Part IV

Gaming

8

The Mechanical Theorem Prover

This part of the book is concerned with the mechanization of the logic. Our goal is to teach you how to use the theorem prover. We start, in this chapter, by sketching how the theorem prover works. Of course, knowing how something works—*e.g.*, an automobile, a programming language, a violin—is quite different from knowing how to use it effectively. So, in Chapter 9, we begin to explain how to use the theorem prover. Finally, in Chapter 10, we use the theorem prover to do many example proofs.

As you read this chapter you may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing. But you may find that the machine's inability to fill in the gap is because your "proof" was simply wrong. Indeed, you may even find that the formula you "proved" is not even a theorem!

You may come to think of the proof process as a game. The theorem is the "opponent." It will use all legal means to dodge your weapons and squirm free of your traps and fences. It can hide amid innocuous detail, shatter into a swarm of subproblems, or stand crystalline still and shimmering in front of you, daring you to find a chink in its armor. In recognition of this view of theorem proving we have named this part of the book "Gaming." You will be hard pressed to find a more challenging game.

8.1 A Sample Session

Here is how the theorem prover responds to the command to prove that app (defined on page 104) is associative. The user input consists of the first three lines of text following the ACL2 > prompt. Everything else was produced automatically. You should read this proof and compare it to the one on page 105.

```
ACL2 >(defthm associativity-of-app
        (equal (app (app a b) c)
               (app a (app b c))))
```

```
Name the formula above *1.

Perhaps we can prove *1 by induction.  Three induction schemes
are suggested by this conjecture.  Subsumption reduces that
number to two.  However, one of these is flawed and so we
are left with one viable candidate.

We will induct according to a scheme suggested by (APP A B).
If we let  (:P A B C) denote *1 above then the induction
scheme we'll use is
(AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A) B C))
              (:P A B C))
     (IMPLIES (ENDP A) (:P A B C))).
This induction is justified by the same argument used to
admit APP, namely, the measure (ACL2-COUNT A) is decreasing
according to the relation E0-ORD-< (which is known to be
well-founded on the domain recognized by E0-ORDINALP).  When
applied to the goal at hand the above induction scheme produces
the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP A))
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP A)
              (EQUAL (APP (APP (CDR A) B) C)
                     (APP (CDR A) (APP B C))))
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

But simplification reduces this to T, using the :definition
APP, the :rewrite rules CDR-CONS and CAR-CONS and primitive
type reasoning.

Subgoal *1/1
(IMPLIES (ENDP A)
         (EQUAL (APP (APP A B) C)
                (APP A (APP B C)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
```

```
        (EQUAL (APP (APP A B) C)
               (APP A (APP B C)))).

But simplification reduces this to T, using the :definition
APP and primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM ASSOCIATIVITY-OF-APP ...)
Rules: ((:REWRITE CDR-CONS)
        (:REWRITE CAR-CONS)
        (:DEFINITION NOT)
        (:DEFINITION ENDP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:DEFINITION APP))
Warnings:  None
Time:  0.04 seconds (prove: 0.03, print: 0.00, other: 0.01)
 ASSOCIATIVITY-OF-APP
```

8.2 Organization of the Theorem Prover

As noted earlier and as depicted in Figure 8.1, the theorem prover takes input from both you and a data base, called the *logical world* or simply *world*. The world embodies a theorem proving strategy, developed by you and codified into *rules* that direct certain aspects of the theorem prover's behavior. When trying to prove a theorem, the theorem prover applies your strategy, possibly using hints you supply with the theorem, and prints its proof attempt. You have no interactive control over the system's behavior once it starts a proof attempt, except that you can interrupt it and abort the attempt. When the system succeeds, new rules, derived from the just-proved theorem, are added to the world according to directions supplied by you. When the system fails, you must inspect the proof attempt to see what went wrong.

Your main activity when using the theorem prover is designing your theorem proving strategy and expressing it as rules derived from theorems. There are over a dozen kinds of rules, each identified by a *rule class* name. The most common are rewrite rules, but other classes include type-prescription, linear, elim, and generalize rules. The basic command for telling the system to (try to) prove a theorem and, if successful, add rules to the data base is the `defthm` command.

(`defthm` *name formula*
  `:rule-classes` (*class*$_1$ ... *class*$_n$))

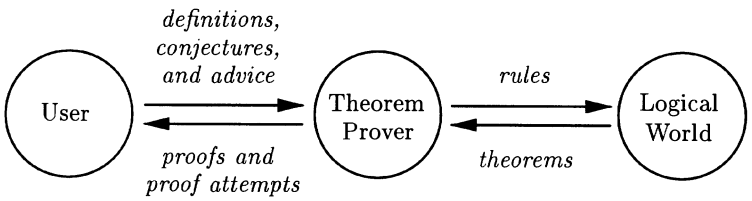

Figure 8.1: Data Flow in the Theorem Prover

The command directs the system to try to prove the given formula and, if successful, remember it under the name *name* and build it into the data base in each of the ways specified by the $class_i$. We will discuss many of the common rule classes in this chapter. But to find out details of the various rule classes, see `rule-classes`, and the documentation links under it.

Every rule has a *status* of either *enabled* or *disabled*. The theorem prover only uses enabled rules. So by changing the status of a rule or by specifying its status during a particular step of a particular proof with a "hint" (see `hints`), you can change the strategy embodied in the world. A set of rules can be collected together into a *theory* and the entire theory can be enabled or disabled. This allows a world to offer alternative strategies from which you may choose by enabling the appropriate theory. See `in-theory` for details. We ignore these issues during our description of the theorem prover but remind the reader that only enabled rules are relevant.

The theorem prover is organized as shown in Figure 8.2. At the center is a *pool* of formulas to be proved. Initially, your conjecture is the only formula in the pool. Surrounding the pool are six proof techniques. Although not depicted in the figure, each of the proof techniques uses rules in the logical world. To operate on a non-empty pool, a formula is drawn out and given to the first technique. Each technique is either applicable to the formula, in which case it reduces the formula to a set of n other formulas and deposits them into the pool, or else the technique is inapplicable to the formula, in which case it passes the formula to the next technique. In the case where n is 0, the formula is actually proved by the technique. The original conjecture is proved when there are no formulas left in the pool and the proof techniques have all halted. This organization is sometimes called "the waterfall" because in [5] it was described in those terms.[1]

The six proof techniques surrounding the pool are described at a high level in the successive sections below.

[1]Not discussed here is the modification to the waterfall to accommodate `force` and `case-split`.

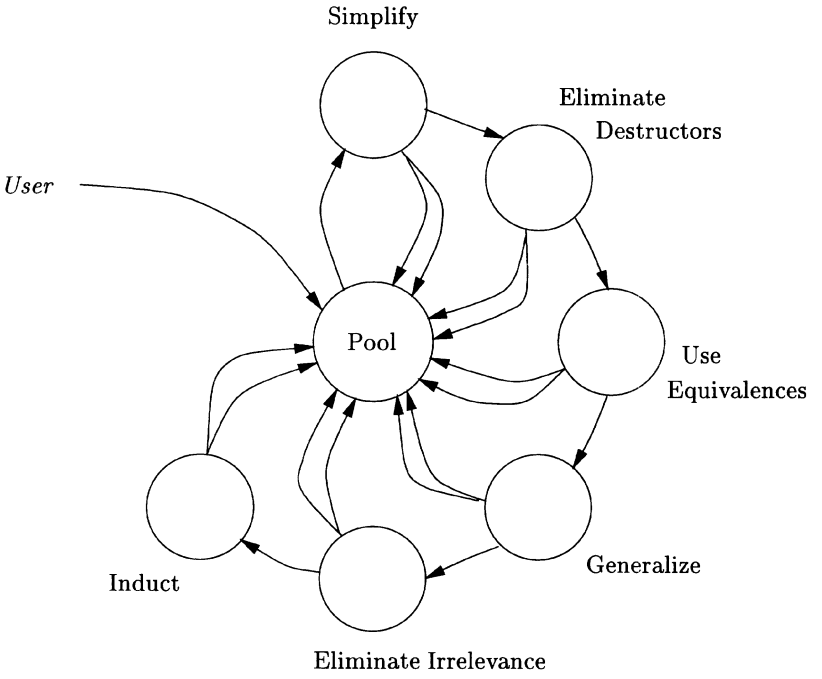

Figure 8.2: Organization of the Theorem Prover

The descriptions are tied together by a single classic Boyer-Moore example, the so-called `rev-rev` example. Consider the following recursive function definition:

```
(defun rev (x)
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))
```

This function reverses a list. For example, `(rev '(1 2 3))` evaluates to `(3 2 1)`. It has the property that if `a` is a true list, *i.e.*, one whose "final cdr" is `nil`, then `(rev (rev a))` is `a`. The hypothesis is necessary: `(rev (rev '(1 2 3 . 4)))` is `(1 2 3)`, not the original input.

The `rev-rev` formula we shall prove is

```
(implies (true-listp a)
         (equal (rev (rev a)) a)).
```

When this formula is put into the pool, it is drawn out and given to the simplification technique. That technique can do nothing with it and passes it to the next technique. In fact none of the first five techniques apply and the formula arrives at induction.

8.2.1 Induction

The key to a successful inductive argument is figuring out how to construct a proof of the formula from certain instances of the same formula. Those instances must be "smaller." The recursive functions in the formula provide suggestions for which instances to use: supply the instances obtained by expanding one or more of the recursive functions. Each recursive function decomposes its arguments in a well-founded way. Not only is this established when the function is admitted but the admission process identifies a set of measured arguments whose "size" is decreasing. Thus, an induction is suggested by each application of the function in which those measured arguments are occupied by distinct variable symbols: under the case split used in the function definition, provide inductive hypotheses corresponding to each recursive call (*e.g.*, replace each measured variable by the term used in its slot in each recursion). This suggested induction is justified by the same measure used to justify the function admission. See Exercise 6.25 for an exploration of the duality between recursion and the suggested induction.

Often, more than one set of arguments could be measured to justify a function definition. To each such set there corresponds an induction. But ACL2 only finds one justification at definition-time and hence, initially, each recursive function suggests just one induction. It is possible to prove an induction rule (see `induction`) so that a term suggests other inductions.

When a formula arrives at the induction technique, ACL2 computes all the inductions suggested by the terms in the formula. It then compares them, possibly combining several into one, and selects one regarded as most appropriate. It then prints the selected induction scheme, applies it to the formula at hand, uses simple propositional calculus to normalize the result, and puts each of the new formulas back into the pool. You can override its choice of induction by supplying an induction hint. See hints.

The "propositional calculus normalization" sometimes makes the instantiation of the induction scheme look different than the scheme itself. Suppose the induction step of the scheme is to assume test q and inductive instance p' to prove p. Suppose that the formula to be proved, p, is of the form $\alpha \rightarrow \beta$. Then the scheme would seem to call for the induction step $(q \wedge (\alpha' \rightarrow \beta')) \rightarrow (\alpha \rightarrow \beta)$. But the propositional normalization actually produces two formulas to prove: $(q \wedge \neg\alpha' \wedge \alpha) \rightarrow \beta$ and $(q \wedge \beta' \wedge \alpha) \rightarrow \beta$. The conjunction of these two is propositionally equivalent to the step required by the scheme.

The application of the induction technique to the `rev-rev` formula produces the following output. We have manually added the line numbers on the left.

```
 4. Name the formula above *1.
 5.
 6. Perhaps we can prove *1 by induction.  Two induction schemes
 7. are suggested by this conjecture.  These merge into one derived
 8. induction scheme.
 9.
10. We will induct according to a scheme suggested by (REV A).
11. If we let  (:P A) denote *1 above then the induction scheme
12. we'll use is
13. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
14.               (:P A))
15.      (IMPLIES (ENDP A) (:P A))).
16. This induction is justified by the same argument used to
17. admit REV, namely, the measure (ACL2-COUNT A) is decreasing
18. according to the relation E0-ORD-< (which is known to be
19. well-founded on the domain recognized by E0-ORDINALP).  When
20. applied to the goal at hand the above induction scheme produces
21. the following three nontautological subgoals.
```

Line 4 is printed when a subgoal enters the induction mechanism. Lines 6–8 describe the candidate inductions. The candidates were suggested by `(true-listp a)` and `(rev a)`; but both make the same suggestion: induction on the cdr structure of `a`. Lines 13–15 give the induction scheme selected. This scheme calls for two formulas to be proved (the induction step, on the first two lines, and the base case, on the last).

Lines 16–19 explain why the induction is legal under the induction principle. Finally, lines 20–21 indicate how many goals are being put into the

pool. Note that three goals are added here, not two as might indicated by the induction scheme. Propositional normalization is responsible for the difference.

The three subgoals are not printed until they are removed from the pool for proof, but the goals and the names they are assigned are shown below.

```
 23. Subgoal *1/3
 24. (IMPLIES (AND (NOT (ENDP A))
 25.               (EQUAL (REV (REV (CDR A))) (CDR A))
 26.               (TRUE-LISTP A))
 27.          (EQUAL (REV (REV A)) A))

 95. Subgoal *1/2
 96. (IMPLIES (AND (NOT (ENDP A))
 97.               (NOT (TRUE-LISTP (CDR A)))
 98.               (TRUE-LISTP A))
 99.          (EQUAL (REV (REV A)) A))

103. Subgoal *1/1
104. (IMPLIES (AND (ENDP A) (TRUE-LISTP A))
105.          (EQUAL (REV (REV A)) A))
```

`Subgoal *1/1` is the base case. The other two, together, are the induction step. `Subgoal *1/3` is the "interesting" part of the induction step, in which one uses the conclusion of the induction hypothesis to prove the conclusion of the induction conclusion. `Subgoal *1/2` is a frequently overlooked case in which one must show that the hypothesis of the induction conclusion implies the hypothesis of the induction hypothesis. All three of these are put in the pool by induction. They are drawn out in the order listed above.

8.2.2 Simplification

Simplification is the heart of the theorem prover. We will discuss simplification in more detail later. The main things it does are:

- apply propositional calculus, equality, and rational linear arithmetic decision procedures,
- use type information and forward chaining rules to construct a "context" describing the assumptions governing each occurrence of each subterm,
- rewrite each subterm in the appropriate context, using definitions, conditional rewrite rules, and metafunctions,
- use propositional calculus normalization to convert the resulting formula to an equivalent set of formulas, reduce the set under subsumption, and deposit the surviving formulas back in the pool.

By "conditional rewrite rules" we mean rules that cause the system to replace certain terms by other terms, provided certain hypotheses can be established. For example, the axiom `(equal (car (cons x y)) x)` gives rise to a rewrite rule in the data base that directs the system to replace every term of the form `(car (cons` α β`))` by α. The axiom `(implies (not (consp x)) (equal (car x) nil))` gives rise to a rule that directs the system to replace `(car` α`)` by `nil`, if the system can prove that `(consp` α`)` is false. When you command the system to prove a theorem and to store it as a rewrite rule, the system generates such a rule. The generated rule is sensitive to the exact syntactic form of the theorem.

ACL2 supports congruence-based rewriting: it supports "substitution of equivalents," not just substitution of `equal`s. That is, rewrite rules can be generated from theorems that conclude not just with an `equal`-term but an *equiv*-term, where *equiv* is an arbitrary user-defined equivalence relation. You may define special equivalence relations and prove congruence rules permitting substitution of equivalents rewriting deep inside of terms. See Section 8.3.1 and see also and `equivalence` `congruence`.

The simplifier above is not guaranteed to produce formulas that are stable under simplification; repeated trips through the simplifier, via insertion into and extraction from the pool, are used to reach the final stable form (if any).

When `Subgoal *1/3`, above, arrives at the simplifier, it is simplified in two successive steps. The first merely expands `(ENDP A)` to `(NOT (CONSP A))` and removes the double `NOT`'s, naming the resulting formula `Subgoal *1/3'`. (When a formula is transformed to exactly one other formula, the new formula is given the same name as the old one with a prime appended at the end.) When `Subgoal *1/3'` is put into the pool, it is immediately extracted and simplified. That simplification expands the recursive functions `TRUE-LISTP` and `REV` to produce `Subgoal *1/3''`, which is put into the pool.

Line 23 below shows `Subgoal *1/3` as it is drawn out of the pool and given to the simplifier. The message printed by the first simplification is on line 29. The formula produced follows that. The second simplification's message and output formula start at line 37.

```
23. Subgoal *1/3
24. (IMPLIES (AND (NOT (ENDP A))
25.               (EQUAL (REV (REV (CDR A))) (CDR A))
26.               (TRUE-LISTP A))
27.          (EQUAL (REV (REV A)) A)).
28.
29. By the simple :definition ENDP we reduce the conjecture to
30.
31. Subgoal *1/3'
32. (IMPLIES (AND (CONSP A)
33.               (EQUAL (REV (REV (CDR A))) (CDR A))
```

```
34.               (TRUE-LISTP A))
35.          (EQUAL (REV (REV A)) A)).
36.
37. This simplifies, using the :definitions TRUE-LISTP and REV,
38. to
39.
40. Subgoal *1/3''
41. (IMPLIES (AND (CONSP A)
42.               (EQUAL (REV (REV (CDR A))) (CDR A))
43.               (TRUE-LISTP (CDR A)))
44.          (EQUAL (REV (APP (REV (CDR A)) (LIST (CAR A))))
45.                 A)).
```

The last subgoal is immediately extracted from the pool and given to the simplifier, but the simplifier does not change it. It is stable and is passed to the next proof technique.

8.2.3 Destructor Elimination

Destructor elimination is a way to get rid of certain function applications by expanding certain variables into terms that make explicit their construction. For example, suppose a formula mentions `(CAR A)` and `(CDR A)`. If `A` is not a cons, those expressions simplify to `NIL`. If `A` is a cons, we could, without loss of generality, replace `A` by `(CONS A1 A2)`, for new variable symbols `A1` and `A2`. Doing so would allow us to get rid of `(CAR A)` and `(CDR A)`, replacing them, respectively, with `A1` and `A2`.

Here is the output produced when destructor elimination is applied to `Subgoal *1/3''` above.

```
47. The destructor terms (CAR A) and (CDR A) can be eliminated
48. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
49. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
50. the following goal.
51.
52. Subgoal *1/3'''
53. (IMPLIES (AND (CONSP (CONS A1 A2))
54.               (EQUAL (REV (REV A2)) A2)
55.               (TRUE-LISTP A2))
56.          (EQUAL (REV (APP (REV A2) (LIST A1)))
57.                 (CONS A1 A2))).
```

Observe the hypothesis on line 53 produced by replacing `A` in `(CONSP A)`. This hypothesis is now manifest in the construction of `A`. When `Subgoal *1/3'''` is put into the pool, it is immediately drawn out and simplified. Simplification eliminates this redundant hypothesis and otherwise changes nothing. The simplified goal is named `Subgoal *1/3'4'`. (The system will

not produce a name with more than three primes, but you can think of `'4'` as four primes, `'5'` as five primes, and so on.)

The transformation above is justified logically by the

Axiom. `CAR-CDR-ELIM`:

```
(implies (consp x)
         (equal (cons (car x) (cdr x)) x)).
```

This axiom is an example of a more general form:

```
(implies (hyp x)
         (equal (constructor (dest1 x) ... (destn x))
                x)).
```

Such theorems can be stored as "destructor elimination" or elim rules. See `elim`. The (*dest*$_i$ *x*) are the *destructor* terms. When destructor elimination is applied to a formula containing an instance of some (*dest*$_i$ *x*) in which the variable *x* is bound to some variable *a*, the technique applies. It "splits" the formula into two cases according to whether (*hyp a*) is true and in the case where it is true, it replaces all of the *a*'s in the formula (except those inside *dest*$_i$ applications) by (*constructor* (*dest*$_1$ *a*) ... (*dest*$_n$ *a*)). Then it generalizes all the (*dest*$_i$ *a*) terms (including the ones just introduced) to distinct new variable symbols, $a_1, \ldots, a_n$. In generalizing it restricts the a_i using generalization rules discussed below. The resulting formulas are put in the pool.

Here is another example of a destructor elimination rule. Suppose `firstn` and `nthcdr` are defined so that the following is a theorem.

```
(implies (and (integerp n)
              (<= 0 n)
              (<= n (len x)))
         (equal (append (firstn n x) (nthcdr n x))
                x))
```

This is in the form of a destructor elimination rule. The destructor terms are `(firstn n x)` and `(nthcdr n x)`. The constructor is `append`. Suppose the destructor elimination technique were applied to the formula `(p (firstn i a) (nthcdr i a) i a)`, *i.e.*, to a formula involving one or more suitable instances of the destructor terms. Then destructor elimination would split the conjecture into two subgoals.

```
(implies (not (and (integerp i)
                   (<= 0 i)
                   (<= i (len a))))
         (p (firstn i a) (nthcdr i a) i a))
(implies (and (and (integerp i)
                   (<= 0 i)
                   (<= i (len (append u v)))))
              ...)
         (p u v i (append u v)))
```

The first of these subgoals handles the "pathological" case where the destructors are being "inappropriately" used. To prove that subgoal it would be best to have rules to reduce (firstn i a) and (nthcdr i a) to other expressions in this case.

The second of these subgoals handles the "normal" case. Note that here a has been replaced by (append u v), and (firstn i a) and (nthcdr i a) have been replaced, respectively, by u and v. The "..." in the hypotheses of the second subgoal stand for hypotheses about u and v that are derived by the generalization technique from rules about (firstn i a) and (nthcdr i a). Observe that the effect of the elimination rule here is to eliminate firstn and nthcdr in favor of append, *i.e.*, to trade "destructors" for "constructors." Whether this is a good move in the context of a proof really depends on which rules are in the data base. We often arrange strategies based on rewriting patterns of constructors.

A more sophisticated destructor elimination rule is shown below.

```
(implies (acl2-numberp x)
         (equal (+ (mod x y) (* y (floor x y))) x))
```

In this theorem, (mod x y) and (floor x y) are the destructor terms and the constructor is the lambda expression (lambda (d1 d2 y) (+ d1 (* y d2))). When this rule is available and the destructor elimination technique is presented with a formula containing, say (MOD I J) or (FLOOR I J), the technique splits on whether I is a number. In the affirmative case, destructor elimination replaces I by (+ R (* J Q)), (MOD I J) by R, and (FLOOR I J) by Q. Provided there are appropriate generalization lemmas available for mod and floor, this eliminates the destructors mod and floor in favor of the constructors + and * without loss of generality.

The destructor elimination technique, like the simplifier, actually supports substitution of equivalents for equivalents rather than just substitution of equals for equals. See page 139 and elim.

8.2.4 Use of Equivalences

The next step in the waterfall attempts the use of equalities appearing in the goal formula. If the formula contains the hypothesis (equal *lhs rhs*) and elsewhere in the formula there is an occurrence of *lhs*, then *rhs* is substituted for *lhs* in every such occurrence with one exception: if *lhs* occurs on one side of an equality, we only substitute into that side of the equality. This restricted substitution method is called *cross-fertilization*. If the equality hypothesis comes from an inductive argument, we throw it away after using it in this fashion. We treat equalities symmetrically and, when it is possible to substitute left-for-right and right-for-left, make a choice based on heuristics.

ACL2 actually supports a more general form of substitution involving equivalence relations. The use of equalities is generalized to the use of any equivalence relation, *equiv*, and substitution is correspondingly restricted to *equiv*-hittable occurrences. See page 139.

