Mechanically Verifying Real-Valued Algorithms in ACL2

by

Ruben Antonio Gamboa, B.S., M.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 1999

Mechanically Verifying Real-Valued Algorithms in ACL2

Approved by
Dissertation Committee:

This work is dedicated to my daughter, who inspired me to finish,

and to my wife, who would not let me quit.

Acknowledgments

The members of my committee have all contributed greatly to my dissertation, either directly with comments and ideas, or indirectly by providing me with the necessary background. Special thanks are due to Robert Boyer and Matt Kaufmann, the former for sticking with me through the early years of learning ACL2 and choosing a topic, and the latter for serving as a bountiful source of ideas and enthusiasm.

Many other people deserve recognition. Among them are my family for believing in me and supporting me, and my best friend and boss for giving me the time away from work to indulge my academic hobby.

RUBEN ANTONIO GAMBOA

The University of Texas at Austin
May 1999

Mechanically Verifying Real-Valued Algorithms in ACL2

Publication No. ____________________

Ruben Antonio Gamboa, Ph.D.
The University of Texas at Austin, 1999

Supervisor: Robert S. Boyer

ACL2 is a theorem prover over a total, first-order, mostly quantifier-free logic, supporting defined and constrained functions, equality and congruence rewriting, induction, and other reasoning techniques. With its powerful induction engine and direct support for the rational and complex-rational numbers, ACL2 can be used to verify recursive rational algorithms. However, ACL2 can not be used to reason about real-valued algorithms that involve the irrational numbers. For example, there are elegant proofs of the correctness of the Fast Fourier Transform (FFT) which could be formulated in ACL2, but since ACL2's sparse number system permits the proof that the square root of two, and hence the eighth principal root of one, does not exist, it is impossible to reason directly about the FFT in ACL2.

This research extends ACL2 to allow reasoning about the real and complex irrational numbers. The modifications are based on non-standard analysis, since infinitesimals are more natural than limits in a quantifier-free context. It is also shown how the trigonometric functions can be defined in the modified ACL2. These definitions are then used to prove that the FFT correctly implements the Fourier Transform.

Contents

List of Tables

Chapter 1

Introduction

ACL2 is a total, first-order, mostly quantifier-free logic based on the programming language Common LISP. ACL2 is also an automated theorem prover for this logic. The theorem prover excels in using induction to prove theorems about recursive functions. At its heart is a rewriter, which uses a database of previously proved theorems to transform a term while maintaining an appropriate equivalence relation, e.g., equality or if-and-only-if. In addition, it supports many other inference mechanisms. For instance, numeric inequalities are automatically verified using a linear arithmetic decision procedure, and propositional tautologies can be proved using a decision procedure based on binary decision diagrams (BDDs). Moreover, ACL2 supports the introduction of constrained functions, allowing a limited amount of higher-order reasoning. ACL2 is a direct descendant of Nqthm, also known as the Boyer-Moore theorem prover. It is fair to say that ACL2 grew out of a desire to "do Nqthm, only better" [40, 12].

Among the enhancements of ACL2 over Nqthm is a richer number system. Nqthm had some native support for the integers, and it was primarily designed for working with the naturals. ACL2, on the other hand, introduced the rational and the complex-rational numbers. However, the irrationals were deliberately excluded from the ACL2 number system. This exclusion stems from the desire to keep ACL2 as close to Common LISP as possible, and Common LISP does not include the irrationals. Since the primary goal of ACL2 is to prove properties about Common LISP functions, it makes sense to exclude objects that do not exist in the Common LISP universe.

However, the omission of the reals places an artificial limit on the theories that ACL2 can verify. For example, an important application of ACL2 is the proof of correctness of various micro-processor algorithms, such as floating-point arithmetic operations, roots, and other transcendental functions. In [54], Russinoff uses ACL2 to prove the correctness of the square root floating-point microcode in the AMD K5 processor. However, since the number $\sqrt{x}$ is possibly irrational for a given rational x, his theory "meticulously avoids any reference" to $\sqrt{x}$. In particular, the correctness theorem has the (simplified) form

$$l^2 \leq x \leq h^2 \Rightarrow rnd(l) \leq sqrt(x) \leq rnd(h)$$

where rnd rounds a number to its nearest floating-point representation and $sqrt$ is the AMD K5 microcode algorithm for finding floating-point square roots. This theorem is equivalent to the desired statement of correctness:

$$sqrt(x) = rnd(\sqrt{x})$$

However, this equivalence can not be stated in ACL2.

The difference between these two theorems is more than a matter of aesthetics. Consider the norm given by $||x|| = \sqrt{Re^2(x) + Im^2(x)}$, where $Re(x)$ and $Im(x)$ are the real and imaginary parts of x, respectively. An algorithm may require the value $||x \cdot y||$, but it may compute it using $||x|| \cdot ||y||$ because those values are already known. The results are mathematically equivalent, even if it is false when the approximation $sqrt(x)$ is used instead of the function $\sqrt{x}$.

This is a general phenomenon. The correctness of many algorithms depend on properties that are true of real-valued functions, even though the algorithms are implemented entirely in terms of rational or floating-point values. That is, an algorithm may compute the value of the function f at a point x not directly by approximating the equation defining

$f(x)$, but indirectly by approximating the equation defining $g(x)$, where it can be proved that the functions f and g are equal. In the example above, the equation $f(x, y)$ would suggest multiplying x and y and taking the norm of the product, while $g(x, y)$ multiplies the norms x and y. The key fact is that in order to prove the correctness of the algorithm following the defining equation $g(x)$, it may be necessary to prove that $f(x) = g(x)$ and such arguments are often impossible without using the language of irrational numbers and irrational functions.

Note, it is quite possible that the defining equation for $g(x)$ is more amenable for approximation using floating-point arithmetic than $f(x)$. This happens when the computation of $f(x)$ accumulates errors, whereas the errors tend to cancel out during the computation of $g(x)$. In the language of numerical analysis, $g(x)$ is called stable and $f(x)$ unstable. For example, stability is the reason pivoting is important when finding the inverse of a matrix, even though the algorithms with and without pivoting are mathematically equivalent [25]. When the equation for $f(x)$ is unstable and that of $g(x)$ stable, the floating-point implementation of $f(x)$ may be very different than that of $g(x)$; ironically, the approximation to $g(x)$ may be closer to the true, possibly irrational, value $f(x)$.

This thesis describes an extension to ACL2 so that it can reason about the irrational real and complex numbers. The extension rests on the theory of non-standard analysis, first proposed by Robinson [52] and later advanced by Nelson [49] among others. Non-standard analysis provides a natural mechanism to reason about the irrationals in ACL2 for two reasons. First, many of the arguments in traditional analysis are simplified in non-standard analysis by the replacement of functions with numbers. For example, to prove that a function f is continuous in standard analysis, it is necessary to find a function mapping a positive ϵ to a positive δ with certain properties. However, in non-standard analysis it is only necessary to show that $f(x) - f(x - \epsilon)$ — a *number* — is "sufficiently small." In ACL2, the proof using non-standard analysis will generally be much easier. The second reason why non-standard analysis fits well with ACL2 is that many arguments in non-standard analysis are proved using induction. For example, using standard analysis, the intermediate value theorem is proved by looking at least upper bounds of sets. In contrast, the non-standard analysis proof proceeds by using induction on a step-function approximating the continuous function in question. This plays to the strength of ACL2 and avoids ACL2's weaknesses with quantification, leading to the natural proof of the result, as presented in section 6.1.3.

Using the techniques of non-standard analysis, it will also be shown how some transcendental functions can be defined in ACL2, such as the square root, exponential, and trigonometric functions. Many of the classical results about these functions will also be verified in ACL2, including Euler's beautiful equation $e^{i\pi} = -1$.

To demonstrate how these results about abstract mathematical functions are relevant to the mechanical verification of real-valued algorithms, this thesis will also present a proof in ACL2 of the correctness of the Fast Fourier Transform (FFT). This proof also illustrates an important point about ACL2. The proof is based on the notation of powerlists, introduced by Misra in [46] to express and reason about data-parallel, recursive algorithms. ACL2 is sufficiently powerful that it can embed the theory of powerlists and reason effectively about them. Such an embedding is presented in chapter 7, where many difficult theorems from [46] are mechanically verified.

There is a synergy between non-standard analysis and ACL2. That is why ACL2 is an ideal medium for mechanically verifying real-valued algorithms. ACL2 is expressive enough to describe complex algorithms, and it boasts a theorem prover powerful enough to prove difficult theorems about such algorithms. Witness, for example, the theorems about floating-point algorithms proved in [47, 54, 55]. The addition of the irrationals and the principles of non-standard analysis makes it powerful enough to define the common transcendental functions and to prove their fundamental properties. It opens a new domain to ACL2, the verification of real-valued algorithms. It is because of the traditional strengths of ACL2 — term rewriting, linear arithmetic, induction, etc. — that it succeeds in this new domain.

1.1 Related Work

There is a long history of real analysis in automated theorem proving, some projects using analysis as a test-bed for theorem provers, others pursuing analysis for more pragmatic reasons, for example, because the correctness of an algorithm hinges on a property of continuous functions.

In his wonderful dissertation [28], Harrison showed how the real numbers can be formalized in the theorem prover HOL [26]. Harrison's approach was to construct the real numbers, rather than to introduce them axiomatically. Real numbers are defined as equivalence classes of Cauchy, rational sequences. All their properties are then developed from first principles.

His approach differs dramatically from the one presented in this thesis. HOL is a higher-order theorem prover, allowing a natural vehicle for reasoning about the equivalence classes of functions. In contrast, ACL2 does not provide sufficiently strong set-theoretic axioms for this task. Moreover, Harrison's primary interest was in developing a mechanized theory of analysis, whereas this thesis is interested in developing the pragmatic subset that is needed for the verification of real-valued algorithms.

The theorem prover PVS offers built-in support for the real numbers [50]. In PVS, the reals are axiomatized using the usual field and ordering axioms, as well as a version of the completeness axiom. PVS makes similar tradeoffs as ACL2, emphasizing issues of pragmatics and usability. As with ACL2, a large portion of the reasoning engine is encoded not in axioms, but in decision procedures built into the theorem prover. This includes decision procedures for theories about the reals, such as linear rewriting. In [21], Dutertre describes a theory of simple mathematical analysis developed using PVS. This theory presents basic notions of analysis, such as sequences, convergence, continuity, and differentiability. Dutertre's motivation in developing his theory of the reals echoes the current motivation in extending ACL2 to include the reals. Rather than focusing on analysis per se, the goal is to prove enough results from analysis to reason about certain algorithms or computer systems.

IMPS offers axiomatic support for the reals [22]. IMPS has a partial and higher-order logic, which supports a proof style close to the one used by working mathematicians. Moreover, IMPS allows separate pieces of the theory to be developed in isolation, the so-called "little-theory approach." Separate theories can be related through theory interpretations, which allow the reuse of theorems in different settings. Real and abstract analysis have been a focus throughout the development of IMPS. Among the many theorems proved are the Bolzano-Weierstrass theorem, some fixpoint theorems similar to Banach's, and the mean value theorem for integration. Proofs in IMPS are developed by a sequence of goal transformations. The user is directly responsible for guiding IMPS through the "high-level" inferences, while IMPS takes care of the "low-level" computational details.

MIZAR also contains axiomatic support for the reals [53, 57]. The emphasis in MIZAR is on the development of a syntax suitable for the formalization of large portions of mathematics. The MIZAR system, available on PCs, encourages the development of large theories which depend on previously proved theorems. It allows the user to define new syntax and parsing rules, to accommodate new notions. On the other hand, MIZAR provides a relatively weak reasoning engine, relying on the user to provide explicit proofs, including references to previous theorems.

Bledsoe developed several theorem provers that were able to prove many results from elementary analysis, such as the intermediate value theorem [8, 9]. Hines and Bledsoe also pursued inference mechanisms that would be useful in an analysis theorem prover, such as the inequality prover STRIVE [30]. Ballantyne and Bledsoe describe a version of Bledsoe's theorem prover IMPLY which proves theorems in the theory of non-standard analysis [3, 2]. Using this system, they were able to prove several theorems of elementary analysis, including the equivalence of the "standard" and "non-standard" definitions of the basic analysis concepts. For example, they present a mechanical proof that the traditional definition of sequence convergence is equivalent to the non-standard version. These results are impressive.

More recently, Fleuriot and Paulson used non-standard analysis to formalize an infinitesimal geometry based on Newton's Principia [23]. Using the Isabelle theorem prover, they were able to prove many of the theorems in the Principia using Newton's original arguments.

A different approach to theorem proving over the reals is to use a computer algebra system, such as Macsyma or Mathematica. Such an approach is suggested by Clarke and Zhao, whose theorem prover Analytica is coded in Mathematica [14]. Analytica uses the Mathematica rewriting engine, which makes minor concesions to soundness in favor of utility. Analytica compensates for this potential unsoundness by checking certain steps in Mathematica's computation and avoiding the more common errors, such as division by an expression which can be equal to zero. The approach seems promising, in that Analytica is able to prove an impressive array of theorems, including key steps in the construction of a continuous, but differentiable nowhere function. Future work on Analytica may entail a closer coupling between the theorem prover and computer algebra system, so that each simplification performed by the computer algebra system can be rigorously justified.

Another approach based on computer algebra is given by Beeson, whose MATHPERT system is designed for pedagogical use [5]. MATHPERT is capable of solving elementary algebra, trigonometry, and calculus problems. The problems are solved by applying a sequence of operators, elsewhere referred to as rewrite rules. Since the primary interest of MATHPERT is in pedagogy, extreme care has been spent in choosing the appropriate operators, so that they correspond, roughly, to "nuggets" of knowledge that a student should acquire. Moreover, the way in which these operators are applied is also given great consideration, so that the results mimic the way students would tackle the problem. The end result is that MATHPERT provides more than just a solution to a given problem. It provides an intelligible solution from which students can learn and profit. Of particular interest is that MATHPERT uses non-standard analysis internally to formally justify that the answer returned is correct. While these details are kept from the student, non-standard analysis is used, for example, to verify that terms inside a limit are well-defined [6].

Cowles has proved the irrationality of the square root of two in ACL2. In fact, he has formalized several versions of this proof. He has also proved some properties of McCarthy's "91 function." These proofs were originally done in Nqthm in the context of integers and Archimedean ordered fields. They have also been proved in ACL2 in the context of Archimedean ordered fields, but this foregoes ACL2's special treatment of the numbers, such as linear rewriting. The modifications to ACL2 described in this thesis will probably benefit his research [17, 16, 18].

Others have tried using a rewrite-based theorem prover with support for induction to reason mechanically about powerlists. Kapur and Subramaniam have used the theorem prover RRL to verify some of the powerlist theorems proved by Misra [33, 32, 35]. In addition, they have used this foundation to prove the correctness of arithmetic circuits [34].

1.2 Outline

There are two main products of this thesis. The first is a modified version of ACL2 that can reason about non-standard analysis. The second is a set of theories verified with the modified ACL2. These theories can be used by others who want to reason mechanically about real-valued algorithms. The complete source tree of the modified version of ACL2 as well as the source files describing the theories described in this thesis can be found in the accompanying CD-ROM. The latest versions of these can also be found on the web at `http://www.lim.com/~ruben/research/thesis`.

These two products are described in this thesis as follows. In chapter 2, it is shown that the trascendental functions can not be introduced into ACL2. This is demonstrated by proving that the fundamental property of $\sqrt{x}$ — i.e., $x \geq 0 \Rightarrow \sqrt{x} \cdot \sqrt{x} = x$ — can be disproved by ACL2. However, it is also shown that ACL2 can define rational approximations that are arbitrarily close to the square root function.

In chapter 3, the basic principles of non-standard analysis are introduced. Moreover, the modifications to ACL2 necessary to include non-standard analysis are described. This chapter also offers a proof that the modifications are sound. Particularly interesting is how induction needs to be modified to support non-standard analysis. The chapter presents a soundness proof for the modified induction principle.

Chapter 4 uses the rational approximation scheme for $\sqrt{x}$ described in chapter 2 to define the $\sqrt{x}$ function in ACL2, using the non-standard modifications to ACL2 described in chapter 3. The basic properties of $\sqrt{x}$ are also verified in ACL2.

