*

Exercise

Problem 7.1 Run the ACL2 theorem prover on each of Problems 5.1 – 5.6.

You will find that ACL2's strategy finds proofs for each of these automatically – at least if you defined the various functions the way we did. See Appendix A.

Here is the output produced by ACL2 Version 2.8 on Problem 4.6. The first form is our input, typed as a `defthm` command at the ACL2 prompt. Note that

ACL2's initial simplification splits the conjecture into two parts, `Subgoal 2` and `Subgoal 1`, according to whether `(memp e a)`. Upon exploring the proof space a little further, ACL2 learns it will have to tackle both by induction. It then discards the simplification work, backs up to the original theorem, and sets up an induction argument on that instead.[3]

It uses `:p` to represent the theorem schematically, where we used ψ above. Its induction argument has two base cases; `Subgoal *1/3` handles the case when `a` is not a `consp`; `Subgoal *1/1` handles the case when `a` is a `consp` but its first element is `e`. This is the induction scheme necessary to unwind `(mem e a)`.

ACL2 prints terms in uppercase. We have lowered the case below to keep the typography consistent with this paper. In the subset of ACL2 used in this paper, Lisp is case insensitive except for string constants.

```
ACL2 !>(defthm memp-app
         (equal (memp e (app a b))
                (or (memp e a) (memp e b))))

This simplifies, using the :type-prescription rule memp,
to the following two conjectures.

Subgoal 2
(implies (memp e a)
         (equal (memp e (app a b)) t)).

This simplifies, using primitive type reasoning and the
:type-prescription rule memp, to

Subgoal 2'
(implies (memp e a) (memp e (app a b))).

Name the formula above *1.

Subgoal 1
(implies (not (memp e a))
         (equal (memp e (app a b)) (memp e b))).

Normally we would attempt to prove this formula by induction.
However, we prefer in this instance to focus on the original
input conjecture rather than this simplified special case.
We therefore abandon our previous work on this conjecture
and reassign the name *1 to the original conjecture.
(See :DOC otf-flg.)

Perhaps we can prove *1 by induction.  Four induction
schemes are suggested by this conjecture.  Subsumption
```

[3] In general, the original theorem is stronger than any single special case of it and is often the better theorem to try by induction. In this particular case, the two subgoals are each strong enough to be inductively provable.

```
reduces that number to three.  These merge into two derived
induction schemes.  However, one of these is flawed and
so we are left with one viable candidate.

We will induct according to a scheme suggested by (memp e a),
but modified to accommodate (app a b).  These suggestions
were produced using the :induction rules app and memp.
If we let  (:p a b e) denote *1 above then the induction
scheme we'll use is
(and (implies (not (consp a)) (:p a b e))
     (implies (and (consp a)
                   (not (equal e (car a)))
                   (:p (cdr a) b e))
              (:p a b e))
     (implies (and (consp a) (equal e (car a)))
              (:p a b e))).
This induction is justified by the same argument used
to admit memp, namely, the measure (acl2-count a) is decreasing
according to the relation o< (which is known to be well-
founded on the domain recognized by o-p).  When applied
to the goal at hand the above induction scheme produces
the following three nontautological subgoals.

Subgoal *1/3
(implies (not (consp a))
         (equal (memp e (app a b))
                (or (memp e a) (memp e b)))).

But simplification reduces this to t, using the :definitions
app and memp and primitive type reasoning.

Subgoal *1/2
(implies (and (consp a)
              (not (equal e (car a)))
              (equal (memp e (app (cdr a) b))
                     (or (memp e (cdr a)) (memp e b))))
         (equal (memp e (app a b))
                (or (memp e a) (memp e b)))).

But simplification reduces this to t, using the :definitions
app and memp, the :executable-counterpart of equal, primitive
type reasoning, the :rewrite rules car-cons and cdr-cons
and the :type-prescription rule memp.

Subgoal *1/1
(implies (and (consp a) (equal e (car a)))
         (equal (memp e (app a b))
                (or (memp e a) (memp e b)))).

But simplification reduces this to t, using the :definitions
```

```
app and memp, the :executable-counterpart of equal, primitive
type reasoning and the :rewrite rule car-cons.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( defthm memp-app ...)
Rules: ((:definition app)
        (:definition memp)
        (:executable-counterpart equal)
        (:fake-rune-for-type-set nil)
        (:induction app)
        (:induction memp)
        (:rewrite car-cons)
        (:rewrite cdr-cons)
        (:type-prescription memp))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.01, other: 0.00)
 memp-app
ACL2 !>
```

8 Decomposition into Lemmas – The Method

Hint on How To Prove Things: *When considering a new conjecture to prove, look for general theorems that can be used to prove it by simplification – rewriting and chaining – before you consider proving it by induction.*

This is probably the most important, and most vague, advice we have. We illustrate it below by proving a little theorem. In our illustration, we use the ACL2 theorem prover to do a lot of the work for us, but the general principles apply whenever you are doing an "induct and simplify" proof. The specific output we display was produced by ACL2 Version 2.8 after defining the functions and proving the theorems in the exercises above.

Let us prove `(equal (rev (rev (rev x))) (rev x))`. We call this theorem "triple `rev`." What more general fact does it suggest?

Hint on How To Prove Things: *Look for pairs of adjacent function symbols and try to think of rules that simplify those expressions.*

You have probably thought of the conjecture `(equal (rev (rev z)) z)`. But is `(equal (rev (rev z)) z)` a theorem? Is it always true? Consider the possibility that `z` is `7`. The left-hand side computes to `nil` but the right-hand side is `7`. So this is not a theorem. We might then restrict `z` to satisfy `consp`. But would that be a theorem? Consider the case when `z` is `'(1 . 7)`. The left-hand side is `(1)` and the right-hand side is `(1 . 7)`.