Returning to the `rev-rev` proof, recall that destructor elimination produced `Subgoal *1/3'''`, which was further simplified to `Subgoal *1/3'4'`. That subgoal cannot be further simplified and has no destructors in it. Thus, ACL2 tries to use the equivalences in it. Below is the subgoal, the message printed by equivalence substitution, and the formula produced and added to the pool.

```
62. Subgoal *1/3'4'
63. (IMPLIES (AND (EQUAL (REV (REV A2)) A2)
64.               (TRUE-LISTP A2))
65.          (EQUAL (REV (APP (REV A2) (LIST A1)))
66.                 (CONS A1 A2))).
67.
68. We now use the first hypothesis by cross-fertilizing
69. (REV (REV A2)) for A2 and throwing away the hypothesis.
70. This produces
71.
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
```

Observe the rather peculiar substitution chosen: A variable was replaced by a non-variable term on only one side of an `equal` even though the variable occurred on both sides.[2] The fact that this proof techniques discards a hypothesis makes it even more interesting. It is possible that the input formula is a theorem but the output formula is not. This does not mean that ACL2 is unsound! It means ACL2 may adopt a goal that it cannot prove. If the output formula is a theorem, the input formula is too. That is, the output may be more general than the input. Because we are likely, at this point in the waterfall, to appeal eventually to induction, ACL2's heuristics have been designed to try to strengthen the goal.

`Subgoal *1/3'5'` is added to the pool, extracted, and given in turn to simplification, destructor elimination, and equivalence substitution. None of them apply and so it arrives at the fourth technique.

[2]The substitution done by this heuristic may "undo" a previous rewrite or, as here, may appear to use the equality "backwards" (*i.e.*, not in the left-to-right sense imposed on any rewrite rule that may later be generated from the equality). While the rewriter gives special significance to the left/right orientation of certain equalities when generating rewrite rules, equalities in the formula being proved are used symmetrically.

8.2.5 Generalization

The fourth proof technique explicitly attempts to generalize the goal. The basic heuristic is to find a subterm that appears in both the hypothesis and the conclusion, in two different hypotheses, or on opposite sides of an equivalence, and replace that subterm by a new variable symbol. Furthermore, if type information (see type-prescription) or generalization rules (see generalize) can be used to restrict the type of the new variable, then it is so restricted. The generalized formula is then added to the pool.

Here is the contribution of generalization to the rev-rev proof.

```
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
76.
77. We generalize this conjecture, replacing (REV A2) by RV.
78. This produces
79.
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
```

Observe that (REV A2) occurs on both sides of an equal. It is generalized to the new variable RV. This new subgoal is added to the pool. None of the proof techniques discussed so far are applicable to it and the prover arrives at the fifth technique. But before we discuss that technique we discuss the restrictions imposed by generalize rules.

In this example there are no applicable generalize rules. But suppose we had previously proved the theorem that REV preserves the length of its argument,

```
(equal (len (rev x)) (len x))
```

and stored it as a generalize rule. Then when (REV A2) is replaced by RV, the generalization heuristic would add the following additional hypothesis.

```
(EQUAL (LEN RV) (LEN A2))
```

What actually happens is that the applicable generalize rules are instantiated so as to contain the term being generalized (*e.g.*, x in the rule is replaced by A2 so the rule mentions (REV A2)); that instance of the rule is added as a hypothesis to the goal; finally, the target term, (REV A2), is replaced by a new variable, RV.

The same restrictions are imposed when destructor terms are eliminated by the introduction of new variable symbols.

8.2.6 Elimination of Irrelevance

The fifth proof technique is the last one before induction. It attempts to eliminate irrelevant hypotheses in the conjecture, by partitioning them into cliques according to the variables they mention. If it can find an isolated clique of hypotheses, then either the formula is a theorem because those hypotheses are collectively false, or else they are irrelevant. It uses type information (see page 145) in a trivial way to guess that a clique is not false.

This occurs when `Subgoal *1/3'6'`, above, arrives at the elimination of irrelevance. Here is the exchange:

```
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
84.
85. We suspect that the term (TRUE-LISTP A2) is irrelevant to
86. the truth of this conjecture and throw it out.  We will thus
87. try to prove
88.
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
91.        (CONS A1 (REV RV))).
```

Observe that the hypothesis, `(TRUE-LISTP A2)`, is irrelevant once `(REV A2)` is generalized to `RV`. `Subgoal *1/3'7'` is then put into the pool.

When it is drawn out and passed around, none of the first five proof techniques apply to it. Induction will be tried. The order in which formulas are drawn from the pool is irrelevant, since all must be proved. But the draw is so orchestrated that we do not try to prove a subgoal by induction until we have processed every subgoal produced by the last induction. For that reason, this subgoal `Subgoal *1/3'7'` is given a special name indicating that it is destined for induction and that it is the first such subgoal arising from the induction attempt on `*1`.

```
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
91.        (CONS A1 (REV RV))).
92.
93. Name the formula above *1.1.
```

Recall that we have been following the progress of the first subgoal produced by the induction on `*1`, namely `Subgoal *1/3`, the "interesting" induction step. The pool at this point contains two other formulas from that induction: `Subgoal *1/2` and `Subgoal *1/1`. Those two formulas are drawn out before induction is applied to `*1.1`. Both simplify to `t`.

8.2.7 The Rev-Rev Log

Here is the complete log of the `rev-rev` proof, starting with the user's input and annotated with line numbers. You should read it to acquaint yourself with its structure and remind yourself of the waterfall underlying this structure.

```
 1. ACL2 >(defthm rev-rev
 2.         (implies (true-listp a) (equal (rev (rev a)) a)))
 3.


 4. Name the formula above *1.
 5.
 6. Perhaps we can prove *1 by induction.  Two induction schemes
 7. are suggested by this conjecture.  These merge into one derived
 8. induction scheme.
 9.
10. We will induct according to a scheme suggested by (REV A).
11. If we let  (:P A) denote *1 above then the induction scheme
12. we'll use is
13. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
14.               (:P A))
15.      (IMPLIES (ENDP A) (:P A))).
16. This induction is justified by the same argument used to
17. admit REV, namely, the measure (ACL2-COUNT A) is decreasing
18. according to the relation E0-ORD-< (which is known to be
19. well-founded on the domain recognized by E0-ORDINALP).  When
20. applied to the goal at hand the above induction scheme produces
21. the following three nontautological subgoals.
22.
23. Subgoal *1/3
24. (IMPLIES (AND (NOT (ENDP A))
25.               (EQUAL (REV (REV (CDR A))) (CDR A))
26.               (TRUE-LISTP A))
27.          (EQUAL (REV (REV A)) A)).
28.
29. By the simple :definition ENDP we reduce the conjecture to
30.
31. Subgoal *1/3'
32. (IMPLIES (AND (CONSP A)
33.               (EQUAL (REV (REV (CDR A))) (CDR A))
34.               (TRUE-LISTP A))
35.          (EQUAL (REV (REV A)) A)).
36.
37. This simplifies, using the :definitions TRUE-LISTP and REV,
38. to
39.
40. Subgoal *1/3''
41. (IMPLIES (AND (CONSP A)
```

```
42.               (EQUAL (REV (REV (CDR A))) (CDR A))
43.               (TRUE-LISTP (CDR A)))
44.          (EQUAL (REV (APP (REV (CDR A)) (LIST (CAR A))))
45.                 A)).
46.
47. The destructor terms (CAR A) and (CDR A) can be eliminated
48. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
49. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
50. the following goal.
51.
52. Subgoal *1/3'''
53. (IMPLIES (AND (CONSP (CONS A1 A2))
54.               (EQUAL (REV (REV A2)) A2)
55.               (TRUE-LISTP A2))
56.          (EQUAL (REV (APP (REV A2) (LIST A1)))
57.                 (CONS A1 A2))).
58.
59. This simplifies, using the :type-prescription rule REV and
60. primitive type reasoning, to
61.
62. Subgoal *1/3'4'
63. (IMPLIES (AND (EQUAL (REV (REV A2)) A2)
64.               (TRUE-LISTP A2))
65.          (EQUAL (REV (APP (REV A2) (LIST A1)))
66.                 (CONS A1 A2))).
67.
68. We now use the first hypothesis by cross-fertilizing
69. (REV (REV A2)) for A2 and throwing away the hypothesis.
70. This produces
71.
72. Subgoal *1/3'5'
73. (IMPLIES (TRUE-LISTP A2)
74.          (EQUAL (REV (APP (REV A2) (LIST A1)))
75.                 (CONS A1 (REV (REV A2))))).
76.
77. We generalize this conjecture, replacing (REV A2) by RV.
78. This produces
79.
80. Subgoal *1/3'6'
81. (IMPLIES (TRUE-LISTP A2)
82.          (EQUAL (REV (APP RV (LIST A1)))
83.                 (CONS A1 (REV RV)))).
84.
85. We suspect that the term (TRUE-LISTP A2) is irrelevant to
86. the truth of this conjecture and throw it out.  We will thus
87. try to prove
88.
89. Subgoal *1/3'7'
90. (EQUAL (REV (APP RV (LIST A1)))
```

```
 91.        (CONS A1 (REV RV))).
 92.
 93. Name the formula above *1.1.
 94.
 95. Subgoal *1/2
 96. (IMPLIES (AND (NOT (ENDP A))
 97.               (NOT (TRUE-LISTP (CDR A)))
 98.               (TRUE-LISTP A))
 99.          (EQUAL (REV (REV A)) A)).
100.
101. But we reduce the conjecture to T, by primitive type reasoning.
102.
103. Subgoal *1/1
104. (IMPLIES (AND (ENDP A) (TRUE-LISTP A))
105.          (EQUAL (REV (REV A)) A)).
106.
107. By the simple :definition ENDP we reduce the conjecture to
108.
109. Subgoal *1/1'
110. (IMPLIES (AND (NOT (CONSP A)) (TRUE-LISTP A))
111.          (EQUAL (REV (REV A)) A)).
112.
113. But simplification reduces this to T, using the :executable-
114. counterparts of REV, EQUAL and CONSP, primitive type reasoning
115. and the :definition TRUE-LISTP.
116.
117. So we now return to *1.1, which is
118.
119. (EQUAL (REV (APP RV (LIST A1)))
120.        (CONS A1 (REV RV))).
121.
122. Perhaps we can prove *1.1 by induction.  Two induction schemes
123. are suggested by this conjecture.  Subsumption reduces that
124. number to one.
125.
126. We will induct according to a scheme suggested by (REV RV).
127. If we let  (:P A1 RV) denote *1.1 above then the induction
128. scheme we'll use is
129. (AND (IMPLIES (AND (NOT (ENDP RV)) (:P A1 (CDR RV)))
130.                (:P A1 RV))
131.      (IMPLIES (ENDP RV) (:P A1 RV))).
132. This induction is justified by the same argument used to
133. admit REV, namely, the measure (ACL2-COUNT RV) is decreasing
134. according to the relation EO-ORD-< (which is known to be
135. well-founded on the domain recognized by EO-ORDINALP).  When
136. applied to the goal at hand the above induction scheme produces
137. the following two nontautological subgoals.
138.
139. Subgoal *1.1/2
```

```
140. (IMPLIES (AND (NOT (ENDP RV))
141.               (EQUAL (REV (APP (CDR RV) (LIST A1)))
142.                      (CONS A1 (REV (CDR RV)))))
143.          (EQUAL (REV (APP RV (LIST A1)))
144.                 (CONS A1 (REV RV)))).
145.
146. By the simple :definition ENDP we reduce the conjecture to
147.
148. Subgoal *1.1/2'
149. (IMPLIES (AND (CONSP RV)
150.               (EQUAL (REV (APP (CDR RV) (LIST A1)))
151.                      (CONS A1 (REV (CDR RV)))))
152.          (EQUAL (REV (APP RV (LIST A1)))
153.                 (CONS A1 (REV RV)))).
154.
155. But simplification reduces this to T, using the :definitions
156. REV and APP, primitive type reasoning and the :rewrite rules
157. CAR-CONS and CDR-CONS.
158.
159. Subgoal *1.1/1
160. (IMPLIES (ENDP RV)
161.          (EQUAL (REV (APP RV (LIST A1)))
162.                 (CONS A1 (REV RV)))).
163.
164. By the simple :definition ENDP we reduce the conjecture to
165.
166. Subgoal *1.1/1'
167. (IMPLIES (NOT (CONSP RV))
168.          (EQUAL (REV (APP RV (LIST A1)))
169.                 (CONS A1 (REV RV)))).
170.
171. But simplification reduces this to T, using the :definitions
172. REV and APP, primitive type reasoning, the :rewrite rules
173. CAR-CONS and CDR-CONS and the :executable-counterparts of
174. CONSP and REV.
175.
176. That completes the proofs of *1.1 and *1.
177.
178. Q.E.D.
179.
180. Summary
181. Form:  ( DEFTHM REV-REV ...)
182. Rules: ((:DEFINITION IMPLIES)
183.         (:ELIM CAR-CDR-ELIM)
184.         (:TYPE-PRESCRIPTION REV)
185.         (:DEFINITION TRUE-LISTP)
186.         (:EXECUTABLE-COUNTERPART EQUAL)
187.         (:DEFINITION NOT)
188.         (:DEFINITION ENDP)
```

```
189.          (:DEFINITION REV)
190.          (:EXECUTABLE-COUNTERPART CONSP)
191.          (:REWRITE CAR-CONS)
192.          (:EXECUTABLE-COUNTERPART REV)
193.          (:REWRITE CDR-CONS)
194.          (:FAKE-RUNE-FOR-TYPE-SET NIL)
195.          (:DEFINITION APP))
196. Warnings:  None
197. Time:  0.13 seconds (prove: 0.08, print: 0.03, other: 0.02)
198.
199. Proof succeeded.
200. ACL2 >
```

As we see from the proof above, the proof of `rev-rev` is not only inductive but it involves invention of an interesting lemma. The lemma was produced by simplifying the induction step of `rev-rev`, using destructor elimination, equality substitution, generalization, and elimination of irrelevance to arrive at:

```
 89. Subgoal *1/3'7'
 90. (EQUAL (REV (APP RV (LIST A1)))
 91.        (CONS A1 (REV RV))).
 92.
 93. Name the formula above *1.1.
```

Formula `*1.1` is a worthy fact. It says that if you append the singleton list containing `A1` to the right end of an arbitrary list, `RV`, and reverse the result, you get the same thing you would get if you consed `A1` onto the front of the reverse of `RV`. We do not know of a proof of `rev-rev` that does not make use of a comparable lemma about `rev` and `app`. Formula `*1.1` is proved by a second, automatically selected induction, starting on line 122.

8.3 Simplification Revisited

In this section we expand our description of the simplifier. On page 126 we described the simplifier as having four steps: decision procedures, establishing context, rewriting, and normalization and subsumption. Without doubt, rewriting is the most important aspect: successful use of the theorem prover requires successful control of the rewriter.

Without loss of generality, we assume the formula to which the simplifier is applied is of the form `(implies (and` $p_1 \ \ldots \ p_n$`)` q`)`. The p_i are the hypotheses and q is the conclusion.

Our presentation is organized as follows. First we discuss equivalence relations and congruence rules, since these are fundamental to several aspects of the simplifier. Then we will discuss each of the four steps (decision procedures, context, rewriting, and normalization and subsumption) in the order in which they occur.

8.3.1 Congruence-Based Reasoning

The most frequently used rule of inference in most proofs is the familiar idea of substitution of equals for equals. ACL2 supports a general form of substitution of equals for equals, based on the ideas of user-defined equivalence relations and congruence rules. The importance of user-defined equivalence and congruence rules is difficult to appreciate at first. They inherit their importance, in part, from the fact that ACL2 does not provide abstract data types. New kinds of objects must be represented in terms of existing ACL2 primitives, *e.g.*, sets are represented as lists. The operations on these objects are defined as functions on the existing data types, *e.g.*, the set operations are functions on lists that ignore duplication and order. Because such representations are often not unique, ACL2's equality predicate, `equal`, is too strong: it distinguishes objects that all the operations of the new type treat equivalently. Thus, one must define an equivalence relation on the new type and prove that the new operations respect this notion of equivalence. Once that is done, ACL2 can use that equivalence to substitute into nests of the new operations. Use of equivalence relations and congruence rules is fundamental to the simplifier as well as other proof techniques in the waterfall.

The need for generalized equivalence relations is easily seen by considering the representation of finite sets as linear lists. Thus, we might think of `(1 2 3)` and `(3 1 2 1)` as equivalent representations of the set $\{1, 2, 3\}$.

Suppose we define `(un a b)` to return (a representation of the) union of (the sets represented by) `a` and `b`. Thus, `(un '(1 2) '(3 4))` might be `(1 2 3 4)`. Is un commutative? Depending on how we actually define it, it may not be commutative. For example, `(un '(1 2) '(3 4))` might be `(1 2 3 4)` but `(un '(3 4) '(1 2))` might be `(3 4 1 2)`. These two objects are not `equal`. But we could define a relation, `(set-equal a b)`, that returns `t` or `nil` according to whether the set represented by `a` is the same as the set represented by `b`. Then we would have the theorem

```
(set-equal (un b a) (un a b)).
```

Suppose we use `mem` to test for "set membership," *i.e.*, `(mem e x)` is `t` or `nil` according to whether `e` is an element of `x`. Consider proving

```
(implies (mem e (un a b))
         (mem e (un b a))).
```

The "natural" proof is "use the commutativity of un to replace `(un b a)` by `(un a b)`." If asked to justify such a move, we might say "substitution of equals for equals." But the commutativity result we have for un is not an equality! What allows us to use it to replace `(un b a)` by `(un a b)` in our conjecture?

One might respond by observing that `set-equal` is an equivalence relation: it is symmetric, reflexive, and transitive. While true, that is not enough. For example, we cannot use commutativity to replace `(un b a)` by

`(un a b)` in `(equal (car (un a b)) (car (un b a)))` or else we could prove a non-theorem. It is important that `mem` respects `set-equal` in the sense that `(mem e x)` returns the same result as `(mem e y)` whenever `x` is `set-equal` to `y`. This is a *congruence rule* and might be phrased as

```
(implies (set-equal x y)
         (equal (mem e x) (mem e y))).
```

This congruence rule allows us to substitute `set-equals` for `set-equals` in the second argument of a `mem` expression, without changing the value of the `mem` expression. Given the congruence rule, we can use the commutativity rule—as stated in terms of `set-equal`—just as though it were an equality. That is, we can use it as a rewrite rule.

There is one final twist to discuss. The twist has to do with the fact that the congruence rule uses two difference senses of equality. In the rule above we see both `set-equal` and `equal`. In general, we might see any two equivalence relations here. For example, in Lisp the membership "predicate" is named `member`. But instead of returning `t` to indicate success, `(member e x)` returns the first tail of `x` that starts with `e`. This way `member` can be used to determine both whether and where `e` occurs in `x`. For this sense of membership, the congruence rule above does not hold. Let `x` be `'(1 2)` and `y` be `'(2 1)`. Let `e` be `1`. Then `(set-equal x y)` is true. But `(member e x)` returns `(1 2)` while `(member e y)` returns `(1)`. These two objects are not `equal`. But they are "propositionally equal" in the sense that they are `nil` or non-`nil` in unison. So we have another congruence rule.

```
(implies (set-equal x y)
         (iff (member e x) (member e y)))
```

This congruence rule allows the substitution of `set-equals` for `set-equals`, in the second argument of `member` expressions, while preserving `iff`. Put another way, if a `member` expression occurs in a position in which only its propositional value is important, then its second argument occurs in a position in which only its set value is important.

At the top of a conjecture, only the propositional value is important. As we explore the subterms occurring in the conjecture, we can use congruence rules to keep us appraised of which equivalence relations we must preserve to maintain top-level propositional equivalence.

A binary relation is known to be an equivalence relation if it has been proved Boolean, symmetric, reflexive, and transitive and those four facts have been marked as an `equivalence` rule. To prove and store the necessary rules, use the `defequiv` command. Some familiar relations that can be defined in ACL2 and proved to be equivalence relations are

- `(iff a b)`, expressing the idea that `a` and `b` are both `nil` or both non-`nil` (without requiring that they both be Boolean). For example, `t` is equivalent (modulo `iff`) to `'(1 2 3)`.

- `(equall a b)`, expressing the idea that two lists are equal except for the atom marking the end of the lists. Thus, `'(1 2 1 3)` and `'(1 2 1 3 . ABC)` are `equall`.
- `(perm a b)`, expressing the idea that lists `a` and `b` are permutations of one another. For example `'(a b c a c)` is a permutation of `'(a a b c c)`.
- `(alist-equal a b)`, expressing the idea that when `a` and `b` are regarded as association lists they assign the same value to every key. For example, `'((A . 1) (B . 2) (A . 3))` is `alist-equal` to `'((B . 2) (A . 1))`.
- `(set-equal a b)`, expressing the idea that lists `a` and `b` contain the same set of elements. For example, `'(a a b c c)` is `set-equal` to `'(b a c)`.

We say an occurrence of *lhs* in a formula is *equiv-hittable* if congruence rules are available to establish that the occurrence can be replaced by any equivalent (modulo *equiv*) term without changing the propositional value of the formula.

Congruence rules are derived from theorems of the form

```
(implies (equiv1 x y)
         (equiv2 (f ... x ...)
                 (f ... y ...)))
```

where $equiv_1$ and $equiv_2$ are known equivalence relations. To prove a congruence rule, use `defcong` and see also `congruence`. Such a congruence rule informs ACL2 that if `x` and `y` are equivalent (modulo $equiv_1$) then the result of replacing one by the other in the indicated argument position of f produces an equivalent term (modulo $equiv_2$). We sometimes characterize such a congruence by saying that it allows $equiv_1$ substitution (in the indicated argument position) into f while *preserving* $equiv_2$. One can thus justify a deep substitution by chaining together congruence rules, starting from a congruence rule that preserves `iff` (propositional equivalence). ACL2 can do such chaining, provided appropriate congruence rules are available for every relevant argument position of every relevant function symbol.

Generally speaking when you represent a new type of object (*e.g.*, sets as lists) you might consider introducing a corresponding equivalence relation. Then when you define the elementary operations on the "new" objects, *e.g.*, `mem` and `un`, you should consider proving the appropriate congruence rules. Here are the rules for `mem` and `un`.

```
(implies (set-equal x y)
         (iff (mem e x)
              (mem e y)))
```

```
(implies (set-equal x y)
         (set-equal (un x a)
                    (un y a)))
(implies (set-equal x y)
         (set-equal (un a x)
                    (un a y)))
```

The first says that propositional equivalence is preserved when set equivalence is preserved in the second argument of mem. The second says that set equivalence is preserved when set equivalence is preserved in the first argument of un. The third says that set equivalence is preserved when set equivalence is preserved in the second argument of un. These three rules may be conveniently expressed as shown below. The names of the variables are unimportant.

```
(defcong set-equal iff (mem e x) 2)
(defcong set-equal set-equal (un x a) 1)
(defcong set-equal set-equal (un a x) 2)
```

Defcong is defined as a macro that expands into a defthm form. This is a common use of macros. See defcong.

Now suppose that the rewriter encounters the term

(mem α
     (un (un β γ) δ))

while it is trying to preserve propositional equivalence. The three congruence rules tell the rewriter that it can replace β, γ, and δ (as well as the un-terms containing them) by set-equal terms. You should think of congruence rules as providing a road-map with which ACL2 can figure out the equivalence relations to preserve while rewriting given subterm occurrences.

Why bother? The knowledge that it is sufficient to preserve set-equal while rewriting, say, β, is only important if you have also proved rules that allow the rewriter to replace one term by a set-equal term. Here are some such rules.

```
(set-equal (un b a) (un a b))
(set-equal (un (un a b) c) (un a (un b c)))
(set-equal (un b nil) b)
```

These are just rewrite rules; they direct the system to replace the left-hand side by the right-hand side in set-equal-hittable contexts.

Congruence rules just permit these rules to be used.