The exponential function is introduced in chapter 5. This is defined in terms of the Taylor expansion for e^x, and it requires the development of a significant theory of converging series in the complex plane, including geometric series and the comparison test of absolutely convergent series. Comparisons on the complex numbers are based on a norm defined using the square root function introduced in chapter 4. This chapter also proves a fundamental property of the exponential function, namely that $e^{x+y} = e^x \cdot e^y$. This difficult proof depends on many facts about the binomial function and the theory of nested sums. The property plays a key role in showing that the exponential function is continuous.

In chapter 6, the trigonometric functions are defined in terms of the exponential function. Particularly interesting is the definition of the constant π, as it depends on the continuity of the cosine function, the theory of alternating series, and the intermediate value theorem. This chapter also develops a comprehensive theory of trigonometry, finding explicit values of the trigonometric functions at the common angles, as well as the sign of the trigonometric functions in the four quadrants. Moreover, it verifies many of the familiar identities from trigonometry.

Chapter 7 develops the theory of powerlists in ACL2. It also proves generalizations of many of the theorems presented by Misra in [46]. The proof of the correctness of Batcher sorting deserves notice. Also proved in this chapter are the correctness of some parallel prefix sum algorithms, as well as a carry-lookahead adder.

Chapter 8 uses the trigonometric functions defined in chapter 6 and the theorems about powerlists proved in chapter 7 to prove the correctness of the Fast Fourier Transform. This illustrates how the traditional strengths of ACL2, represented by the reasoning about powerlists, combine with the enhancements based on non-standard analysis to prove theorems about real-valued algorithms.

The syntax of ACL2 will be used throughout this dissertation. This syntax is essentially that of Common LISP, with a few primitives defined to support theorem proving, such as the function `defthm` which introduces a new theorem. This choice of syntax may frustrate some readers who are unfamiliar with ACL2 or any LISP dialect. Those readers who feel uncomfortable with the notation may wish to read appendix A, which presents a gentle introduction to ACL2 and its syntax.

In the following chapters, only the main theorems and lemmas are included. The remaining lemmas, which are required to guide ACL2 towards the eventual proof, are omitted for the sake of brevity and clarity. For the same reason, the ACL2 statements shown have been stripped of the ACL2-specific annotations, such as hints, guards, and rule-classes. The complete ACL2 input can be found elsewhere on the CD-ROM, as well as from `http://www.lim.com/˜ruben/research/thesis`.

Chapter 2

Learning from the Square Root Function

This chapter illustrates how the lack of irrational numbers in ACL2 can lead to surprising results. In particular, it is shown that ACL2 can prove that $x \cdot x$ is not equal to 2 for any value of x. This result is more limiting than it appears. It demonstrates that the introduction of the square root function into ACL2 — through the addition of an axiom — would result in a contradictory theory. In [54] Russinoff uses ACL2 to prove the correctness of the AMD K5 square root implementation. However, he observes the lack of the square root function in ACL2 prevented him from mechanically proving the square root microcode against the precise IEEE specification.

The results in this chapter have been verified in ACL2 versions 1.8 through 2.1. Chapter 3 will introduce ACL2 v2.1(r), which supports the irrationals and therefore fails to prove some of the theorems presented in this chapter. Although the explicit version number will not be used, it should be clear that "ACL2" refers to ACL2 v2.1 and not to ACL2 v2.1(r) for the remainder of this chapter.

2.1 The Non-Existence of the Square Root of Two

To prove that

```
(not (equal (* x x) 2))
```

is a theorem in ACL2, it is sufficient to rule out suitable candidates for `x`. The first step is the most interesting from a mathematical viewpoint – no rational number satisfies $x \cdot x = 2$:

```
(implies (rationalp x)
         (not (equal (* x x) 2)))
```

The proof follows the classic argument of the irrationality of $\sqrt{2}$. After ruling out the rationals, the proof is nearly complete. The complex rationals can be ruled out, since all their squares are either complex or negative. Since all other objects (i.e., non-numbers) in ACL2 have zero squares, that will complete the proof.

Begin by considering the rationals and showing that none of them can be equal to $\sqrt{2}$. Suppose for now that $\sqrt{2}$ is rational. Then, for some relatively prime integers p and q, $(\frac{p}{q})^2 = 2$. This implies that $p^2 = 2q^2$, so p^2 is even. But since p is an integer, this implies that p must be even as well. That is, there is some integer p' with $p = 2p'$. Then it follows that $4p'^2 = 2q^2$, and hence q^2 is even. Again, this implies that q is even. But then, p and q are not relatively prime as claimed, and therefore $\sqrt{2}$ can not be rational.

The first step in formalizing this argument is to prove the following lemmas:

```
(defthm even-square-implies-even
  (implies (and (integerp p)
                (divisible (* p p) 2))
           (divisible p 2)))

(defthm even-implies-square-multiple-of-4
  (implies (and (integerp p)
                (divisible p 2))
           (divisible (* p p) 4)))
```

Given the equation $p^2 = 2q^2$, the lemma `even-square-implies-even` establishes that p is even, and `even-implies-square-multiple-of-4` that q^2 is even. Applying `even-square-implies-even` again, it follows that q is even. That is, ACL2 can prove the following lemmas:

```
    (defthm sqrt-lemma-1.1
      (implies (and (integerp p)
                    (integerp q)
                    (equal (* p p) (* 2 (* q q))))
               (divisible q 2)))

    (defthm sqrt-lemma-1.2
      (implies (and (integerp p)
                    (integerp q)
                    (equal (* p p) (* 2 (* q q))))
               (divisible p 2)))
```

To complete the argument, only the key property that p and q are relatively prime is needed. Equivalently, $\frac{p}{q}$ must be expressed in lowest terms. The ACL2 functions `numerator` and `denominator` can be used to express an arbitrary rational number in lowest terms. Using these functions converts the lemmas above into the following:

```
    (defthm sqrt-lemma-1.3
      (implies (and (rationalp x)
                    (equal (* x x) 2))
               (equal (* (numerator x) (numerator x))
                      (* 2 (* (denominator x)
                              (denominator x))))))
```

Combining all the results above yields the basic fact contradicting the rationality of $\sqrt{2}$:

```
(defthm sqrt-lemma-1.4
  (implies (and (rationalp x)
                (equal (* x x) 2))
           (and (divisible (numerator x) 2)
                (divisible (denominator x) 2))))


(defthm sqrt-lemma-1.5
  (implies (and (divisible (numerator x) 2)
                (divisible (denominator x) 2))
           (not (rationalp x))))


(defthm sqrt-2-is-not-rationalp
  (implies (rationalp x)
           (not (equal (* x x) 2))))
```

Having ruled out the rationals from the list of candidates for $\sqrt{2}$, it is only necessary to eliminate the remaining ACL2 objects, namely the complex rationals and non-numeric objects. This is much easier. A complex rational has the form $a + bi$, where $b \neq 0$ and a and b are rationals. None of these objects can be the square root of 2, because their squares all have the form $a^2 - b^2 + 2abi$, and for that to be equal to 2, a must be zero. But then, the square of bi is equal to $-b^2$, which is negative since b is rational. This argument can be easily verified in ACL2:

```
(defthm complex-squares-rational-iff-imaginary
  (implies (and (complex-rationalp x)
                (rationalp (* x x)))
           (equal (realpart x) 0)))

(defthm imaginary-squares-are-negative
  (implies (and (complex-rationalp x)
                (equal (realpart x) 0))
           (< (* x x) 0)))
```

From these theorems, it is easy to rule out the complex rational numbers from the list of candidates:

```
    (defthm sqrt-2-is-not-complex-rationalp
      (implies (complex-rationalp x)
                (not (equal (* x x) 2))))
```

Since the ACL2 number system includes only the rational and complex rational numbers, this establishes that no number can be equal to $\sqrt{2}$ in the ACL2 universe. But by simple type analysis, ACL2 can verify that only numbers can have non-zero squares. Therefore, all ACL2 objects are ruled out. This establishes the following theorem:

```
    (defthm there-is-no-sqrt-2
      (not (equal (* x x) 2)))
```

As mentioned earlier, this theorem does more than simply rule out the possibility that some ACL2 object is equal to $\sqrt{2}$. It also explicitly rules out the possibility of introducing — by definition or otherwise — a function with the properties of the square root function. Similar arguments would rule out other irrational functions. This is especially sad when the power of ACL2 is considered. This power will become evident in the next section, where ACL2 demonstrates the existence of rational functions arbitrarily close to the square root function. That is, it is possible to define arbitrarily good approximations to the square root function and other irrational functions in ACL2.

2.2 Approximating the Square Root Function

The previous section showed that the fundamental theorem of square roots — $x \geq 0 \Rightarrow \sqrt{x} \cdot \sqrt{x} = x$ — is inconsistent with the axioms of ACL2. However, it is possible to prove weaker versions of this theorem. One such approach is to require that it only hold when both x and $\sqrt{x}$ are rational; at other points, no claims are made about the function, so it is free to take on any value, say zero. That such a function exists in the ACL2 logic is clear, since for rational p/q in least terms, $\sqrt{p/q}$ is given by $\sqrt{p}/\sqrt{q}$ and since p and q are relatively prime, this is rational if and only if $\sqrt{p}$ and $\sqrt{q}$ are integers. Unfortunately, the likelihood that an arbitrary integer has an integer square root is small, so this would only cover a small fraction of the rationals, and the modified theorem would be too weak.

A better alternative is to substitute closeness for strict equality. For example, require that $|\sqrt{x} \cdot \sqrt{x} - x| < \epsilon$ for some fixed $\epsilon > 0$. There are many different approximation schemes that can be used to come close to the square root function. Although other schemes offer better performance, the simplicity of the bisection algorithm makes it a promising approximation scheme in ACL2.

The convergence criterion is interesting, since the result $\sqrt{x}$ to which the approximation converges is not necessarily in the ACL2 universe, so it is not able to guarantee something similar to $|\hat{x} - \sqrt{x}| < \epsilon$ (cf. [54]). For this reason, $|\hat{x}^2 - x| < \epsilon$ will serve as the test of convergence.

An iterative approximation to the square root function can be defined as follows:

```
(defun iterate-sqrt-range (low high x num-iters)
  (if (<= (nfix num-iters) 0)
      (cons (rfix low) (rfix high))
    (let ((mid (/ (+ low high) 2)))
      (if (<= (* mid mid) x)
          (iterate-sqrt-range mid high x (1- num-iters))
        (iterate-sqrt-range low mid x
                                    (1- num-iters))))))
```

Because the convergence of `iterate-sqrt-range` is not obvious to ACL2 and ACL2 only accepts definitions when it can prove their termination on all inputs, it is convenient to divorce the convergence result from the terminating condition of the iterative function. In particular, `iterate-sqrt-range` accepts a non-empty range of real numbers specified by `low` and `high`, and it divides the range in half `num-iters` times. It returns the resulting `low-high` range, and subsequently the returned `low` will be used as the approximation to $\sqrt{x}$. At each split of the range, the parameter `x` is used to decide which half to keep and which to discard. For example, `(iterate-sqrt-range 0 2 2 1)` will return `(1 . 2)`, since after the first iteration $\sqrt{2}$ can be contained in this range. Letting the function iterate 5 times as in `(iterate-sqrt-range 0 2 2 5)` yields `(11/8 . 23/16)`. Notice the original range of width 2 has been reduced to one with width of 1/16, or 2^5 times smaller.

The proof of convergence can be split into two parts. First, if `num-iters` is large enough, the difference between the final `high` and `low` can be made arbitrarily small. Second, if the `high` and `low` are very close to each other, the square of `low` is very close to `x`.

Before proceeding, some basic properties of `iterate-sqrt-range` need to be established. For example, if the original `high-low` range is not vacuous, then neither is the final `high-low` range after iterating any number of times:

```
(defthm iterate-sqrt-range-reduces-range
  (implies (and (rationalp low)
                (rationalp high)
                (< low high))
           (< (car (iterate-sqrt-range low high x
                                       num-iters))
              (cdr (iterate-sqrt-range low high x
                                             num-iters)))))
```

A particularly crucial lemma is that the final `high` estimate is not larger than the initial one. That is, no iteration can increase the current upper estimate.

```
(defthm iterate-sqrt-range-non-increasing-upper-range
  (implies (and (rationalp low)
                (rationalp high)
                (< low high))
           (<= (cdr (iterate-sqrt-range low high x
                                        num-iters))
               high)))
```

Nonetheless, the final `high` estimate is large enough that it does not cross below the square root of `x`, so long as the initial `high` estimate is not below the square root of `x`:

```
(defthm iterate-sqrt-range-upper-sqrt-x
  (implies (and (rationalp low)
                (rationalp high)
                (rationalp x)
                (<= x (* high high)))
           (<= x
               (* (cdr (iterate-sqrt-range low high x
                                           num-iters))
                  (cdr (iterate-sqrt-range low high x
                                           num-iters))
               ))))
```

This provides a tight bound on how far the values of `high` can range. Similarly, the analogous theorems for the `low` bound of the range can be established.

With these lemmas, the continuity of x^2 is sufficient to show that if the `high-low` range is small enough, then the range of their squares is arbitrarily small. Specifically, for any $\epsilon > 0$ and $a > 0$, it is possible to find a δ such that for any b with $0 < b - a < \delta$, it follows that $b^2 - a^2 < \epsilon$. In fact, algebraic manipulation will show that this is true for any $\delta \le \epsilon/(a + b)$. Moreover, for ranges $[a, b]$ such that $a \le \sqrt{x} \le b$, the term b^2 can be replaced by the smaller x to conclude that $x - a^2 < \epsilon$. The continuity condition can be stated as follows:

```
(defthm sqrt-epsilon-delta-aux-4
  (implies (and (rationalp a)
                (rationalp b)
                (rationalp x)
                (rationalp epsilon)
                (<= 0 a)
                (< a b)
                (<= x (* b b))
                (< (- b a) delta)
                (<= delta (/ epsilon (+ b a))))
           (< (- x (* a a)) epsilon)))
```

Unfortunately, this result is stated in terms of `(+ b a)`, which will correspond to the *final* `high-low` estimates of the approximation. It would be more convenient to define δ in terms of the original estimates or guesses. Since the `high` estimates are monotonically decreasing and the `low` estimates are monotonically increasing, it is not possible to readily conclude anything about the sum of the final `high` and `low`. However, observe that the claim remains true for $\delta \le \epsilon/2b$, since for $0 \le a \le b$, $\epsilon/2b \le \epsilon/(a + b)$. Now, δ will only depend on the final `high` estimate, and since we know `high` is monotically decreasing we can replace the final `high` estimate with the initial guess. This is important, because it allows the number of iterations required to be computed before the square root approximation is begun. Combining these observations results in the first half of the convergence result:

```
(defthm iter-sqrt-epsilon-delta
  (implies (and (rationalp low)
                (rationalp high)
                (rationalp epsilon)
                (rationalp delta)
                (rationalp x)
                (< 0 epsilon)
                (<= 0 low)
                (< low high)
                (<= x (* high high))
                (<= delta (/ epsilon (+ high high))))
           (let ((range (iterate-sqrt-range low high x
                                                num-iters)))
             (implies (< (- (cdr range) (car range))
                         delta)
                      (< (- x
                            (* (car range) (car range)))
                         epsilon)))))
```

It remains only to be shown that the final `high-low` estimate will be sufficiently close together so that the theorem above can apply, as long as enough iterations are performed. It is possible to define a function `guess-num-iters` that computes the required number of iterations for a specific x and ϵ. Since the iteration scheme halves the estimate range at each step, only the size of the initial estimate is needed. The function, which essentially computes the base-2 logarithm of the initial range, is given below:

```
(defun guess-num-iters-aux (range num-iters)
  (if (and (integerp range)
           (integerp num-iters)
           (> num-iters 0)
           (> range (2-to-the-n num-iters)))
      (guess-num-iters-aux range (1+ num-iters))
    (1+ (nfix num-iters))))
(defmacro guess-num-iters (range delta)
   ‘(guess-num-iters-aux (ceiling ,range ,delta) 1))
```

The function `2-to-the-n` returns 2^n for non-negative integer n; its definition is omitted in favor of brevity. Before proving that `guess-num-iters` returns a sufficiently large value for any choice of `range` and `epsilon`, it is important to consider how `iterate-sqrt-range` reduces the `high-low` range after a number of iterations. The following theorem proves that the range is halved at each step:

```
(defthm iterate-sqrt-reduces-range-size
  (implies (and (<= (* low low) x)
                (<= x (* high high))
                (rationalp low)
                (rationalp high)
                (integerp num-iters))
           (let ((range (iterate-sqrt-range low high x
                                                    num-iters)))
             (equal (- (cdr range) (car range))
                    (/ (- high low)
                       (2-to-the-n num-iters))))))
```

With this result and some algebraic rewriting, the second half of the convergence theorem can be proved. Specifically, it can be shown that by iterating `guess-num-iters` times the final `high-low` range is sufficiently small for the first convergence theorem to apply:

```
(defthm iterate-sqrt-range-reduces-range-size-to-delta
  (implies (and (rationalp high)
                (rationalp low)
                (rationalp delta)
                (< 0 delta)
                (< low high)
                (<= (* low low) x)
                (<= x (* high high)))
           (let ((range (iterate-sqrt-range
                         low
                         high
                         x
                         (guess-num-iters (- high low)
                                          delta))))
             (< (- (cdr range) (car range)) delta))))
```

The only remaining task is choosing appropriate starting values for `high` and `low`. Given an $x > 0$, an initial range containing $\sqrt{x}$ can be $[0, x]$ if $x > 1$, and $[0, 1]$ otherwise. It is clear that this range includes $\sqrt{x}$, is not empty, and includes only non-negative numbers. Hence it can be used to begin the iteration.