Hint on How To Prove Things: *When forming new conjectures, test them on constants.*

Since ACL2 is a programming language, you can usually run your conjectures on a few examples. One could type `(let ((x '(1 . 7))) (equal (rev (rev x)) x))` to run the second test above; the result is `nil`. We often do `:set-guard-checking nil` when we are running tests, so that ACL2 does not reject the test simply because it violates the implicit type constraints on Lisp primitives. See set-guard-checking.

Consider for a moment the expected input to `rev`. `Rev` `cdr`s down its argument until it is no longer a `consp` and then claims its reverse is `nil`. Implicitly then, `rev` "expects" its argument to be a proper list. Thus, the first part of our attack on the triple `rev` problem is to prove:

```
(defthm rev-rev
  (implies (properp z)
           (equal (rev (rev z)) z)))
```

In a fully-typed language, we might not need the explicit hypothesis that `z` is a proper list. But we are in an untyped language and must make these restrictions explicit.

Hint on How To Prove Things: *Often you will have to invent new concepts – concepts not involved in your main theorem – to state the lemmas you need.*

It is surprising how often people resist adding a new definition, even of a concept they clearly have in mind. If you restrict your predicates to things like `consp`, there is no way you can state an inductively provable version of `(equal (rev (rev x)) x)`. You have to introduce the new concept of "proper list" to state the theorem.

Suppose we had proved `rev-rev`. Could we prove the triple `rev` theorem? Answer: Not unless we knew the following.

```
(defthm properp-rev
  (properp (rev x)))
```

Hint on How To Prove Things: *If you have introduced hypotheses in your lemmas, be sure you prove that the appropriate terms satisfy those hypotheses.*

Is the `properp-rev` conjecture true? Does `rev` always return a proper list, even when its input is improper? Yes! Because it either returns `nil` or a list it produces by appending a proper list (a singleton list) to the right of the recursively produced answer. So here is a theorem, `properp-rev`, that is stronger than we might have produced in a strongly typed language, i.e., there is no restriction that `x` be proper.

Thus, we have designed the following proof script:

```
; --- Script for proving triple-rev ---
(defthm rev-rev
  (implies (properp z)
           (equal (rev (rev z)) z)))
(defthm properp-rev
  (properp (rev x)))
(defthm triple-rev
  (equal (rev (rev (rev a))) (rev a)))
; --- The End ---
```

This plan illustrates an important adage:

Hint on How To Prove Things: *Separate your concerns!*

In the triple `rev` problem, the concept of `properp` neatly divides the problem. We first prove that the composition of two `rev`s is the identity *on proper lists.* We then prove that `(rev a)` is a proper list. To separate the two parts of the problem we had to think of the idea of a proper list. Notice that this is not only a nice plan for proving the triple `rev` theorem but it leaves us in excellent shape to prove other theorems, like `(equal (rev (rev (dup a))) (dup a))`, where all we have to do is prove that `dup` returns a proper list.

No further decomposition of our plan comes to mind, so now let us prove the first one inductively. What we are doing is following our method of using the `script` buffer, with the "cursor" positioned just before the `rev-rev` theorem. When we submit the `rev-rev` event, a successful proof description flashes by. We cannot read it as it flashes by, but the final two lines are:

```
Time:  0.03 seconds (prove: 0.02, print: 0.01, other: 0.00)
 REV-REV
```

whereas the final line in a failed proof is always

```
*** FAILED *** See :DOC failure *** FAILED ***
```

This success may appear to be good news, but a few lines above the successful conclusion are the lines

```
That completes the proofs of *1.1 and *1.
Q.E.D.
```

which tell the informed reader that ACL2 did (at least) a second induction (to prove `*1.1`) and so the proof is at least of the form "induct, simplify, induct, simplify." We don't know what intermediate lemma ACL2 discovered and proved, but we know it used induction twice in this proof! Since we recommend that novices stick to the "simplify, induct, simplify" strategy, you should undo the newly proved theorem – remember, it has just added a rule to ACL2 database!

– with `:u` (see u, ubt, pbt) and then read the generated proof script *from the top down.*

Hint on How To Prove Things: *We recommend that the novice ACL2 user not rely on ACL2's creative contributions in the beginning. As the problems become harder, ACL2's creative contributions count for less and less – and its ability to carry out massive automatic simplifications using user-specified rules counts for more and more. So the novice is encouraged to learn to spot the need for rules and to program ACL2 to use them.*

Here is the first part of ACL2's proof attempt on `rev-rev`. Read it, just as you would a human-generated proof sketch.

```
ACL2 !>(defthm rev-rev
         (implies (properp z)
                  (equal (rev (rev z)) z)))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Two induction schemes
are suggested by this conjecture.  Subsumption reduces
that number to one.

We will induct according to a scheme suggested by (rev z).
This suggestion was produced using the :induction rules
properp and rev.  If we let  (:p z) denote *1 above then
the induction scheme we'll use is
(and (implies (not (consp z)) (:p z))
     (implies (and (consp z) (:p (cdr z)))
              (:p z))).
This induction is justified by the same argument used
to admit rev, namely, the measure (acl2-count z) is decreasing
according to the relation o< (which is known to be well-
founded on the domain recognized by o-p).  When applied
to the goal at hand the above induction scheme produces
the following three nontautological subgoals.

Subgoal *1/3
(implies (and (not (consp z)) (properp z))
         (equal (rev (rev z)) z)).

But simplification reduces this to t, using the :definition
properp, the :executable-counterparts of consp, equal
and rev and primitive type reasoning.

Subgoal *1/2
(implies (and (consp z)
              (equal (rev (rev (cdr z))) (cdr z))
              (properp z))
         (equal (rev (rev z)) z)).
```

```
This simplifies, using the :definitions properp and rev,
to

Subgoal *1/2'
(implies (and (consp z)
              (equal (rev (rev (cdr z))) (cdr z))
              (properp (cdr z)))
         (equal (rev (app (rev (cdr z)) (list (car z))))
                z)).

The destructor terms (car z) and (cdr z) can be eliminated
by using car-cdr-elim to replace z by (cons z1 z2), (car z)
by z1 and (cdr z) by z2.  This produces the following
goal.
```

The step after `Subgoal *1/2'` is not a simplification, so the proof does not have the "simplify, induct, simplify" form we recommend for novices. In particular, `Subgoal *1/2'` is the first formula after the induction that was not proved by simplification. We call this formula a checkpoint. Learn to recognize them!