Finally, it is possible that if you have several different equivalence relations, then some will refine others. For example, perm is a refinement of set-equal in the sense that if (perm a b) holds, then so does (set-equal a b). Thus, if an occurrence of a term is set-equal-hittable then it is also perm-hittable. That is, the system can use rules for perm and for set-equal when in a set-equal-hittable position. It is useful therefore to bring to the system's attention the refinement relations between your equivalence

relations. This is done by proving refinement rules; see defrefinement and refinement.

Having sketched the role of equivalence relations and congruence rules, we now return to the details of the simplification process. Recall that simplification proceeds in four steps: use of decision procedures, establishment of a context, rewriting, and normalization and subsumption.

8.3.2 Decision Procedures

When a formula is given to the simplifier three decision procedures are applied. The first is propositional calculus. The second is congruence closure, using equivalence relations. The third is rational linear arithmetic.

Propositional Calculus
The default propositional procedure is based on the normalization of `if` expressions: (a) propositional connectives are expanded in terms of `if`; (b) the `if` terms are distributed, so (*f* (`if` *a b c*)) becomes (`if` *a* (*f b*) (*f c*)) and (`if` (`if` *a b c*) *x y*) becomes (`if` *a* (`if` *b x y*) (`if` *c x y*)); and (c) the resulting tree is explored to determine whether every reachable tip is non-`nil`. The user may direct the system on a particular subgoal to use ordered binary decision diagrams [11, 32] instead. See bdd. Bdds are most effective on large propositional problems; we do not recommend using them until you are familiar with the rest of the system. ACL2 extends BDDs to cons trees, and involves term rewriting in their construction.

Congruence Closure
The congruence closure procedure uses the context to compute equivalence classes, chooses a canonical representative of every equivalence class (namely the lexicographically smallest term), and substitutes that representative for all members of the class into all function applications allowing it. For example, if (*equiv a b*) and (*equiv b c*) are known from the present context, then (*equiv a c*) is added to the context; and moreover, if *a* is lexicographically less than *b* and *c*, then *equiv*-hittable occurrences of *b* and *c* are replaced by *a*. This is done iteratively. See defequiv, defcong, and defrefinement to introduce an equivalence relation, congruence rule, or refinement.

Linear Arithmetic
The linear arithmetic procedure is a decision procedure for rational linear inequalities, *i.e.*, formulas made up of variables, constants, sums, differences, products of constants with variables, equalities, and inequalities. In this discussion we use the word "inequality" loosely to include (`equal x y`), since, when x and y are rational, that equality is equivalent to the conjunction of (`<= x y`) and (`<= y x`).

The procedure works by trying to derive a contradiction from the negation of the goal. The procedure organizes the inequalities from the conjecture into a *linear data base* and then combines them by cross-multiplication and addition so as to create new inequalities. The linear data base for a formula contains all the hypothesis inequalities and the negation of the conclusion inequality, if any. If a contradiction can be derived from this data base, the formula is a theorem of linear arithmetic. All function applications other than sums, differences, and products with constants are treated as variables.

For example, linear arithmetic can be used to prove the following.

```
(implies (and (< (* 3 a) (* 2 b))
              (<= (* 5 b) (+ (* 7 a) c)))
         (< a (* 2 c)))
```

The formula above would be proved the same way if `a`, `b`, and `c` were replaced by more complicated expressions.

Sometimes it is necessary to augment the linear data base with inequalities derived from theorems about other function symbols. For example, let `(pos e x)` be the position at which `e` occurs in the list `x` or the length of `x`, `(len x)`, if `e` does not occur. The following is a theorem but is not a consequence of linear arithmetic.

```
(implies (and (< 0 j)
              (< (* 2 (len a)) k))
         (< (pos e a) (+ k j)))
```

However if we add the hypothesis that `(<= (pos e a) (len a))`, the new formula is a consequence of linear arithmetic. You can bring such inequalities to the theorem prover's attention by proving linear rules.

A *linear rule* is a theorem that concludes with an inequality. If an instance of one of the terms mentioned as an addend in the inequality arises in the linear data base, the rule is instantiated so as to create that instance. If the hypotheses of the rule can be established, the instantiated inequality is added to the linear data base. The hypotheses are established by rewriting them, as described in the section on rewriting below. See **linear** for more details.

8.3.3 Context

If the formula is not proved by one of the foregoing procedures, the simplifier will rewrite each hypothesis and then the conclusion. Rewriting is done in a *context* that specifies what may be assumed true. When rewriting the conclusion, we assume all of the hypotheses. When rewriting a hypothesis, we assume the other hypotheses and the negation of the conclusion.

The context actually consists of two kinds of information, each of which is derived from the assumptions described above. One kind of information

is arithmetic in nature. The other is type theoretic. The former is derived from the arithmetic inequalities among the assumptions, and linear rules, as described above. The type theoretic information is discussed here.

Sometimes the construction of the context proves the theorem. For example, type information may establish that two hypotheses are contradictory. In this case, no rewriting is done and success is reported immediately.

Before we descend further into the derivation of this type theoretic information it should be noted that many successful users have only a vague idea of how this part of ACL2 works. This section will give you a fairly good model. But you should not be discouraged by its length: it is possible to use ACL2 successfully without pulling the levers described here.

A *type statement* is a claim that a term has a certain type. We might record the assumption `(orderedp a)` by making the type statement that "`(orderedp a)` has type non-`nil`." Similarly, we might record the assumption `(rationalp x)` with the statement that "`x` has type rational." What then do we mean by "type?" In the next section we explain what a "type" is and sketch how we deduce type information from assumptions.

However, the type deduction algorithm, called *primitive type reasoning* or *type set reasoning* in the theorem prover output, can be extended by two kinds of rules.

- Type-prescription rules allow you to inform the type algorithm of the type of the output produced by a function. A type-prescription rule about `orderedp` might assert that it is Boolean-valued. Then when we assume `(orderedp a)` we deduce that `(orderedp a)` has type `t` rather than the much larger type non-`nil`. See `type-prescription`.

- Compound-recognizer rules are applicable to Boolean-valued functions of one argument. These rules allow you to tell the system how to deduce type information about the argument. For example, a compound-recognizer rule might tell the type mechanism to deduce from `(orderedp a)` that `a` is a true list. See `compound-recognizer`.

We discuss these two types of rules after discussing types. Then we describe how we assemble our assumptions into a context for the rewriter.

Types

We partition the ACL2 universe into fourteen *primitive types*. A *type* is any union of these primitive types. The fourteen primitive types are: {`0`}, the positive integers, the positive ratios (*i.e.*, non-integer rationals), the negative integers, the negative ratios, the complex rationals, the characters, the strings, {`nil`}, {`t`}, the symbols other than `nil` and `t`, the proper conses (*i.e.*, conses that are true lists), the improper conses, and all others. Note that three primitive types contain only a single object: {`0`}, {`nil`}, and {`t`}

These primitive types are used to build familiar types. For example, the naturals are the union of {`0`} and the positive integers. The rationals

are the union of the first five primitive types listed. The ACL2 numbers are the union of the first six primitive types. The symbols are the union of {`nil`}, {`t`}, and the other symbols. The conses are the union of the proper and improper conses. The true lists are the union of {`nil`} and the proper conses. The type non-`nil` is the union of all the primitive types except {`nil`}.

The ACL2 user refers to the type of a term x by formulas about x composed from `0`, `<`, `integerp`, `rationalp`, `complex-rationalp`, `acl2-numberp`, `characterp`, `stringp`, `equal`, `null`, `nil`, `t`, `symbolp`, `consp`, `atom`, `listp`, and `true-listp`.[3] Such terms are called *type expressions* about x. In such expressions, x is called the *typed term*. For example, `i` is the typed term in the type expression `(and (integerp i) (< 0 i))`. A type expression about x can be turned into a unique statement of the form "x has type s." It is because of type expressions that the user does not necessarily need to understand the type system.

We can compute the intersection, union, and complement of types. The satisfiability of conjunctions, disjunctions, and negations of type expressions can be deduced by an obvious computation on the underlying types. For example, it is impossible for x to be both an integer and a symbol (because the intersection of the types is empty). But it is possible for x to be both an integer and a positive rational (indeed, such an x has type positive integer). Similarly, if x is a true list and a symbol, then x is `nil`.

When we say the context records type information about the assumptions we mean it records the strongest type expressions it can deduce from the assumptions.

Here is a simple example. Suppose we have the assumptions

1. `(integerp i)`
2. `(rationalp x)`
3. `(< 0 x)`
4. `(primep j)`
5. `(equal i x)`
6. `(equal a 'ABC)`
7. `(not (consp e))`

After processing the first, we know that `i` is an element of the integers, *i.e.*, a negative integer, zero, or a positive integer. After processing the next two we know that `x` is either a positive integer or a positive ratio. Upon processing assumption 4, in the absence of any information about `primep`, we know that `(primep j)` is non-`nil`. Upon processing assumption 5 we

[3] Of course, the user may use macros such as `>` that expand to calls of these functions.

know that (equal i x) has type t. In addition, we now know that i and x must have the same type, so we can intersect their types to create a new type for each: both i and x are positive integers. Upon processing assumption 6 we know that (equal a 'ABC) is t and we also know that a is a symbol other than nil and t. The type algorithm can make no other use of the information that a is ABC. After assumption 7 we know that e is in the union of all the primitive types except the two containing conses.

Type-Prescription Rules

Consider a function application, $(f\ a_1\ \ldots\ a_n)$, and a type expression, *expr*, about it. A type-prescription rule for f may be derived from a theorem of the form (implies (and $hyp_1\ \ldots\ hyp_n$) *expr*). When the type algorithm must deduce the type of a term, $(f\ a'_1\ \ldots\ a'_n)$, that is an instance of the typed term, it instantiates the rule accordingly and then attempts to determine, by type reasoning alone, that each instantiated hyp_i is true in the current context. If so, it deduces a type statement about $(f\ a'_1\ \ldots\ a'_n)$ from the instantiated type expression *expr* and conjoins (intersects) that type statement with the current context.

More than one type-prescription rule may be applicable and all are used and conjoined. See type-prescription for restrictions and details.

An example type-prescription rule is (true-listp (rev x)). It allows the type algorithm to deduce that the type of (rev (app a b)) is either nil or a proper cons. Another type-prescription rule is the following.

```
(implies (and (integerp i)
              (integerp j)
              (not (equal j 0)))
         (integerp (rem i j)))
```

It allows the type algorithm to deduce that the type of (rem a b) is the set of integers, provided, in the current context, the type of a is a subset of the integers and the type of b is a subset of the non-zero integers.

The theorem prover can deduce type-prescription rules at definition time. For example, it may be able to deduce that primep returns either t or nil. In that case the processing of assumption 4 above will tell us that (primep j) is t rather than merely non-nil.

Compound-Recognizer Rules

We frequently define application-specific "recognizers," *i.e.*, Boolean-valued functions of one argument that recognize certain kinds of objects. Examples are suggested by the terms (primep j) and (btreep a). Sometimes the truth or falsity of such expressions imply primitive type information about the argument. Perhaps when (primep j) is true it is known that j is a positive integer. Perhaps when (btreep a) is false it is known that a is a cons. (This would be the case, for example, if btreep were defined to return t on all atoms but put some restriction on conses.)

Suppose f is a unary Boolean function symbol, x is a variable symbol, and $expr_i$ is a type expression about x. A compound-recognizer rule may be derived from a theorem of one of the following forms.

- `(implies (`f x`)` $expr_1$`)`
- `(implies (not (`f x`))` $expr_1$`)`
- `(and (implies (`f x`)` $expr_1$`) (implies (not (`f x`))` $expr_2$`))`
- `(iff (`f x`)` $expr_1$`)`
- `(equal (`f x`)` $expr_1$`)`

When $(f\ a)$ is assumed true (or false) the rule may allow the deduction of the corresponding type information about a, depending on the parity of the assumption and available rules, in the obvious way. See `compound-recognizer` for restrictions and details.

A particularly useful compound recognizer rule in some applications is the one that represents the definition of what it is to be `booleanp`. The function is defined as follows.

```
(defun booleanp (x)
  (if (equal x t) t (equal x nil)))
```

Unless `booleanp` is disabled, the hypothesis `(booleanp` e`)` is expanded and splits the goal formula into two cases. But if `booleanp` is disabled, the fact that e is `t` or `nil` is hidden. However, by proving

```
(iff (booleanp x) (or (equal x t) (equal x nil)))
```

as a compound-recognizer rule and then disabling `booleanp`, the type information about e is made available without case splits.

Assembling the Context

To construct a context from some assumptions, we compute a linear data base from the inequalities among our assumptions, using linear rules as appropriate. We also compute type information from our assumptions, using type prescription and compound recognizer rules. We then elaborate the type information with forward chaining as described below.

A *forward chaining* rule can be constructed from virtually any theorem. A typical forward chaining theorem has the form

`(implies (and` p_1 ... p_n`)` q`)`.

We call p_1 the *trigger term*. The trigger term can actually be specified by the user and can be any term whatsoever. If an instance of the trigger term occurs in the current context and the corresponding instances of the p_i are all true in the current context, then the corresponding instance of q is represented as a type statement and added to the context. The process is iterated until no changes occur. Care is taken not to loop forever. See `forward-chaining` for restrictions and details.

8.3.4 Rewriting

Recall that if the simplifier cannot prove the formula with one of the decision procedures, then it rewrites each hypothesis and the conclusion, under a context derived for each as described above.

The simplifier sweeps across the formula calling the rewriter on each hypothesis and the conclusion in turn.[4] Though it hardly ever matters, the sweep is left to right and the rewritten hypotheses are used to construct the contexts for the unrewritten ones and the conclusion.

Here is a user-level description of how the rewriter works. The following description is not altogether accurate but is relatively simple and predicts the behavior of the rewriter in nearly all cases you will encounter.

If given a variable or a constant to rewrite, the rewriter returns it. Otherwise, it is dealing with a function application, $(f\ a_1\ \ldots\ a_n)$. In most cases it simply rewrites each argument, a_i, to get some a_i' and then "applies rewrite rules" to $(f\ a_1'\ \ldots\ a_n')$, as described below.

But a few functions are handled specially. If f is `if`, the test, a_1, is rewritten to a_1' and then a_2 and/or a_3 are rewritten, depending on type reasoning about whether a_1' is `nil`. If f is `equal` or a recognizer (such as `integerp`), type reasoning is tried after rewriting the arguments and before rewrite rules are applied to the call of f. Finally, if f is a lambda expression, (`lambda` $(v_1\ \ldots\ v_n)$ *body*), then rewriting is applied to *body* after binding each v_i to a_i'.

Now we explain how rewrite rules are applied to $(f\ a_1'\ \ldots\ a_n')$. We call this the *target term* and are actually interested in a given occurrence of that term in the formula being rewritten.

Associated with each function symbol f is a list of rewrite rules. The rules are all derived from axioms, definitions, and theorems, as described below, and are stored in reverse chronological order – the rule derived from the most recently proved theorem is the first one in the list. The rules are tried in turn and the first one that "fires" produces the result.

A rewrite rule for f may be derived from a theorem of the form

```
(implies (and hyp1 ... hypk)
         (equiv (f b1 ... bn)
                rhs))
```

where *equiv* is a known equivalence relation. Note that the definition of f is of this form, where $k = 0$ and *equiv* is `equal`. A theorem concluding with a term of the form (`not` $(p\ \ldots)$) is considered, for these purposes, to conclude with (`iff` $(p\ \ldots)$ `nil`). A theorem concluding with $(p\ \ldots)$, where p is not a known equivalence relation and not `not`, is considered to conclude with (`iff` $(p\ \ldots)$ `t`).

[4]Note the distinction between *simplification* and *rewriting* as we use the terms here. The former is a formula-level activity while the latter is a term-level activity. The former orchestrates the latter.

Such a rule causes the rewriter to replace instances of the *pattern*, (*f* b_1 ... b_n), with the corresponding instance of *rhs* under certain conditions as discussed below.

Suppose the target term occurs in an *equiv*-hittable position in the formula. Suppose in addition that it is possible to instantiate variables in the pattern so that the pattern matches the target. We will depict the instantiated rule as follows.

```
(implies (and hyp'1 ... hyp'k)
         (equiv (f a'1 ... a'n)
                rhs'))
```

To apply the instantiated rule the rewriter must establish its hypotheses. To do so, rewriting (and propositional calculus) is used recursively to establish each hypothesis in turn, in the order in which they appear in the rule. This is called *backchaining*. If all the hypotheses are established, the rewriter then recursively rewrites rhs' to get rhs''. Certain heuristic checks are done during the rewriting to prevent some loops. Finally, if certain heuristics approve of rhs'', we say the rule *fires* and the result is rhs''. This result replaces the target term.

Special Hypotheses

A few interesting special cases arise in the process of trying to establish the hypotheses.

The first special case is that hyp_i is an arithmetic inequality, *e.g.*, (< *u* *v*). In this case, the two arguments are rewritten, to u' and v', and then the linear arithmetic decision procedure is applied to (< u' v') using the linear data base in the context. During this process, new linear lemmas may be added temporarily to the data base, in support of the terms introduced in u' and v'. Rewrite rules are not applied to the hypothesis itself, (< u' v'), unless the linear procedure cannot decide it.

The second special case arises when the instantiated hypothesis hyp_i' contains *free variables*, that is, variables that do not occur in the pattern or in any previous hypothesis. A theorem illustrating this problem is

```
(implies (and (divides p q)
              (not (equal p 1))
              (not (equal p q)))
         (not (primep q))).
```

Interpreted as a rule, this means that we can rewrite `(primep q)` to `nil`, provided we can rewrite `(divides p q)` to `t`, `(equal p 1)` to `nil`, and `(equal p q)` to `nil`. When we apply this rule to the target `(primep a)`, we substitute *a* for `q` to make the pattern of the rule match the target. But note that we have not substituted anything for `p`. In this rule, `p` is a free variable. The partially instantiated hyp_1 is `(divides p a)`. Chances are this will not rewrite to true unless we substitute for `p` some term related to the formula we are trying to prove! So the system must guess a choice for

the free variables. It does this in a very weak way. It simply searches the type information in the current context, looking for a term that matches the partially instantiated hypothesis. That is, it tries to find terms that, when substituted for the free variables in hyp_i' produce a term that is explicitly assumed true in the current context. A hypothesis containing a free variable is not rewritten at all! If the system finds a way to instantiate the free variables of hyp_i', it instantiates subsequent hypotheses accordingly, and tries to establish them. It never backs up to consider other choices for the free variables.

The third special case is that the hypothesis is of one of three forms `(syntaxp` *hyp*`)`, `(force` *hyp*`)`, or `(case-split` *hyp*`)`. Hypotheses of the first form are not logical restrictions at all but pragmatic metatheoretic restrictions on when to use the rule. `Syntaxp` always returns `t`. But when the rewriter encounters such a hypothesis it evaluates the form inside the `syntaxp` to decide whether the rule should fire. See `syntaxp` for details. Hypotheses of the other two forms are logically restrictive. Both `force` and `case-split` are defined as the identity function, so, for example, `(force` *hyp*`)` is true if and only if *hyp* is true. But when the rewriter encounters a hypothesis marked with `force` or `case-split` it tries to establish it as above and if that fails it assumes *hyp* and goes on. It returns to prove *hyp* later. See `force` and `case-split` for details.

Heuristic Checks

Recall that the rewriter makes a few final checks before firing the rule. Two deserve mention here. If the rule is a function definition then firing is tantamount to "opening up" or "expanding" a call of the function. If the definition is recursive, care must be taken not to expand indefinitely. For example, something must prevent `(app a b)` from opening to introduce `(app (cdr a) b)` and that in turn opening to introduce `(app (cdr (cdr a)) b)`, and so on. The final heuristic check prevents that. The rewriter does not fire the rule if the rule is a recursive definition and the rewritten rhs, rhs'', fails certain tests. One test permitting firing is that the arguments to the rewritten recursive call already appear in the formula being proved by the simplifier. Another test permitting the firing is that the arguments be symbolically simpler. Occasionally these heuristics will disallow an expansion that is important to your proof. You must then explicitly direct the system to expand the function call. See `expand`.

The second noteworthy final check concerns rules like `(equal (f x y) (f y x))` that commute, or more generally permute, the arguments to a function. Care must be taken not to indefinitely permute the arguments using such a rule. The rewriter will not fire such a rule unless the rewritten right hand side occurs before the target term in a total ordering on ACL2 terms based on lexicographic comparisons. You may think of the system as using permutative rules only to swap arguments into alphabetical order. See `loop-stopper`.

Before we leave the rewriter a supremely important point must be made: Basically, *it just does what you tell it to do with your rewrite rules.* If you tell it to loop forever, by rewriting a to b, b to c, and c to a, then it will loop forever, or as long as the resources of time and memory allow.

8.3.5 Normalization and Subsumption

We now leave the rewriter and return to the level of simplification. The simplifier is working on some goal formula, (`implies` (`and` hyp_1 ... hyp_k) *concl*), by rewriting the parts, in turn. Let us assume it has just rewritten hyp_k.[5] Suppose the result is a term that involves some `if`-expressions. For simplicity, suppose the result is (`p` (`if` *a b c*)).

You might expect the simplifier to move on to *concl*, rewriting it in a context in which hyp_1, ..., (`p` (`if` *a b c*)) are assumed true. But that is not what it does.

Instead it first lifts the `if`-expressions out of the rewritten term and splits the problem into as many cases as there are paths through the `if`-expressions. In our simple case above there are two paths: a is true and (`p` b) is true, or a is false and (`p` c) is true. The simplifier then proceeds to rewrite *concl* under each such extension of the hypotheses. Thus, in the simple case above, *concl* is rewritten in two different contexts. In the first case, the context contains hyp_1, ..., a, and (`p` b). In the second case, it contains hyp_1, ..., (`not` a), and (`p` c). The process of simplifying a formula thus yields a set of formulas whose conjunction is equivalent to the original.

The simplifier tries to clean up the set of formulas by throwing out or combining certain of them. For example, if one formula is (`implies` p q) and another is (`implies` (`and` p r) q), then clearly we might as well just prove the former. Moreover, if one formula is (`implies` (`and` p r) q) and another is (`implies` (`and` p (`not` r)) q), then we might as well just prove (`implies` p q). This is called *subsumption/replacement.*

If the result of subsumption/replacement is a set containing a single formula that is identical to the input formula, then the simplifier does not apply and passes the formula on to destructor elimination.

If the result is the empty set of formulas, then the simplifier proved the input formula.

Otherwise, the simplifier deposits each of the formulas into the pool.

[5] The analogous processing is applied after the conclusion is rewritten, just as though the conclusion were negated and appeared as hypothesis $k + 1$ and `nil` was inserted as the new *concl*.

8.4 Comments

This completes our sketch of how the theorem prover works. Recall the remark made at the beginning of this chapter.

> As you read this chapter you may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing.

It is perhaps impressive that the theorem prover can prove `rev-rev` completely automatically. But the validity of the `rev-rev` formula is obvious to most programmers. Unfortunately, so is the "validity" of the same formula without the crucial `true-listp` hypothesis. The latter fact justifies the use of a mechanical theorem prover: you will probably find that you frequently believe in the validity of formulas that are not theorems! The former fact, that validity is often obvious to you, justifies the use of a theorem prover that tries to fill in the gaps in your arguments. In constructing complicated arguments in support of practical applications, you will toss off observations like `rev-rev` without giving them a second thought and sometimes the theorem prover, using all of its power, will be able to prove them.

9

How to Use the Theorem Prover

It is one thing to understand how a tool works. It is another to know how to use it to get a particular job done. This chapter begins to explain how to use the tool just described. Here are some key ideas to keep in mind.

- The theorem prover is automatic only in the sense that you cannot steer it once it begins a proof attempt. You can only interrupt it and abort.
- You are responsible for guiding it, usually by getting it to prove the necessary lemmas. Get used to thinking that it rarely proves anything substantial by itself.
- Never prove a lemma without also thinking of what kind of rules should be made from it. You *always* specify the kind of rules to produce from a lemma, even when you say nothing about rules. The command `(defthm` *name p*`)` means "prove formula *p*, give it the name *name*, and *make it a rewrite rule*." If you do not want a theorem to be turned into a rule use `(defthm` *name p* `:rule-classes nil)`.