The resulting ACL2 function to approximate square root can be defined as follows:

```
(defun iter-sqrt (x epsilon)
  (if (and (rationalp x)
           (<= 0 x))
      (let ((low 0)
            (high (if (> x 1) x 1)))
        (let ((range (iterate-sqrt-range
                      low high x
                      (guess-num-iters (- high low)
                                       (/ epsilon
                                          (+ high
                                             high))))))
          (car range)))
    nil))
```

For example, an estimate to $\sqrt{2}$ with precision $\pm 1/1000$ can be found with `(iter-sqrt 2 1/32)`; the value returned is 11585/8192, roughly 1.41418 which is indeed close to $\sqrt{2}$. With little more than propositional reasoning, ACL2 can now prove the main convergenge result:

```
(defthm convergence-of-iter-sqrt
  (implies (and (rationalp x)
                (rationalp epsilon)
                (< 0 epsilon)
                (<= 0 x))
           (and (<= (* (iter-sqrt x epsilon)
                       (iter-sqrt x epsilon))
                    x)
                (< (- x (* (iter-sqrt x epsilon)
                           (iter-sqrt x epsilon)))
                   epsilon))))
```

In section 2.1, it is established that the square root function can not be defined in ACL2. Nevertheless, as seen above it is possible to define approximation schemes that are as close to the square root function as desired. The next chapter will show how ACL2 can be modified to reason about irrational as well as rational numbers. Subsequently, the results of this chapter will be used to actually define the square root function in the modified version of ACL2.

Chapter 3

Non-Standard Analysis in ACL2

This chapter describes the non-standard analysis modifications to ACL2. It begins by introducing an axiomatic treatment of non-standard analysis. This serves as the foundation for non-standard analysis in ACL2, which is described next. The chapter concludes by describing an ACL2 library consisting of non-standard analysis axioms and theorems.

3.1 An Introduction to Non-Standard Analysis

The formalism of non-standard analysis in ACL2 follows the axiomatic approach pioneered by Nelson in Internal Set Theory (IST) [49]. This section presents a simple introduction to Internal Set Theory, so that the following material is self-contained. However, this section is not intended to be a comprehensive introduction to non-standard analysis. Several good introductions to non-standard analysis are readily available, including [51, 19, 48].

Internal Set Theory (IST) is a conservative extension to Zermelo-Fraenkel set theory (ZF)[1]. It introduces the unary predicate *standard* which is left undefined, just as the $\in$ predicate is undefined. Note, it is possible to ask whether any set is *standard* or not. In particular, this means that all mathematical objects built using set theory — sets, numbers,

[1]The specific set theory that is extended is largely irrelevant. ZF is used for concreteness.

functions, graphs — may be *standard* or not.

The axioms of naive set theory describe how sets may be constructed. They include the following [27, 56]:

- **Extension:** Two sets are equal if and only if they contain the same elements.
- **Specification:** Given a set A and a condition $P(x)$, there exists a set B whose elements are those elements a of A for which $P(a)$ is true. This is written as $B = \{a \in A \mid P(a)\}$.
- **Replacement:** Given a set A and a unary function f, there exists a set that contains $f(a)$ for each element a of A.

The remaining axioms are used to define valid ways of constructing sets from other sets. The specification axiom provides the only mechanism to construct a set from a predicate. In Internal Set Theory, this axiom is restricted so that the predicate $P(x)$ in $\{x \in S \mid P(x)\}$ can be neither *standard* nor any predicate defined from *standard*. Notice, this does not disallow the construction of any set that was possible in ZF, since the only terms that are affected are terms in the language of IST that are not already in the language of ZF.

To simplify future discussions, define a formula to be *classical* if it does not contain the predicate *standard* nor any functions or predicates defined using the predicate *standard*. This is a purely syntactic notion on the formulas of the language of Internal Set Theory, and it should be differentiated from the notion *standard*, which is a formal property of the objects, i.e., sets, of Internal Set Theory. It will be shown later that all classically constructed objects are *standard*. However, the converse is not true. For example, a non-classical formula may be used to construct a classical function, as the non-classical formula *standard*$(x) \vee \neg$*standard*(x) defining the classical function "true" illustrates.

In addition to the axioms from set theory, Internal Set Theory introduces three new axioms to deal explicitly with the predicate *standard*. The first axiom is the idealization axiom.

- **Idealization:** For any classical binary relation $R(x, y)$, the following are equivalent:
  - For every *standard*, finite set F there is a y so that $R(x, y)$ is true for all x in F.
  - There is a y so that $R(x, y)$ holds for all *standard* x.

This axiom guarantees the existence of at least one non-*standard* element. As an example, let R be the $<$ relation. It is certainly the case that there is an upper bound for every finite set of reals. The idealization axiom asserts the existence of a, necessarily non-standard, real y that is greater than all *standard* reals.

The second axiom is the standardization axiom.

- **Standardization:** Given a *standard* set A and a condition $P(x)$, there exists a unique *standard* subset $B \subset A$ whose *standard* elements are precisely the *standard* elements a of A for which $P(a)$ is true. This is written as $B = {}^\circ\{a \in A \mid P(a)\}$.

Note that this axiom is strictly weaker than the specification axiom when $P(x)$ is a classical property. For non-classical properties $P(x)$, it serves as a replacement to the specification axiom. However, the axiom does not guarantee anything about the non-*standard* elements of B. In particular, B may contain a non-*standard* element x for which $P(x)$ is false, or it may fail to contain a non-*standard* element $x \in A$ for which $P(x)$ is true. For example, let $P(x)$ be the property *standard*(x), and consider the set $A' = \{x \in \mathbb{R} \mid \textit{standard}(x)\}$. Since *standard* is a non-classical predicate, A' is not an admissible set. However, the standardization principle guarantees the existence of a unique *standard* set B so that, for *standard* x, $x \in \mathbb{R}$ if and only if $x \in \mathbb{R}$ and *standard*(x). That is, B is a *standard* subset of the reals containing all the *standard* reals. Since $\mathbb{R}$ is itself a *standard* subset of $\mathbb{R}$ containing all *standard* reals, it follows that $B = \mathbb{R}$, since the axiom guarantees the subset B is unique.

The standardization axiom permits a very useful construction, known as *shadow* sets. It is defined as ${}^\circ S = \{x \in U \mid x \in S\}$, where U is an arbitrary *standard* superset of S. The notation ${}^\circ S$ is read as "the shadow of S." ${}^\circ S$ is the unique *standard* set that agrees with S on all *standard* elements. Note, if S is *standard*, then ${}^\circ S$ is necessarily equal to S.

The notation $^\circ\{x \in S \mid P(x)\}$ where $P(x)$ is not a classical formula will be used to refer to the unique set guaranteed by the standardization axiom for the formula $P(x)$. The notation offers the intuitive appeal of taking the shadow of the "set" of elements satisfying $P(x)$, even though this "set" can not be formally constructed. Such "sets" are commonly referred to as "external sets" in contrast to "internal sets," which are the ordinary sets of set theory. The name "Internal Set Theory" reflects this convention.

The third and final axiom introduced in Internal Set Theory is the transfer axiom.

- **Transfer:** Let $P(x)$ be a classical formula referencing only *standard* parameters. If $P(x)$ is true for all *standard* values of x, it is also true for all possible values of x.

A useful corollary of this axiom is that any classical predicate $P(x)$ with only *standard* parameters that is satisfied by some $P(x_0)$ must also be satisfied by a *standard* x_1. Otherwise, $\neg P(x)$ would be a classical formula with *standard* parameters that is true of all *standard* values of x, so by the transfer axiom it would be true of all values of x, including x_0. In particular, this means that if the property $P(x)$ is satisfied by a *unique* element x_0, then this element must be *standard*. Examples of such elements include 0, 1, π, $\mathbb{R}$, etc.

Another useful corollary of the transfer axiom is a modified version of the extension axiom. Two *standard* sets A and B are equal if and only if they contain the same *standard* elements. This follows from the transfer axiom, since the formula $x \in A \Leftrightarrow x \in B$ is true of all x if it is true of all *standard* x, as it is a classical formula mentioning only the *standard* parameters A and B.

The restriction imposed on the axioms of specification and transfer that limit their use to only classical properties is crucial. Consider, for example, the following flawed "proof" that all natural numbers are *standard*. 0 is a *standard* natural number. If n is a natural number and n is *standard*, so is $n + 1$. (This follows from the axiom of transfer, since $n + 1$ is uniquely determined and n and 1 are *standard*.) Appealing to the principle of induction, therefore, it can be concluded that all the natural numbers are *standard*. This is false. To understand the error, recall that the induction principle is based on the wellfoundedness of the naturals: every non-empty set of naturals has a least element. Induction is sound, because the induction hypothesis guarantees the set of counter-examples to the theorem can not have a least element, hence it must be empty. However, in this case the set of counterexamples is $S = \{n \in \mathbb{N} \mid \neg \textit{standard}(n)\}$, and since *standard* is not a classical property, this set is not well-formed. What this means is that the principle of induction can not be used to prove non-classical properties. As was the case with the specification axiom, notice that this restriction does not invalidate any inductive proof that was possible before the introduction of the *standard* predicate.

Using the concept of a shadow set, it is possible to derive a weaker induction principle that is applicable to any predicate. Let $P(n)$ be a classical or non-classical property defined over the natural numbers $n \in \mathbb{N}$, and further assume that $P(0)$ and $P(n) \Rightarrow P(n+1)$ have been established. Let $S = {}^\circ\{n \in \mathbb{N} \mid P(n)\}$. That is, S is the *shadow* of the "set" of natural numbers n satisfying $P(n)$. Observe, S is the set of all naturals, since S is a *classical* set, and therefore membership in S can be established using the classical induction principle. But since S is a shadow set, it can only be concluded that $P(n)$ is true for *standard* n. Therefore, the non-standard induction principle can conclude $\textit{standard}(n) \Rightarrow P(n)$ from $P(0)$ and $P(n) \Rightarrow P(n+1)$ for any property $P(n)$.

The concept of shadows allows more powerful constructions. Consider a non-classical function $f : \mathbb{R} \to \mathbb{R}$ — that is, one whose definition uses the function *standard* or some other non-classical function — so that $f(x)$ is *standard* for every *standard* x. The function f can be used to implicitly define a *classical* function ${}^\circ f$ that agrees with f on *standard* arguments. The function ${}^\circ f$ is classical in the sense that it has a *standard* graph, and it could be given an explicit definition in the language of set theory, without the use of the predicate *standard*. The construction of ${}^\circ f$ is as follows. The function f is a set of tuples $(x, f(x))$, with the restriction that no two tuples have the same first element. Observe, the shadow of this set ${}^\circ f$ is also a function. This follows because a tuple $(x, f(x))$ is *standard* precisely when both x and $f(x)$ are, which from the hypothesis is precisely when x is *standard*. Since $^\circ f$ is *standard* (as it is the result of a shadow construction) and the set $\{x\} \times \mathbb{R}$ is *standard* for *standard* x, it follows that $X = {}^\circ f \cap (\{x\} \times \mathbb{R})$ is necessarily *standard*. But $(x, f(x))$ can be the only *standard* element of X with x as its first coordinate, since $(x, f(x))$ is *standard*, and it is the only element of f with x as its first coordinate. Similarly, $(x, f(x))$ must be in X. This means that $X = \{(x, f(x))\}$, as both X and $\{(x, f(x))\}$ are *standard* sets containing the same *standard* elements. It follows that for *standard* x, there is only one y such that $(x, y) \in {}^\circ f$. Equivalently, for *standard* x, the cardinality of $^\circ f \cap (\{x\} \times \mathbb{R})$ must be equal to 1. From the transfer principle again, the cardinality of $^\circ f \cap (\{x\} \times \mathbb{R})$ is equal to 1 for *all* x, and so $^\circ f$ is a function. What this shows is that given any function f so that $f(x)$ is *standard* whenever x is, it is possible to implicitly define a *standard* function g so that $g(x) = f(x)$ for all *standard* x. As in the case with all shadow constructions, it is not possible to say what the value of g is for a non-*standard* x, except by indirect means. For example, if $g(x) = x^2$ for all *standard* x, then using the transfer principle it follows that $g(x) = x^2$ for *all* x, even though $f(x)$ may not be x^2 for a non-*standard* x.

The *standard* predicate and the idealization, standardization, and transfer principles are surprisingly powerful. The real benefit of Internal Set Theory to analysis, however, is that it is possible to define (non-classical) predicates which correspond to many of the intuitive notions from analysis, such as "infinitely small" and "infinitely close." Analysis in the language of Internal Set Theory is commonly referred to as "non-standard analysis."[2]

A number is ***i-small*** if it is smaller in magnitude than all positive *standard* numbers. That is, ϵ is ***i-small*** if $|\epsilon| < x$ is true for all *standard* $x > 0$. Clearly 0 is ***i-small***, but it is not the only ***i-small*** number. Recall, the $idealization$ axiom demonstrates the existence of a number $y_<$ which is greater than all *standard* reals. Consequently, $1/y_<$ is smaller in

[2]The phrase shows the historical development of Internal Set Theory, which followed from the study of "non-standard" models of arithmetic. In this view of real analysis, the predicate *standard* is used to recognize the numbers in the real number line. Quantifiers, however, range not over the real numbers, but over the "hyperreals," which include the reals as well as "infinitesimals" and their arithmetic closure.

magnitude than all non-zero *standard* reals, and so it is *i-small*. The formal notion of *i-small* captures the informal notion of “infinitesimal.” Similarly, a number x is called *i-large* if it is larger in magnitude than all *standard* numbers. The number $y_<$ serves as an example. It is clear that y is *i-large* if and only if it is non-zero and $1/y$ is *i-small*, and that y is *i-small* if and only if $1/y$ is *i-large* or $y = 0$. A number that is not *i-large* is called *i-limited*. It is clear that all *standard* numbers must be *i-limited*. Two numbers x and y are *i-close* when $x - y$ is *i-small*. Since 0 is the only *standard i-small* number, it follows that two *standard* numbers are *i-close* if and only if they are equal.

In addition, it can be shown that there is a function *standard-part* which assigns a *standard* number *i-close* to each *i-limited* real. That is, for *i-limited* x, *standard-part*(x) is *standard* and *i-close* to x. The number *standard-part*(x) can be defined as the supremum of ${}^\circ\{y \in \mathbb{R} \mid y \leq x\}$. This set is bounded above, since x is *i-limited*, and so there must be a *standard* number M with $|x| \leq M$. From the transfer principle, *standard-part*(x) is *standard*, since it is the supremum of a *standard* set. That *standard-part*(x) is *i-close* to x follows from the fact that for any *standard* $c > 0$, $x -$ *standard-part*$(x) \geq c$ implies that $x \geq$ *standard-part*$(x) + c$ and so *standard-part*$(x) + c$ is in ${}^\circ\{y \in \mathbb{R} \mid y \leq x\}$, since it is *standard* and at most x, but then *standard-part*(x) would not be a supremum of this set. Similarly, if *standard-part*$(x) - x \geq c$, it must be that *standard-part*$(x) - c \geq x$ so ${}^\circ\{y \in \mathbb{R} \mid y \leq x\}$ would be bounded by *standard-part*$(x) - c$, again contradicting *standard-part*(x) as the supremum of the set. Therefore, *standard-part*(x) and x are *i-close*. Since *standard-part*(x) is *standard*, it follows that it is the unique *standard* number *i-close* to x.