There are two main kinds of checkpoint formulas. The first is any formula proved by induction. In reading ACL2 output, we avoid reading past an induction without asking ourselves whether the formula being proved "needs" to be proved by induction and whether the selected induction is appropriate for it.

The other kind of checkpoint formula is the first formula after an induction that is not proved by repeated simplification. That is the case for `Subgoal *1/2'` above.

There is an Emacs utility that will automatically take you to the checkpoints of a proof attempt; see `proof-tree`.

Hint on How To Prove Things: *Whenever a proof fails (or you want to reduce a proof to the recommended form), read the formula at the first checkpoint and look for a lemma decomposition. Sometimes, it helps to read a few formulas past the first checkpoint – often ACL2's heuristics come fairly close to generating the needed lemma, or at least creating a term that will suggest the lemma to you. So if the checkpoint does not suggest anything, read on.*

What lemma is suggested by the checkpoint formula?

```
Subgoal *1/2'
(implies (and (consp z)
              (equal (rev (rev (cdr z))) (cdr z))
              (properp (cdr z)))
         (equal (rev (app (rev (cdr z)) (list (car z))))
                z)).
```

Recall an earlier Hint on How To Prove Things: *Look for pairs of adjacent function symbols and try to think of rules that simplify those expressions.* We see a subterm of the above checkpoint formula that has the form (rev (app ...)). Note that in the checkpoint formula, if we could move that outer rev past the app so that it nestles around the inner rev, we will have reproduced the induction hypothesis term, (rev (rev (cdr z))) and could use our induction hypothesis!

So what is a lemma that simplifies (rev (app a b))? And, if you care to use the "Note" just above, you could ask yourself: What is a lemma that relates (rev (app a b)) to (rev a)? Some thought, and perhaps some testing, produces the beautiful lemma:

```
(defthm rev-app
  (equal (rev (app a b)) (app (rev b) (rev a))))
```

Add this form to the script buffer, just in front of rev-rev, and repeat.

This proof fails, and the checkpoint formula is

```
Subgoal *1/2'
(implies (not (consp a))
         (equal (rev b) (app (rev b) nil))).
```

What does this suggest? Remember the hints on how to prove things!

The lemma suggested is that nil is a right identity for app. But of course it is not quite! It is a right identity for proper lists.

```
(defthm app-right-identity
  (implies (properp x)
           (equal (app x nil) x)))
```

But to use this theorem we must also know that (rev b), from Subgoal *1/2', is proper. We've already posed that theorem in our script – we needed it for our proof sketch of triple-rev – and so we move it forward in our script.[4]

So now our script looks like this and we are still at the top!

```
; --- Script for proving triple-rev ---
(defthm app-right-identity
  (implies (properp x)
           (equal (app x nil) x)))
(defthm properp-rev
  (properp (rev x)))
(defthm rev-app
  (equal (rev (app a b))
         (app (rev b) (rev a))))
(defthm rev-rev
  (implies (properp z)
           (equal (rev (rev z)) z)))
(defthm triple-rev
  (equal (rev (rev (rev a))) (rev a)))
```

[4] This fact, that lemmas invented for one proof may actually be useful in earlier proofs, is one of the reasons it is hard to build a rigid interface that enforces some kind of stack of proof plans: evolving plans can cause non-local rearrangements.

```
; --- The End ---
```

When we submit `properp-rev` it succeeds but we see the line that means multiple inductions were used:

```
That completes the proofs of *1.1 and *1.
```

so we undo with `:u` and read the checkpoint.

```
Subgoal *1/1'
(implies (and (consp x) (properp (rev (cdr x))))
         (properp (app (rev (cdr x)) (list (car x)))))
```

The pair of function symbols that leap out now are `properp` and `app`. Note that this is an instance of our advice about proving rules to simplify terms involving pairs of adjacent function symbols. Under what conditions is `(properp (app a b))` true? It depends on whether `(properp b)` is true. This suggests `(implies (properp b) (properp (app a b)))`. But that is a fairly weak theorem and as a rewrite rule it means: whenever you see `(properp (app a b))` backchain to `(properp b)` and if you can establish it, rewrite the target to `t`. Can we do better?

Yes! Consider the theorem

```
(defthm properp-app
  (equal (properp (app a b))
         (properp b)))
```

This is stronger than the mere implication. Furthermore, it is indeed a theorem! The rule generated from it allows the unconditional elimination of `(properp (app a b))` in favor of the simpler `(properp b)`.