Bear in mind that this chapter provides only some basic information for getting started. In particular, it does not describe a way to monitor the application of rewrite rules (see **break-rewrite**). However, the last section of this chapter does suggest an approach to finer-grained interaction with the system. Further assistance in using ACL2 may be found later in this part. We also recommend the tutorial [27], which shows the use of ACL2's predecessor (Pc-)Nqthm ([4]) on a non-trivial example in considerable detail.

9.1 The Method

There are many different styles among ACL2 users. Some users tend to exercise every feature of the system while others exercise as few as possible. We recommend and will teach a single high-level proof style in this book. As you become a more sophisticated user, you will discover and possibly adopt other features of ACL2.

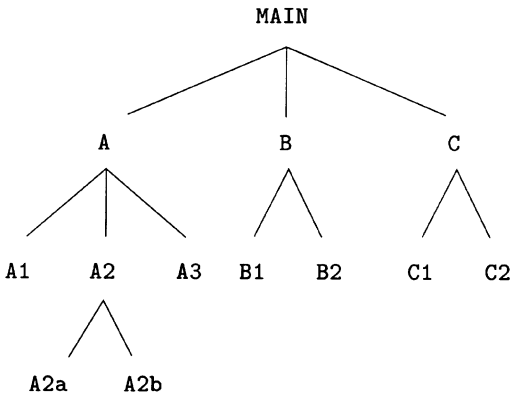

Figure 9.1: A Proof Tree

Prerequisite: To lead ACL2 to a proof you must know where the proof is. More generally, you must be able to prove theorems in this logic. Some find it helpful to practice hand proofs of simple ACL2 theorems. See the exercises in Chapters 6 and 7.1 and throughout the case studies in the companion book ([22]).

Imagine a full proof tree of some goal named `MAIN`, as depicted in Figure 9.1. `MAIN` is proved using the lemmas named `A`, `B`, and `C`, which themselves are proved with the lemmas indicated. To lead ACL2 to a proof of `MAIN`, you must ultimately prove every lemma in this tree. As a practical matter, you may not have worked out the proof of every lemma before you start to use ACL2. In fact, most users discover the structure of the proof tree by interacting with ACL2. Merely keeping track of the evolving tree, what has been proved and what remains to be proved is often daunting. We will describe one procedure, which will help you discover proofs that can be checked with ACL2. The goal of the procedure is to produce a sequence of `defthm` commands that lead ACL2 to a proof of the main theorem. The procedure will produce a postorder traversal of the tree. That is, using the procedure you will prove the lemmas of Figure 9.1 in the order: `A1`, `A2a`, `A2b`, `A2`, `A3`, `A`, `B1`, `B2`, `B`, `C1`, `C2`, `C`, `MAIN`. We note this only to make it obvious that there are many other styles one might follow. Furthermore, ACL2 contains proof structuring devices that allow you to structure the commands into the very tree shown, should you decide that is how you wish to present them. These devices are exploited, for example, in the top-down methodology presented by Kaufmann in his case study in the companion volume, [22]. But the fundamental problem is discovering the tree in the first place. ACL2 and this procedure can help you.

We use ACL2 in conjunction with a text editor. We prefer Emacs because we can run ACL2 as a process under Emacs and have the output piped into a buffer, called the *`*shell*` buffer*, that we can explore with Lisp-specific search and move commands. We generally prepare our input commands in a second buffer, called the *script buffer*.

When we are done, the script buffer will contain the postorder traversal of the proof tree for the main theorem. But during the project, the script is logically divided into two parts by an imaginary line we call the *barrier*. The part above the barrier consists of commands that have been carried out successfully by ACL2 in the `*shell*` buffer. The part below the barrier consists of commands that we intend to carry out. We sometimes refer to the first part as the *done* list and the second part as the *to-do* list.

Initially, the script buffer should contain the main theorem, p, written as a `defthm` command, *e.g.*, `(defthm main` p`)`. The barrier is at the top of the buffer, *i.e.*, the done list is empty and the to-do list contains just the main theorem.

Here is "The Method" often used to tackle a proof project.

1. Think about the proof of the first theorem in the to-do list. Structure the proof either as an induction followed by simplification or just simplification. Have the necessary lemmas been proved? That is, are the necessary lemmas in the done list already? If so, proceed to Step 2. Otherwise, add the necessary lemmas at the front of the to-do list and repeat Step 1.

2. Call the theorem prover on the first theorem in the to-do list and let the output stream into the `*shell*` buffer. Abort the proof if it runs more than a few seconds.

3. If the theorem prover succeeded, advance the barrier past the successful command and go to Step 1.

4. Otherwise, inspect the output of the failed proof attempt in the `*shell*` buffer, starting from the beginning, not the end. Basically you should look for the first place the proof attempt deviates from your imagined proof. Modify the script appropriately. We discuss this at length below. It usually means adding lemmas to the to-do list, just in front of the theorem just tried. It could mean adding hints to the current theorem. In any case, after the modifications go to Step 1. (We discuss a variant of Step 4 in Section 9.4.)

The most important part of The Method is the first word of Step 1. We use the term "The Method" partly in jest to poke gentle fun at the very idea that there is an algorithm for discovering proofs of deep and beautiful theorems. But The Method is a good starting point.

One might ask why we do not provide a user interface that supports The Method. No design we have ever contemplated was sufficiently flexible

to allow what really goes on in a major proof effort. We often evaluate small expressions typed directly into the `*shell*` to query the current world or test the behavior of defined functions. We often use Emacs to grab expressions from theorem prover output to help us create such tests or create appropriate lemmas for insertion into the script buffer. Failed proofs often reveal that some key goal is not a theorem. Modifications may then be necessary on both sides of the barrier and in the `*shell*`. Such considerations argue for an unrestricted interface used with discipline.

One might also criticize The Method as being too flat. A tree is being built but it is being encoded in a linear traversal. There is a good reason for this. Consider the proof tree of Figure 9.1. Often, lemmas `A`, `B`, and `C` involve the same collection of concepts. The lemmas necessary to prove `A`, *i.e.*, `A1`, `A2`, and `A3`, are often used in the proofs of `A`'s peers. The flat structure is often good because it allows sharing: all the lemmas necessary to prove a goal are available during the proof of that goal's subsequent peers. But there is a bad effect and you must look out for it: in large proofs the logical world gets so complicated you cannot control the simplifier.

For example, the strategy you adopt to prove one goal, *e.g.*, `A`, may conflict with the strategy you adopt for a subsequent one, *e.g.*, `B`. The most dramatic example of such a conflict is perhaps when the combined strategies simply loop indefinitely or cause catastrophic expansion. Perhaps your proof of `A` calls for `app`-nests to be right-associated but your proof of `B` calls for these nests to be left-associated. In successfully carrying out your plan for `A` you would introduce a rewrite rule, say `A1`, to right-associate `app`. Upon focusing your attention on `B`, you might prove `B1` to left-associate `app`. When the rewriter next encounters an `app` nest it will go into an infinite loop. Such loops will show up in The Method when rewriting fails to terminate. This often manifests itself by a stack overflow in the host Common Lisp. In some Lisps, such as GCL, a rewrite loop can manifest itself in a segmentation error (in which the Lisp process is aborted). A less dramatic example of conflicting strategies is when the rules for `A` simply transform the subgoals of `B` into forms that you find hard to recognize or reconcile with your intended proof.

The moral is: if you adopt conflicting strategies, it is best to be aware of it when you do it and localize the strategies to their intended goals. In small proofs (*e.g.*, those involving only dozens of rules) or in proofs whose structure you can keep clearly in mind, the simplest way to address this problem is to disable conflicting rules from prior goals before starting a new goal; this leaves those rules available should you ever need them again. See `in-theory`.

But in large proofs it is best, eventually, to use books (or encapsulation) to structure the proof, layering it appropriately, and isolating the proofs of independent peers. You might find, for example, that the lemmas at a given level in the tree, *e.g.*, lemmas `A1`, ..., `C2`, constitute a useful simplification strategy about a certain collection of concepts. You may choose to develop

a book containing all of those and then use the book to prove A, B, and C. You may also find that the proof of A requires additional "tactical" lemmas that conflict with those of B and so wish to isolate the two peers. Therefore, you might prove A in one book and B in another, importing the basic lemmas into each book and then augmenting the two worlds appropriately for their separate goals. The two books would then export only their main results, A and B, hiding the details of their proofs. C might be handled analogously. Then your proof of MAIN might end up as a very short script: three include-book commands bring in A, B, and C, and then the defthm for MAIN. See books for details. Encapsulation is an alternative to books that allows you to hide the proof of a given theorem without producing a corresponding file. See encapsulate.

We say "eventually" above because you should not be too rigid about introducing layers in a proof or else you may get lost in the layers! You must simply use good sense to structure your proofs.

A final criticism of The Method is that it is bottom-up. You find yourself proving the low-level lemmas for some sub-subgoal, like A2a, before you have seen A, B, and C mechanically assembled into MAIN. There is a reason for this. Very often—much more so than you might think—your formalization of the problem will be wrong, either in the sense that the defined concepts do not have properties you think they do or they are defined in an intractable way. By encouraging you to get your hands dirty and actually start using the definitions you have a chance to assess the whole plan in the back of your mind. It is possible to experiment in a top-down way, assuming formulas (and thus adding their rules to the world) and then using them to prove the main results. Sometimes this is useful. It often brings your attention to additional key lemmas omitted from your initial proof sketch. It more often highlights the need for many routine lemmas that will be discovered and proved in the natural course of events by The Method. See skip-proofs for details of how to assume that which must, ultimately, be proved, and remember that until it is proved you cannot be sure it is even valid! See also the case study by Kaufmann in the companion volume, [22], for a top-down methodology along these lines that has tool support, as well as Moore's case study in that volume for the use of a top-down macro.

When you are comfortable with ACL2 you will not use The Method as rigidly as it is described above, largely because it only produces flat proofs. You will probably use it by default to explore the problem, possibly adding unproved rules so the theorem prover can accompany you in your explorations. You will identify key layers or theories that need to be developed. You will recognize when it is important to do proofs in isolation. You will then use The Method, or your modification of it, to develop the appropriate scripts for each of the books you envision.

But ultimately you must learn how to build scripts that lead ACL2 to a given realistic goal from a reasonable but not quite perfect initial world. The Method is a good way to approach that problem.

9.2 Inspecting Failed Proofs

You will spend most of your time looking at ACL2 output, trying to figure out why ACL2 did not prove something that (a) you think is a theorem and (b) you think is now completely obvious given all the work ACL2 and you have already done. This section is intended to help you take advantage of that output.

In a nutshell, the most common activity is to focus on the first subgoal that ACL2 cannot simplify which does not ultimately get proved using other techniques. ACL2 provides a tool that automatically selects parts of the output on which you should probably focus; see `proof-tree`. However, in this section we focus directly on the linear output, in particular from a failed proof of the following theorem.

Theorem. `Main`

```
(equal (app (app a a) a)
       (app a (app a a)))
```

Note that the main theorem does not state that `app` is associative but states a weaker property. We try to prove this theorem in ACL2's initial world, right after defining `app`. In that world, the theorem that `app` is associative has not been proved.

The user who submits the following theorem is not following The Method because he or she could not have a proof in mind! Nevertheless, it is tempting to expect the theorem prover to do your thinking for you and this example should quickly disabuse you of that expectation!

```
 1. ACL2 >(defthm main
 2.         (equal (app (app a a) a)
 3.                (app a (app a a)))
 4.         :rule-classes nil)

 5.
 6. Name the formula above *1.
 7.
 8. Perhaps we can prove *1 by induction.  Three induction schemes
 9. are suggested by this conjecture.  Subsumption reduces that
10. number to one.
11.
12. We will induct according to a scheme suggested by
13. (APP A (APP A A)).  If we let  (:P A) denote *1 above then
14. the induction scheme we'll use is
15. (AND (IMPLIES (AND (NOT (ENDP A)) (:P (CDR A)))
16.               (:P A))
17.      (IMPLIES (ENDP A) (:P A))).
18. This induction is justified by the same argument used to
19. admit APP, namely, the measure (ACL2-COUNT A) is decreasing
20. according to the relation E0-ORD-< (which is known to be
21. well-founded on the domain recognized by E0-ORDINALP).  When
```

```
22. applied to the goal at hand the above induction scheme produces
23. the following two nontautological subgoals.
24.
25. Subgoal *1/2
26. (IMPLIES (AND (NOT (ENDP A))
27.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
28.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
29.          (EQUAL (APP (APP A A) A)
30.                 (APP A (APP A A)))).
31.
32. By the simple :definition ENDP we reduce the conjecture to
33.
34. Subgoal *1/2'
35. (IMPLIES (AND (CONSP A)
36.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
37.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
38.          (EQUAL (APP (APP A A) A)
39.                 (APP A (APP A A)))).
40.
41. This simplifies, using the :definition APP, primitive type
42. reasoning and the :rewrite rules CDR-CONS and CAR-CONS, to
43.
44. Subgoal *1/2''
45. (IMPLIES (AND (CONSP A)
46.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
47.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
48.          (EQUAL (CONS (CAR A) (APP (APP (CDR A) A) A))
49.                 (APP A (CONS (CAR A) (APP (CDR A) A))))).
50.
51. This simplifies, using the :definition APP, primitive type
52. reasoning and the :rewrite rule CONS-EQUAL, to
53.
54. Subgoal *1/2'''
55. (IMPLIES (AND (CONSP A)
56.                (EQUAL (APP (APP (CDR A) (CDR A)) (CDR A))
57.                       (APP (CDR A) (APP (CDR A) (CDR A)))))
58.          (EQUAL (APP (APP (CDR A) A) A)
59.                 (APP (CDR A)
60.                      (CONS (CAR A) (APP (CDR A) A))))).
61.
62. The destructor terms (CAR A) and (CDR A) can be eliminated
63. by using CAR-CDR-ELIM to replace A by (CONS A1 A2),
64. generalizing (CAR A) to A1 and (CDR A) to A2.  This produces
65. the following goal.
66.
67. Subgoal *1/2'4'
68. (IMPLIES (AND (CONSP (CONS A1 A2))
69.                (EQUAL (APP (APP A2 A2) A2)
70.                       (APP A2 (APP A2 A2))))
```

```
          (EQUAL (APP (APP A2 (CONS A1 A2)) (CONS A1 A2))
                 (APP A2 (CONS A1 (APP A2 (CONS A1 A2))))))).

This simplifies, using primitive type reasoning, to

Subgoal *1/2'5'
(IMPLIES (EQUAL (APP (APP A2 A2) A2)
                (APP A2 (APP A2 A2)))
          (EQUAL (APP (APP A2 (CONS A1 A2)) (CONS A1 A2))
                 (APP A2 (CONS A1 (APP A2 (CONS A1 A2))))))).

We generalize this conjecture, replacing (APP A2 (CONS A1 A2))
by L and (APP A2 A2) by AP and restricting the type of the new
variable L to be that of the term it replaces, as established
by primitive type reasoning and APP.  This produces

Subgoal *1/2'6'
(IMPLIES (AND (CONSP L)
              (EQUAL (APP AP A2) (APP A2 AP)))
          (EQUAL (APP L (CONS A1 A2))
                 (APP A2 (CONS A1 L)))).

Name the formula above *1.1.

Subgoal *1/1
(IMPLIES (ENDP A)
          (EQUAL (APP (APP A A) A)
                 (APP A (APP A A)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP A))
          (EQUAL (APP (APP A A) A)
                 (APP A (APP A A)))).

But simplification reduces this to T, using the :definition
APP and primitive type reasoning.

So we now return to *1.1, which is

(IMPLIES (AND (CONSP L)
              (EQUAL (APP AP A2) (APP A2 AP)))
          (EQUAL (APP L (CONS A1 A2))
                 (APP A2 (CONS A1 L)))).

Perhaps we can prove *1.1 by induction.  Four induction schemes
are suggested by this conjecture.  Subsumption reduces that
```

```
... ⟨ 497 lines deleted ⟩

616. Subgoal *1.1.2.5/1'''
617. (CONSP A2).
618.
619. We suspect that this conjecture is not a theorem.  We might
620. as well be trying to prove
621.
622. Subgoal *1.1.2.5/1'4'
623. NIL.
624.
625. Obviously, the proof attempt has failed.
626.
627. Summary
628. Form:  ( DEFTHM MAIN ...)
629. Rules: ((:TYPE-PRESCRIPTION APP)
630.         (:ELIM CAR-CDR-ELIM)
631.         (:REWRITE CONS-EQUAL)
632.         (:REWRITE CDR-CONS)
633.         (:REWRITE CAR-CONS)
634.         (:FAKE-RUNE-FOR-TYPE-SET NIL)
635.         (:DEFINITION NOT)
636.         (:DEFINITION ENDP)
637.         (:DEFINITION APP))
638. Warnings:  None
639. Time:  0.65 seconds (prove: 0.43, print: 0.21, other: 0.01)
640.
641. ****** FAILED ******  See :DOC failure  ****** FAILED ******
```

On lines 1–4 we issue the **defthm** command to prove our main theorem. Lines 5–641 contain the theorem prover's response. The proof attempt fails and we have deleted much of it for brevity. But many points about the theorem prover can be made from this unsuccessful attempt to prove a simple theorem.

- It is not "smart enough" to prove even this simple theorem without help from the user!
- The output is produced in real time. It describes an ongoing proof attempt, not a proof.
- At 60 lines per page this failed attempt produced more than 10 pages of output.
- Unless you are piping the output into a scrollable window, text editor, or file, most of it is lost.
- From the summary at the bottom (line 639) we see that it takes only 0.65 seconds to produce this output, a rate of about 15 pages a

second. During this time the system does four successive inductions. You cannot read it fast enough to steer it.

- Do not spend much time reading the theorem prover's output unless it fails or runs for a "long time."
- A "long time" in this setting is several seconds.
- Ridiculously long subgoal names indicate an unsuccessful strategy.
- `Subgoal *1.1.2.5/1'4'` (line 622) is `nil` and the proof attempt stops. This does not mean the original formula was not a theorem! In fact, we know the original formula is a theorem. Failure simply means the system could not prove it. Sometimes the system's search strategy will lead it to try to prove subgoals that are manifestly false. When that happens, it fails.
- The system stops automatically in this example. But it does not always stop: it can "run forever" or exhaust physical resources on your machine. That is why the output is produced in real time and you should pay cursory attention to it. We often let it scroll by at full speed and abort when the subgoal numbering gets deep.

Suppose you were confronted with this failed proof attempt. Given The Method, the appropriate response is to begin to read the output *from the top*. Do not start reading the output at `Subgoal *1.1.2.5/1'4'`.

On line 6 the system gives the formula the temporary name `*1` and at line 8 begins an inductive proof. Recall the waterfall (page 123). The system only tries induction when all else fails. Is this a theorem to be proved by induction?

The induction message (lines 8–23) is one of two main *checkpoints* in the output and it should always raise a red flag when you see it. It is crucial that you not read past an induction argument until you are convinced that induction is the appropriate mathematical technique for the goal at hand. Perhaps the goal can be proved by appeal to other lemmas.

If you decide induction is plausible—most likely because you have an informal proof in mind—you must next determine whether the particular scheme chosen is an appropriate one. Read lines 15–17.

The system is inducting on the structure of `A`. The base case is line 17, when `A` is not a cons. The induction step is on lines 15–16. Assuming `A` is a cons and the formula holds for `(CDR A)`, the system will attempt to prove the formula.

A little experience with induction will teach you that this is probably not going to work. By "experience" we do not mean experience with the mechanization of induction in ACL2. We mean experience with the mathematical technique. The mathematically experienced user would stop reading upon seeing that the system was trying to prove this theorem by

induction. That is simply the wrong attack and everything else that follows is pointless. We will see why soon.

However, let us assume that we do not see anything wrong with this inductive attack and just continue reading the output.

`Subgoal *1/2` (lines 25–30) is the inductive step, printed with our particular formula rather than in schematic form. Following that are three successive simplifications, `Subgoal *1/2'`, `Subgoal *1/2''`, and `Subgoal *1/2'''`, each obtained from the former by opening up function definitions and applying a few axioms. The last is just a simplification of the first.

- Simplification is good. Skip past it, for now.
- The first message that mentions destructor elimination, use of equalities, cross-fertilization, generalization, irrelevant terms, or induction should raise a red flag. Simplification has done all it can. The formula just above that message is the most important checkpoint in the output. We call it the *simplification checkpoint*. The important thing about the formula at the simplification checkpoint is that it is stable under simplification.

The crucial message appears at line 62, when the system reports that it will try to eliminate destructors. The formula just before that, `Subgoal *1/2'''`, is as simple as the goal is going to get with simplification alone. You know that destructor elimination and whatever else the system will try will ultimately fail: you are reading a failed proof! So you must figure out how to prove `Subgoal *1/2'''` (lines 55–60).

Study the conjecture at the checkpoint. Ask yourself

- Is the formula even valid? Perhaps your "theorem" is not a theorem.
- If it is valid, why? Sketch a little proof.
- Which theorems are used in that little proof that are not in the logical world?
- If all the theorems you need are in the world, why were they not applied? There are three common answers:
  - The pattern of a key rule does not match what is in the formula being proved. Perhaps another rule fired, messing up the pattern you expected.
  - Some hypothesis of the rule cannot be established.
  - The rule is disabled.
- If there is a missing theorem, is it suspiciously like the one you are trying to prove? If so, perhaps the wrong induction was done or the formula you are trying to prove is too weak.

- On the other hand, if there is a missing theorem and it is different from the one you are trying to prove, then you have probably identified a key lemma. Most often, such lemmas are about new combinations of functions from the original theorem and the functions introduced by rewriting.

- Can you phrase the missing theorem as a rewrite rule so that the checkpoint goal simplifies further, ideally to true?

If answers come to you, repair the script accordingly and proceed with Step 1 of The Method.

In the example being discussed, no proof of `Subgoal *1/2'''` comes to mind. The only missing lemma that might come to mind is the associativity of `app`. But suppose we do not think of it. What else should we do? Since we know we are in an induction, we ought to figure out how to use the induction hypothesis (lines 56–57) to prove the induction conclusion (lines 58-60).

But there is no way to use the hypothesis in the conclusion. They do not match up. For instance, look at the left-hand side of each.

hypothesis: `(APP (APP (CDR A) (CDR A)) (CDR A))`
conclusion: `(APP (APP (CDR A) A) A)`

The first `(CDR A)` in the one matches the corresponding `(CDR A)` in the other. But the next two `(CDR A)`'s are mismatched with `A`'s. Things are even more mismatched on the right-hand side. In this situation you should

- contemplate whether there are rewrite rules that would allow the rewriter to transform the conclusion into (something involving) the hypothesis.

Again, none come to mind.

The offending `(CDR A)`'s came into the hypothesis from our choice of inductive instance. This suggests we should choose a different induction hypothesis.

Evidently, an appropriate induction hypothesis could be obtained if we could substitute `(CDR A)` for the *first* `A` in the conjecture and *leave the other two* `A's alone`. That is, the left-hand side of the "induction hypothesis" we wish we had is `(APP (APP (CDR A) A) A)`.

But that is not legal! Induction must uniformly replace the induction variable by something smaller. To get the instance we need, we must distinguish the first `A` from the other `A`'s on the left-hand side.

Therefore, we are proving the wrong theorem! We should be proving a theorem in which some of the `A`'s here are replaced by some other variable or variables.

The experienced user of the theorem prover would not read past the induction checkpoint in this example. Even the novice should not read past the simplification checkpoint. In fact, most of the time, the problem will be evident at the simplification checkpoint.