These new functions — *i-small*, *i-large*, *i-limited*, *i-close* and *standard-part* — obey simple algebraic properties. For example, it is obvious that $x + y$ is *i-small* (*i-limited*) if both x and y are *i-small* (*i-limited*). If x is *i-limited* and ϵ is *i-small*, $\epsilon \cdot x$ is *i-small* and x/ϵ is *i-large* for $\epsilon \neq 0$. If x is *i-close* to y and y is *i-close* to z, then x is *i-close* to z. Less obvious is the fact that for *i-limited* x and y, the *standard-part* of $x + y$ is the sum of *standard-part*(x) and *standard-part*(y).

The convenience of non-standard analysis becomes evident when traditional notions from analysis are written using the new language of non-standard analysis. For example, a *standard* sequence $\{a_n\}$ converges to the *standard* point A if A is *i-close* to a_N for all *i-large* integers N. A *standard* function f is continuous at a *standard* point x if $f(y)$ is *i-close* to $f(x)$ for all y *i-close* to x. These definitions are easier to use than the traditional definitions from analysis. Consider the function $f(x) = \sin(1/x)$. It is a classical result from analysis that this function can not be extended continuously at $x = 0$. The traditional proof is to find two sequences $\{a_n\}$ and $\{b_n\}$ converging to 0, so that $\{f(a_n)\}$ and $\{f(b_n)\}$ converge to different values. In non-standard analysis, the argument is considerably more direct. Consider $x_0 = \frac{1}{2\pi n}$ and $x_1 = \frac{1}{2\pi n+\pi/2}$, where n is an *i-large* integer. Then x_0 and x_1 are *i-close* (both being *i-small*), but $f(x_0) = 0$ is not *i-close* to $f(x_1) = 1$. Therefore, no value of $f(0)$ can be *i-close* to $f(y)$ for all y *i-close* to 0, and hence f can not be continuously extended at $x = 0$.

With all the new predicates of non-standard analysis, it is important to re-create the traditional mental picture of the real number line. The integers are divided into two groups. The *standard* integers include 0, ± 1, ± 2, ... There is at least one non-*standard* integer N. Necessarily, $\pm N, \pm(N \pm 1), \pm(N \pm 2), \ldots$ are also non-*standard*. Notice in particular that there is no least non-*standard* integer. Also notice that the properties *standard* and *i-limited* coincide for the integers.

The corresponding picture for the reals is a little more complex. Certainly, there are *i-large* numbers, such as the *i-large* integer N, as well as $N/2$, $\sqrt{N}$, etc. As is the case with the integers, all reals larger in magnitude than N are also *i-large*, as is any number $N - x$ for *i-limited* x. Moreover, there are *i-small* reals, all of which are *i-close* to 0. All *i-limited* reals are *i-close* to a *standard* real. That is, if x is *i-limited*, it can be written as $x = x^* + \epsilon$, where x^* is *standard* and ϵ is *i-small*. The number x^* is equal to *standard-part*(x).

3.2 Non-Standard Analysis Primitives in ACL2

As chapter 2 showed, the numeric system of ACL2 v2.1 is too restrictive to permit the real numbers. Hence the first step towards analysis in ACL2 is the extension of the numeric system to include the irrationals.

In addition to the type recognizers `rationalp` and `complex-rationalp` of ACL2 v2.1, ACL2 v2.1(r) includes the new type recognizers `realp` and `complexp` and modifies the type `acl2-numberp` to include them both. Moreover, the arithmetic axioms of ACL2 v2.1(r) have been modified to reflect the new elements in the ACL2 universe. For many axioms, the required modifications are straight-forward. For example, the `Positive` axiom, stating that if x and y are positive rationals, $x \cdot y$ is a positive rational, is extended to the reals simply by replacing `rationalp` with `realp` everywhere. Some axioms, however, are too useful in their own right to simply replace rational with real everywhere. For example, an axiom built into the ACL2 type system states that the product of two rationals is rational. Rather than weakening this axiom by replacing it with the corresponding axiom for the reals, ACL2 v2.1(r) adds the axiom asserting that the product of two reals is real. Of the arithmetic axioms in ACL2 v2.1, only those explicitly dealing with numerator and denominator can not be extended to the reals.

The initial ACL2 theory contains more than the basic arithmetic axioms. It also contains many useful arithmetic functions, such as `abs`, `floor`, and `trunc`. All of these functions need to be extended to accept irrational arguments. This is trivial in the case of functions like `abs`. However, the functions `floor` and `trunc` are defined using `integer-quotient`, which performs division by repeated subtraction. For example, the floor of $17/2$ is found by dividing 2 into 17 using repeated subtraction, giving a value of 8. Clearly, a similar trick will not work for the reals. A simple solution to this problem is to introduce a new undefined function `floor1` which is axiomatized to return the correct value of $\lfloor x \rfloor$ for an arbitrary number x. Of course, it would be easier to axiomatize `floor` directly, but that would make the executable version of the function `floor` undefined for all arguments. By introducing `floor1`, it is possible to allow ACL2 to have an executable version of `floor`, at least for rational constants. With the executable version, ACL2 can evaluate a constant expression that appears in the middle of a proof, such as `(floor 3/2 1)`.

With these modifications, ACL2 is able to reason about the irrational numbers, but it can not construct irrational numbers. In particular, there are no irrational constants, and there is no mechanism to allow an ACL2 function to return an irrational result given rational arguments. Nevertheless, it can reason effectively about the real and complex numbers. For example, it shown in chapter 2 that $x \cdot x \neq 2$ is a theorem of ACL2 v2.1. However, this result can not be proved after the introduction of `realp`. Instead, it is possible to show that if $x \cdot x = 2$ then x must be real but not rational. However, as described so far, ACL2 v2.1(r) can not prove that there must be some x with $x \cdot x = 2$, nor can it define the function `sqrt` with the required property. To do that requires the knowledge that the real number line is complete. This proof is accomplished in ACL2 v2.1(r) using non-standard analysis.

The primitive non-standard functions in ACL2 v2.1(r) are `standard-numberp`, `standard-part`, and `i-large-integer`. `Standard-numberp` is a function that tests whether a number is *standard* or not. Note, this is a strict numeric type, so ACL2 treats all non-numeric objects as non-*standard*. The function `standard-part` returns the standard part of a real or complex number, provided such a number exists, i.e., provided the number is *i-limited*. For *i-large* numbers, `standard-part` is not defined, but it is convenient to think of it as the identity function. As its name suggests, the constant `i-large-integer` is an integer axiomatized to be *i-large*; it is also assumed to be positive. The functions `i-small`, `i-large`, `i-limited`, and `i-close` are given explicit definitions in terms of `standard-part`. A number is `i-small` if its `standard-part` is zero; it is `i-large` if its inverse is `i-small` and non-zero; and it is `i-limited` if it is not `i-large`. Two numbers are `i-close` if their difference is `i-small`.

All of these functions are special in two ways. First, none of the primitive non-standard functions is given an explicit definition. Instead, they are all treated as constrained functions, as if they had been introduced using `encapsulate` or `defstub`; ACL2 can not evaluate the value of any term that depends non-trivially on one of these functions. Secondly, ACL2 introduces the notion of classical and non-classical functions. These new functions are considered to be non-classical, as are any functions defined in terms of non-classical functions. Note, any ACL2 v2.1(r) functions (formulas) that are also ACL2 v2.1 functions (formulas) are necessarily classical.

ACL2 v2.1(r) restricts the use of non-classical functions to prevent inadvertent uses of the specification axiom on non-classical properties. A non-classical function can not be defined recursively. In effect, non-classical functions in ACL2 are similar to macros, since any term involving a non-classical function can always be "flattened" into a term involving only the primitive non-classical functions. Moreover, non-classical constrained functions are not permitted in ACL2 v2.1(r). When a function symbol is introduced using `encapsulate`, ACL2 v2.1(r) considers the function to be classical, and it ensures that the local witness function used to justify the introduction is also classical.

Moreover, the use of induction on non-classical formulas is restricted. Recall that in Internal Set Theory induction can only be used over the *standard* integers. Similarly, the induction principle in ACL2 can be used to establish the truth of non-classical formulas only for *standard* instances of their variables. The remaining cases are treated separately, similar to the "base" cases. That is, for each variable appearing in the formula, the non-*standard* induction principle of ACL2 adds the proof obligation that the formula is true for non-*standard* instances of the variable.

Consider the function `factorial` defined as follows:

```
(defun factorial (n)
  (if (and (integerp n) (< 0 n))
      (* n (factorial (- n 1)))
    1))
```

Suppose an attempt is made to prove that `factorial` always returns a *standard* value:

```
(defthm standard-numberp-factorial-false
  (standard-numberp (factorial n)))
```

The classical induction principle of ACL2 would reduce this theorem to the following two goals:

```
(implies (not (and (integerp n) (< 0 n)))
         (standard-numberp (factorial n)))


(implies (and (and (integerp n) (< 0 n)))
              (standard-numberp (factorial (+ -1 n))))
         (standard-numberp (factorial n)))
```

In addition, since the theorem uses the non-classical function `standard-numberp`, the following goal is added by ACL2 v2.1(r):

```
(implies (not (standard-numberp n))
         (standard-numberp (factorial n)))
```

Intuitively, the first two goals prove the theorem for any *standard* value of n, and the last goal proves it for any non-*standard* n. In this case, the last goal can not be established, and so ACL2 v2.1(r) does not prove that all integers have a *standard* factorial. As can be seen, this modification to the induction principle is crucial in preserving soundness.

The theorem `standard-numberp-factorial-false` is false, because the factorial of a non-*standard* integer is also non-*standard*. However, the theorem is true if only *standard* integers are considered, as ACL2 v2.1(r) can prove:

```
(defthm standard-numberp-factorial
  (implies (standard-numberp n)
           (standard-numberp (factorial n))))
```

In this case, the basis and induction steps are as follows:

```
(implies (and (not (and (integerp n) (< 0 n)))
              (standard-numberp n))
         (standard-numberp (factorial n)))


(implies (and (and (integerp n) (< 0 n))
              (standard-numberp n)
              (implies (standard-numberp (+ -1 n))
                       (standard-numberp (factorial
                                           (+ -1 n)))))
         (standard-numberp (factorial n)))
```

ACL2 can quickly prove both of these goals. In addition, ACL2 v2.1(r) adds the following goal, since the theorem is non-classical:

```
(implies (and (not (standard-numberp n))
              (standard-numberp n))
         (standard-numberp (factorial n)))
```

This time, ACL2 is able to prove this goal, since the hypotheses are quickly found to be contradictory, completing the proof of the original conjecture. In general, the only non-classical theorems that can be proved by induction are those that specifically apply to *standard* values, for example by having `standard-numberp` as a hypothesis, as in `standard-numberp-factorial`.

Formally, the ACL2 non-standard induction principle is as follows:

Suppose:

- p is a term;
- r is a function symbol that denotes a classical well-founded relation;

- m is a classical function symbol of n arguments;
- $x_1, \ldots, x_n$ are distinct variables;
- $q_1, \ldots, q_k$ are terms;
- $h_1, \ldots, h_k$ are positive integers;
- for $1 \leq i \leq k$ and $1 \leq j \leq h_i$, $s_{i,j}$ is a classical substitution, and it is a theorem that

  ```
  (IMPLIES $q_i$
           ($r$ ($m$ $x_1 \ldots x_n$)/$s_{i,j}$ ($m$ $x_1 \ldots x_n$)))
  ```

  and
- $y_1, \ldots, y_u$ are the variables occurring in p that are one of the x_i or are changed by the $s_{i,j}$.

Then p is a theorem if

```
(IMPLIES (AND (NOT $q_1$) ...(NOT $q_k$))
         $p$)
```

is a theorem, for each $1 \leq i \leq u$,

```
(IMPLIES (NOT (STANDARD-NUMBERP $y_i$))
         $p$)
```

is a theorem, and for each $1 \leq i \leq k$,

```
(IMPLIES (AND $q_1$ $p$/$s_{i,1}$ ...$p$/$s_{i,h_i}$)
         $p$)
```

is a theorem.

Compare this to the formal definition of the induction principle of Nqthm or ACL2, found in [10, 38].

To understand why the non-standard induction principle is sound, consider a specific choice p, r, m, x_i, q_i, h_i, $s_{i,j}$, and h_k such that the conditions above as well as the basis and induction steps can be established. Then the following proof in internal set theory establishes the validity of p.

PROOF: Without loss of generality, assume that the x_i are X_1, X_2, …, X_n; that r is R; that m is M; that X_{n+1}, X_{n+2}, …, X_z are all of the variables other than X_1, X_2, …, X_n in p, the q_i and either component of any pair in any $s_{i,j}$; that p is (P X_1 … X_z); that q_i is (Q_i X_1 … X_z); that $s_{i,j}$ replaces X_v with some term $d_{i,j,v}$; and that the Y_i are given by X_1, X_2, …, X_u, for some $n \leq u \leq z$. Note that $d_{i,j,v}$ is equal to X_v for $u < v \leq z$.

Let RM be the function on u-tuples defined by

$$(RM\ \langle U_1 \ldots U_u\rangle\ \langle V_1 \ldots V_u\rangle) = (R\ (M\ U_1 \ldots U_n)\ (M\ V_1 \ldots V_n)).$$

Note that RM is classical and well-founded.

Let the tuple $C = \langle C_{u+1}\ C_{u+2} \ldots C_z\rangle$ be a binding for the tuple of variables $\langle X_{u+1}\ X_{u+2} \ldots X_z\rangle$. Define the set GC as the shadow set of all u-tuples U for which $(p\ U\ C)$ is false. That is, it is defined as follows:

$$GC = {}^\circ\{\langle U_1 \ldots U_u\rangle \mid (P\ U_1 \ldots U_u\ C_{u+1}\ C_{u+2} \ldots C_z) \text{ is false}\}$$

Since GC is a standard set, membership in GC can be decided using the classical principle of induction. In particular, if GC is non-empty, it must have an RM-minimal tuple. Moreover, GC is a standard set, so by the transfer principle, if it is non-empty, it must have a standard RM-minimal tuple. Let $\langle X_1\ X_2 \ldots X_u\rangle$ be such a tuple. There are two cases to consider.

Case 1: Suppose none of the q_i is true. By the base case, $(P\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ is true, and so $\langle X_1 \ldots X_u\rangle$ should not be in GC, yielding a contradiction.

Case 2: Suppose at least one of the q_i is true. Without loss of generality, assume that the term $(Q_1\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ is true. From the conditions on r, m, q, and $s_{i,j}$, it follows that

$$(R\ (M\ d_{1,1,1} \ldots d_{1,1,n})\ (M\ X_1 \ldots X_n))$$
$$(R\ (M\ d_{1,2,1} \ldots d_{1,2,n})\ (M\ X_1 \ldots X_n))$$
$$\vdots$$
$$(R\ (M\ d_{1,h_1,1} \ldots d_{1,h_1,n})\ (M\ X_1 \ldots X_n))$$

are all true. By the definition of RM,

$$(RM\ \langle d_{1,1,1} \ldots d_{1,1,u}\rangle\ \langle X_1 \ldots X_u\rangle)$$
$$(RM\ \langle d_{1,2,1} \ldots d_{1,2,u}\rangle\ \langle X_1 \ldots X_u\rangle)$$
$$\vdots$$
$$(RM\ \langle d_{1,h_1,1} \ldots d_{1,h_1,u}\rangle\ \langle X_1 \ldots X_u\rangle)$$

are all true as well. Observe, the terms $d_{1,i,j}$ are all standard, since the X_i are standard, and the substitutions $s_{1,i}$ are assumed classical, hence they return standard values for standard arguments. Since $\langle X_1 \ldots X_u\rangle$ is an RM-minimal u-tuple such that $(p\ U\ C)$ is false and the $d_{1,i,j}$ are all standard, it follows that

$$(P\ d_{1,1,1} \ldots d_{1,1,u}\ C_{u+1} \ldots C_z)$$
$$(P\ d_{1,2,1} \ldots d_{1,2,u}\ C_{u+1} \ldots C_z)$$
$$\vdots$$
$$(P\ d_{1,h_1,1} \ldots d_{1,h_1,u}\ C_{u+1} \ldots C_z)$$

are all true. Hence, $(P\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ follows from the first induction hypothesis, contradicting the assumption that $\langle X_1 \ldots X_u\rangle$ is in GC.