So we change our `script` again to what is shown below. This time, every successive form in it is processed successfully in the "simplify, induct, simplify" strategy.

```
; --- Script for proving triple-rev ---
(defthm properp-app
  (equal (properp (app a b))
         (properp b)))
(defthm app-right-identity
  (implies (properp x)
           (equal (app x nil) x)))
(defthm properp-rev
  (properp (rev x)))
(defthm rev-app
  (equal (rev (app a b))
         (app (rev b) (rev a))))
(defthm rev-rev
  (implies (properp z)
           (equal (rev (rev z)) z)))
(defthm triple-rev
```

```
  (equal (rev (rev (rev a))) (rev a)))
; --- The End ---
```

The last proof above consists of a single simplification, just as we planned.

```
ACL2 !>(defthm triple-rev
         (equal (rev (rev (rev a))) (rev a)))

But simplification reduces this to t, using primitive
type reasoning and the :rewrite rules properp-rev and
rev-rev.

Q.E.D.

Summary
Form:  ( defthm triple-rev ...)
Rules: ((:fake-rune-for-type-set nil)
        (:rewrite properp-rev)
        (:rewrite rev-rev))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 triple-rev
```

However, if we look one more time at the proof output, we again see the line that means multiple inductions were used

```
That completes the proofs of *1.1 and *1.
```

— this time in the proof of `rev-app`. So we undo with `:u` and read the checkpoint.

```
Subgoal *1/1'
(implies (and (consp a)
              (equal (rev (app (cdr a) b))
                     (app (rev b) (rev (cdr a)))))
         (equal (app (rev (app (cdr a) b))
                     (list (car a)))
                (app (rev b)
                     (app (rev (cdr a)) (list (car a)))))).

The destructor terms (CAR A) and (CDR A) can be eliminated by
using CAR-CDR-ELIM to replace A by (CONS A1 A2), (CAR A) by A1
and (CDR A) by A2.  This produces the following goal.
```

This time, no rules occur to us that would further the simplification process, so we allow the theorem prover to use destructor elimination, and read further to see if any subgoals suggest a rule that can avoid multiple inductions.

```
Subgoal *1/1'5'
(equal (app (app rv0 rv) (list a1))
       (app rv0 (app rv (list a1)))).
```

`Name the formula above *1.1.`

We can now will discover a lemma stating that `app` is an associative operation, shown as lemma `app-assoc` below. With this addition, the proof succeeds without multiple inductions.

```
; --- Script for proving triple-rev ---
(defthm properp-app
  (equal (properp (app a b))
         (properp b)))
(defthm app-right-identity
  (implies (properp x)
           (equal (app x nil) x)))
(defthm properp-rev
  (properp (rev x)))
(defthm app-assoc
  (equal (app (app a b) c)
         (app a (app b c))))
(defthm rev-app
  (equal (rev (app a b))
         (app (rev b) (rev a))))
(defthm rev-rev
  (implies (properp z)
           (equal (rev (rev z)) z)))
(defthm triple-rev
  (equal (rev (rev (rev a))) (rev a)))
; --- The End ---
```

Note that one side-effect of our recommended reliance on "simplify, induct, simplify" is that it causes us to think about the general rules for manipulating the function symbols of the problem and to state them as lemmas. Had we relied on ACL2's creative contributions, we would not have identified so many good rules about `rev`, `app`, and `properp`.

Note also that since the final theorem is proved by rewriting with two existing rules, there is no need to enshrine `triple-rev` as a rule itself. Our subsequent rule library is a little smaller if we change the last form in the script to

```
(defthm triple-rev
  (equal (rev (rev (rev a))) (rev a))
  :rule-classes nil)
```

The general procedure we have just described is called "The Method" in [6] and described in the documentation for `The-Method`.

Exercises

Use The Method to find proofs for each of the theorems below using ACL2.

Problem 8.1

```
(implies (memp e c)
         (memp e (rev (dup (dup c))))
```

Problem 8.2

```
(equal (leaves (swaptree x))
       (rev (leaves x)))
```

where `leaves` is defined as

```
(defun leaves (x)
  (if (consp x)
      (app (leaves (car x)) (leaves (cdr x)))
      (cons x nil)))
```

Problem 8.3

```
(subp x x)
```

Problem 8.4

```
(implies (properp x)
         (equal (int x x) x))
```

Problem 8.5

```
(implies (and (subp x y)
              (subp y z))
         (subp x z))
```

Problem 8.6

```
(subp (app a a) a)
```

Problem 8.7

```
(seteqp (rev a) a)
```

where `seteqp` ("set equality") is defined as

```
(defun seteqp (x y)
  (and (subp x y)
       (subp y x)))
```

Note: Note that after proving that `(rev a)` is set-equivalent to `a` there is a natural expectation that ACL2 will replace `(rev a)` by `a` where ever it sees it used as a set. But `seteqp` is not `equal`! But ACL2 supports the introduction of user-defined equivalence rules and the kind of generalized rewriting just hinted at. See equivalence and congruence.

Problem 8.8

```
(seteqp (app a b) (app b a))
```

Problem 8.9

```
(equal (leaves (leaves x)) (app (leaves x) '(nil)))
```

Problem 8.10

```
(iff (memp e (lonesomes x))
     (and (memp e (leaves x))
          (lonesomep e (leaves x))))
```

Problem 8.11

```
(equal (raise b (add i j))
       (mult (raise b i) (raise b j)))
```

where `add`, `mult`, and `raise` are defined to be list-based analogues of addition, multiplication, and exponentiation. In this analogical setting, numbers (recognized by `nump`) are represented by lists of `nil`s of the appropriate length. Of course, ACL2 supports arithmetic but by simulating it in these exercises we force you to invent a lot of lemmas!

```
(defun nump (x)
  (if (consp x)
      (and (equal (car x) nil)
           (nump (cdr x)))
    (equal x nil)))
(defun add (x y)
  (if (consp x)
      (cons nil (add (cdr x) y))
    (mapnil y)))
(defun mult (x y)
  (if (consp x)
      (add y (mult (cdr x) y))
    nil))
(defun raise (x y)
  (if (consp y)
      (mult x (raise x (cdr y)))
    '(nil)))
```