However, returning to our example, we briefly explain what the system does after the simplification checkpoint. In this case, it is not very enlightening. At line 62 it begins to eliminate the CARs and CDRs by renaming A to be a cons of two other variables. After a little simplification, it generalizes (line 82), producing `Subgoal *1/2'6'`. Nothing more can be done for that goal, so the system gives it the temporary name `*1.1` and will ultimately try to prove it by induction. Unfortunately, inspection of `*1.1` will reveal that it is not a theorem. None of this should be surprising since we know we are proving the wrong theorem! In fact, generalization often produces goals that are not theorems.

Meanwhile, the system had another top-level subgoal to prove: the base case of the induction, `Subgoal *1/1` (lines 95–98). The proof of the base case proceeds without difficulty and finishes by line 108.

So on line 110 the system attempts to prove the doomed `*1.1`. It tries induction. We have deleted almost 500 lines of output. The system tried two more inductions before finally producing a subgoal that was manifestly false.

Many readers are swamped by the output of ACL2. They feel obliged to read it all. That is a waste of your time. Read it until you understand why the proof attempt failed; better yet, read it until you understand why the system deviated from your intended proof. There was a remote chance when `main` was submitted that the system would somehow hit upon a generalization that it could prove. The whole exercise only costs 0.65 seconds so we see no harm in letting it run; cycles are free and it is virtually impossible, given human reaction time, to stop it more quickly. But do not read more than you need!

Now, recall what we have learned: we are trying to prove the wrong theorem. We need a formula like

```
(equal (app (app a a) a)
       (app a (app a a)))
```

except with the first `a` of the left-hand side distinguished from the next two. A candidate formula is

```
(equal (app (app a b) b)
       (app a (app a a)))
```

but of course that is not a theorem (it is easy to construct a counterexample exploiting the fact that `b` occurs on only one side). We need some `b`'s on the right and the "obvious" modification is

```
(equal (app (app a b) b)
       (app a (app b b)))
```

We cannot be more specific in our guidance or justification of these ideas. They are the result of mathematical insight gained simply by doing proofs and understanding what the function app does.

The formula above is provable by induction. The proof is enlightening because the problems raised in the last attempt are so completely solved by this strange little formula. However, you probably recognize that the associativity of app is a still more general formula that would do the job.

- It is almost always best to prove the most general theorems you can state conveniently.

Therefore, if we were following The Method we would type

```
(defthm associativity-of-app
  (equal (app (app a b) c)
         (app a (app b c))))
```

into the script buffer just in front of `main`. We would be cognizant of the fact that the rule generated from this theorem will right-associate all app nests as long as it is enabled.

Virtually every time we type a theorem that will become a rewrite rule we give thought to the termination of the rewriting scheme we are evolving.

- This is most often done by selecting a normal form for all the terms that arise in the problem and "orienting" every rule to drive the pattern (left-hand side of the conclusion) closer to the normal form.

- Furthermore, every subexpression in the pattern should be written in the selected normal form because the pattern is matched *after* the arguments of the target have been rewritten. If you are rewriting `(g x)` to `(h x)` then a rewrite rule with a pattern (left-hand side) of `(f (g x))` will never be used: the pattern will not be seen because the `g` in a potential target term will be rewritten to `h`!

- Sometimes no single normal form is adequate. In that case we choose an arbitrary normal form and *stick to it.* We do not just orient the rules randomly.

- When we need a lemma that relates terms not in normal form or that, when used as a rewrite rule, would drive a term out of normal form, we either specify `:rule-classes nil` or immediately disable it, so as not to introduce a loop in the rewriter. Such lemmas must be specified explicitly in subsequent `:use` hints (see `hints`) or enabled with attention paid to the conflicting rules.

This kind of thinking in connection with your rewrite rules is absolutely crucial to using The Method successfully. Time and time again the answer to vexing questions about why ACL2 failed to find a proof can be traced back to the failure to follow the four items of advice above.

Recall that ACL2 supports congruence-based reasoning, as described in Section 8.3.1.

- Most rewrite rules conclude with an `equal`, but do not forget that you can use `iff` and your own equivalence relations.

- If you intend to rewrite with equivalence relations, be sure to prove the necessary congruence rules.

Having added `associativity-of-app` to the script, it becomes the first theorem in the to-do list. We go to Step 1 of The Method: Think. The proof of the associativity of `app` is just a standard induction down the cdr of `a` followed by simplification using the definition and axioms. We admit that in simple cases like this it is reasonable to skip Step 1 and just throw the proposed theorem at the theorem prover. But if such a proof attempt does not succeed quickly then we revert to Step 1.

So we are ready for Step 2: command ACL2 to try. It produces a proof (not shown here) in 0.05 seconds.

- Skim successful proofs cursorily looking for the highlights: induction and simplification, or just simplification, as you expected.

Why read successful proofs? It can happen that the system finds a different proof than the one you are expecting. This could be a sign that something is seriously wrong, *e.g.*, a hypothesis is false because it was entered incorrectly. It could also be a sign that your model of the logical world is not accurate. You cannot afford *not* to understand the proof strategy you are constructing!

- If the system's proof involves an unexpected induction, it is a sign that a crucial lemma may be missing or inapplicable. While the system could fill in the gap this time, you might really need that lemma for some other theorem.

- If you expected an induction and there was none, then there are already sufficient lemmas in the world to prove this theorem. Their use is reported in the proof. Study them.

- If the system's proof was entirely by rewriting, you might not need this lemma as a rewrite rule. Earlier rules may always do the rewriting for you.

- If rewriting participated in the proof before the first induction or other techniques were used, you might want to contemplate whether there is a conflict between the rewrite rules used and the rewrite rule (if any) introduced by this lemma. Evidently, existing rules transform this one. Will its pattern ever match anything if those existing rules fire? Perhaps now that you have proved this lemma you can disable the earlier rules.

Having proved associativity-of-app we follow The Method by advancing the barrier and repeating Step 1: we submit the next command, which is main again. We expect it to succeed by simplification. This time we get:

```
 1. ACL2 >(defthm main
 2.         (equal (app (app a a) a)
 3.                (app a (app a a)))
 4.         :rule-classes nil)

 5.
 6. By the simple :rewrite rule ASSOCIATIVITY-OF-APP we reduce
 7. the conjecture to
 8.
 9. Goal'
10. (EQUAL (APP A (APP A A))
11.        (APP A (APP A A))).
12.
13. But we reduce the conjecture to T, by primitive type reasoning.
14.
15. Q.E.D.
16.
17. Summary
18. Form:  ( DEFTHM MAIN ...)
19. Rules: ((:REWRITE ASSOCIATIVITY-OF-APP)
20.         (:FAKE-RUNE-FOR-TYPE-SET NIL))
21. Warnings:  None
22. Time:  0.02 seconds (prove: 0.00, print: 0.00, other: 0.02)
23. MAIN
```

9.3 Another Example

Here is another illustration of The Method, with more focus on inspecting output to find missing theorems. Suppose we have added the definitions of app and rev and have proved

```
(defthm rev-rev
  (implies (true-listp a)
           (equal (rev (rev a)) a))).
```

Note that rev-rev is a conditional rewrite rule. Our goal is to prove

```
(defthm main
  (equal (rev x)
         (if (endp x)
             nil
           (if (endp (cdr x))
               (list (car x))
```

```
              (cons (car (rev (cdr x)))
                    (rev
                     (cons (car x)
                           (rev (cdr (rev (cdr x))))))))))
  :rule-classes nil).
```

This is an ugly-looking theorem! But it expresses a surprising fact about `rev`. Indeed, as first observed to one of us by Rod Burstall, you could use this equality as a definition of reverse: it is a way to define how to reverse a list without using the auxiliary function **append**. Here we will not try to admit this as a definition or add it as an alternative definition (see definition) but just focus on proving it as a theorem about the `rev` we have already defined. We make the theorem have `:rule-classes nil` because we do not want `(rev x)` to be rewritten this way.[1]

Step 1 is to sketch a proof. If either of the two conditions tested in the `if`-nest is true, the formula is easy to prove. Otherwise, we have to look at the messy `(cons (car (rev (cdr x))) (rev (cons ...)))` expression. We know that the theorem prover will simplify the `(rev (cons ...))` by expanding the definition of `rev`. Rather than work out the expansion by hand, we just call the theorem prover on the goal above, expecting (correctly) that the system will fail to prove the theorem. This is an important and common use of the system and its verbose output: submit formulas merely to see how your current rules simplify them. Just do not be misled into letting the system dictate your strategy.

When this formula is submitted, the system runs for three seconds. We ignore all but the first few lines of output. The first simplification checkpoint is:

```
Subgoal 3''
(IMPLIES (AND (CONSP X)
              (CONSP (REV (CDR X)))
              (CONSP (CDR X)))
         (EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
                (APP (REV (REV (CDR (REV (CDR X)))))
                     (LIST (CAR X))))).
```

Two things should catch your eye. The first is `(CONSP (REV ...))`. The system does not know that `rev` returns a cons precisely when its argument is a cons. In the goal above, it helps little to know this, since it just removes a true hypothesis. But we can anticipate that the negation of this hypothesis will be considered later (indeed, it characterizes `Subgoal 2`). So we might as well "teach" the theorem prover how to simplify this question. We add to our to-do list the following theorem.

[1] Readers interested in logic might consider whether it is even possible to admit such a function under the ACL2 principle of definition. See the case study by Moore in [22].

```
(defthm consp-rev
  (equal (consp (rev x))
         (consp x)))
```

This is a rewrite rule that replaces any instance of `(consp (rev x))` by the corresponding instance of `(consp x)`. This lemma will be proved by induction followed by simplification. If we consider its proof, then we anticipate needing a lemma about `(consp (app ...))`, but expect the system will generate it. We return to this point below.

Meanwhile, the second thing that should catch your attention in the subgoal above is the `(REV (REV (CDR (REV ...))))` term. How did this term escape being hit by `rev-rev`? The only possible answer is that the theorem prover could not establish the hypothesis, in this case, `(true-listp (CDR (REV ...)))`. We could prove that directly, but a better strategy is to break it up into two observations, which we expect the system to chain together to get the needed conclusion.

```
(defthm true-listp-rev
  (true-listp (rev x)))
(defthm true-listp-cdr
  (implies (true-listp x)
           (true-listp (cdr x))))
```

The first will be proved by induction followed by simplification (with a lemma about `true-listp` and `app`) and the second by simplification.

It is not a good idea to add too many rewrite rules at once, without studying their interaction, so we now proceed with Step 1 again, for the three new lemmas. We have already considered their proofs—or perhaps we have not, but our intuition is that the proofs are likely to be easy for the theorem prover to find—and so take Step 2 and submit the commands. All three succeed, as expected.

So now we submit `main` again, simply to see the effect of our new rules on our checkpoint. In fact, the system proves `main` by simplification. Looking more closely at `Subgoal 3''`, above, we might note that once `rev-rev` is applied, the concluding equality is an identity. The other two subgoals deal with the special cases when `x` is empty or a singleton.

More can be said of the three lemmas we added. First, we anticipated needing some lemmas above but expected ACL2 to bridge the gap. That can be a frustrating strategy. It is better to be disciplined about adding the rules you think the system will need. To prove `consp-rev`, a disciplined user might first prove

```
(defthm consp-app
  (iff (consp (app a b))
       (or (consp a) (consp b))))
```

The question `(consp (app ...))` will come up in a proof about `(consp (rev ...))` because `rev` expands to `app`. The theorem prover happens

to generates a sufficient lemma about app to prove consp-rev. But that lemma is not added to the data base—only *you* add things to the data base. The next time (consp (app ...)) comes up, you might not be so lucky! It is better to arrange for it to simplify away.

The form of both consp-app and consp-rev is surprising to many new users. Weaker theorems that come to mind (for the app case) are as follows.

```
(implies (consp a) (consp (app a b)))
(implies (consp b) (consp (app a b)))
(implies (not (consp (app a b))) (not (consp a)))
(implies (not (consp (app a b))) (not (consp b)))
```

No single one of these four rules captures the logical content of consp-app. Even then, consp-app is pragmatically stronger because it eliminates the term (consp (app a b)) and introduces a case split. The four rules above allow ACL2 to settle certain consp questions, but only if they are raised by other rules. In addition, having all four rules would be very inefficient because they all cause backchaining.

- Use unconditional rewrite rules when possible.

The true-listp-rev theorem is another example where the disciplined user would have first proved a lemma about true-listp and app before expecting to prove something about true-listp and rev. The lemma we would choose is

```
(defthm true-listp-app
  (equal (true-listp (app a b))
         (true-listp b)))
```

to simply eliminate the question without backchaining.

Finally, consider true-listp-cdr. It too causes back chaining. Logically speaking, we could strengthen it to an unconditional rewrite rule.

```
(defthm true-listp-cdr
  (equal (true-listp (cdr x))
         (or (atom x) (true-listp x))))
```

But pragmatically, we do not want this rule because it rewrites in the opposite direction of the definition of true-listp. That is, the definition replaces (true-listp x) by something involving (true-listp (cdr x)); the strengthened rule above "undoes" that expansion. Introducing the strengthened rule above would cause the rewriter to loop indefinitely. We could have, alternatively, used the strengthened rule and disabled the definition of true-listp. But we tend to leave recursive functions enabled so that they unwind properly in inductive proofs.

9.4 Finer-Grained Interaction: The "Proof-Checker"

We have seen that using The Method may involve the inspection of ACL2 output from a failed proof attempt. Such inspection often leads the user to discover useful rewrite rules to prove, problems with the theorem prover's choice of induction scheme, or even modifications to make in the statement of the alleged theorem.

In this section we introduce an interactive utility, the *proof-checker*. This tool can help you discover why a proof attempt failed, by providing a number of ways to guide a new proof attempt interactively. We illustrate the proof-checker by using the example of the preceding section. See `proof-checker` for details. An important aspect of the proof-checker is that it can free you from much of the need to inspect the theorem prover's output stream. In fact we elide some prover output in the example presented below.

First we make a critical point. Although the proof-checker can free you from the need to anticipate necessary lemmas, the downside is that you can be lulled into the false notion that you no longer need to pay attention to Step 1 of The Method: *Think*. Experience can provide a sense for when the conjecture is somehow sufficiently simple that one can rely on guidance provided by a proof-checker session. We say more on this issue in remarks at the end of this section.

We begin the interactive proof session by using `verify`, which gives us the prompt "`->:`". All user input to the proof-checker is shown below on lines starting with this prompt.

```
ACL2 !>(verify
        (equal (rev x)
               (if (endp x)
                   nil
                 (if (endp (cdr x))
                     (list (car x))
                   (cons (car (rev (cdr x)))
                         (rev
                          (cons (car x)
                                (rev (cdr (rev (cdr x))))))))))))
->:
```

Although our goal is to let the proof-checker be an active assistant, we think just a little about how to proceed. Do we want to start the proof with induction, or should we simplify first? The prover starts with simplification, of course, but often it seems clear that the argument is fundamentally an inductive one and one issues the proof-checker `induct` command first. See `acl2-pc::induct`.[2] We are welcome to follow Step 1 of The Method and

[2] In some documentation, in particular the Emacs Info version, the proof-checker commands are prefixed by "`acl2-pc||`" rather than "`acl2-pc::`". All proof-checker commands are documented in the section `proof-checker-commands`.

work out the proof outline before we start, which tells us whether to start with induction or simplification. In this case, however, we use our intuition and start with simplification, expecting it to bring to light some missing rewrite rules that may be useful.

Hence, we begin by issuing the command **bash**, which calls the simplifier. All goals that would normally be put into the pool (see Section 8.2) are instead presented to the user as new goals to be proved.

```
->: bash
***** Now entering the theorem prover *****

[[[theorem prover output omitted here]]]

Creating two new goals:  (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.
However, the following subgoals remain to be proved:
  (MAIN . 1) and (MAIN . 2).
Now proving (MAIN . 1).
->:
```

What has happened? The proof-checker maintains a stack of named goals. This stack initially contains a single goal called **MAIN** which is the formula to be proved, *i.e.*, the argument of **verify**. The message above indicates that **MAIN** has been removed from the stack but new goals **(MAIN . 1)** and **(MAIN . 2)** have been pushed onto the stack. The top goal on the stack is the one being proved, in this case, the goal named **(MAIN . 1)**.

We can inspect all the goals (see acl2-pc::print-all-goals), but instead we start by looking at the current goal using the command **th** (which is mnemonic for "theorem"). Notice that a goal consists of hypotheses and a conclusion.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (CONSP (REV (CDR X)))
3. (CONSP (CDR X))

The current subterm is:
(EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
       (APP (REV (REV (CDR (REV (CDR X)))))
            (LIST (CAR X))))
->:
```

We now inspect this goal as we might have inspected a checkpoint from a failed proof in the preceding section. We notice a term in the conclusion of the form **(REV (REV ...))**. Of course, we are surprised to see such a term, since we vaguely recall having already proved a rule for it.

Before finding that rule, we first "move" to the subterm in question. The astute reader may have noticed the labeling of the conclusion above: "The current subterm." In fact the proof-checker maintains a pointer to a subterm of each goal's conclusion, which points initially to the entire conclusion (as above). But the proof-checker command `dv` ("dive") allows us to move that pointer. The command `(dv 2 1)` moves the pointer from the top of the conclusion to the second argument of the `EQUAL` and then to the first argument of that (second) `APP`.

```
->: (dv 2 1) ;; or, type 2 and then 1
->: p ;; print the current subterm
(REV (REV (CDR (REV (CDR X)))))
->: p-top ;; show entire conclusion, highlighting current subterm
(EQUAL (APP (CDR (REV (CDR X))) (LIST (CAR X)))
       (APP (*** (REV (REV (CDR (REV (CDR X)))))
                 ***)
            (LIST (CAR X))))
->:
```

Now that we are pointing to the `(REV (REV ...))` term, we can explore the issue of rewriting this term.

```
->: sr ;; or, show-rewrites
1. REV-REV
  New term: (CDR (REV (CDR X)))
  Hypotheses: ((TRUE-LISTP (CDR (REV (CDR X)))))
->:
```

Evidently the rule `REV-REV` does apply, as we might expect. Let us direct the proof-checker to apply this rewrite rule.

```
->: p
(REV (REV (CDR (REV (CDR X)))))
->: r ; or, rewrite; or, (rewrite 1); or, (rewrite rev-rev)
Rewriting with REV-REV.

Creating one new goal:  ((MAIN . 1) . 1).
->: p
(CDR (REV (CDR X)))
->:
```

Aha! The rewrite succeeded in simplifying the `(REV (REV ...))` term, but it created a new goal.

```
->: goals ;; display the names on the goal stack

(MAIN . 1)
((MAIN . 1) . 1)
(MAIN . 2)
->:
```

Let us see if the current goal can be simplified, preferably to `t`. Note that we move the pointer to the top of the conclusion before calling `bash` once again.

```
->: top ;; move to the top of the conclusion
->: bash
***** Now entering the theorem prover *****

[Note:  A hint was supplied for our processing of the goal
above.  Thanks!]

But we reduce the conjecture to T, by primitive type reasoning.

Q.E.D.


The proof of the current goal, (MAIN . 1), has been completed.
However, the following subgoals remain to be proved:
  ((MAIN . 1) . 1).
Now proving ((MAIN . 1) . 1).
->:
```

So far, so good; but we have not yet discovered any useful rewrite rules to prove. The new goal, obtained from the hypothesis of the rule `rev-rev` applied above, is about to suggest such a rule.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (CONSP (REV (CDR X)))
3. (CONSP (CDR X))

The current subterm is:
(TRUE-LISTP (CDR (REV (CDR X))))
->:
```

If we attempt to use the `bash` command to simplify (or prove) this goal, the prover is stumped.

We have been led somewhat more directly than in the preceding section to the need for rules `true-listp-rev` and `true-listp-cdr`. We temporarily exit the proof-checker in order to prove these rules (which are shown on page 172).

```
->: exit ;; return to ACL2 top-level
Exiting....
 NIL
ACL2 !>
```

After proving the two rewrite rules mentioned above, we re-enter the proof-checker session and attempt to prove the goal above. Notice that with no arguments, `verify` re-enters the previous proof-checker session.

```
ACL2 !>(verify)
->: bash
***** Now entering the theorem prover *****

[Note:  A hint was supplied for our processing of the
goal above.  Thanks!]

But simplification reduces this to T, using the :rewrite
rules TRUE-LISTP-CDR and TRUE-LISTP-REV.

Q.E.D.


The proof of the current goal, ((MAIN . 1) . 1), has
been completed, as have all of its subgoals.
Now proving (MAIN . 2).
->: goals

(MAIN . 2)
->:
```

There is one goal remaining.

```
->: th
*** Top-level hypotheses:
1. (CONSP X)
2. (NOT (CONSP (REV (CDR X))))

The current subterm is:
(NOT (CONSP (CDR X)))
->:
```

The second hypothesis contains a term that could be simplified with an appropriate rewrite rule. We have thus been led by the proof-checker to discovery of the rule `consp-rev`, which was discovered with somewhat less guidance in the preceding section (see page 172). Again we exit the proof-checker in order to prove this rule and subsequently re-enter the proof-checker using `(verify)`. Then we may issue the `bash` command as before. This time we rather arbitrarily use `prove`, which invokes the full power of the prover and either proves the goal completely or causes no change to the proof-checker state.

```
->: prove
***** Now entering the theorem prover *****
```

```
But we reduce the conjecture to T, by the simple :rewrite
rule CONSP-REV.

Q.E.D.

*!*!*!*!*!*!* All goals have been proved! *!*!*!*!*!*!*
You may wish to exit.
->:
```

The proof has succeeded! We have the option of creating a **defthm** event at this point, which will have associated :**instructions** recording the commands we gave in the interactive session. However, we prefer to avoid these low-level :**instructions** in order to increase the likelihood that a proof will replay if modifications are made to our proof script. We exit the proof-checker loop and the proof now succeeds automatically.

```
->: exit
Exiting....
 NIL
ACL2 !>(defthm main
         (equal (rev x)
                (if (endp x)
                    nil
                  (if (endp (cdr x))
                      (list (car x))
                    (cons (car (rev (cdr x)))
                          (rev
                           (cons (car x)
                                 (rev (cdr (rev (cdr x))))))))))
         :rule-classes nil)

But simplification reduces this to T, using the :definitions
APP, ENDP and REV, primitive type reasoning and the :rewrite
rules CAR-CONS, CDR-CONS, CONSP-REV, REV-REV, TRUE-LISTP-
CDR and TRUE-LISTP-REV.

Q.E.D.

Summary
Form:  ( DEFTHM MAIN ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION ENDP)
        (:DEFINITION REV)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS)
        (:REWRITE CONSP-REV)
        (:REWRITE REV-REV)
        (:REWRITE TRUE-LISTP-CDR)
```

```
        (:REWRITE TRUE-LISTP-REV))
Warnings:  None
Time:  0.06 seconds (prove: 0.02, print: 0.03, other: 0.01)
 MAIN
ACL2 !>
```

The proof-checker provides many other commands; see `proof-checker-commands`. The advanced user has the option of extending the proof-checker's power by defining compound commands, called *macro commands*, that can use non-trivial control structures. More importantly, among the basic capabilities not illustrated above are:

- undo (`undo` and `restore`);
- choose the goal to consider next (`cg`, `change-goal`);
- substitute equals for equals (=), or more generally, using equivalence relations (`equiv`);
- use induction (`induct`) to replace the current goal by goals for base and induction steps;
- consider cases (`casesplit`, `claim`, `split`);
- use binary decision diagrams for (primarily) propositional reasoning (`bdd`);
- manipulate the hypotheses (`contradict`, `demote`, `promote`, `drop`);
- invoke steps of the waterfall (`elim`, `generalize`; see also `use`);
- set the current theory (`in-theory`);
- expand function calls (`expand`, `x`, `x-dumb`);
- simplify the current subterm (`s`, `s-prop`, `sl`);
- use forward chaining (`forwardchain`); and
- save sessions (`ex`, `save`; see also `retrieve`).