Therefore, it can be concluded that GC is empty, since it is a *standard* set containing no *standard* elements. From the definition of GC, it follows that for any standard tuple

$\langle X_1 \ldots X_u \rangle$, $(P\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ must be true. Moreover, from the assumptions, if $\langle X_1 \ldots X_u \rangle$ is a non-standard tuple, it also follows that $(P\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ is true. This is because if $\langle X_1 \ldots X_u \rangle$ is non-standard, one of the X_i must be non-standard, and then the theorem follows from the hypothesis

```
(IMPLIES (NOT (STANDARD-NUMBERP $X_i$))
         $p$)
```

This establishes that $(P\ X_1 \ldots X_u\ C_{u+1} \ldots C_z)$ is true for all tuples $\langle X_1 \ldots X_u \rangle$.

Since this argument can be carried out for any values of $\langle C_{u+1} \ldots C_z \rangle$, it follows that $(P\ X_1 \ldots X_u\ X_{u+1} \ldots X_z)$ is true for all values of the X_i. This establishes the validity of p. Q.E.D.

ACL2 v2.1(r) introduces two new events that deal exclusively with non-classical formulas. The event `defun-std` is used to define a *standard* function using a non-classical body. The newly introduced function is considered to be a classical function. This is justified by the Internal Set Theory concept of shadow functions only when the function body returns a *standard* value for *standard* arguments. In these cases, the function is explicitly defined only for the *standard* arguments; that is, the function is defined by its body only for *standard* arguments. For the remaining arguments, the function is implicitly defined, as being the (unique) *standard* function that agrees with the body for *standard* arguments.

Consider, for example, the function introduced as follows:

```
(defun-std std-pt (x)
  (standard-part x))
```

This function is accepted, because for *standard* x, `(standard-part x)` is *standard*. It is important to realize that `std-pt` is *not* the same as the function `standard-part`. The *standard* function `std-pt` is guaranteed equal to `standard-part` only for *standard* values for `x`. Since `standard-part` returns `x` for these values, it follows that `std-pt` is the identity function for *standard* x. But, since `std-pt` is considered classical, it must also be the identity function for all values of `x`, since by the transfer axiom, two classical functions that have equal values for *standard* arguments must be equal to each other.

The other event introduced by ACL2 2.1(r) is `defthm-std`, which serves as an explicit invocation of the transfer axiom. Using `defthm-std`, it is possible to prove a theorem by proving it only for *standard* arguments; however, as is the case with the transfer axiom, `defthm-std` can only be used to prove classical formulas. For example, consider the following theorem:

```
(defthm-std std-pt-is-identity
  (implies (acl2-numberp x)
           (equal (std-pt x) x)))
```

Since `std-pt` is treated as a classical function, this theorem can be proved using `defthm-std`. ACL2 v2.1(r) will prove this theorem by considering only *standard* values of `x`. That is, it attempts to prove the following formula instead:

```
(implies (and (standard-numberp x)
              (acl2-numberp x))
         (equal (std-pt x) x))
```

Since `x` is known to be *standard*, the term `(std-pt x)` can be expanded using the body of `std-pt`, and the proof becomes trivial. Note, the theorem could not have been proved using `defthm` instead of `defthm-std`, because the term `(std-pt x)` can not be expanded without the hypothesis `(standard-numberp x)`, since the body of `std-pt` can only be expanded for *standard* arguments, as `std-pt` was introduced using `defun-std`. Moreover, a similar theorem about `standard-part` instead of `std-pt` could not have been proved using `defthm-std`, since the resulting formula is not classical.

The combination of `defun-std` and `defthm-std` is particularly powerful. Using `defun-std`, it is possible to introduce functions mapping rationals to irrationals. A useful technique is to construct a rational approximation $\{f_n(x)\}$ to a particular function $f(x)$. The limit of the sequence $\{f_n(x)\}$, namely $f(x)$, can be expressed in ACL2 using `defun-std` as *standard-part*$(f_N(x))$ for any *i-large* natural N, such as `i-large-integer`. Properties of $f(x)$ can be established using `defthm-std`. It is only necessary to prove the given property for *standard* values of x, for which cases the definition of $f(x)$ can be opened up to *standard-part*$(f_N(x))$. This approach is illustrated in chapter 4, where the square root function is defined in ACL2 and its relevant properties are proved. This same approach will later be used to define the exponential function.

3.3 The Non-Standard Analysis ACL2 Library

The previous section described the non-standard analysis theory that ACL2 v2.1(r) recognizes on startup. This theory by itself is not strong enough to do useful analysis in ACL2. The theory is extended by a set of axioms[3] and lemmas collected in the ACL2 “books” or libraries `nsa.lisp` and `nsa-complex.lisp`. This section describes those libraries.

It is in `nsa.lisp` that the existence of a non-standard number is assumed; the axiom `i-large-integer-is-large` asserts that `i-large-integer` is an *i-large* number.

Another group of axioms relates how the arithmetic operators combine numbers that are *standard*. For example, the following axiom asserts that the sum of two *standard* numbers is *standard*:

```
    (defaxiom standard-numberp-plus
      (implies (and (standard-numberp x)
                    (standard-numberp y))
```

[3]Adding the necessary axioms to the file `axioms.lisp` would build them into the theorem prover, so they would be available to users on start-up. This will be the likely approach when ACL2(r) is released to the general public; however, the current approach proved more friendly in the development of the theory, because axioms could be tried out without requiring ACL2 to be recompiled.

```
                (standard-numberp (+ x y))))
```

Similar theorems treat unary minus, multiplication, and inversion. It is also assumed that a complex number is *standard* if its real and imaginary parts are *standard*.

An important axiom states that the number 1 is *standard* in ACL2. Using this fact as well as the arithmetic axioms above, it is possible to prove that all rational numbers with an arbitrary but fixed bound on their numerator and denominator must be standard. The following two theorems are very useful in ACL2 v2.1(r) to prove that a particular number is *standard*:

```
(defthm standard-numberp-integers-to-10000
  (implies (and (integerp x) (<= -10000 x) (<= x 10000))
           (standard-numberp x)))

(defthm standard-numberp-rationals-num-demom-10000
  (implies (and (rationalp x)
                (<= -10000 (numerator x))
                (<= (numerator x) 10000)
                (<= (denominator x) 10000))
           (standard-numberp x)))
```

Note, the second theorem subsumes the first, but the rewrite engine of ACL2 favors the first theorem when only integer values are present.

The properties of `standard-part` are also axiomatized. The two fundamental properties are that the `standard-part` of an *i-limited* number is *standard* and that the `standard-part` of a *standard* number is the number itself. This results in the following axioms:

```
(defaxiom standard-part-of-standard-numberp
  (implies (standard-numberp x)
           (equal (standard-part x) x)))

(defaxiom standardp-standard-part
  (implies (i-limited x)
           (standard-numberp (standard-part x))))
```

It is also assumed that the `standard-part` of a complex number is formed by taking the `standard-part` of its real and imaginary parts.

This leads up to the `standard-part` of arithmetic expressions. In formulating these axioms, it is important that the function `standard-part` be applied only to *i-limited* numbers. So for example, the *standard-part* of the sum of two numbers is the sum of their *standard-part*, but only if both numbers are *i-limited*:

```
(defaxiom standard-part-of-plus
  (implies (and (i-limited x)
                (i-limited y))
           (equal (standard-part (+ x y))
                  (+ (standard-part x)
                     (standard-part y)))))
```

Similar axioms are introduced for multiplication and inverses. However, in the case of unary minus, the axiom introduced is a little stronger. In particular, it is expected that the `standard-part` of a negation is the negation of the `standard-part`:

```
(defaxiom standard-part-of-uminus
  (equal (standard-part (- x))
         (- (standard-part x))))
```

This axiom is justified because function `standard-part` can be extended to return the *standard-part* of x for *i-limited* x and simply x when x is not *i-limited*.

An important axiom states that `standard-part` is a monotonic function on the reals. In particular, if $x \leq y$, then *standard-part*$(x) \leq$ *standard-part*(y).

```
(defaxiom standard-part-<=
  (implies (and (realp x) (realp y) (<= x y))
           (<= (standard-part x) (standard-part y))))
```

Using this axiom, it is possible to prove a "squeeze" theorem for `standard-part`. In particular, if y is between x and z, and the *standard-part* of x is the same as that of z, then the *standard-part* of y must also be equal to that value.

```
(defthm standard-part-squeeze
  (implies (and (realp x) (realp y) (realp z)
                (<= x y) (<= y z)
                (= (standard-part x) (standard-part z)))
           (equal (standard-part y) (standard-part x))))
```

The remaining axioms and theorems define the theory of the predicates `i-small`, `i-limited`, and `i-large`. The first axiom asserts that `i-small` numbers are also `i-limited`. Similarly, it is assumed that `standard-numberp` numbers are also `i-limited`. The converse of this axiom is not true in general, but it is assumed that `i-limited` integers are also `standard-numberp`.

With the axioms introduced up to this point, ACL2 v2.1(r) can prove that the sum of two `i-small` numbers is `i-small`, as is the product of an `i-small` number and an `i-limited` number. It can also be proved that an `i-limited` number is `i-close` to its `standard-part`.

To prove that the sum of two `i-limited` numbers is also `i-limited`, it is necessary to add an axiom asserting that the sum of a `standard-numberp` and an `i-small` number is `i-limited`. In particular, if x is *i-limited*, it can be written as ${}^*x + \epsilon_x$, where ${}^*x$ is *standard* and ϵ_x is *i-small*. Similarly, an *i-limited* y can be written as ${}^*y + \epsilon_y$.

Hence, $x+y$ is equal to ${}^*x+\epsilon_x+{}^*y+\epsilon_y$, which is equal to $({}^*x+{}^*y)+(\epsilon_x+\epsilon_y)$, the sum of a *standard* and an *i-small* number which is *i-limited* by the above axiom. A similar argument allows ACL2 v2.1(r) to prove that the negative of an `i-limited` number, the inverse of an `i-limited` and not `i-small` number, and the product of two `i-limited` numbers are all `i-limited`.

No further axioms are required for non-standard analysis in ACL2 v2.1(r). The libraries contain an additional number of useful theorems. In particular, it is proved that the sum of an `i-limited` and an `i-large` number must be `i-large`. Also, it is shown that the product of an `i-large` and an `i-limited` number that is not `i-small` must be `i-large`. Moreover, it is shown that a number that is `i-close` to an `i-small` (`i-limited` or `i-large`) number is also `i-small` (`i-limited` or `i-large` respectively). The predicate `i-close` is proved to define an equivalence relation.

The predicates `i-small`, `i-limited`, and `i-large` split the real numbers into broad orders of magnitude. ACL2 v2.1(r) is able to prove that if a number is smaller in magnitude than an `i-small` (`i-limited`) number, it must also be `i-small` (`i-limited`). Similarly, a number larger in magnitude than an `i-large` number must be `i-large`. It is also shown that an `i-small` number is smaller in magnitude than an `i-limited` number, which in turn is smaler in magnitude than an `i-large` number.

Whether a complex number is an `i-small`, `i-limited`, or `i-large` number can be decided by inspecting its real and imaginary parts. It can be shown that a complex number is `i-small` if and only if both its real and imaginary parts are `i-small`. Similarly, it is `i-limited` precisely when both its real and imaginary parts are `i-limited`. It is `i-large` when either of its real and imaginary parts is `i-large`.

This develops a non-standard analysis theory sufficiently powerful to prove many useful theorems. The next chapters will illustrate this development, touching on the intermediate value theorem, an order-of-magnitude preserving norm on the complex plane, and the exponential and trigonometric functions.

Chapter 4

The Square Root Function Revisited

This chapter combines the results of the previous two chapters to define the square root function in ACL2 v2.1(r). Recall, chapter 2 demonstrated that the square root function could not be soundly introduced into ACL2 v2.1, even though an arbitrarily good approximation to the square root could be defined. In this chapter, the non-standard analysis techniques introduced in chapter 3 will be used to derive the square root function from the approximation defined in chapter 2.

4.1 Defining the Square Root Function in ACL2

In chapter 2, the following ACL2 v2.1 theorem was proved:

```
(defthm there-is-no-sqrt-2
  (not (equal (* x x) 2)))
```

Recall, the proof of this theorem proceeded by eliminating all the possible candidates for such an `x`. That the square of no rational number is equal to 2 followed from the classical observation that $\sqrt{2}$ is irrational. The complex numbers were eliminated using simple algebraic means — the square of a non-real complex number is either a non-real complex number or a negative real. ACL2 objects other than numbers have zero squares. Thus were all possibilities eliminated.

However, ACL2 v2.1(r) contains more objects in its universe than ACL2 v2.1. In particular, it adds the numeric types `realp` and `complexp`. Therefore, the process of elimination described above fails in ACL2 v2.1(r). Objects of type `complexp` can still be eliminated with the same algebraic argument by which the complex rationals were eliminated earlier. However, the possibility remains that the square of some irrational is equal to 2. The ACL2 v2.1 theorem `there-is-no-sqrt-2` is *not* a theorem of ACL2 v2.1(r). In its stead, a weaker statement can be proved:

```
(defthm irrational-sqrt-2
  (implies (equal (* x x) 2)
           (and (realp x)
                (not (rationalp x)))))
```

Given the above, it appears possible to axiomatize a function in ACL2 v2.1(r) corresponding to the square root function. In fact, such a function can be defined. In chapter 2 the function `iter-sqrt` was introduced, and it was shown that this function is a good approximation to the square root function. Recall in particular the following theorem, updated to the real numbers:

```
(defthm convergence-of-iter-sqrt
  (implies (and (realp x)
                (realp epsilon)
                (< 0 epsilon)
                (<= 0 x))
           (and (<= (* (iter-sqrt x epsilon)
                       (iter-sqrt x epsilon))
                    x)
                (< (- x (* (iter-sqrt x epsilon)
                           (iter-sqrt x epsilon)))
                   epsilon))))
```

The number `(iter-sqrt x epsilon)` can be bounded by a tight range. In particular, it is non-negative, and it can be no larger than the maximum of `x` and 1. This shows that when `x` is an *i-limited* number, so is `(iter-sqrt x epsilon)`. In ACL2, we can establish the following theorems, which verify the claimed bounds for the expression:

```
(defthm iter-sqrt-type-prescription
  (and (realp (iter-sqrt x epsilon))
       (<= 0 (iter-sqrt x epsilon))))

(defthm iter-sqrt-upper-bound-1
  (implies (and (realp x)
                (<= 1 x))
           (<= (iter-sqrt x epsilon) x)))

(defthm iter-sqrt-upper-bound-2
  (implies (and (realp x)
                (< x 1))
           (<= (iter-sqrt x epsilon) 1)))
```

From these lemmas, it is easy to show in ACL2 v2.1(r) that `iter-sqrt` returns an *i-limited* value when its argument is *i-limited*, as claimed above:

```
(defthm limited-iter-sqrt
  (implies (and (i-limited x)
                (realp x)
                (<= 0 x))
           (i-limited (iter-sqrt x epsilon))))
```

This allows the function `acl2-sqrt` to be introduced as the limit of `iter-sqrt`. This is accomplished using the ACL2 v2.1(r) event `defun-std`:

```
    (defun-std acl2-sqrt (x)
      (standard-part (iter-sqrt x (/ (i-large-integer)))))
```

This is the first example of a construction that will be used repeatedly in future chapters, so it is useful to pause and reflect on what has happened. Recall, `defun-std` permits the definition of *classical* functions from non-classical bodies. The body of `acl2-sqrt` is clearly non-classical, as it refers to the primitive non-standard analysis functions `standard-part` and `i-large-integer`. However, the definition is only accepted when the body returns a *standard* value for *standard* arguments. The body of `acl2-sqrt` does just that, because the `iter-sqrt` term will return an *i-limited* result — since the *standard* value of `x` is also *i-limited* — and the *standard-part* of an *i-limited* number is known to be *standard*. Hence, the definition is accepted by ACL2 v2.1(r). The body defines the value of `acl2-sqrt` only for *standard* arguments `x`. In these cases, the number returned by `iter-sqrt` is *i-close* to the square root of `x` — this is true, since its square is within `(/ (i-large-integer))` of the square of `x` hence they are *i-close* to each other, and the squares of two non-negative *i-limited* numbers are *i-close* to each other only when the two numbers are *i-close* to each other. Therefore, the *standard-part* of the `iter-sqrt` term must be equal to $\sqrt{x}$, since $\sqrt{x}$ is *standard* when `x` is *standard* and no two different *standard* numbers are *i-close* to each other. Hence `acl2-sqrt` *is* equal to the mathematical square root function.