Problem 8.12

```
(defthm add-commutativity
  (equal (add i j) (add j i)))
```

Hint on How To Prove Things: *When you know a function is commutative, use that fact to arrange the arguments in some canonical order. Thus, if* `add` *is known to be commutative, then whenever you see* `(add b a)`*, rewrite it to* `(add a b)`*. But do not use the rule the other way – to move things out of order – or you will loop forever! ACL2 uses this heuristic and uses a lexicographic ordering on terms.*

Problem 8.13

```
(defthm add-commutativity2
  (equal (add i (add j k)) (add j (add i k))))
```

Hint on How To Prove Things: *The heuristic advice about commutative functions does not help you if you are rewriting* `(add b (add a c))` *because the ordering (probably) will not prefer* `(add a c)` *over* `b`*. But the theorem above, which we call a "commutativity2" theorem, allows an appropriate swap and ACL2 uses such theorems to arrange the ordering of terms. Given associativity, commutativity, commutativity2, and these heuristics, you can arrange nests of such functions into right-associated form with the arguments ascending in the order. That is what ACL2 does.*

Problem 8.14

```
(defthm mult-commutativity
  (equal (mult i j) (mult j i)))
```

Problem 8.15

```
(defthm mult-commutativity2
  (equal (mult i (mult j k)) (mult j (mult i k))))
```

9 Accumulators

A common form of recursion is to decrement some argument while building the final answer in another. This allows the function to inspect the partially computed answer. In addition, such functions are often the functional expression of an iterative process and are preferred over other forms of recursion because they are tail-recursive, thus allowing compilers to make optimizations that (for example) avoid stack overflows. The arguments that are being built up are called "accumulators." Proving theorems about accumulator-using functions frequently requires care in stating sufficiently general theorems.

Consider, for example, the challenge of reversing a list. One definition is

```
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
    nil))
```

This definition suffers two inefficiencies in terms of the resources required to execute it. The first is that it requires stack space proportional to the length of the list, because for every `cons` in the `cdr`-chain of the input, execution must push a new stack frame so that it can "remember" to do the `app`. The second is that the `app` copies the list returned by the recursive call and returns the copy; the memory allocated to creating the recursive call's answer is unreachable "garbage" as soon as the `app` has finished with it.

The following tail-recursive version eliminates both of these drawbacks.

```
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
    a))
```

This is the functional expression of the code fragment:

```
while consp(x) { a = cons(car(x),a); x = cdr(x);};
return a;
```

It is the same as `rev` if `a` is initialized to `nil`. For example, `(rev1 '(1 2 3) nil)` is `'(3 2 1)`.

Suppose we wanted to prove `(equal (rev1 x nil) (rev x))`. Think about proving this by induction. It is clear we should induct on `x` by `cdr`. The induction hypothesis is about `(rev1 (cdr x) nil)`. The induction conclusion is about `(rev1 x nil)`. When we expand that term in the conclusion it becomes `(rev1 (cdr x) (cons (car x) nil))`. Note that the `rev1` term in the hypothesis does not match the `rev1` term in the conclusion. In the hypothesis, the accumulator is `nil` but in the expanded conclusion it is `(cons (car x) nil)`. We would like to "instantiate" `nil` to be `(cons (car x) nil)`, but of course we cannot instantiate anything but a variable.

This theorem is not strong enough to be inductively provable. If we think of `rev1` as an expression of an iteration, then the main theorem we are proving is about the *first* entry to the loop (when `a` is `nil`) and we must think about an arbitrary entry to the loop. Put another way, instead of thinking about `(rev1 x nil)` we must think about `(rev1 x a)`. This is just the generalization problem.

So what is the relation between `(rev1 x a)` and `(rev x)`? One way to help discover the general form of the theorem we are seeking is to try a few examples. For example, `(rev1 '(1 2 3) '(4 5 6))` is `(3 2 1 4 5 6)`.

The obvious general form of the theorem is `(equal (rev1 x a) (app (rev x) a))`.

Hint on How To Prove Things: *When dealing with a function that has an accumulator argument, never try to prove a theorem about the function by induc-*

tion unless the accumulator argument is a variable symbol. That is, think about the most general legal call of the function, not the initial call.

As soon as you replace the `nil` in `(rev1 x nil)` by a new variable symbol `a` you are confronted with the problem: what happens to `a` on the other side of the theorem? In this case, we use `app` to "connect" the expected answer, `(rev x)`, to the initial value of `a`. It is nice that `app` is already known to us. Often, however, you will have to invent a new function to relate the final answer to the initial value of the accumulator. This suggests the advice given earlier: do not be afraid to introduce new concepts to explain what is happening.

Now consider proving `(equal (rev1 x a) (app (rev x) a))` by induction. The Induction Principle allows us to replace `a` in the hypothesis by any term we wish, if we are inducting on `x` (replacing `x` by the smaller `(cdr x)`). So which `a` do we choose? The answer is obvious: the `a` that we will need if we expand `(rev1 x a)` in the conclusion. In particular, the choice of `a` in the hypothesis is `(cons (car x) a)`.