As was done here, the proof checker is often used to find rules that will ultimately be used in an "automatic" proof. This mixing of the two proof engines can be extremely helpful: use the proof checker to explore proofs but record your strategies as general rules for the theorem prover. We often use the proof checker on unproved subgoals extracted from failed proofs. No sign of this appears in the final list of theorems.

Notice that the worked example in this section did not lead us to create the lemma `consp-app` of the preceding section (see page 172). The preceding section emphasized *thinking* about proof strategies, which for example could lead one to discover the lemma `consp-app`. The assistance given by

the proof-checker frees the user from some of that thinking, but as a result we did not find the lemma `consp-app`, a lemma that might turn out to be useful in later proofs even though it was not needed for this one. There is clearly a trade-off here between thinking and letting the system do some of the work. Individual style and experience will govern the extent to which one uses the proof-checker (or theorem prover, for that matter) to avoid some thinking, or uses thinking to avoid some potential interaction entirely. Experience suggests that new users tend to err on the side of doing insufficient thinking, which is why we have stressed Step 1 of The Method throughout most of this chapter. However, proof-checker interaction can provide a balance in the trade-off as one gains experience with ACL2.

10

Theorem Prover Examples

This chapter contains several examples and their solutions. Each section will start with an English description of a problem or will contain phrases such as "Why?" and "Prove the following." When you reach such a phrase, we recommend that you stop and work out a solution before reading further. Usually this will require that you define functions, translate informal correctness criteria into ACL2, and perhaps prove (on paper) the main theorem. Once you have a pencil and paper proof, think about how to decompose the proof into ACL2 rules and use the theorem prover to check your proof; you can then compare your results with ours.

We will take the "brain-dead" approach to using ACL2, *i.e.*, we will not think about how to structure our proofs, but rather, we will blindly try to prove theorems and will then react to ACL2's responses. This approach works fine for small examples and gives us the opportunity to stumble onto (and discuss) important issues.

Each section contains a set of related problems whose solutions are developed independently of the other sections in a new ACL2 session, *i.e.*, in a `ground-zero` theory (see `theories`). You are encouraged to evaluate the functions we define and to experiment with other approaches to solving the exercises. The examples will consist of proving two versions of the factorial function equivalent, proving a theorem about *, proving insertion sort correct, proving some theorems about functions that manipulate trees, proving an adder and multiplier correct, and proving a compiler for a stack-based machine correct. As a point of reference, an ACL2 expert can solve all of the problems in this chapter in about half a day.

To make the presentation more concise, we will present only the relevant parts of the output produced by ACL2 (prefaced by a line number). Therefore, it may help to run ACL2 while going through our solutions.

10.1 Factorial

Define the factorial function; define a tail recursive version; prove the two are "equivalent."

Since functions in ACL2 are total, we have to define the factorial function on any possible input. The standard way of dealing with this is to

M. Kaufmann et al., *Computer-Aided Reasoning*

coerce any value outside the intended domain to a base element, which in this case is 0. Notice the use of zp to test for 0 (and non-naturals); see zp and zero-test-idioms.

```
(defun fact (x)
  (if (zp x)
      1
    (* x (fact (- x 1)))))
```

For the tail recursive version, we introduce a variable that contains a partial product.

```
(defun tfact (x p)
  (if (zp x)
      p
    (tfact (- x 1) (* x p))))
```

We can test the functions to see if they seem to be the same (tfact is given an initial partial product of 1).

```
ACL2 >(fact 10)
3628800
ACL2 >(tfact 10 1)
3628800
```

If we try to evaluate fact on large numbers, we will get an error (which can look as follows in GCL).

```
Error: Invocation history stack overflow.
```

This occurs because the underlying Common Lisp places limits on the sizes of its stacks. One way to alleviate this problem is to compile the functions (see comp), which we can do as follows.

```
:comp (fact tfact)
```

Because tfact is tail-recursive, the recursion is replaced by iteration. This greatly eases the restriction on computation imposed by the size of the invocation stack and makes it possible to execute the function on large numbers.

```
;; Note: Tail-recursive call of TFACT was replaced by iteration.
```

To prove that fact and tfact are "equivalent," we will prove the following theorem.

```
(defthm fact=tfact
  (equal (tfact x 1) (fact x)))
```

Submitting fact=tfact to ACL2 produces the following. We only show the simplification checkpoint (see 165).

```
42. Subgoal *1/2''
43. (IMPLIES (AND (INTEGERP X)
44.               (< 0 X)
45.               (EQUAL (TFACT (+ -1 X) 1)
46.                      (FACT (+ -1 X))))
47.          (EQUAL (TFACT (+ -1 X) X)
48.                 (* X (FACT (+ -1 X))))).
```

The same goal may be obtained without perusing ACL2 output by instead entering the proof-checker with

```
(verify
  (equal (tfact x 1) (fact x)))
```

and issuing the command `(then induct bash)`. See page 174. We use ACL2 without the proof-checker in the present chapter.

That the proof attempt failed will come as no surprise to readers experienced in constructing proofs by induction because they will notice that the above theorem needs to be strengthened: opening `tfact` will change the second argument from a constant to `x`, and no induction hypothesis will match this new term (one cannot substitute for a constant). This is what happens on `Subgoal *1/2''`, above.

We strengthen the theorem so that instead of a `1` in `fact=tfact`, we have a variable.

```
(defthm fact=tfact-lemma
  (equal (tfact x p)
         (* p (fact x))))
```

ACL2 fails to prove `fact=tfact-lemma`. If we look at the simplification checkpoint, we see:

```
67. Subgoal *1/2'4'
68. (IMPLIES (AND (INTEGERP X) (< 0 X))
69.          (EQUAL (* X P (FACT (+ -1 X)))
70.                 (* P X (FACT (+ -1 X))))).
```

What lemma is suggested by the failed proof attempt? Notice that the conclusion of `Subgoal *1/2'4'` has the form `(equal (* A B C) (* B A C))`; this is a "simple" theorem about multiplication, *i.e.*, it is a theorem we expect elementary school students to know. Since such theorems are very useful to have around, ACL2 experts have written several arithmetic books. Some of these books, *e.g.*, `top-with-meta`, are part of the ACL2 source code distribution, but are not loaded into the initial data base. If we load `top-with-meta`, then ACL2 "knows" the above theorem. We use include-book to load `top-with-meta` (which probably resides in a different directory in your filesystem).

```
(include-book "/local/acl2/books/arithmetic/top-with-meta")
```

We will discuss how to prove the above theorem without the use of books in the next section.

If we then submit `fact=tfact-lemma` to ACL2, we get:

```
60. Subgoal *1/1.2'
61. (IMPLIES (NOT (INTEGERP X))
62.          (ACL2-NUMBERP P)).
63.
64. We suspect that this conjecture is not a theorem.  We might
65. as well be trying to prove
66.
67. Subgoal *1/1.2''
68. NIL.
69.
70. Obviously, the proof attempt has failed.
```

This output tells us exactly why `fact=tfact-lemma` is not a theorem. `Subgoal *1/1.2'` suggests that `fact=tfact-lemma` is false if `x` is not an integer and `p` is not a number. In fact, if `x` is not an integer, then `tfact` will return `p`, which can be anything, but `(* p (fact x))` is always an ACL2 number (*i.e.*, `(acl2-numberp (* x y))` is a theorem). We have made the classic mistake of strengthening the theorem to the point where the resulting term is not true. Our advice on using induction is: simplify and strengthen as much as possible, but no more. We try the following.

```
(defthm fact=tfact-lemma-for-acl2-numberp
  (implies (acl2-numberp p)
           (equal (tfact x p)
                  (* p (fact x)))))
```

ACL2 proves this lemma, which is stronger than `fact=tfact`, hence, ACL2 can easily prove the main result.

```
1. (defthm fact=tfact
2.   (equal (tfact x 1) (fact x)))
3.
4. But simplification reduces this to T, using the :type-prescription
5. rule FACT, the :definition FIX, the :rewrite rules FACT=TFACT-
6. LEMMA-FOR-ACL2-NUMBERP and UNICITY-OF-1 and primitive type
7. reasoning.
8.
9. Q.E.D.
```

There are other ways of defining a tail recursive version of `fact`, *e.g.*,

```
(defun tfact2 (x p)
  (if (zp x)
      (fix p)
    (tfact2 (- x 1) (* x p)))).
```

With this definition, we can prove the following theorem, which does not have a hypothesis:

```
(thm (equal (tfact2 x p)
            (* p (fact x)))).
```

10.2 Associative and Commutative Functions

In the previous section we were confronted with the following failed proof attempt:

```
67. Subgoal *1/2'4'
68. (IMPLIES (AND (INTEGERP X) (< 0 X))
69.          (EQUAL (* X P (FACT (+ -1 X)))
70.                 (* P X (FACT (+ -1 X))))).
```

We used the `top-with-meta` book to bypass the problem; in this section we will prove the required theorem without the use of books. It turns out that what we do is applicable to any associative and commutative function.

You might think that `*` is a function symbol, but if you type `:pe *` (see pe and history), you see that `*` is really a macro. Since macros are just syntactic sugar, *i.e.*, they are expanded into expressions, you may wonder why ACL2 insists on printing `(BINARY-* X (BINARY-* P (FACT (+ -1 X))))` as `(* X P (FACT (+ -1 X)))`. The answer is that internally, ACL2 manipulates the translated version of the term, but as a service to the user of the theorem prover, it "pretty prints" the term. Certain other macros are also treated this way, *e.g.*, `+` and `append`. See also `macro-aliases-table`.

Why does ACL2 get stuck on `Subgoal *1/2'4'`? This situation is one that you may encounter with any associative and commutative function. We start by asking: what rewrite rules do I need in order to put a function that is commutative and associative into canonical form?

What about `(equal (* x y) (* y x))`? Strictly speaking this is a bad rewrite rule because if applicable, it will keep on rewriting a term forever. However, since such rules are often useful, ACL2 has heuristics that prevent such rules from looping. See `loop-stopper`. Roughly, the heuristics allow such a rule to fire only if the resulting term is lexicographically "smaller" than the original term, *e.g.*, `(* a b)` is smaller than `(* b a)`, but not the other way around.

To deal with associativity we use `(equal (* (* x y) z) (* x (* y z)))`, which pushes parentheses to the right. Are these two rules enough to put terms into canonical form? It depends on what we mean by canonical form, but what will happen if ACL2 is given the above two theorems and asked to prove `(equal (* y (* x z)) (* x (* y z)))`? Notice that neither of the two rules apply, hence the two rules are not enough to prove

this example. Let us prove the above theorem on paper.

Proof

```
    (* y (* x z))
= { Associativity of * }
    (* (* y x) z)
= { Commutativity of * }
    (* (* x y) z)
= { Associativity of * }
    (* x (* y z)) □
```

The proof only requires the associativity and commutativity of *, but the rewrite rules we get from commutativity and associativity do not by themselves put terms into a canonical form. We also need the following theorem.

```
(defthm commutativity-of-*-2
  (equal (* y (* x z))
         (* x (* y z))))
```

Are these three rules enough? Prove `commutativity-of-*-2`.

ACL2 cannot prove `commutativity-of-*-2` without assistance. This should not be surprising since it was not able to prove `Subgoal *1/2'4'`. Does ACL2 "know" that * is associative and commutative? One way to determine what ACL2 knows is to scan its source file `axioms.lisp`, a file that describes the theory of ACL2 by enumerating the axioms and definitions. Another way is to use the history command :pl by typing the following.

```
:pl binary-*
```

ACL2 will print out all of the rules whose top function symbol is `binary-*`; this includes `associativity-of-*` and `commutativity-of-*`.

```
62. Rune:        (:REWRITE COMMUTATIVITY-OF-*)
63. Status:      Enabled
64. Lhs:         (* X Y)
65. Rhs:         (* Y X)
66. Hyps:        T
67. Equiv:       EQUAL
68. Outside-in:  NIL
69. Subclass:    BACKCHAIN
70. Loop-stopper: ((X Y BINARY-*))
71.
72. Rune:        (:REWRITE ASSOCIATIVITY-OF-*)
73. Status:      Enabled
74. Lhs:         (* (* X Y) Z)
75. Rhs:         (* X Y Z)
76. Hyps:        T
```

```
77. Equiv:      EQUAL
78. Outside-in: NIL
79. Subclass:   ABBREVIATION
```

While our hand proof of `commutativity-of-*-2` only requires the associativity and commutativity of `*`, the theorem prover does not prove `commutativity-of-*-2` even with the rewrite rules `associativity-of-*` and `commutativity-of-*` present, because the rewrite rules are applied in one direction. We have to force ACL2 to reproduce our hand proof. We do this below by using a hint to force ACL2 to take the first and last steps of our hand proof.

```
(defthm commutativity-of-*-2
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :use ((:instance associativity-of-* (y x) (x y))
                 (:instance associativity-of-*))
           :in-theory (disable associativity-of-*))))
```

The `:hints` argument to `defthm` allows us to give hints to the theorem prover. Each hint is attached to the name of some goal, with `"Goal"` being the name of the top-level conjecture. For more details, see hints.

The `:use` hint instantiates the named theorems and adds each as a hypothesis to the goal in question. This is a simple but subtle way to use previously proved theorems. Write down the goal produced and find a proof of it.

The `:in-theory` hint allows us to change the status of rules. Notice that we disable `associativity-of-*`. If we keep it enabled, it removes the instantiated hypotheses just added, by rewriting them away (to `t`).

We can mimic the above proof to get a set of rewrite rules that produce canonical terms for any associative and commutative function. We start in a ground-zero theory by using `encapsulate` to define `ac-fun`, a constrained function about which we know only that it is associative and commutative.

```
(encapsulate
 ((ac-fun (x y) t))
 (local (defun ac-fun (x y) (declare (ignore x y))
          nil))
 (defthm associativity-of-ac-fun
   (equal (ac-fun (ac-fun x y) z)
          (ac-fun x (ac-fun y z))))
 (defthm commutativity-of-ac-fun
   (equal (ac-fun x y)
          (ac-fun y x))))
```

We have the same problems with ac-fun that we had with *, namely that ac-fun terms are not rewritten into a canonical form. For example, the following theorem is not proved.

```
(thm
 (equal
  (ac-fun (ac-fun f (ac-fun c d)) (ac-fun (ac-fun c b) a))
  (ac-fun (ac-fun (ac-fun a c) b) (ac-fun c (ac-fun d f)))))
```

Therefore, we prove commutativity-2-of-ac-fun as follows.

```
(defthm commutativity-2-of-ac-fun
  (equal (ac-fun y (ac-fun x z))
         (ac-fun x (ac-fun y z)))
  :hints (("Goal"
           :in-theory (disable associativity-of-ac-fun)
           :use ((:instance associativity-of-ac-fun)
                 (:instance associativity-of-ac-fun
                            (x y) (y x))))))
```

Ac-fun terms are now rewritten into a canonical form, hence, ACL2 can easily prove the previous thm. Note that it is not the case that equivalent terms containing only ac-fun (and equal) are rewritten to t (that is a complicated issue), as the following example shows.

```
(thm
 (implies (equal (ac-fun a d) (ac-fun a e))
          (equal (ac-fun a (ac-fun c d))
                 (ac-fun a (ac-fun c e)))))
```

We can use functional instantiation (see lemma-instance) to show that any associative and commutative function also satisfies commutativity-2-of-ac-fun, as follows.

```
(defthm commutativity-2-of-*
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :by (:functional-instance
                commutativity-2-of-ac-fun
                (ac-fun binary-*)))))
```

ACL2 generates and establishes the constraints required, namely, that * is associative and commutative.

Notice that for the :functional-instance hint, we had to associate ac-fun with binary-* (instead of *, which is a macro). Another way to prove the above theorem, which bypasses this problem and highlights a very powerful feature of ACL2, is to use a pseudo-lambda expression (see lemma-instance) as follows.

```
(defthm commutativity-2-of-*
  (equal (* y (* x z))
         (* x (* y z)))
  :hints (("Goal"
           :by (:functional-instance
                commutativity-2-of-ac-fun
                (ac-fun (lambda (x y) (* x y)))))))
```

Since there are many functions that are associative and commutative, it may help to have a macro that automates this process. The macro will have one argument, the name of a function, and will generate the appropriate `defthm` form, using functional instantiation. Write this macro.

We must name the `defthm` produced by the macro. If the name of function is *op*, then the name of our `defthm` will be `commutativity-2-of-`*op*. We define `make-name` to make it convenient to create symbols. See the documentation for `intern-in-package-of-symbol`, `concatenate`, and other unfamiliar functions that appear below.

```
(defun make-name (prefix name)
  (intern-in-package-of-symbol
   (concatenate 'string
                (symbol-name prefix)
                "-"
                (symbol-name name))
   prefix))
```

The macro can now be defined as follows.

```
(defmacro commutativity-2 (op)
  `(defthm ,(make-name 'commutativity-2-of op)
     (equal (,op y (,op x z))
            (,op x (,op y z)))
     :hints (("Goal"
              :by (:functional-instance
                   commutativity-2-of-ac-fun
                   (ac-fun (lambda (x y) (,op x y))))))))
```

We can use the above macro on `*` as follows.

```
(commutativity-2 *)
```

Another solution to this problem, which we recommend you look at, can be found in the file `cowles/acl2-asg.lisp` in the `book/` directory of the ACL2 distribution.

10.3 Insertion Sort

Define an insertion sort on integers and prove it correct.

We define `insert`, a function that inserts a number into a list, and we use it to define `insertion-sort`.

```
(defun insert (a x)
  (cond ((atom x) (list a))
        ((<= a (car x)) (cons a x))
        (t (cons (car x) (insert a (cdr x))))))

(defun insertion-sort (x)
  (cond ((atom x) nil)
        (t (insert (car x) (insertion-sort (cdr x))))))
```

What does it mean for this function to be correct? At the least, the function must return an ordered list. We define a predicate to recognize ordered lists.

```
(defun orderedp (x)
  (cond ((atom (cdr x)) t)
        (t (and (<= (car x) (cadr x))
                (orderedp (cdr x))))))
```

We use `orderedp` to state a correctness condition.

```
(defthm insertion-sort-is-ordered
  (orderedp (insertion-sort x)))
```

The first simplification checkpoint in the proof attempt is

```
36. Subgoal *1/2''
37. (IMPLIES (AND (CONSP X)
38.               (ORDEREDP (INSERTION-SORT (CDR X))))
39.          (ORDEREDP (INSERT (CAR X)
40.                            (INSERTION-SORT (CDR X))))).
```

Not surprisingly we need to know that `insert` preserves `orderedp`. We therefore prove the following (which ACL2 guesses if we let it).

```
(defthm insert-ordered
  (implies (orderedp x)
           (orderedp (insert a x))))
```

ACL2 now proves `insertion-sort-is-sorted`.

But this is not enough. For example, if `insertion-sort` always returned `nil`, then we would be able to prove the above theorem even though we would not consider the function correct. We have to show that `insertion-sort` returns a permutation of its input. First, we define the notion of a permutation.

```
(defun in (a b)
  (cond ((atom b) nil)
        ((equal a (car b)) t)
        (t (in a (cdr b)))))
```

```
(defun del (a x)
  (cond ((atom x) nil)
        ((equal a (car x)) (cdr x))
        (t (cons (car x) (del a (cdr x))))))

(defun perm (x y)
  (cond ((atom x) (atom y))
        (t (and (in (car x) y)
                (perm (cdr x) (del (car x) y))))))
```

We now prove that `insertion-sort` returns a permutation of its input.

```
(defthm insertion-sort-is-perm
  (perm (insertion-sort x) x))
```

The first simplification checkpoint is:

```
36. Subgoal *1/2''
37. (IMPLIES (AND (CONSP X)
38.               (PERM (INSERTION-SORT (CDR X)) (CDR X)))
39.          (PERM (INSERT (CAR X)
40.                        (INSERTION-SORT (CDR X)))
41.                X)).
```

Although ACL2 completes the proof on its own, we take it as a challenge to save ACL2 from attempting subsidiary induction arguments. The goal above suggests the need to show that `insert` preserves `perm`. If we formulate a rewrite rule that looks like this goal, then it is not clear that ACL2 will be able to prove it—the `car` and `cdr` may somehow get in the way of induction—and more seriously, we wonder a bit if the rule may somehow loop when it is used later. An example of such a looping rule is discussed below (page 197). We can let the prover's next goal guide us in finding a suitable rule.

```
43. The destructor terms (CAR X) and (CDR X) can be eliminated
44. by using CAR-CDR-ELIM to replace X by (CONS X1 X2), generalizing
45. (CAR X) to X1 and (CDR X) to X2.  This produces the following
46. goal.
47.
48. Subgoal *1/2'''
49. (IMPLIES (AND (CONSP (CONS X1 X2))
50.               (PERM (INSERTION-SORT X2) X2))
51.          (PERM (INSERT X1 (INSERTION-SORT X2))
52.                (CONS X1 X2))).
```

We can derive the following theorem by generalizing the goal above. In fact ACL2 comes up with this exact generalization (using different variable names), but it is rare that ACL2's generalization heuristics are so on-target.

```
(defthm insert-perm-cons
  (implies (perm x y)
           (perm (insert a x) (cons a y))))
```

ACL2 can now prove `insertion-sort-is-perm` without any induction other than the one at the top level.

10.4 Tree Manipulation

The function `flatten` (page 49) returns a list of the tips of a tree.

```
(defun flatten (x)
  (cond ((atom x) (list x))
        (t (append (flatten (car x)) (flatten (cdr x))))))
```

In Exercise 7.9, the following, more efficient, function (why is it more efficient?) is introduced.

```
(defun mc-flatten (x a)
  (cond ((atom x) (cons x a))
        (t (mc-flatten (car x)
                       (mc-flatten (cdr x) a)))))
```

We will give a solution to Exercise 7.9, by proving that the above functions are equivalent.

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

As we saw with the factorial example, this theorem should raise some flags: do we need to prove a stronger theorem? Since opening the definition of `mc-flatten` will replace the above `nil` with a term involving the variable `x`, the answer is yes. We therefore attempt to come up with a rule for rewriting the term `(mc-flatten x y)` in terms of `flatten`. After a little thought, we try:

```
(defthm flatten-mc-flatten-lemma
  (equal (mc-flatten x a)
         (append (flatten x) a)).
```

ACL2 proves the `flatten-mc-flatten-lemma`, but has to prove the associativity of `append` as a separate induction. Since this is a useful theorem to have around, we prove it.

```
(defthm associativity-of-append
  (equal (append (append x y) z)
         (append x (append y z)))).
```

ACL2 can now prove `flatten-mc-flatten`, but it uses induction. Why? What other fact is required so that the theorem prover can prove `flatten-mc-flatten` entirely by simplification? We leave you to contemplate this and move on to another tree processing function.