Thus far, it has only been mechanically verified that the function exists. Its properties follow from a formalization of the argument above. The first step is a restatement of the convergence result for `iter-sqrt`. The restatement uses the language of non-standard analysis:

```
    (defthm convergence-of-iter-sqrt-strong
       (implies (and (realp x)
                     (realp epsilon)
```

```
                    (< 0 epsilon)
                    (i-small epsilon)
                    (<= 0 x))
               (i-close (* (iter-sqrt x epsilon)
                           (iter-sqrt x epsilon))
                        x)))
```

The remainder of the argument can be formalized easily in ACl2 v2.1(r). The result is the fundamental theorem of the square root function:

```
    (defthm-std sqrt-sqrt
      (implies (and (realp x)
                    (<= 0 x))
               (equal (* (acl2-sqrt x) (acl2-sqrt x)) x)))
```

The use of `defthm-std` instead of `defthm` to introduce the theorem `sqrt-sqrt` is important. It allows ACL2 v2.1(r) to restrict consideration to *standard* values of `x`. Only for those values of `x` can the definition of `acl2-sqrt` be opened. When it is, the result follows from the lemma `convergence-of-iter-sqrt-strong`.

4.2 Properties of the Square Root Function

ACL2 v2.1(r) can prove more properties of the square root function. Particularly useful are theorems describing how to decide whether a number is less than or greater than the square root of another. A familiar trick from algebra is to conclude that $\sqrt{x} < y$ by verifying that $x < y^2$ — and a familiar error in algebra is to forget to verify that x and y are non-negative before squaring both sides. This trick can be used in ACL2 by proving the following theorem:

```
    (defthm sqrt-<-y
  (implies (and (realp x)
                (<= 0 x)
                (realp y)
                (<= 0 y))
           (equal (< (acl2-sqrt x) y)
                  (< x (* y y)))))
```

Similar theorems can be proved to reason about $y < \sqrt{x}$, as well as the cases involving $>$ instead of $<$.

In many cases, these theorems can be used to evaluate the value of $\sqrt{x}$. What is necessary is to find a candidate value y so that $y^2 = x$. It then follows that $y = \sqrt{x}$:

```
(defthm y*y=x->y=sqrt-x
  (implies (and (realp x)
                (<= 0 x)
                (realp y)
                (<= 0 y)
                (equal (* y y) x))
           (equal (acl2-sqrt x) y)))
```

This theorem will serve as the only way to reduce constant expressions involving `acl2-sqrt`. It is immediate, for example, that `(acl2-sqrt 0)` is equal to 0 and `(acl2-sqrt 1)` is equal to 1. The case of equalities involving non-constant `acl2-sqrt` terms can also be solved by squaring both sides of the equality.

The theorem `y*y=x->y=sqrt-x` is extremely useful. From `y*y=x->y=sqrt-x`, it is easy to verify that the square root of a product is the product of the square roots. Similarly, the inverse of a square root is the square root of the inverse. Moreover, it can also be shown that $\sqrt{x^2} = |x|$. A surprising theorem that follows from `y*y=x->y=sqrt-x` is that for *i-limited* numbers, the *standard-part* of the square root is the same as the square root of the *standard-part*.

In order to introduce the function `acl2-sqrt`, it was required to show that `iter-sqrt` returned *i-limited* values for *i-limited* arguments. ACL2 v2.1(r) can also prove that `acl2-sqrt` returns *i-limited* results for *i-limited* arguments. The proof follows from the following two lemmas:

```
(defthm-std acl2-sqrt-x-<-1
  (implies (and (realp x)
                (<= 0 x)
                (< x 1))
           (<= (acl2-sqrt x) 1)))

(defthm-std acl2-sqrt-x-<-x
  (implies (and (realp x)
                (<= 0 x)
                (<= 1 x))
           (<= (acl2-sqrt x) x)))
```

Notice the use of `defthm-std`. It permits the body of the function `acl2-sqrt` to be opened up, since it restricts `x` to *standard* values. After opening up the body of `acl2-sqrt`, the theorems follows from the analogous results for `iter-sqrt` and the monotonicity of *standard-part*. From these two lemmas, it becomes clear that `acl2-sqrt` is *i-limited*:

```
(defthm limited-sqrt
  (implies (and (realp x)
                (<= 0 x)
                (i-limited x))
           (i-limited (acl2-sqrt x))))
```

This illustrates an approach that will be repeated in the next chapters. First an approximation function — `iter-sqrt` — is defined. It is shown that this function returns *i-limited* values for *i-limited* arguments. Then, the desired function — `acl2-sqrt` — is defined using `defun-std` by taking the *standard-part* of the approximation function. The definition is accepted, since the approximation function returns *i-limited* values for *standard* arguments, so its *standard-part* is *standard*. When a family of approximation functions exists, the specific approximation chosen is one that give values that are *i-close* to the desired values for *standard* arguments. In the case of `iter-sqrt`, the quality of the approximation was specified with the `epsilon` argument, so the way to ensure the approximated result was *i-close* to the true value was to choose an *i-small* value of `epsilon`. Later, examples using sequences will be shown, and in these cases the *i-close* value is selected by choosing an element of the sequence with *i-large* index. Once the function is defined, its properties are proved using `defun-std`. The properties need only be proved for *standard* values, and the result will follow for all values because of the transfer axiom.

Chapter 5

The Exponential Function

This chapter shows how the exponential function can be introduced into ACL2. The procedure follows the paradigm presented in chapter 4 for the square root function. First, an approximation to the exponential function is defined. This is based on the Taylor series to e^x. Then, it is shown that the approximation converges, hence using the principles of non-standard analysis it is possible to define the function e^x as the *standard-part* of the partial sum of the Taylor series up to an arbitrary *i-large* integer N. Important properties of the exponential function are also established. In section 5.2, it is shown that $e^{x+y} = e^x \cdot e^y$. The proof proceeds by examining the partial sums of the Taylor series approximation of e^{x+y} as well as the product of the partial sums of the approximations for e^x and e^y. Using this result, it is proved in section 5.3 that the function e^x is continuous.

5.1 Defining the Exponential Function in ACL2

The function e^x can be defined in ACL2 by considering approximations to e^x. The Taylor series approximation to e^x is given by

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots .$$

Let $T_n(x)$ be the partial sum of this Taylor series through the $\frac{x^n}{n!}$ term. That is, let $T_n(x)$ be defined as follows:

$$e^x \approx T_n(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}.$$

Since the Taylor series for e^x is convergent, $T_n(x)$ is *i-limited* for *i-limited* values of x, regardless of the value of n. In particular, if x is *i-limited*, $T_N(x)$ is *i-limited* for an *i-large* natural N. Therefore, *standard-part*$(T_N(x))$ is a *standard* number when x is *standard*, and it is possible to define $e^x =$ *standard-part*$(T_N(x))$.

The key step in this construction is the assertion that $T_n(x)$ is *i-limited* when x is *i-limited*. This can be proved by comparing $T_n(x)$ to a geometric series. However, this is complicated by the fact that the terms in $T_n(x)$ are complex, not necessarily real numbers. Since the numbers $x^n/n!$ are potentially complex, it is not possible to compare them directly with a term $a_0 \cdot r^n$ from a geometric series. What is required is a norm $||x||$ on the complex numbers. It is then possible to show that $||x^n/n!|| < ||a_0 \cdot r^n||$. Section 5.1.1 develops the theory of a suitable norm over the complex numbers. Section 5.1.2 develops the theory of geometric series, as well as some important lemmas, such as the comparison test. These results are used in section 5.1.3 to define the exponential function.

5.1.1 A Complex Norm

A *norm* $||x||$ is a real-valued function with the following properties:

- $||x||$ is real and $||x|| \geq 0$ for all values of x.
- $||0||$ is equal to 0. Moreover, if $||x|| = 0$, then x is necessarily equal to 0.
- $||x||$ obeys the triangle inequality. That is, $||x + y|| \leq ||x|| + ||y||$.

In addition, $||x||$ is called *magnitude-preserving* if $||x||$ is *i-small* (*i-limited*, or *i-large*) if and only if x is *i-small* (respectively, *i-limited*, or *i-large*).

Although any norm that satisfies the required properties will serve, this section will focus on the norm defined by $||a + bi|| = \sqrt{a^2 + b^2}$. In ACL2, this norm can be defined as follows:

```
(defun norm (x)
  (acl2-sqrt (+ (* (realpart x) (realpart x))
                (* (imagpart x) (imagpart x)))))
```

This definition uses the function `acl2-sqrt`, defined in chapter 4.

It is immediate that $||x||$ is a non-negative real since the function `acl2-sqrt` always returns a non-negative real result. It is also clear that $||x|| = 0$ if and only if $x = 0$. It is a less obvious but a well-known fact that $||x||$ obeys the triangle inequality. To see this, define the conjugate of a complex number $a + bi$ as $(a + bi)' = a - bi$. Observe that conjugates obey simple algebraic properties, including $(x + y)' = x' + y'$, $(xy)' = x'y'$, $x + x' = Re(x)$ where $Re(x)$ is the real part of x, and $x'' = x$. Since $||x||^2 = x \cdot x'$, it follows that $||x + y||^2 = (x + y)(x + y)' = (x + y)(x' + y') = xx' + xy' + x'y + yy' = xx' + xy' + (xy')' + yy' = xx' + yy' + 2Re(xy')$. Clearly, $Re(xy') \leq ||xy'|| = ||x|| \cdot ||y||$. Therefore, $||x+y||^2 \leq xx'+yy'+2||x||\cdot||y|| = ||x||^2+||y||^2+2||x||\cdot||y|| = (||x||+||y||)^2$. Since both sides of the inequality involved non-negative reals, it is possible to take square roots of both sides, proving that $||x + y|| \leq ||x|| + ||y||$ as required.

To recognize that $||x||$ is a magnitude-preserving norm, first notice that $\sqrt{x}$ is a magnitude-preserving function. That is, $\sqrt{x}$ is *i-small* if and only if x is *i-small*, and similarly for *i-limited* and *i-large*. Likewise, x^2 is a magnitude-preserving function. Since $a+bi$ is *i-small* if and only if both a and b are *i-small*, it follows that $a + bi$ is *i-small* if and only if $a^2 + b^2$ is *i-small*. Hence, $a + bi$ is *i-small* if and only if $||a + bi||$ is *i-small*. Similar arguments work for *i-limited* and *i-large* values of $a + bi$.

When x is *i-limited*, the *standard-part* of $||x||$ is given by $||\textit{standard-part}(x)||$. This follows from the fact that $\sqrt{\textit{standard-part}(y)} = \textit{standard-part}(\sqrt{y})$.

Several other lemmas about the norm $||x||$ will be needed in the following sections. Norms are idempotent; that is, $||\,||x||\,|| = ||x||$. Moreover, when x and y are non-negative reals, the norm is monotonic; if $x \leq y$, $||x|| \leq ||y||$. A very important lemma is that norms distribute over products. That is, $||xy|| = ||x|| \cdot ||y||$. Moreover, when a and b are positive reals, $||ax + bx|| = ||a|| \cdot ||x|| + ||b|| \cdot ||x||$. All of these results can be easily proved in ACL2 using nothing more than simple algebra.

5.1.2 Geometric Series

Mathematically, a real (complex) sequence is a function from the positive integers into the real (complex) numbers. A sequence is commonly written as the enumeration of its values, e.g., $a_1, a_2, a_3, \ldots$. Using this notation, the sequence is specified as $\{a_n\}$. The sequence is said to converge to a value A if the term a_n is ***i-close*** to A for all ***i-large*** integers n.

Informally, a series is the sum of all the terms in a sequence. Formally, a series is defined by the partial sums of a sequence. Given the sequence $\{a_n\}$, the partial sums are defined as the sequence $a_1, a_1 + a_2, a_1 + a_2 + a_3, \ldots$. If this sequence of partial sums converges to some value S, the series $\{a_n\}$ is said to converge to S.

It is natural to represent a sequence in ACL2 as a function mapping a positive integer argument i into the i^{th} element of the sequence, a_i. It is more convenient, however, to write the function to return the first i elements of the sequence, not just a_i. This allows properties of sequences — more precisely, properties of finite prefixes of sequences — to be written as first-order predicates. Instead of saying that the function `seq` is a geometric sequence, for example, it is possible to say that the sequence returned by `seq` is geometric. Note the shift from second-order to first-order logic: the expression `(geometricp seq)`, which is inadmissible in ACL2, is replaced by `(geometricp (seq n))`. Using this approach, properties about sequences can be stated as properties about lists, an area where ACL2 is particularly capable.

A sequence is geometric if the ratio of successive terms is constant. It can be written as $a_1, a_1 \cdot r, a_1 \cdot r^2, a_1 \cdot r^3, \ldots$, where a_1 is the initial element of the sequence and r is the constant ratio. The following ACL2 function tests whether a sequence is geometric:

```
(defun geometric-sequence-p (seq ratio)
  (if (consp seq)
      (if (consp (cdr seq))
          (and (acl2-numberp (car seq))
               (equal (* (car seq) ratio)
                      (car (cdr seq)))
               (geometric-sequence-p (cdr seq) ratio))
        (acl2-numberp (car seq)))
    nil))
```

The function expects two arguments, the sequence and the expected ratio. It is possible to generate a geometric sequence from its first element and constant ratio. The following function generates the first `nterms` elements of such a sequence:

```
(defun geometric-sequence-generator (nterms a1 ratio)
  (if (zp nterms)
      nil
    (cons a1
          (geometric-sequence-generator (1- nterms)
                                        (* a1 ratio)
                                        ratio))))
```

For example, the sequence generated by `(geometric-sequence-generator 3 7 1/2)` is `(7 7/2 7/4)`. It is a simple matter to prove that any sequence generated by this function is, in fact, geometric.

```
(defthm geometric-sequence-generator-is-geometric
   (implies (and (not (zp nterms))
                 (acl2-numberp x))
            (geometric-sequence-p
             (geometric-sequence-generator nterms x a)
             a)))
```

It is a well-known result that the sum of the first n elements of a geometric sequence is given by $\frac{a_1 - a_1 \cdot r^n}{1-r}$, where a_1 is the first element of the sequence and r is the constant ratio. The result follows because the sum S is given by $S = a_1 + a_1 \cdot r + a_1 \cdot r^2 + \cdots + a_1 \cdot r^{n-1}$. It follows that $rS = a_1 \cdot r + a_1 \cdot r^2 + \cdots + a_1 \cdot r^n$. Subtracting rS from S and simplifying, $S(1-r) = a_1 - a_1 \cdot r^n$, therefore $S = \frac{a_1 - a_1 \cdot r^n}{1-r}$. This argument can be easily formalized in ACL2, yielding the following theorem:

```
(defthm sumlist-geometric
  (implies (and (geometric-sequence-p seq ratio)
                (acl2-numberp ratio)
                (not (equal ratio 1)))
           (equal (sumlist seq)
                  (if (consp seq)
                      (/ (- (car seq)
                            (* ratio (last-elem seq)))
                         (- 1 ratio))
                    0))))
```

As their names suggest, the function `sumlist` adds up all the elements of a sequence, and `last-elem` returns the last element of a sequence. Since the last element of a geometric sequence can be derived from the first element, the constant ratio, and the length of the sequence, it is possible to find a simpler formula for the geometric sum. In particular, the last element of an geometric sequence with n elements is given by $a_1 \cdot r^{n-1}$, as proved by the following theorem:

```
(defthm last-geometric
 (implies (and (geometric-sequence-p seq ratio)
               (consp seq))
          (equal (last-elem seq)
                 (* (car seq)
                    (expt ratio (- (len seq) 1))))))
```

Combining these two theorems produces the classic result for the sum of a geometric sequence:

```
(defthm sumlist-geometric-useful
  (implies (and (geometric-sequence-p seq ratio)
                (acl2-numberp (car seq))
                (acl2-numberp ratio)
                (not (equal ratio 1)))
           (equal (sumlist seq)
                  (* (car seq)
                     (/ (- 1 (expt ratio (len seq)))
                        (- 1 ratio))))))
```

While geometric series are important in their own right, their significance to the exponential function is indirect. Consider the Taylor approximation to e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots$$

In particular, consider the successive terms $\frac{x^n}{n!}$ and $\frac{x^{n+1}}{(n+1)!}$. These differ by a factor of $\frac{x}{n+1}$. It is clear that the terms in the Taylor sequence after $\frac{x^n}{n!}$ are no larger than the terms in the geometric series with starting number $\frac{x^n}{n!}$ and constant ratio $\frac{x}{n+1}$. It should be possible to argue, therefore, that the Taylor series converges to some value less than the sum of the geometric series.