Here is ACL2's proof of the theorem. Note how trivial it is.

```
(defthm rev1-is-app-rev
  (equal (rev1 x a) (app (rev x) a)))

Name the formula above *1.

Perhaps we can prove *1 by induction.  Two induction schemes
are suggested by this conjecture.  Subsumption reduces
that number to one.

We will induct according to a scheme suggested by (rev1 x a).
This suggestion was produced using the :induction rules
rev and rev1.  If we let  (:p a x) denote *1 above then
the induction scheme we'll use is
(and (implies (not (consp x)) (:p a x))
     (implies (and (consp x)
                   (:p (cons (car x) a) (cdr x)))
              (:p a x))).
This induction is justified by the same argument used
to admit rev1, namely, the measure (acl2-count x) is decreasing
according to the relation o< (which is known to be well-
founded on the domain recognized by o-p).  Note, however,
that the unmeasured variable a is being instantiated.
When applied to the goal at hand the above induction scheme
produces the following two nontautological subgoals.

Subgoal *1/2
(implies (not (consp x))
         (equal (rev1 x a) (app (rev x) a))).

But simplification reduces this to t, using the :definitions
app, rev and rev1, the :executable-counterpart of consp
```

```
and primitive type reasoning.

Subgoal *1/1
(implies (and (consp x)
              (equal (rev1 (cdr x) (cons (car x) a))
                     (app (rev (cdr x)) (cons (car x) a))))
         (equal (rev1 x a) (app (rev x) a))).

But simplification reduces this to t, using the :definitions
app, rev and rev1, the :executable-counterpart of consp,
primitive type reasoning and the :rewrite rules associativity-
of-app, car-cons and cdr-cons.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( defthm rev1-is-app-rev ...)
Rules: ((:definition app)
        (:definition rev)
        (:definition rev1)
        (:executable-counterpart consp)
        (:fake-rune-for-type-set nil)
        (:induction rev)
        (:induction rev1)
        (:rewrite associativity-of-app)
        (:rewrite car-cons)
        (:rewrite cdr-cons))
Warnings:  None
Time:  0.01 seconds (prove: 0.01, print: 0.00, other: 0.00)
 rev1-is-app-rev
ACL2 !>
```

Note how we oriented the rewrite rule generated from `rev1-is-app-rev`: we eliminate `rev1` in favor of the nicer functions `app` and `rev`. While `rev1` is computationally efficient, it is often hard to state inductively provable theorems about it because the accumulator argument must always be occupied by a variable symbol.

Hint on How To Prove Things: *When you discover a general theorem about an accumulator-using function relating it to primitive recursive functions, use the new theorem to eliminate the accumulator-using function from future problems.*

Exercises

Use The Method to find proofs for each of the theorems below.

Problem 9.1 The following two functions are `nump` analogues of factorial. Prove they are equivalent:

```
(equal (fact1 n '(nil)) (fact n))  ; '(nil) is ''one''
```

where

```
(defun fact (n)
  (if (consp n)
      (mult n (fact (cdr n)))
    '(nil)))
(defun fact1 (n a)
  (if (consp n)
      (fact1 (cdr n) (mult n a))
    a))
```

Problem 9.2

```
(equal (mc-flatten x nil) (leaves x))
```

where

```
(defun mc-flatten (x a)
  (if (consp x)
      (mc-flatten (car x)
                  (mc-flatten (cdr x) a))
    (cons x a)))
```

The function `mc-flatten` is an "almost tail-recursive" version of `leaves` first written by John McCarthy. It has the interesting property that it produces no garbage: every `cons` it creates is in the final answer, unlike `leaves`.

10 Conclusion

We have illustrated how to find simple rigorous proofs. We first repeat all the hints given so far, and then we add a few more.

Hint on How To Prove Things: *Induction must be applied to strong theorems, not weak ones! Always try to invent the strongest theorems you can think of!*

Hint on How To Prove Things: *Give careful thought to the "preferred" forms you use in your proofs and provide yourself with lemmas that allow you, insofar as possible, to canonicalize terms.*

Hint on How To Prove Things: *When using the ACL2 system,* never *prove a named theorem without understanding its effect as a rule!*

Hint on How To Prove Things: *Realize that when you are dealing with arithmetic, your sense of what is "straightforward" has been honed by many* *years of drill-and-practice with manipulating algebraic properties of numbers. Be prepared to "explain" formally why some arithmetic relations hold!*

Hint on How To Prove Things: *If you are doing arithmetic proofs with ACL2, start by including one of the arithmetic books into your script. The most commonly used book is included by adding the command* `(include-book "arithmetic/top-with-meta" :dir :system)`.

Hint on How To Prove Things: *Realize that when you are dealing with arithmetic you may be doing inequality chaining, not replacement of equals by equals, and make that form of reasoning explicit in your notation.*

Hint on How To Prove Things: *Every time you encounter a new theorem you should give thought to how it is to be used in subsequent proofs.*

Hint on How To Prove Things: *Ensure that your rules do not loop! One way to do this is to keep in mind some ordering on your preferred terms and be sure that the right-hand side of each rule is lower in this ordering than the left-hand side.*

Hint on How To Prove Things: *Definitions are (generally) used as expansion rules, i.e., function calls are replaced by their instantiated bodies. This imposes a restriction on your choice of preferred forms: function bodies are preferable to function calls. If you want to override that built-in preference in ACL2, you should disable the functions after proving the rules you need about them.*

Hint on How To Prove Things: *When designing your rewrite rules, be sure the left-hand sides are in your preferred form!*

Hint on How To Prove Things: *Keep your proofs in the "simplify, induct, simplify" form. That is, identify each inductively proved lemma you need in a proof, write it down, and give it a name. Do not get into the habit of inventing and inductively proving lemmas "on the fly" in the middle of other proofs.*

It is better that you understand and control the lemma decomposition of your theorems.