Admit the following function.

```
(defun gopher (x)
  (if (or (atom x)
          (atom (car x)))
      x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
```

ACL2 does not admit `gopher` because `acl2-count` does not decrease on the recursive call. Note that `gopher` recurs exactly the way `flat` does on one of its recursive calls (see page 108), hence, we use the `acl2-count` of the `car` as the measure.

```
(defun gopher (x)
  (declare (xargs :measure (acl2-count (car x))))
  (if (or (atom x)
          (atom (car x)))
      x
    (gopher (cons (caar x) (cons (cdar x) (cdr x))))))
```

The following function uses `gopher` to determine if two trees have the same fringe; admit it.

```
(defun samefringe (x y)
  (if (or (atom x)
          (atom y))
      (equal x y)
    (and (equal (car (gopher x))
                (car (gopher y)))
         (samefringe (cdr (gopher x))
                     (cdr (gopher y))))))
```

If we look at the first simplification checkpoint during admission, we see:

```
26. Goal'
27. (IMPLIES (AND (CONSP X)
28.               (CONSP Y)
29.               (EQUAL (CAR (GOPHER X))
30.                      (CAR (GOPHER Y))))
31.          (< (ACL2-COUNT (CDR (GOPHER X)))
32.             (ACL2-COUNT X))).
33.
34. Name the formula above *1.
```

The suggested rewrite rule is shown below.

```
(defthm gopher-acl2-count-cdr
  (implies (consp x)
           (< (acl2-count (cdr (gopher x)))
              (acl2-count x))))
```

ACL2 can prove the above and can subsequently admit `samefringe`.

Many experienced users would make the rule above a linear rule. As such it would cause the linear arithmetic procedure to add the indicated inequality to the linear data base whenever some inequality mentioned (an instance of) `(acl2-count (cdr (gopher x)))` and the hypothesis `(consp x)` can be proved. If stored as a rewrite rule, as above, the rule is quite restricted in its applicability: it conditionally rewrites the indicated <-expression to `t`. This is all we need in the current situation.

Prove that `samefringe` is correct.

We choose the following theorem.

```
(defthm correctness-of-samefringe
  (equal (samefringe x y)
         (equal (flatten x)
                (flatten y))))
```

ACL2 cannot prove this theorem with its current rules. Looking at the first simplification checkpoint, we see:

```
70. Subgoal *1/3'
71. (IMPLIES (AND (CONSP X)
72.               (CONSP Y)
73.               (NOT (EQUAL (CAR (GOPHER X))
74.                           (CAR (GOPHER Y)))))
75.          (NOT (EQUAL (FLATTEN X) (FLATTEN Y)))).
76.
77. Name the formula above *1.1.
```

This suggests the following rewrite rule.

```
(defthm car-gopher-car-flatten
  (implies (consp x)
           (equal (car (gopher x))
                  (car (flatten x)))))
```

It turns out that, for this simple example, it does not make a difference which way we orient the rewrite rule. In general, you should rewrite more complicated terms into simpler onesand you should design rewrite rules that massage terms into canonical forms.

We try proving `correctness-of-samefringe` again. Once again, ACL2 cannot prove the theorem and once again, we check the first simplification checkpoint:

```
102. Subgoal *1/2.2'
103. (IMPLIES (AND (CONSP X)
104.                (CONSP Y)
105.                (EQUAL (CAR (FLATTEN X))
106.                       (CAR (FLATTEN Y)))
107.                (EQUAL (FLATTEN (CDR (GOPHER X)))
108.                       (FLATTEN (CDR (GOPHER Y))))
109.                (SAMEFRINGE (CDR (GOPHER X))
110.                            (CDR (GOPHER Y))))
111.           (EQUAL (FLATTEN X) (FLATTEN Y))).
```

We decide to try moving the `cdr` to the left (outside) of the `flatten` in lines 107 and 108; this suggests that we prove the following theorem.

```
(defthm cdr-flatten-gopher
  (implies (consp x)
           (equal (flatten (cdr (gopher x)))
                  (cdr (flatten (gopher x))))))
```

ACL2 proves the above theorem, but when we try to prove the theorem `correctness-of-samefringe`, we get the following peculiar error. (This is a GCL error; if you are using a different Common Lisp, you may get a different error message.)

```
90. Error: Value stack overflow.
91. Fast links are on: do (si::use-fast-links nil) for debugging
92. Error signalled by ACL2_*1*_ACL2::DEFTHM-FN.
93. Broken at COND.  Type :H for Help.
94. ACL2>>
```

In GCL it is even possible to get a segmentation error that aborts the Lisp process.

What is going on? This is something that happens to everyone (although it happens much more frequently to beginners): the rewriter has entered an infinite loop. Often, by inspection one can figure out what combination of rewrite rules is leading to an infinite loop, but if not, one can use `brr` to examine the rewriter. See `break-rewrite` for the details, but briefly, with `brr` one can `monitor` rewrite rules: when a monitored rule is tried, the rewriter will enter an interactive break, where you can inspect the context (there are many things you can do, see `brr-commands`). In many situations, the following trick is all that is needed. After a stack overflow, reenter the top-level ACL2 loop[1], type `:brr t` and try proving the theorem again. This will lead to another stack overflow, but now, enter raw Lisp and type `(cw-gstack *deep-gstack* state)`. This will print out the rewrite stack, which usually makes it clear why the rewriter is looping. After doing this, we observe the following loop.

[1] In GCL it is best to enter raw Lisp first and execute `(si::use-fast-links nil)` to prevent a stack overflow from manifesting itself as a segmentation error.

```
31. 9. Attempting to apply (:REWRITE CDR-FLATTEN-GOPHER) to
32.      (FLATTEN (CDR (GOPHER X)))
33. 10. Rewriting (to simplify) the rhs of the conclusion,
34.      (CDR (FLATTEN (GOPHER X))),
35.    under the substitution
36.      X : X
37. 11. Rewriting (to simplify) the first argument,
38.      (FLATTEN (GOPHER X)),
39.    under the substitution
40.      X : X
41. 12. Attempting to apply (:DEFINITION FLATTEN) to
42.      (FLATTEN (GOPHER X))
43. 13. Rewriting (to simplify) the rewritten body,
44.      (BINARY-APPEND (FLATTEN (CAR #))
45.                     (FLATTEN (CDR #))),
46. 14. Rewriting (to simplify) the second argument,
47.      (FLATTEN (CDR (GOPHER X))),
48. 15. Attempting to apply (:REWRITE CDR-FLATTEN-GOPHER) to
49.      (FLATTEN (CDR (GOPHER X)))
```

What is the rewriter really doing here?
Given the term

```
1. (flatten (cdr (gopher x))),
```

ACL2 uses the rewrite rule `cdr-flatten-gopher` to rewrite it to

```
2. (cdr (flatten (gopher x))).
```

ACL2 then applies the definition of `flatten` to get

```
3. (cdr (append (flatten (car (gopher x)))
                (flatten (cdr (gopher x))))).
```

ACL2 then tries to simplify the second argument to **append**, but this is the same term we started with, so we managed to pump 1, our original term, to 3, a bigger term that contains 1 as a subterm.

Having identified the loop, we can introduce a new rewrite rule that prevents the loop from occurring or we can throw away the offending rewrite rule. Let us try the first approach. The following rule comes to mind.

```
(defthm flatten-gopher
  (equal (flatten (gopher x))
         (flatten x)))
```

Notice that since this rule was added to the theorem prover's rules after the definition of `flatten`, it will be used before the definition of `flatten` and will therefore keep the above loop from occurring.

ACL2 proves `correctness-of-samefringe`, the main theorem (by performing seven inductions, but since this is the last theorem we want to prove in this section, we do not investigate further).

It is instructive to try the second approach to stopping the loop. We have to replace `cdr-flatten-gopher` by a rewrite rule that does not loop. We can combine the above two rewrite rules into the following rewrite rule.

```
(defthm cdr-flatten-gopher
  (implies (consp x)
           (equal (flatten (cdr (gopher x)))
                  (cdr (flatten x)))))
```

The observant reader may have thought of this rewrite rule previously, when we chose `cdr-flatten-gopher` instead. ACL2 can now prove the main theorem, `correctness-of-samefringe`.

10.5 Binary Adder and Multiplier

In this section we will define and prove correct a binary adder and multiplier.

10.5.1 Binary Adder

Define a binary adder and prove that it adds.

We use `t` and `nil` to represent 1 and 0, respectively, and will represent binary numbers as lists of `t`'s and `nil`'s. Our plan is to define a serial adder. A serial adder works by adding two binary numbers bit by bit. Hence, it can be built out of a full adder using recursion, with an algorithm very similar to the one taught to children for adding base-10 numbers. (If you are not familiar with computer arithmetic, consult a book on computer architecture, *e.g.*, [18].)

In order to define a full adder, we define the following Boolean-valued functions.

```
(defun band (p q) (if p (if q t nil) nil))
(defun bor  (p q) (if p t (if q t nil)))
(defun bxor (p q) (if p (if q nil t) (if q t nil)))
(defun bmaj (p q c)
  (bor (band p q)
       (bor (band p c)
            (band q c))))
```

A full adder has three inputs (bits) and returns the sum of its inputs as two bits: a sum and a carry. We define a full adder as follows.

```
(defun full-adder (p q c)
  (mv (bxor p (bxor q c))
      (bmaj p q c)))
```

Recall that the serial adder will take as input two lists of Booleans and a carry-in bit. If we were designing hardware, we would probably know something about the binary numbers given as input to the adder, *e.g.*, their length, or that they have the same length. However, ACL2 functions have to be total, hence, we have to decide what to do even with inputs that are not of the same length. We decide to design an adder that correctly adds any pair of binary numbers, even if they are not the same length (we are thinking ahead here, because when we define a multiplier, it will be useful to have such an adder). We have to decide if the first bit of a list is the high-order bit or the low-order bit. It is more convenient to make the first bit the low-order bit: otherwise, we would have to align the numbers before adding them. Our definition is as follows.

```
(defun serial-adder (x y c)
  (if (and (endp x) (endp y))
      (list c)
    (mv-let (sum cout)
            (full-adder (car x) (car y) c)
            (cons sum (serial-adder (cdr x) (cdr y) cout)))))
```

ACL2 does not admit the above function because it cannot prove termination. If we scan through the output produced, we see:

```
 8. For the admission of SERIAL-ADDER we will use the relation
 9. EO-ORD-< (which is known to be well-founded on the domain
10. recognized by EO-ORDINALP) and the measure (ACL2-COUNT X).
```

If we think about why `serial-adder` terminates, we realize that termination also depends on `y`, *e.g.*, if the length of `y` is greater than the length of `x`, then `serial-adder` may take more than `(acl2-count x)` steps. In this case, the heuristics used by ACL2 to guess a measure do not work and we are forced to give ACL2 a measure explicitly.

```
(defun serial-adder (x y c)
  (declare (xargs :measure (+ (len x) (len y))))
  ...)
```

There are many other measures that work. Two examples are `(max (len x) (len y))` and `(+ (acl2-count x) (acl2-count y))`.

Execute the adder on a few examples. What does it mean for the adder to be correct? Well, it means that it adds! More specifically, the binary number returned is the sum of the binary numbers given as input plus the initial carry-in. In order to write this formally, we have to define a function that transforms a binary number into a number.

```
(defun n (v)
  (cond ((endp v) 0)
        ((car v) (+ 1 (* 2 (n (cdr v)))))
        (t (* 2 (n (cdr v))))))
```

We state correctness as follows.

```
(defthm serial-adder-correct
  (equal (n (serial-adder x y c))
         (+ (n x) (n y) (if c 1 0))))
```

ACL2 does not prove this theorem. Scanning through the output produced by ACL2 until we reach the simplification checkpoint, we see the following:

```
1310. Subgoal *1/1.4'5'
1311. (EQUAL (* 2 (N (SERIAL-ADDER X2 NIL T)))
1312.        (+ 1 1 (* 2 (N X2)))).
1313.
1314. Name the formula above *1.1.
```

If we scan ahead to the next checkpoint, we see:

```
1378. Subgoal *1/1.3'6'
1379. (EQUAL (+ 1 (* 2 (N (SERIAL-ADDER X2 NIL NIL))))
1380.        (+ 1 (* 2 (N X2)))).
1381.
1382. Name the formula above *1.2.
```

`*1.3` is the same as `*1.2`; finally we have:

```
1490. Subgoal *1/1.1'5'
1491. (EQUAL (* 2 (N (SERIAL-ADDER X2 NIL NIL)))
1492.        (* 2 (N X2))).
1493.
1494. Name the formula above *1.4.
```

The appropriate thing to do now is to determine which lemmas are suggested by `*1.1`, `*1.2`, and `*1.4`. Notice that `*1.4` has the form `(equal (* 2 a) (* 2 b))`; this suggests that the multiplication is not important and that the appropriate lemma is the simpler `(equal a b)`. Notice that the suggested lemma will allow ACL2 to discharge `*1.2` and `*1.3`. (As an aside, we point out that sometimes, on very large problems or with rules that match too often or take too much time[2], it is wiser to prove an unsimplified lemma because it is less applicable and therefore the theorem prover is faster.) After a similar analysis of `*1.1`, we decide to prove the following two lemmas.

```
(defthm serial-adder-correct-nil-nil
  (equal (n (serial-adder x nil nil))
         (n x)))
```

[2]Statistics supplied by `accumulated-persistence` can be used to identify rules that blow up ACL2's search space.

```
(defthm serial-adder-correct-nil-t
  (equal (n (serial-adder x nil t))
         (+ 1 (n x))))
```

We really need to prove the above lemmas in the order indicated because ACL2 cannot directly prove `serial-adder-correct-nil-t`. Why? ACL2 can now prove `serial-adder-correct`.

10.5.2 Binary Multiplier

Define a binary multiplier and prove that it multiplies.

Multiplication can be performed by repeatedly shifting and adding. That is, to multiply y by x, we initialize p, a partial sum, to 0 and iterate over x, one bit at a time from the low-order bit to the high order bit. If the current bit is 0, we multiply y by 2, otherwise we add y to p and multiply y by 2. Note that multiplying a binary number by 2 is the same as inserting a 0 low order bit (*i.e.*, shifting). A multiplier in ACL2 is defined as follows.

```
(defun multiplier (x y p)
  (if (endp x)
      p
    (multiplier (cdr x)
                (cons nil y)
                (if (car x)
                    (serial-adder y p nil)
                  p))))
```

The multiplier is correct if it multiplies. More specifically, the binary number returned is the product of the binary numbers given as input plus the initial partial sum. Here is a formal statement.

```
(defthm multiplier-correct
  (equal (n (multiplier x y p))
         (+ (* (n x) (n y)) (n p))))
```

ACL2 tries to prove this theorem, but after a second or so, it is clear from the output streaming by that many inductions will be necessary for this proof attempt to succeed (ACL2 actually runs for a long time and eventually runs out of memory). We therefore interrupt ACL2 and scan through the output until we reach the first simplification checkpoint, where we see the following:

```
77. Subgoal *1/3''
78. (IMPLIES (AND (CONSP X)
79.               (NOT (CAR X))
80.               (EQUAL (N (MULTIPLIER (CDR X) (CONS NIL Y) P))
81.                      (+ (N P) (* (N (CDR X)) 2 (N Y)))))
```

```
82.          (EQUAL (+ (N P) (* (N (CDR X)) 2 (N Y)))
83.                 (+ (N P) (* (N Y) 2 (N (CDR X))))))).
```

Notice that the conclusion of `Subgoal *1/3''` has the form `(equal (+` a `(* x y z)) (+` a `(* z y x)))`; this suggests that the addition is not important and that we need a lemma that allows us to conclude `(equal (* x y z) (* z y x))`. But, we have already seen such a lemma, namely `commutativity-of-*-2`. Once we get ACL2 to prove `commutativity-of--*-2`, `multiplier-correct` follows.

10.5.3 Miscellaneous

We can generate a gate-level design of an adder and multiplier for fixed length binary numbers by unrolling the recursive definitions the appropriate number of times. This idea was used to generate a netlist description of a formally verified chip [19]. Note that if we unroll the serial adder, we get a ripple carry adder. A formal netlist description language, similar to that described in Hunt's case study in the companion volume [22], has been used to describe and verify a chip which was then fabricated [20].

If one is interested in proving more complicated fixed-length hardware modules correct, then BDDs can be useful. BDDs [12, 32] are data structures used for the simplification of Boolean expressions. They have been found to work well in practice, especially with hardware. In ACL2, BDDs are generalized: they can represent not only Boolean values, but arbitrary ACL2 terms, and they are integrated with rewriting. The ACL2 distribution contains, in the directory `books/bdd`, examples highlighting the use of BDDs, *e.g.*, there are specifications of a simple ripple-carry ALU and a tree-structured propagate-generate ALU, as well as proofs—employing BDDs—of their equivalence.

10.6 Compiler for Stack Machine

We will define a simple stack-based machine and a compiler that given an expression generates code for the machine. Of course we will prove that our compiler is correct.

We want the stack machine to have instructions with names such as `push` and `pop`. Since we cannot define functions with such names in the `ACL2` package (these symbols are pre-defined in Common Lisp), we will define a new package as follows.

```
(defpkg "compile"
  (set-difference-eq
   (union-eq *acl2-exports*
       (union-eq '(acl2-numberp len)
```

```
          *common-lisp-symbols-from-main-lisp-package*))
  '(pop push top compile step eval)))
```

The constant `*acl2-exports*` is a list of symbols that is convenient to import into other packages. It includes many commonly used symbols—such as `defun`, `defthm`, `iff`, and so on—that you would otherwise have to prefix by `"ACL2::"`. `Union-eq` is a function that returns the set union of two lists; `set-difference-eq` is a function that takes the set difference of two lists. The above `defpkg` defines the new symbol package `compile` and imports exactly the symbols we want. We select this as the current package.

```
(in-package "compile")
```

10.6.1 Expressions

Our expressions are built out of symbols, numbers, and the functions `inc`, `sq`, `+`, and `*`. Notice that binary functions in expressions are written in infix notation.

```
(defun exprp (exp)
  (cond
   ((atom exp)
    (or (symbolp exp) (acl2-numberp exp)))
   ((equal (len exp) 2)
    (and (or (equal (car exp) 'inc)
             (equal (car exp) 'sq))
         (exprp (cadr exp))))
   (t
    (and (equal (len exp) 3)
         (or (equal (cadr exp) '+)
             (equal (cadr exp) '*))
         (exprp (car exp))
         (exprp (caddr exp))))))
```

Define the semantics of expressions: `inc` increments by one, `sq` squares, `+` adds, and `*` multiplies.

Since expressions can contain symbols, they have a value in the context of an *environment*, an alist (see page 31) that relates symbols to numbers. Here is the function to look up the value of a symbol in an environment.

```
(defun lookup (var alist)
  (cond ((endp alist)
         0)  ; default
        ((equal var (car (car alist)))
         (cdr (car alist)))
        (t (lookup var (cdr alist)))))
```

Alists are often used for representing environments, memories, functions, and related concepts. Note that you can update the value of a variable by consing a cons to the beginning of the alist, because `lookup` finds the first cons matching a variable. Another nice property of the alist representation is that the alist only has to contain the variables you are interested in; all other variables have a default value (in our case, 0). This can be very useful, *e.g.*, suppose you want to model a memory assigning values to 64-bit wide addresses.

The following function evaluates an expression in an environment.

```
(defun eval (exp alist)
  (cond
   ((atom exp)
    (cond ((symbolp exp) (lookup exp alist))
          (t exp)))
   ((equal (len exp) 2)
    (cond ((equal (car exp) 'inc)
           (+ 1 (eval (cadr exp) alist)))
          (t  ; 'sq
           (* (eval (cadr exp) alist)
              (eval (cadr exp) alist)))))
   (t  ; (equal (len exp) 3)
    (cond ((equal (cadr exp) '+)
           (+ (eval (car exp) alist)
              (eval (caddr exp) alist)))
          (t  ; *
             (* (eval (car exp) alist)
                (eval (caddr exp) alist)))))))
```

Evaluate `eval` on several examples.

10.6.2 Stack Machine

The stack machine will have six instructions: `pushv` pushes the value of a variable on the stack, `pushc` pushes a constant on the stack, `dup` duplicates the top of the stack, `add` adds the top two elements of the stack, `mul` multiplies the top two elements of the stack, and anything else acts as a skip. Define a function that given an instruction, an environment, and a stack, steps the machine for a single step and returns the new stack.

We start by defining some simple stack manipulation functions.

```
(defun pop (stk) (cdr stk))
(defun top (stk) (if (consp stk) (car stk) 0))
(defun push (val stk) (cons val stk))
```

We define step, the function that steps the machine for a single step as follows.

```
(defun step (ins alist stk)
  (let ((op (car ins)))
    (case op
      (pushv (push (lookup (cadr ins) alist) stk))
      (pushc (push (cadr ins) stk))
      (dup   (push (top stk) stk))
      (add   (push (+ (top (pop stk)) (top stk))
                   (pop (pop stk))))
      (mul   (push (* (top (pop stk)) (top stk))
                   (pop (pop stk))))
      (t stk))))
```

Define a function that given a program (a list of instructions), an environment, and a stack runs the program to completion. The function should return the final stack.

```
(defun run (program alist stk)
  (cond ((endp program) stk)
        ((run (cdr program)
              alist
              (step (car program) alist stk)))))
```

The machine that we are defining does not allow any looping, but in general—if we were defining a more complicated machine—we cannot admit a function such as run, because there are programs that never terminate. In such a situation, we would instead define a similar function that takes an extra argument indicating how many times to step the machine.

For an example of a more complex state machine in ACL2, see the article by Greve, Wilding and Hardin in [22]. In addition, for a comprehensive description of how to define such machines in ACL2 and how to configure the rewriter to facilitate proofs about their programs, see [6].

10.6.3 Compiler

Write a compiler that takes expressions and returns programs (for the stack machine) that evaluate the expressions.

```
(defun compile (exp)
  (cond
   ((atom exp)
    (cond ((symbolp exp)
           (list (list 'pushv exp)))
          (t (list (list 'pushc exp)))))
   ((equal (len exp) 2)
```

```
(cond ((equal (car exp) 'inc)
       (append (compile (cadr exp)) '((pushc 1) (add))))
      (t (append (compile (cadr exp)) '((dup) (mul))))))
(t (cond ((equal (cadr exp) '+)
          (append (compile (car exp))
                  (compile (caddr exp))
                  '((add))))
         (t (append (compile (car exp))
                    (compile (caddr exp))
                    '((mul)))))))
```

Prove that `compile` compiles.

Our notion of correctness is the following.

```
(defthm compile-is-correct
  (implies (exprp exp)
           (equal (top (run (compile exp) alist stk))
                  (eval exp alist))))
```

`Compile-is-correct` says that the value of expression `exp` in environment `alist` is equivalent to the top of the stack of a machine running the program produced by compiling `exp`, in environment `alist`. Notice that the initial value of the stack seems to play no important role in the above condition; all we care about is the top of the final stack.

Following our "brain-dead" approach to proving theorems, we try to prove this theorem and look at the first simplification checkpoint, where we see:

```
107. Subgoal *1/6.7
108. (IMPLIES (AND (CONSP EXP)
109.               (CONSP (CDR EXP))
110.               (NOT (CONSP (RUN (COMPILE (CAR EXP)) ALIST STK)))
111.               (EQUAL 0 (EVAL (CAR EXP) ALIST))
112.               (NOT (CONSP (RUN (COMPILE (CADDR EXP))
113.                                ALIST STK)))
114.               (EQUAL 0 (EVAL (CADDR EXP) ALIST))
115.               (EQUAL (+ 1 1 (LEN (CDDR EXP))) 3)
116.               (EQUAL (CADR EXP) '*)
117.               (EXPRP (CAR EXP))
118.               (EXPRP (CADDR EXP))
119.               (CONSP (RUN (APPEND (COMPILE (CAR EXP))
120.                                   (COMPILE (CADDR EXP))
121.                                   '((MUL)))
122.                           ALIST STK)))
123.          (EQUAL (CAR (RUN (APPEND (COMPILE (CAR EXP))
124.                                   (COMPILE (CADDR EXP))
125.                                   '((MUL)))
126.                           ALIST STK))
127.                 0)).
```

There are a few terms of the form (run (append x y) a s) present above. It is almost always the case when proving machines correct that we need a theorem about the composition of programs which says you can run a program composed of two parts by first running the first and then the second. We prove the following.

```
(defthm composition
  (equal (run (append prg1 prg2) alist stk)
         (run prg2 alist (run prg1 alist stk))))
```

We try to prove the main result again. Looking at the output produced by ACL2 induces panic and we instead decide to a step back and look at the big picture. What approach is ACL2 taking to this problem?

```
32. Perhaps we can prove *1 by induction.  Three induction schemes
33. are suggested by this conjecture.  Subsumption reduces that
34. number to two.  These merge into one derived induction scheme.
35.
36. We will induct according to a scheme suggested by
37. (EVAL EXP ALIST).  If we let  (:P ALIST EXP STK) denote *1
38. above then the induction scheme we'll use is
39. (AND (IMPLIES (AND (NOT (ATOM EXP))
40.                    (NOT (EQUAL (LEN EXP) 2))
41.                    (NOT (EQUAL (CADR EXP) '+))
42.                    (:P ALIST (CAR EXP) STK)
43.                    (:P ALIST (CADDR EXP) STK))
44.               (:P ALIST EXP STK))
```

If we look at the first conjunct (lines 39–44) of the induction scheme, we notice that ACL2 is trying to prove (:P ALIST EXP STK)) from (:P ALIST (CAR EXP) STK) and (:P ALIST (CADDR EXP) STK)) (in the case where we are multiplying). Is this reasonable? What happens when we run a compiled program? Try an example.