To formalize this argument, it is necessary to formalize the notion of “no larger” used above. This can be done by using the complex norm introduced in section 5.1.1. What is also needed is a version of the comparison test: if $||a_n|| \leq ||b_n||$ for all indices n, then $\sum_n ||a_n||$ converges if $\sum_n ||b_n||$ converges. When $\sum_n ||a_n||$ converges, the series is called absolutely convergent. Absolute convergence is a stronger property than simple convergence: a sequence is convergent whenever it is absolutely convergent.

The needed theory of absolute convergence can be readily developed in ACL2. First, the notion of the sum of the norms of a sequence is required; i.e., the value $\sum_n ||a_n||$ for a given sequence $\{a_n\}$:

```
(defun sumlist-norm (x)
  (if (consp x)
      (+ (norm (car x))
         (sumlist-norm (cdr x)))
    0))
```

From the definition, it is apparent that `sumlist-norm` is a non-negative real. That the `sumlist-norm` of a list is at least equal to the norm of the sum of the list is a simple generalization of the triangle inequality for norms:

```
(defthm norm-sumlist-<=-sumlist-norm
  (<= (norm (sumlist l))
      (sumlist-norm l)))
```

As was the case with the sum of a geometric series, it is possible to find a closed form solution for the `sumlist-norm` of a geometric sequence, provided the constant ratio is a real between 0 and 1. This restriction is necessary to ensure the term $1 - r^n$ is always positive. The result is similar to the sum of a geometric series:

```
(defthm sumlist-norm-real-geometric
  (implies (and (geometric-sequence-p seq ratio)
                (acl2-numberp (car seq))
                (realp ratio)
                (<= 0 ratio)
                (< ratio 1))
           (equal (sumlist-norm seq)
                  (* (norm (car seq))
                     (norm (/ (- 1 (expt ratio
                                         (len seq)))
                              (- 1 ratio)))))))
```

This theorem has some important consequences. When the first element of a geometric sequence is *i-limited*, it follows that the `sumlist-norm` is also *i-limited*, provided the constant ratio is not *i-close* to 1. This also holds when the first element of the sequence is *i-small*, in which case the `sumlist-norm` is also *i-small*.

The only remaining detail is the comparison test. The following function can be used to recognize when a sequence is bounded below another sequence:

```
(defun seq-norm-<= (x y)
  (if (consp x)
      (and (consp y)
           (<= (norm (car x)) (norm (car y)))
           (seq-norm-<= (cdr x) (cdr y)))
    t))
```

From the definition, it is simple to deduce that if a sequence is bounded by another, its `sumlist-norm` is bounded by the other's:

```
(defthm seq-norm-<=-sumlist-norm
  (implies (seq-norm-<= x y)
           (<= (sumlist-norm x)
                    (sumlist-norm y))))
```

An important consequence is that if the `sumlist-norm` of a sequence is *i-limited*, the `sumlist-norm` of any sequence that is bounded by it must also be *i-limited*. A similar result holds when a sequence has an *i-small* `sumlist-norm`. These two results combined form the non-standard analysis equivalent of the comparison test for convergence.

5.1.3 The Definition of the Exponential Function

The stage is almost set for the introduction of the exponential function into ACL2. It is a simple matter to define the Taylor series approximation to e^x in ACL2. Moreover, as argued in the previous section, the series $\frac{x^i}{i!} + \frac{x^{i+1}}{(i+1)!} + \frac{x^{i+2}}{(i+2)!} + \cdots$ is bounded by a geometric series with first element $\frac{x^i}{i!}$ and constant ratio $\frac{||x||}{(i+1)}$. However, this geometric series is not guaranteed to converge absolutely unless $||x||$ is less than $i+1$. So the first $||x||+1$ terms of the Taylor series approximation must be accounted for differently. The following argument suffices. When x is limited, so is $||x|| + 1$. Since each of the $\frac{x^n}{n!}$ terms is limited when x and n are limited, it follows that the first $||x|| + 1$ terms of the Taylor approximation is the sum of an *i-limited* number of *i-limited* numbers, so it must be *i-limited*.

The Taylor approximation can be defined as follows:

```
(defun taylor-exp-term (x counter)
  (* (expt x counter)
     (/ (factorial counter))))

(defun taylor-exp-list (nterms counter x)
  (if (or (zp nterms)
          (not (integerp counter))
          (< counter 0))
      nil
    (cons (taylor-exp-term x counter)
          (taylor-exp-list (1- nterms)
                           (1+ counter)
                           x))))
```

For example, `(taylor-exp-term 2 3)` is equal to $8/6 = 4/3$ and `(taylor-exp-list 4 0 2)` is `'(1 2 2 4/3)`. Since the term x^n is *i-limited* when x and n are *i-limited* and n is non-negative, it follows that `taylor-exp-term` is *i-limited* under these circumstances. In particular, the following theorem holds:

```
(defthm limited-taylor-exp-term
  (implies (and (<= 0 counter)
                (i-limited counter)
                (i-limited x))
           (i-limited (taylor-exp-term x counter))))
```

Since each term in the sum is *i-limited*, the sum of an *i-limited* number of terms is also *i-limited*. This shows that an *i-limited* prefix of the Taylor expansion adds up to an *i-limited* number:

```
(defthm taylor-exp-list-limited-up-to-limited-counter
  (implies (and (i-limited nterms)
                (integerp counter)
                (i-limited counter)
                (i-limited x))
           (i-limited (sumlist
                        (taylor-exp-list nterms
                                         counter
                                         x)))))
```

All that remains is to show that the remaining terms in the Taylor approximation really are bounded by a geometric sequence. The proof is simplified if a different definition of `taylor-exp-list` is used. In particular, consider the following function:

```
(defun taylor-exp-list-2 (nterms prev i x)
  (if (or (zp nterms)
          (not (integerp i))
          (< i 0))
      nil
    (cons prev
          (taylor-exp-list-2 (1- nterms)
                             (* prev (/ x (+ 1 i)))
                             (+ 1 i)
                             x))))
```

This definition makes it immediately apparent that the ratio of successive terms in the Taylor approximation to e^x is $\frac{x}{i+1}$. Simple induction verifies that this function is identical to `taylor-exp-list`. It is easy to show that this function is bounded above by a geometric sequence:

```
(defthm taylor-exp-list-2-seq-<=geom-sequence-generator
  (implies (and (<= (norm prev) (norm a1))
                (integerp i)
                (<= 0 i)
                (realp ratio)
                (<= (norm (/ x (+ 1 i))) (norm ratio)))
           (seq-norm-<= (taylor-exp-list-2 nterms
                                           prev
                                           i
                                           x)
                        (geometric-sequence-generator
                         nterms
```

```
                   a1
                   ratio))))
```

The only restriction on the geometric sequence is that the norm of its first element be no less than the norm of the first element of the Taylor series, and that the norm of its constant ratio be no less than the norm of $\frac{x}{i+1}$.

These theorems are sufficient to prove the Taylor sum of e^x is *i-limited* for *i-limited* values of x. The important lemma tying all these results together is the division of the terms in the Taylor sum into those with exponent less than $||x||$ and the remainder; the sum of both sublists is known to be *i-limited*, so their combined sum is also *i-limited*. In anticipation of the eventual definition of e^x in ACL2, it is convenient to phrase this split using the designated constant `i-large-integer`:

```
(defthm taylor-exp-list-split-for-limited
  (implies (and (i-limited x)
                (integerp counter)
                (<= 0 counter))
           (equal (taylor-exp-list (i-large-integer)
                                   counter
                                   x)
                  (append (taylor-exp-list
                            (next-integer
                             (next-integer (norm x)))
                            counter
                            x)
                           (taylor-exp-list
                            (- (i-large-integer)
                               (next-integer
                                (next-integer (norm x))))
```

```
                                        (+ counter
                                           (next-integer
                                            (next-integer (norm x))))
                                        x)))))
```

Trivially, it is now possible to show that the Taylor approximation of e^x is *i-limited*:

```
    (defthm taylor-exp-list-limited
      (implies (i-limited x)
               (i-limited
                (sumlist
                 (taylor-exp-list (i-large-integer) 0 x)))))
```

In fact, it is possible to derive a stronger result: the Taylor series converges absolutely.

```
    (defthm taylor-exp-list-norm-limited
      (implies (i-limited x)
               (i-limited
                (sumlist-norm
                 (taylor-exp-list (i-large-integer) 0 x)))))
```

The way is now paved for the definition of e^x in ACL2.

```
    (defun-std acl2-exp (x)
      (standard-part
       (sumlist (taylor-exp-list (i-large-integer) 0 x))))
```

This definition uses ACL2's new `defun-std` primitive, which allows a *standard* function to be defined implicitly by specifying its values only for *standard* arguments. In order for the definition to be accepted, it must be shown that for *standard* arguments, the function yields *standard* results. This follows, since the function body is of the form *standard-part*(S), where S is the Taylor approximation to e^x, known to be *i-limited* for *i-limited* values of x.

This definition is sufficient, but it leaves open the question of whether the function defined depends on the value of `i-large-integer`. To see that this is not the case, it is necessary to consider what would happen when some other positive *i-large* integer M is used instead of `i-large-integer`. Specifically, the difference between the two Taylor approximations must be *i-small*.

This is simple to visualize. Suppose N and M are positive *i-large* integers such that $N < M$, and suppose x is an *i-limited* number. The term $\frac{x^{N+1}}{(N+1)!}$ is necessarily *i-small*, because it is less than the term $x^k \cdot \left(\frac{x}{k+1}\right)^{N-k}$ for some *i-limited* k with $k > x$ (e.g., $k = \lceil x \rceil$). But this is the product of the *i-limited* number, x^k, and an *i-small* number, $\left(\frac{x}{k+1}\right)^{N-k}$, hence it is *i-small*. Therefore, the series $\frac{x^{N+1}}{(N+1)!} + \frac{x^{N+2}}{(N+2)!} + \cdots + \frac{x^N}{N!}$ is bounded by a geometric series with an *i-small* starting element and constant ratio with norm less than 1, implying the sum of this series is *i-small*.

This argument can be formalized in ACL2 in a manner similar to the proof of `taylor-exp-list-limited`. The result yields the following convergence theorem:

```
(defthm exp-convergent
  (implies (and (i-limited x)
                (integerp M) (<= 0 M) (i-large M)
                (integerp N) (<= 0 N) (i-large N))
           (i-close (sumlist (taylor-exp-list M 0 x))
                    (sumlist (taylor-exp-list N 0 x)))))
```

An analogous result holds for `sumlist-norm` instead of `sumlist`. This second form of the theorem will prove important in the subsequent development of the theory.

It is now apparent that the use of `i-large-integer` in the definition of e^x was inconsequential. In other words, `(acl2-exp x)` is precisely e^x.

5.2 The Exponent of a Sum: $e^{x+y} = e^x \cdot e^y$

An important property of the exponential function is that $e^{x+y} = e^x \cdot e^y$. This section presents a proof of this theorem in ACL2. The proof proceeds by considering finite Taylor approximations to e^{x+y} and $e^x \cdot e^y$. Consider $\sum_{i=0}^{n} \frac{(x+y)^i}{i!}$. The term $(x+y)^i$ can be expanded using the binomial theorem. The resulting terms can be simplified as follows:

$$\begin{aligned}
\sum_{i=0}^{n} \frac{(x+y)^i}{i!} &= \sum_{i=0}^{n} \frac{\sum_{j=0}^{i} \binom{i}{j} x^{i-j} y^j}{i!} \\
&= \sum_{j=0}^{n} \left(\sum_{i=j}^{n} \frac{\binom{i}{j} x^{i-j}}{i!} \right) \cdot y^j \\
&= \sum_{j=0}^{n} \left(\sum_{i=j}^{n} \frac{x^{i-j}}{j!\,(i-j)!} \right) \cdot y^j \\
&= \sum_{j=0}^{n} \left(\sum_{i=j}^{n} \frac{x^{i-j}}{(i-j)!} \right) \cdot \frac{y^j}{j!} \\
&= \sum_{j=0}^{n} \left(\sum_{i=0}^{n-j} \frac{x^{i}}{i!} \right) \cdot \frac{y^j}{j!}
\end{aligned}$$

This last term is very close to $\sum_{j=0}^{n} \left(\sum_{i=0}^{n} \frac{x^i}{i!} \right) \cdot \frac{y^j}{j!}$, which is the product of $\sum_{i=0}^{n} \frac{x^i}{i!}$ and $\sum_{j=0}^{n} \frac{y^j}{j!}$. The difference between these two terms is $\sum_{j=1}^{n} \left(\sum_{i=n-j+1}^{n} \frac{x^i}{i!} \right) \cdot \frac{y^j}{j!}$. Notice that all terms in this difference are of a high order; each term contains a large exponent on either x or y.

It is best to visualize the situation in a 2-dimensional grid. The columns of the grid correspond to the $\frac{x^i}{i!}$ terms, and the rows to the $\frac{y^j}{j!}$ terms, so the term $t_{i,j}$ in position (i,j) of the grid corresponds to the product $\frac{x^i}{i!} \cdot \frac{y^j}{j!}$. The product of $\sum_{i=0}^{n} \frac{x^i}{i!}$ and $\sum_{j=0}^{n} \frac{y^j}{j!}$ is the sum of all the terms $t_{i,j}$ in the grid. The sum $\sum_{i=0}^{n} \frac{(x+y)^i}{i!}$ corresponds to the sum of the terms "below" the triangular; i.e., those terms with $i + j \leq n$. Now, consider the sum of the $t_{i,j}$ terms in the bottom "quadrant" of this grid; that is, those terms with both i and j less than $n/2$. It is clear that $\sum_{i=0}^{n} \frac{(x+y)^i}{i!}$ lies between the sum of the terms in the bottom quadrant and those in the entire grid. But the sum of the terms in the bottom quadrant correspond to the product of $\sum_{i=0}^{n/2} \frac{y^i}{i!}$ and $\sum_{j=0}^{n/2} \frac{x^j}{j!}$. If n is ***i-large*** so is $n/2$, so the sum of the bottom quadrant and the entire grid are ***i-close*** to each other, since the Taylor series is known to converge. Since $\sum_{i=0}^{n} \frac{(x+y)^i}{i!}$ lies between these, it is also close to both of them. Hence, taking the ***standard-part*** of both expressions yields $e^{x+y} = e^x \cdot e^y$.

The following sections build the ACL2 theory necessary to formalize this argument. First, a proof of the binomial theorem is presented. Then some lemmas dealing with summations and nested summations are derived. With these pieces in place, the main theorem can be summarily proved.

5.2.1 The Binomial Theorem

The binomial theorem states that $(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$ for non-negative integer values of n. This section develops an ACL2 proof of this well-known result.

The binomial function $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ can be defined in ACL2 as follows:

```
(defun choose (k n)
  (if (and (integerp k) (integerp n) (<= 0 k) (<= k n))
      (/ (factorial n)
         (* (factorial k) (factorial (- n k))))
    0))
```

Intuitively, this function counts the number of different k-element subsets that can be formed from an n-element set — there are $\frac{n!}{(n-k)!}$ ways of choosing the k elements, and dividing this by $k!$ eliminates the duplicate counting of permutations. However, without this intuition, it is not immediately obvious that $\binom{n}{k}$ is always an integer.