Hint on How To Prove Things: *When considering a new conjecture to prove, look for general theorems can will prove it by simplification – rewriting and chaining – before you consider proving it by induction.*

Hint on How To Prove Things: *Look for pairs of adjacent function symbols and try to think of rules that simplify those expressions.*

Hint on How To Prove Things: *When forming new conjectures, test them on constants.*

Hint on How To Prove Things: *Often you will have to invent new concepts – concepts not involved in your main theorem – to state the lemmas you need.*

Hint on How To Prove Things: *If you have introduced hypotheses in your lemmas, be sure you prove that the appropriate terms satisfy those hypotheses.*

Hint on How To Prove Things: *Separate your concerns!*

Hint on How To Prove Things: *We recommend that the novice ACL2 user not rely on ACL2's creative contributions in the beginning. As the problems become harder, ACL2's creative contributions count for less and less – and its ability to carry out massive automatic simplifications using user-specified rules counts for more and more. So the novice is encouraged to learn to spot the need for rules and to program ACL2 to use them.*

Hint on How To Prove Things: *Whenever a proof fails (or you want to reduce a proof to the recommended form), read the formula at the first checkpoint and look for a lemma decomposition. Sometimes, it helps to read a few formulas past the first checkpoint – often ACL2's heuristics come fairly close to generating the needed lemma, or at least creating a term that will suggest the lemma to you. So if the checkpoint does not suggest anything, read on.*

Hint on How To Prove Things: *When you know a function is commutative, use that fact to arrange the arguments in some canonical order. Thus, if* `add` *is known to be commutative, then whenever you see* `(add b a)`*, rewrite it to* `(add a b)`*. But do not use the rule the other way – to move things out of order – or you will loop forever! ACL2 uses this heuristic and uses a lexicographic ordering on terms.*

Hint on How To Prove Things: *The heuristic advice about commutative functions does not help you if you are rewriting* `(add b (add a c))` *because the ordering (probably) will not prefer* `(add a c)` *over* `b`*. But the theorem above, which we call a "commutativity2" theorem, allows that swap and ACL2 uses such theorems to arrange the ordering of terms. Given associativity, commutativity, commutativity2, and these heuristics, you can arrange nests of such functions* *into right-associated form with the arguments ascending in the order. That is what ACL2 does.*

Hint on How To Prove Things: *When dealing with a function that has an accumulator argument, never try to prove a theorem about the function by induction unless the accumulator argument is a variable symbol. That is, think about the most general legal call of the function, not the initial call.*

Hint on How To Prove Things: *When you discover a general theorem about an accumulator-using function relating it to primitive recursive functions, use the new theorem to eliminate the accumulator-using function from future problems.*

See `acl2-tutorial` for further introduction, with subtopics containing many other helpful tips for using the ACL2 logic and theorem prover, and with many more examples.

One important tip there is that there are many books of rules developed by ACL2 users. We did not stress the use of books here simply because we are trying to train you to use The Method and to program the simplifier yourself. Those are skills you will need. But as you master those skills and move on to bigger projects, it is very helpful when you can build on the work of others. That is how mathematics has built such a magnificent body of results. So now we enshrine this advice as a hint.

Hint on How To Prove Things: *Build on the work of others instead of inventing everything yourself. In the ACL2 setting, learn about the books available (by visiting the Mathematical Tools link on the ACL2 home page), learn about and use* `include-book` *(see* `include-book`*) to load books into your scripts, and learn to use* `certify-book` *(see* `certify-book`*) to create books others can use.*

If you want to learn more about ACL2, we recommend you buy the book *Computer-Aided Reasoning: An Approach* by Kaufmann, Manolios, and Moore [6]. While Kluwer Academic Press owns the electronic and hardback copyrights, the authors own the paperback rights and sell a spiral-bound version for close to their cost, about $15 plus shipping. (You can find the book in hardback elsewhere, for well over its original price of about $120.) See the Books and Papers link on the ACL2 home page for ordering details. The book contains about 150 exercises and the solutions are on the web.

The companion book, *Computer-Aided Reasoning: ACL2 Case Studies*, edited by the same authors [5], is also a very valuable resource because it presents detailed notes on many large-scale proof projects and the actual source scripts are available on the web. The companion book is available under the same copyrighting terms and approximately the same prices as the first.

A detailed account of proof development for a non-trivial example may be found in [9]. The proof described there was carried out with ACL2's predecessor, Nqthm, but lessons therein pertain to ACL2 usage as well.

The Workshops link on the home page is also a good source of material. ACL2 workshop papers are usually accompanied by complete proof scripts, which are posted on the ACL2 home page.

One more hint is in order.

Hint on How To Prove Things: *Practice makes perfect. There is no substitute for experience. Think of theorems to prove and work out the proofs!*

A Definitions

In this appendix we include definitions of all the functions mentioned in our exercises. This appendix thus answers the exercises that just require definitions! So don't look here until you do those exercises. But to do the proof-based exercises, it might be best to use our definitions.