We will manipulate a term that matches the first conjunct of the induction scheme and will try to rewrite it until the appropriate induction is apparent. Below, we abbreviate compile, append, and stk by c, app, and s, respectively (this allows us to focus on the structure of the term).

```
    (top (run (c '(x * y)) a s))
= { Definition of com }
    (top (run (app (c 'x) (c 'y) '((mul))) a s))
= { Composition }
    (top (run (app (c 'y) '((mul))) a (run (c 'x) a s)))
= { Composition }
    (top (run '((mul)) a (run (c 'y) a (run (c 'x) a s))))
```

We have rewritten the term we started with above in terms of `x` and `y`, the subexpressions of `(x * y)`. We did this because induction allows us to assume what we want to prove for smaller instances of the problem, *e.g.*, we can assume that `(top (run (c 'x) a s))` is `(eval 'x a)` and that `(top (run (c 'y) a (run (c 'x) a s)))` is `(eval 'y a)`. At this point, it should clear that our induction hypotheses are too weak because in order to prove that the machine multiplies on the above example, not only do we need to know what is on top of the stack after we run the code produced by the compiler for `y`, but we also need to know that right below that is `(eval 'x a)`. We are now in the familiar situation where we have to strengthen a theorem in order to apply induction. As we saw, we want our theorem to tell us not only what happens to the top of the stack, but what happens to the rest of the stack. The following comes to mind.

```
(defthm compile-is-correct-general
  (implies (exprp exp)
           (equal (run (compile exp) alist stk)
                  (cons (eval exp alist) stk))))
```

Will this work? Let us rework the above example.

```
  (run (c '(x * y)) a s)
```

= { Definition of `com` }

```
  (run (app (c 'x) (c 'y) '((mul))) a s)
```

= { Composition }

```
  (run (app (c 'y) '((mul))) a (run (c 'x) a s))
```

= { Composition }

```
  (run '((mul)) a (run (c 'y) a (run (c 'x) a s)))
```

We can assume the following inductive hypotheses.

1. `(run (c 'x) a s)` is `(cons (eval 'x a) s)` and
2. `(run (c 'y) a (run (c 'x) a s))` is `(cons (eval 'y a) (cons (eval 'x a) s))`.

We can symbolically run the program to see that the `mul` instruction will pop the values of `x` and `y` off the stack and will push their product, therefore it seems that we can prove this theorem by induction. ACL2, however, does not prove the theorem.

```
12. We will induct according to a scheme suggested by
13. (EVAL EXP ALIST).  If we let  (:P ALIST EXP STK) denote *1
14. above then the induction scheme we'll use is
15. (AND (IMPLIES (AND (NOT (ATOM EXP))
16.                    (NOT (EQUAL (LEN EXP) 2))
17.                    (NOT (EQUAL (CADR EXP) '+))
```

```
18.                    (:P ALIST (CAR EXP) STK)
19.                    (:P ALIST (CADDR EXP) STK))
20.              (:P ALIST EXP STK))
```

Inspection of the first conjunct of the induction scheme shows why the proof attempt fails. Above, when we considered this part of the induction, we assumed induction hypothesis 2, whose stack (`(cons (eval 'x a) s)`) differs from `s`, the stack of the induction conclusion. The first conjunct of the induction scheme generated by ACL2, however, mentions the stack of the induction conclusion. Before continuing, let us make sure that the substitution we are suggesting does not violate the induction principle. The measure used to justify this induction, `(acl2-count exp)` (which is the measure used to admit `eval`), does not mention `stk`, therefore, we can substitute anything for `stk` (as long as we substitute something smaller for `exp`). We have to tell ACL2 what the right induction scheme is. This is done by defining a function that recurs according to the scheme and giving ACL2 a hint (see hints) to use the induction scheme suggested by this function. Define such a function.

```
(defun compiler-induct (exp alist stk)
  (cond
   ((atom exp) stk)
   ((equal (len exp) 2)
    (compiler-induct (cadr exp) alist stk))
   (t  ; Any binary function may be used in place of append below
    (append (compiler-induct (car exp) alist stk)
            (compiler-induct (caddr exp)
                             alist
                             (cons (eval (car exp) alist)
                                   stk))))))
```

We give ACL2 the appropriate hint as follows.

```
(defthm compile-is-correct-general
  (implies (exprp exp)
           (equal (run (compile exp) alist stk)
                  (cons (eval exp alist) stk)))
  :hints (("Goal"
           :induct (compiler-induct exp alist stk))))
```

ACL2 can prove this theorem and then `compile-is-correct` follows.

11

Theorem Prover Exercises

This chapter contains exercises of varying degrees of difficulty. The exercises will allow you to gain experience in using the theorem prover. We cannot over-stress the importance of doing the exercises. It is one thing to understand how the theorem prover works and another to be a competent user. We remind you that solutions to all of the exercises are on the Web (see the link to this book's page on the ACL2 home page). We suggest that you do the exercises without consulting our solutions, but that once you are done, we recommend that you compare your solutions to ours.

Please do not be discouraged if some of these exercises take considerable thought and time. After all, we did warn that you will be hard pressed to find a more challenging game. With practice you will win the game with increasing frequency.

11.1 Starters

The exercises in this first section are generally simpler than those in the sections that follow. We suggest that you use them to get acquainted with the ACL2 theorem prover.

Exercise 11.1 *Are the following functions admissible? If not, why not? If so, admit them.*

```
(defun f (x)
  (if (endp x)
      0
    (1+ (f (cdr x)))))
(defun f (x)
  (if (null x)
      0
    (1+ (f (cdr x)))))
```

Exercise 11.2 *Recall the definitions of* `flatten` *and* `swap-tree` *(pages 49 and 115).*

- *Use ACL2 to prove the following theorem, or an appropriately-fixed theorem, from page 115:*

```
(equal (flatten (swap-tree x)) (rev (flatten x)))
```

- *Admit the function* `flat` *defined on page 108. Prove that* `(flat x)` *is equal to* `(flatten x)` *(a fact proved by hand on page 109).*

Exercise 11.3 *Prove the following. (*Hint: *You may find it helpful to use* `:pe` *to view the definitions of* `append` *and its supporting functions.)*

```
(defthm reverse-reverse
  (implies (true-listp x)
           (equal (reverse (reverse x))
                  x)))
```

Exercise 11.4 *Prove that* `(rev x)` *(see page 124) is equal to* `(reverse x)` *if* `x` *is not a string.*

11.2 Sorting

In this section, you will be asked to prove several sorting algorithms correct. What does it mean for a sorting algorithm to be correct? Correctness is captured by the following two conditions.

1. The output is ordered.
2. The output is a permutation of the input.

In Chapter 10, we proved the correctness of insertion sort and in the process defined both the notion of a permutation, `perm`, and of an ordered list, `orderedp` (on pages 193 and 192, respectively). That proof was relatively easy; however, the proofs in this section are going to require more work. We start by proving that `perm` is an equivalence relation. Recall that in the discussion of congruence-based reasoning (page 139), we saw that the theorem prover can use equivalence relations the way it uses `equal`, in the right contexts. The macro `defequiv` can be used to prove that a relation is an equivalence relation.

Exercise 11.5 *Create a certified book (see* `certify-book`*) that starts with definitions including the definition of* `perm`*, concludes with* `(defequiv perm)`*, and has any number of* `local` *events in between. We suggest proceeding as follows.*

- *Prove* `(perm x x)`.

- *Prove* `(implies (perm x y) (perm y x))`.
- *Prove* `(implies (and (perm x y) (perm y z)) (perm x z)))`.
- *Use* `:trans1` *to print out the immediate expansion (see page 37 of* `(defequiv perm)`.
- *Prove* `(defequiv perm)`.

We will use the fact that `perm` is an equivalence relation by proving some congruence rules. We use the macro `defcong` to prove congruence rules.

Exercise 11.6 *Use* `:trans1` *to print out the immediate expansion of the following.*
`(defcong perm perm (append x y) 1)`

Exercise 11.7 *Open a new book in which to put your solutions to the following.*

- *Prove* `(defcong perm perm (append x y) 1)`.
- *Prove* `(defcong perm perm (append x y) 2)`.

An interesting sorting algorithm is quicksort. The idea is to break a list in two, where the elements of the first list are those that are less than some pivot element and the second list contains the remaining elements. These two lists are then dealt with recursively. Our treatment of this problem ignores the important fact that this processing can be done *in situ*.

Exercise 11.8 *Define the function* `less` *that takes two arguments,* `x` *and* `lst`*, and returns the elements of* `lst` *that are less than* `x` *(in the sense of* `<`*).*

Exercise 11.9 *Define the function* `notless` *that takes two arguments,* `x` *and* `lst`*, and returns the elements of* `lst` *that are not less than* `x` *(in the sense of* `<`*).*

Given the above definitions, we define the function `qsort` as follows.

```
(defun qsort (x)
  (cond ((atom x) nil)
        (t (append (qsort (less (car x) (cdr x)))
                   (list (car x))
                   (qsort (notless (car x) (cdr x)))))))
```

Exercise 11.10 *Prove* `(perm (qsort x) x)`.

Exercise 11.11 *Define the Boolean valued function* `lessp` *that takes two arguments,* `x` *and* `lst`*, and returns* `t` *iff every element of* `lst` *is less than* `x` *(in the sense of* `<`*).*

Exercise 11.12 *Define the Boolean valued function* `notlessp` *that takes two arguments,* `x` *and* `lst`*, and returns* `t` *iff every element of* `lst` *is not less than* `x` *(in the sense of* `<`*).*

Exercise 11.13

- *Prove* `(defcong perm equal (lessp x lst) 2)`.
- *Prove* `(defcong perm equal (notlessp x lst) 2)`.

Exercise 11.14 *Prove* `(orderedp (qsort lst))`.

Exercise 4.15, on page 62, asked that you define the function `mergesort`. Before continuing, make sure that you have done the exercise.

Exercise 11.15 *Prove* `(orderedp (mergesort lst))`.

Exercise 11.16 *Prove* `(perm (mergesort lst) lst)`.

11.3 Compressed Lists

In this section, you will be asked to prove theorems about a function that *compresses* lists, *i.e.*, a function that removes adjacent duplicates.

Exercise 11.17 *Define a function to compress a list. Given a list of elements,* `compress` *returns the list with all adjacent duplicates removed,* e.g., `(compress '(x x x y z y x y y))` *is equal to* `'(x y z y x y)`.

Exercise 11.18 *Prove the following.*

```
(equal (compress (compress x)) (compress x))
```

Exercise 11.19 *Prove the following.*

```
(equal (compress (append (compress x) y))
       (compress (append x y))
```

Exercise 11.20 *Recall the recognizer* `orderedp` *for ordered lists, defined on page 192. An exercise on page 58 asked for the definition of a recognizer* `no-dupls-p` *for duplicate-free lists. Formulate and prove a theorem stating that the application of* `compress` *to an ordered list is duplicate-free. You may need an additional hypothesis.*

Exercise 11.21 *Define* `same-compress`*, a Boolean function of two arguments that returns* `t` *iff* `compress` *applied to one of the arguments is* `equal` *to* `compress` *applied to the other.*

Exercise 11.22 *Prove* `(defequiv same-compress)`.

Exercise 11.23 *Prove the following.*

```
(defcong same-compress same-compress (append x y) 2).
```

Exercise 11.24 *Prove the following.*

```
(defcong same-compress same-compress (append x y) 1).
```

Exercise 11.25 *Prove the following.*

```
(equal (rev (compress x))
       (compress (rev x)))
```

The function `rev` *reverses a list and was defined on page 124.*

11.4 Summations

In this section, we present a sequence of equations containing summations and ask that you formalize the summations in ACL2 and determine whether or not they hold. Note that $\sum_{i=1}^{n} f(i) = f(1) + \cdots + f(n)$. Try doing these exercises in the `ground-zero` theory, *i.e.*, in a new ACL2 session. Afterwards, do the exercises once more, but use one of the arithmetic books that comes with the ACL2 distribution; `top-with-meta` is one such book.

Exercise 11.26 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.27 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} (3i^2 - 3i + 1) = n^3$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.28 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.29 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n}(2i)^2 = \frac{2n(n+1)(2n+1)}{3}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.30 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.31 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.32 *Formalize the following in ACL2.*

$$\sum_{i=1}^{n}(4i-1) = n(2n+1)$$

Is it a theorem? If so, prove it; if not, give a counterexample.

Exercise 11.33 *Formalize the following in ACL2.*

$$\left(\sum_{i=1}^{n} i\right)^2 = \sum_{i=1}^{n} i^3$$

Is it a theorem? If so, prove it; if not, give a counterexample.

11.5 Tautology Checking

In this section, we define a notion of `if`-expressionand prove the correctness of a simple tautology checker for such expressions.

The following function recognizes expressions whose top function symbol is `if`.

```
(defun ifp (x)
  (and (consp x)
       (equal (car x) 'if)))
```

The following functions extract the test, the true branch, and the false branch of an `if`.

```
(defun test (x)
  (second x))

(defun tb (x)
  (third x))

(defun fb (x)
  (fourth x))
```

An expression whose only function symbol is `if` is called an `if`-expression. For example, `(if (if a b c) d e)` is an `if`-expression, but `(if (foo (if a b c)) d e)` is not.

Exercise 11.34 *Define the function* `if-exprp` *to recognize* `if`*-expressions.*

An `if`-expression is *normalized* if no `if` subexpression contains an `if` in its test. Here is a naive attempt at normalizing such expressions.

```
(defun if-n (x)
  (if (ifp x)
      (let ((test (test x))
            (tb (tb x))
            (fb (fb x)))
        (if (ifp test)
            (if-n (list 'if (test test)
                        (list 'if (tb test) tb fb)
                        (list 'if (fb test) tb fb)))
          (list 'if test (if-n tb) (if-n fb))))
    x))
```

Exercise 11.35 *Add the above definition in* :program *mode and execute it on several examples,* e.g., *try* `(if-n '(if (if (if a b c) d e) e b))`.

Notice that

`(if-n '(if (if` α β γ`)` δ ϵ`))`

expands to

`(if-n '(if` α `(if` β δ ϵ`) (if` γ δ ϵ`))).`

Hence, the termination argument requires some thought.

Exercise 11.36 *Admit* `if-n` *in* :logic *mode.*

Exercise 11.37 *Admit* `if-n` *with a natural-number-valued measure function. (We suspect that the measure function you used for the previous exercise returned ordinals past the naturals.)*

Exercise 11.38 *Define* `(peval x a)` *to determine the value of an* `if`*-expression under the alist* `a`*. For example, if* `x` *is* `(if (if t c b) c b)` *and* `a` *is* `((c . t) (b . nil))`*, then* `(peval x a)` *is* `t`*. Notice that* `t` *and* `nil` *retain their status Boolean constants.*

Exercise 11.39 *Define* `(tautp x)` *to recognize* tautologies*:* `if`*-expressions that evaluate to* `t` *under all alists. (*Hint*: Consider normalizing the* `if`*-expressionand then exploring all paths through it.)*

Exercise 11.40 *Prove that* `tautp` *is sound: when* `tautp` *returns* `t`*, its argument evaluates to non-*`nil` *under every alist.*

Exercise 11.41 *Prove that* `tautp` *is complete: when* `tautp` *returns* `nil`*, there is some alist under which the* `if`*-expression evaluates to* `nil`*.*

11.6 Encapsulation

In this section, we will use encapsulation and functional instantiation to prove the equivalence of two functions that apply an associative and commutative function to a list of objects. We will use similar techniques to prove an important theorem about permutations.

Exercise 11.42 *Use* `encapsulate` *to introduce the function* `ac` *which is constrained to be associative and commutative. (See also Section 10.2, page 187.)*

The following function applies `ac` to a list of elements.

```
(defun map-ac (lst)
  (cond ((endp lst) nil)
        ((endp (cdr lst)) (car lst))
        (t (ac (car lst) (map-ac (cdr lst))))))
```

`Map-act`, defined below, is similar to `map-ac`, but it is tail recursive.

```
(defun map-act-aux (lst a)
  (cond ((endp lst) a)
        (t (map-act-aux (cdr lst) (ac (car lst) a)))))

(defun map-act (lst)
  (cond ((endp lst) nil)
        (t (map-act-aux (cdr lst) (car lst)))))
```

Notice that we used a helper function to define `map-act`.

Exercise 11.43 *Prove* `(equal (map-act lst) (map-ac lst))`*.* (Hint*: You may find the macro* `commutativity-2`*, defined on page 191, useful.)*

Consider the following function that returns the maximum of two numbers.

```
(defun maxm (a b)
  (if (< a b)
      (fix b)
    (fix a)))
```

Exercise 11.44 *Prove that* `maxm` *is associative and commutative.*

Exercise 11.45 *Define the functions* `map-maxm` *and* `map-maxmt` *to apply* `maxm` *to a list.* `Map-maxm` *and* `map-maxmt` *correspond to* `map-ac` *and* `map-act`, *respectively.*

Exercise 11.46 *Use functional instantiation (see* `lemma-instance`*) in order to prove the following.*

```
(equal (map-maxmt lst) (map-maxm lst))
```

Notice that we can use the above approach to prove that the tail-recursive version of any function applying an associative and commutative function to a list is equal to the simpler version of the function. Rewriting complicated functions into simpler functions and reasoning about the simpler functions is an example of *compositional reasoning.* Such decomposition replaces problems with manageable pieces.

11.7 Permutation Revisited

Recall the function of `mergesort` (page 62). In this section we revisit the proof that `mergesort` returns a permutation of its input, which you were asked to prove in Exercise 11.16. But this time we are less interested in `mergesort` *per se* than in developing a "new" way to prove theorems about `perm`.

Exercise 11.47 *Define the function* `how-many` *so that* `(how-many e x)` *determines how many times* `e` *occurs as an element of the list* `x`.

Exercise 11.48 *Prove the following.*

```
(equal (how-many e (mergesort lst))
       (how-many e lst))
```

Many people would consider the theorem in Exercise 11.48 to be equivalent to `(perm (mergesort lst) lst)`. Indeed, in a suitable logic permitting ACL2 terms and quantification,

```
 (perm (mergesort lst) lst)
↔
 (∀ e [(how-many e (mergesort lst)) = (how-many e lst)] )
```

is a theorem.

Exercise 11.49 *The universal quantifier in the theorem above is crucial. The similar-looking formula*

```
(iff (perm (mergesort lst) lst)
     (equal (how-many e (mergesort lst))
            (how-many e lst)))
```

is not a theorem. Construct a counterexample.

Exercise 11.50 *Find a general way to use the theorem proved in Exercise 11.48 to prove* `(perm (mergesort lst) lst)`. *If you wish, you may add* `true-listp` *hypotheses to make the problem easier.*

You may not want to do Exercise 11.50 now. But you may someday find yourself struggling to prove theorems about `perm` and you should remember this: it is possible to convert permutation problems into `how-many` problems, which are often easier to solve. Contrast your solutions of Exercises 11.16 and 11.48.

We offer several solutions to Exercise 11.50 on the Web page. In one, we constrain two constants, `(alpha)` and `(beta)`, to have the property `(equal (how-many e (alpha)) (how-many e (beta)))` and then prove `(perm (alpha) (beta))`. In our proof, we define `(bounded-quantifierp x a b)` to check, for each element `e` in `x`, that `(how-many e a)` is equal to `(how-many e b)`. We then relate `bounded-quantifierp` to `perm`.

The next exercise is valuable even if you do not tackle Exercise 11.50 now. It will teach you something very important about functional instantiation.

Exercise 11.51 *How can the theorem* `(perm (alpha) (beta))`, *above, be used to prove* `(perm (mergesort lst) lst)`? *To work on this problem, first constrain* `(alpha)` *and* `(beta)` *as described above. Then pretend you proved the* `perm` *theorem by executing the following.*

```
(skip-proofs
 (defthm perm-alpha-beta
   (perm (alpha) (beta))))
```

Now prove the theorem `(perm (mergesort lst) lst)` *by functional instantiation. (*Hint: *This is easy once you see the power of functional instantiation.)*

In another solution to Exercise 11.50 we use a common "trick" in dealing with quantification in this setting: we define a function that exhibits a "bad guy," *i.e.*, a function that finds an `e` that occurs a different number of times in `a` than in `b` when `(perm a b)` is false.

11.8 The Extractor Problem

The following function builds a list containing the first n natural numbers, in reverse order.

```
(defun nats (n)
  (if (zp n)
      nil
    (cons (- n 1) (nats (- n 1)))))
```

The following function builds a list by recurring on map and consing the n^{th} element of lst to the result, where n is the car of map.

```
(defun xtr (map lst)
  (if (endp map)
      nil
    (cons (nth (car map) lst)
          (xtr (cdr map) lst))))
```

Exercise 11.52 *Prove* `(equal (xtr (nats (len x)) x) (rev x))`.

11.9 Finite Set Theory

We have seen that ACL2 has built-in functions to manipulate sets. Examples of such functions are member and subsetp. These functions assume a *flat* representation of sets. For example, `(subsetp '(1 2) '(2 1))` is t, but `(subsetp '((1 2)) '((2 1)))` is nil. The first expression corresponds to $\{1,2\} \subseteq \{2,1\}$, which in set theory is true, but the second expression can be viewed as corresponding to $\{\{1,2\}\} \subseteq \{\{2,1\}\}$, which in set theory is also true. Below, you will be asked to define *general* finite set theory functions, *i.e.*, ACL2 functions that do not assume sets are flat; for example, they consider `((1 2))` to be a subset of `((2 1))`. This exercise may be harder than it seems. Part of the problem is getting the definitions right and we are intentionally leaving some ambiguity in the next exercise so that you can explore various possibilities.

Exercise 11.53 *Define the functions* in, =<, *and* == *that correspond to set membership, subset, and set equality, respectively. These functions should be general set theory functions, as discussed above. (*Hint: *You may find it useful to use* mutual-recursion.*)*

Exercise 11.54 *Test your functions above on the following examples.*

1. `(== '((1 2)) '((2 1)))`
2. `(=< '((2 1) (1 2)) '((2 1)))`

3. `(in '((1)) '((2 (1)) (1 2)))`

4. `(== '(() (1 2 1) (2 1) x) '((1 2) x ()))`

5. `(== 'x 1)`

6. `(=< 'x 1)`

Exercise 11.55 *Prove* `(=< X X)`.

Exercise 11.56 *Prove the following.*
`(implies (and (=< X Y) (=< Y Z)) (=< X Z))`

Exercise 11.57 *Prove* `(defequiv ==)`.