We have presented the definitions in alphabetical order so you can find them. ACL2 requires definitions to be presented bottom-up: define concepts before using them.

```
(defun add (x y)
  (if (consp x)
      (cons nil (add (cdr x) y))
    (mapnil y)))

(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

(defun collect-lonesomep (a b)
; collect elements of a that are lonesome in b
  (if (consp a)
      (if (lonesomep (car a) b)
          (cons (car a)
                (collect-lonesomep (cdr a) b))
        (collect-lonesomep (cdr a) b))
    nil))

(defun dup (x)
  (if (consp x)
      (cons (car x)
            (cons (car x)
                  (dup (cdr x))))
      nil))

(defun fact (n)
  (if (consp n)
      (mult n (fact (cdr n)))
    '(nil)))
```

```
(defun fact1 (n a)
  (if (consp n)
      (fact1 (cdr n) (mult n a))
    a))

(defun foundp (x a)
  (if (consp a)
      (if (equal x (car (car a)))
          t
        (foundp x (cdr a)))
    nil))

(defun int (x y)
  (if (consp x)
      (if (memp (car x) y)
          (cons (car x) (int (cdr x) y))
        (int (cdr x) y))
    nil))

(defun leaves (x) ; the leaves of x
  (if (consp x)
      (app (leaves (car x))
           (leaves (cdr x)))
    (cons x nil)))

(defun lonesomep (e lst)
  (if (mem e lst)
      (not (mem e (cdr (mem e lst))))
    nil))

(defun lonesomes (x)
  (collect-lonesomep (leaves x) (leaves x)))

(defun lookup (x a)
  (if (consp a)
      (if (equal x (car (car a)))
          (cdr (car a))
        (lookup x (cdr a)))
    nil))

(defun mapnil (x)
  (if (consp x)
      (cons nil (mapnil (cdr x)))
    nil))

(defun mc-flatten (x a)
  (if (consp x)
      (mc-flatten (car x)
                  (mc-flatten (cdr x) a))
```

```
    (cons x a)))

(defun mem (e x) ; where does e occur in x?
  (if (consp x)
      (if (equal e (car x))
          x
        (mem e (cdr x)))
    nil))

(defun memp (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (memp e (cdr x)))
    nil))

(defun mult (x y)
  (if (consp x)
      (add y (mult (cdr x) y))
    nil))

(defun nump (x)
  (if (consp x)
      (and (equal (car x) nil)
           (nump (cdr x)))
    (equal x nil)))

(defun properp (x)
  (if (consp x)
      (properp (cdr x))
    (equal x nil)))

(defun raise (x y)
  (if (consp y)
      (mult x (raise x (cdr y)))
    '(nil)))

(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
    nil))

(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
    a))

(defun seteqp (x y)
  (and (subp x y)
       (subp y x)))
```

```
(defun swaptree (x)
  (if (consp x)
      (cons (swaptree (cdr x))
            (swaptree (car x)))
    x))

(defun subp (x y)
  (if (consp x)
      (if (memp (car x) y)
          (subp (cdr x) y)
        nil)
    t))

(defun treecopy (x)
  (if (consp x)
      (cons (treecopy (car x))
            (treecopy (cdr x)))
     x))

(defun ziplists (x y)
  (if (consp x)
      (cons (cons (car x) (car y))
            (ziplists (cdr x) (cdr y)))
    nil))
```

B Emacs

Emacs has an interactive tutorial. Type `meta-x help-with-tutorial enter`.
Here are some commonly used commands.

To Insert Text

just type it

To Move Around

ctrl-f	move forward one character
ctrl-b	move backward one character
ctrl-n	move down to next line
ctrl-p	move up to previous line
ctrl-meta-f	moves forward over balanced expression
ctrl-meta-b	moves backward over balanced expression
ctrl-meta-u	moves up one level of parens.
meta-<	move to the beginning of the buffer
meta->	move to the end of the buffer

To Cut and Paste

ctrl-d	delete one character
ctrl-k	kill one line – and put it in the “kill ring”
ctrl-meta-k	kill one balanced expression - and put it in the “kill ring”
ctrl-y	to yank (paste) text from the kill ring back into the buffer

Other

ctrl-x b	select another buffer (type buffer name)
ctrl-x ctrl-f	read a file into a buffer of that name
ctrl-x ctrl-s	write a buffer to the file it came from
ctrl-x ctrl-w	write a buffer to the file you name
ctrl-meta-q	indent the list expression immediately after the cursor in a way consistent with the parentheses

The ACL2 source code distribution comes with a pre-defined Emacs library that contains many useful commands for interacting with ACL2. Just put the following in the `.emacs` file in your home directory, replacing `<dir>` by the absolute pathname of your local ACL2 source code directory.

```
(load "<dir>/emacs/emacs-acl2.el")
```

This library also enables navigation of proofs and checkpoints with an emacs-based tool; see proof-tree-emacs.

References

1. B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods.* Springer-Verlag, 1999.
2. B. Brock and J S. Moore. A mechanically checked proof of a comparator sort algorithm. In *Engineering Theories of Software Intensive Systems.* IOS Press, Amsterdam, 2005 (to appear).
3. David Greve and M. Wilding. A separation kernel formal security policy, 2003.
4. M. Kaufmann. Modular proof: The fundamental theorem of calculus. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 75–92, Boston, MA., 2000. Kluwer Academic Press.
5. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Press, Boston, MA., 2000.
6. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.
7. M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. In *http://www.cs.utexas.edu/users/moore/publications/km97a.ps.gz.* Dept. of Computer Sciences, University of Texas at Austin, 1997.
8. M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/.* Dept. of Computer Sciences, University of Texas at Austin, 2004.
9. M. Kaufmann and P. Pecchiari. Interaction with the boyer-moore theorem prover: A tutorial study using the arithmetic-geometric mean theorem. *Journal of Automated Reasoning*, 16(1–2):181–222, 1996.
10. H. Liu and J S. Moore. Executable JVM model for analytical reasoning: A study. In *Workshop on Interpreters, Virtual Machines and Emulators 2003 (IVME '03)*, San Diego, CA, June 2003. ACM SIGPLAN.
11. W. McCune and O. Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 265–282, Boston, MA., 2000. Kluwer Academic Press.

12. J S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-03.
13. J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, September 1998.
14. D. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998. http://www.onr.com/user/russ/david/k7-div-sqrt.html.
15. D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–232, Boston, MA., 2000. Kluwer Academic Press.
16. J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the ACL2 Workshop, 2002*. http://www.cs.utexas.edu/users/moore/acl2/workshop-2002, Grenoble, April 2002.
17. R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. In *Proceedings of ACL2 Workshop 2000*. Department of Computer Sciences, Technical Report TR-00-29, November 2000. http://www.cs.utexas.edu/users/moore/acl2/workshop-2000/final/sumners2/paper.ps.