However, it is possible to provide an alternative and well-known definition of $\binom{n}{k}$ that makes its properties more apparent. Consider the process of choosing a k-element subset S' from a set of n-elements S. A reasonable approach is to pick an arbitrary element $x_0 \in S$ and consider two possibilities. If x_0 is chosen as a member of S', then the remaining elements of S' can be chosen in $\binom{n-1}{k-1}$ different ways — i.e., $k-1$ elements remain to be chosen from $S - \{x_0\}$. Conversely, if x_0 is not chosen as a member of S', then all k elements of S' must be chosen from the $n-1$ elements in $S - \{x_0\}$. Therefore, it appears that $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

This key observation can be proved in ACL2 as follows:

```
(defthm choose-reduction
  (implies (and (integerp k)
                (integerp n)
                (< 0 k)
                (< k n))
           (equal (choose k n)
                  (+ (choose (1- k) (1- n))
                     (choose k (1- n))))))
```

It now becomes a simple matter to observe that `choose` is an integer function:

```
(defthm choose-is-non-negative-integer
  (and (integerp (choose k n))
       (<= 0 (choose k n))))
```

The theorem `choose-reduction` also holds the key for the binomial theorem. Consider the inductive case in the proof of the theorem. From the induction hypothesis, it follows that $(x+y)^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i} x^i y^{n-1-i}$. The binomial theorem can now be proved by the following argument:

$$\begin{aligned}
(x+y)^n &= (x+y) \cdot (x+y)^{n-1} \\
&= x \cdot (x+y)^{n-1} + y \cdot (x+y)^{n-1} \\
&= x \cdot \sum_{i=0}^{n-1} \binom{n-1}{i} x^i y^{n-1-i} + y \cdot \sum_{i=0}^{n-1} \binom{n-1}{i} x^i y^{n-1-i}
\end{aligned}$$

$$\begin{aligned}
&= \sum_{i=0}^{n-1} \binom{n-1}{i} x^{i+1} y^{n-1-i} + \sum_{i=0}^{n-1} \binom{n-1}{i} x^i y^{n-i} \\
&= \sum_{i=1}^{n} \binom{n-1}{i-1} x^i y^{n-i} + \sum_{i=0}^{n-1} \binom{n-1}{i} x^i y^{n-i} \\
&= \binom{n-1}{0} x^0 y^{n-0} + \sum_{i=1}^{n-1} \left(\binom{n-1}{i-1} + \binom{n-1}{i} \right) x^i y^{n-i} + \\
&\quad \binom{n-1}{n-1} x^n y^{n-n} \\
&= \binom{n}{0} x^0 y^{n-0} + \sum_{i=1}^{n-1} \binom{n}{i} x^i y^{n-i} + \binom{n}{n} x^n y^{n-n} \\
&= \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}
\end{aligned}$$

Other than `choose-reduction`, the only facts needed are $\binom{n}{0} = 0$ for all n and $\binom{n}{n} = 1$ for $n \neq 0$.

It remains only to define the binomial expansion of $(x + y)^n$ in ACL2. This can be done with the following function:

```
(defun binomial-expansion (x y k n)
  (if (and (integerp k) (integerp n) (<= 0 k) (<= k n))
      (cons (* (choose k n) (expt x k) (expt y (- n k)))
            (binomial-expansion x y (1+ k) n))
    nil))
```

The function `binomial-expansion` actually computes the value of $\sum_{i=k}^{n} \binom{n}{i} x^i y^{n-i}$. For example, the value of `(binomial-expansion 1 1 0 3)` is `'(1 3 3 1)` and that of `(binomial-expansion 1 2 0 3)` is `'(8 12 6 1)`. In ACL2, the binomial theorem can be expressed as follows:

```
(defthm binomial-theorem
  (implies (and (integerp n) (<= 0 n))
           (equal (expt (+ x y) n)
                  (sumlist
                   (binomial-expansion x y 0 n)))))
```

An interesting corollary follows from this theorem. It is not immediately obvious that $\sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} = \sum_{i=0}^{n} \binom{n}{i} x^{n-i} y^i$. However, this is a trivial observation using the binomial theorem, since it reduces to $(x+y)^n = (y+x)^n$.

5.2.2 Nested Summations

As seen earlier, the proof of $e^{x+y} = e^x \cdot e^y$ depends heavily on properties of nested sums. Particularly pertinent are lemmas that allow summations to be permuted; i.e., $\sum_i \sum_j a_{i,j} = \sum_j \sum_i a_{i,j}$. Moreover, some of the summations to follow will be triangular, for example in the sum $\sum_{i=0}^{n} \sum_{j=0}^{i} a_{i,j} = \sum_{j=0}^{n} \sum_{i=j}^{n} a_{i,j}$.

A generic sum can be captured in ACL2 using the `encapsulate` operator. Take, for example, the following definition that captures the value $a_{i,j}$ above:

```
(encapsulate
 ((binop (i j) t))

 (local
  (defun binop (i j)
    (+ i j)))

 (defthm binop-type-prescription
   (acl2-numberp (binop i j)))
)
```

The only constraint on `binop` is that it return a numeric value.

The sum of the terms $t_{i,j}$ can be computed in one of two ways, either by adding up the rows $t_{i,*}$ one at a time or by adding up the sum of each column $t_{*,j}$. Capturing the sum by adding up the totals in each row can be performed with the following pair of functions:

```
(defun row-expansion-inner (i j n)
  (if (and (integerp j) (integerp n) (<= 0 j) (<= j n))
      (cons (binop i j)
            (row-expansion-inner i (1+ j) n))
    nil))

(defun row-expansion-outer (i m n)
  (if (and (integerp i) (integerp m) (<= 0 i) (<= i m))
      (cons (sumlist (row-expansion-inner i 0 n))
            (row-expansion-outer (1+ i) m n))
    nil))
```

Notice that `row-expansion-inner` collects all the values `(binop i j)` for a fixed `i`. Hence, the `sumlist` of the `row-expansion-inner` is the sum of the terms in that row. Similarly, `row-expansion-outer` collects the sums of all the rows, hence its `sumlist` will be the sum of all the terms.

The two functions `col-expansion-inner` and `col-expansion-outer` are analogous to the functions defined above, but collecting the elements $t_{i,j}$ a column at a time, instead of a row at a time. It is straight-forward to prove that `col-expansion-outer` computes the same sum as `row-expansion-outer`:

```
(defthm ok-to-swap-inner-outer-sums
  (equal (sumlist (row-expansion-outer 0 m n))
         (sumlist (col-expansion-outer 0 m n))))
```

The treatment of triangular summations is similar. The following function defines a "lower-triangular" summation, expanding the elements a row at a time:

```
(defun row-expansion-outer-lt (i m n)
  (if (and (integerp i) (integerp m) (<= 0 i) (<= i m))
      (cons (sumlist (row-expansion-inner
                      i 0 (if (< i n) i n)))
            (row-expansion-outer-lt (1+ i) m n))
    nil))
```

Notice how `row-expansion-outer-lt` uses the same function to add up the elements in a given row; the only difference is in the number of columns assumed in the row, which now changes from row to row. A similar function adds the values a column at a time.

To prove that these functions compute the same value, it is possible to use `ok-to-swap-inner-outer-sums`, the previous result about arbitrary sums. The trick is to find a suitable function to take the place of `binop` above. The following function does as required:

```
(defun lt-binop (i j)
  (if (< i j)
      0
    (binop i j)))
```

This is a classic example of the power of ACL2's `encapsulate` primitive in tandem with functional instantiation hints.

The results above deal with arbitrary ranges for the rows and columns being added. An important special case occurs when these ranges are equal, as in $\sum_{i=0}^{n} \sum_{j=0}^{n} a_{i,j}$. It is easy to derive special results for this case as instances of the more generic theorems.

Besides the main results above, there are a number of other useful lemmas about summations. For example, scalars can be factored out of summations without altering the value of the sum; i.e., $\sum_i c \cdot a_i = c \cdot \sum_i a_i$. These lemmas are easy to prove in ACL2. It is convenient to prove them once in a generic setting — i.e., using `encapsulate`.

5.2.3 Proving $e^{x+y} = e^x \cdot e^y$

In this section, the informal argument given in the beginning of section 5.2 is formalized in ACL2. The proof will follow the overall plan given there almost exactly.

The proof begins with the following definition of the sum $\sum_{k=0}^{n} \frac{(x+y)^k}{k!}$:

```
(defun binomial-over-factorial-unswapped (x y k n)
  (if (and (integerp k) (integerp n) (<= 0 k) (<= k n))
      (cons (/ (sumlist (binomial-expansion x y 0 k))
               (factorial k))
            (binomial-over-factorial-unswapped
             x y (1+ k) n))
    nil))
```

For example, `(binomial-over-factorial-unswapped 1 0 0 3)` is equal to `'(1 1 1/2 1/6)` and `(binomial-over-factorial-unswapped 1 1 0 3)` is equal to `'(1 2 2 4/3)`. Since the Taylor expansion of e^x is given as $\sum_{i=0}^{n} \frac{x^i}{i!}$, it follows that the sum above is equal to the Taylor expansion of e^{x+y}:

```
(defthm exp-x+y-binomial-unswapped-expansion
  (implies (and (integerp nterms)
                (<= 0 nterms)
                (integerp counter)
                (<= 0 counter))
           (equal (taylor-exp-list nterms
                                   counter
                                   (+ x y))
                  (binomial-over-factorial-unswapped
                   x
                   y
```

```
                        counter
                        (1- (+ nterms counter))))))
```

In the sequel, it is convenient to expand $(y + x)^i$ instead of $(x + y)^i$. This leads to the following alternative definition of $\sum_{k=0}^{n} \frac{(x+y)^k}{k!}$:

```
(defun binomial-over-factorial (x y k n)
  (if (and (integerp k) (integerp n) (<= 0 k) (<= k n))
      (cons (/ (sumlist (binomial-expansion y x 0 k))
               (factorial k))
            (binomial-over-factorial x y (1+ k) n))
    nil))
```

From `binomial-sum-commutes`, it follows that `binomial-over-factorial-unswapped` is the same as `binomial-over-factorial`. Therefore, the term containing `binomial-over-factorial-unswapped` can be replaced with an equivalent `binomial-over-factorial` term in `exp-x+y-binomial-unswapped-expansion`.

The function `binomial-over-factorial` follows the pattern of a nested sum; it sums the values of various inner sums. Therefore, the theorems developed in section 5.2.2 apply, as long as the function `binomial-over-factorial` is defined to match the constrained functions defined there. This can be done with the following pair of functions:

```
(defun binomial-over-factorial-inner-sum (x y j i)
  (if (and (integerp i) (integerp j) (<= 0 j) (<= j i))
      (cons (/ (* (choose j i)
                  (expt x (- i j)) (expt y j))
               (factorial i))
            (binomial-over-factorial-inner-sum
             x y (1+ j) i))
```

```
        nil))

    (defun binomial-over-factorial-outer-sum (x y i n)
      (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
          (cons (sumlist
                   (binomial-over-factorial-inner-sum
                    x y 0 i))
                (binomial-over-factorial-outer-sum
                 x y (1+ i) n))
        nil))
```

The first function collects all the terms in the binomial expansion of $\frac{(x+y)^i}{i!}$ while the second collects the sum of these expansions. Notice, the two functions follow the pattern of a triangular nested sum, as discussed in section 5.2.2. It is easily noted that `binomial-over-factorial-outer-sum` is the same function as `binomial-over-factorial`.

The next step simplifies the terms in `binomial-over-factorial-inner-sum` using the identity $\frac{\binom{i}{j}}{i!} = \frac{1}{j!(i-j)!}$. This yields the following function:

```
    (defun inner-sum-1 (x y j i n)
      (if (and (integerp j) (integerp n) (<= 0 j) (<= j n))
          (cons (/ (* (expt x (- i j)) (expt y j))
                   (* (factorial j) (factorial (- i j))))
                (inner-sum-1 x y (1+ j) i n))
        nil))
```

Note, `inner-sum-1` is the same function as `binomial-over-factorial-inner-sum`. An outer sum equivalent to `binomial-over-factorial-outer-sum` using `inner-sum-1` instead of `binomial-over-factorial-inner-sum` will compute the same value as `binomial-over-factorial-outer-sum`. Such a function can be defined as follows:

```
(defun outer-sum-1 (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (cons (sumlist (inner-sum-1 x y 0 i i))
            (outer-sum-1 x y (1+ i) n))
    nil))
```

The next step in the proof is crucial. The inner sum `inner-sum-1` collects the terms of the binomial expansion of $\frac{(x+y)^i}{i!}$. The following function collects all the terms in these expansions containing y^j for a given j. Following the intuition developed in section 5.2.2, this amounts to adding the values a column at a time instead of a row at a time. The function can be defined as follows:

```
(defun inner-sum-2 (x y i j n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (cons (/ (* (expt x (- i j)) (expt y j))
               (* (factorial j) (factorial (- i j))))
            (inner-sum-2 x y (1+ i) j n))
    nil))
```

The function `outer-sum-2` collects the inner sums generated by `inner-sum-2`:

```
(defun outer-sum-2 (x y j n)
  (if (and (integerp j) (integerp n) (<= 0 j) (<= j n))
      (cons (sumlist (inner-sum-2 x y j j n))
            (outer-sum-2 x y (1+ j) n))
    nil))
```

That `outer-sum-2` returns the same values as `outer-sum-1` follows from the lemma `ok-to-swap-inner-outer-expansions-lt-m=n` proved in section 5.2.2.

The term $\frac{y^j}{j!}$ appearing in `inner-sum-2` does not depend on the value of i, which is the index of the inner sum. Therefore, it can be factored out of the sum. This observation leads to the definition of `inner-sum-3` and the corresponding `outer-sum-3`:

```
(defun inner-sum-3 (x i j n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (cons (/ (expt x (- i j)) (factorial (- i j)))
            (inner-sum-3 x (1+ i) j n))
    nil))


(defun outer-sum-3 (x y j n)
  (if (and (integerp j) (integerp n) (<= 0 j) (<= j n))
      (cons (/ (* (sumlist (inner-sum-3 x j j n))
                  (expt y j))
            (factorial j))
          (outer-sum-3 x y (1+ j) n))
    nil))
```

Using the lemma `factor-constant-from-expansion`, it follows that `outer-sum-3` computes the same function as `outer-sum-2`.

To complete the argument, it is only necessary to recognize `inner-sum-3` as a specific portion of the Taylor expansion of e^x:

```
(defthm taylor-exp-list-is-inner-sum-3
  (implies (and (integerp i)
                (integerp j) (<= 0 j) (<= j i)
                (integerp n))
           (equal (inner-sum-3 x i j n)
                  (taylor-exp-list (1+ (- n i))
                                   (- i j)
                                   x))))
```

This leads to a final definition of the outer sum, directly invoking `taylor-exp-list` as the inner sum:

```
(defun exp-x-y-k-n (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (cons (* (sumlist
                 (taylor-exp-list (1+ (- n i)) 0 x))
                (taylor-exp-term y i))
             (exp-x-y-k-n x y (1+ i) n))
    nil))
```

It is an obvious corollary of `taylor-exp-list-is-inner-sum-3` that this function is the same as `outer-sum-3`. Combining all the equalities results in the following main theorem:

```
(defthm exp-k-n-sum-simplification
  (implies (and (integerp nterms) (<= 0 nterms))
           (equal (sumlist
                    (taylor-exp-list nterms 0 (+ x y)))
                   (sumlist
                    (exp-x-y-k-n x y 0 (1- nterms))))))
```

This theorem formalizes the argument that $\sum_{i=0}^{n} \frac{(x+y)^i}{i!} = \sum_{j=0}^{n} \left(\sum_{i=0}^{n-j} \frac{x^i}{i!} \right) \cdot \frac{y^j}{j!}$ informally presented in the beginning of section 5.2.

The product of the Taylor expansions of e^x and e^y can be computed as follows:

```
(defun exp-x-*-exp-y-n (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (cons (* (sumlist (taylor-exp-list (1+ n) 0 x))
                (taylor-exp-term y i))
             (exp-x-*-exp-y-n x y (1+ i) n))
```

```
        nil))
```

It is easy to verify that the `sumlist` of this function does in fact return $\left(\sum_{i=0}^{n} \frac{x^i}{i!}\right) \cdot \left(\sum_{j=0}^{n} \frac{y^j}{j!}\right)$.

```
    (defthm exp-x-*-exp-y-n-=-exp-x-n-*-exp-y-n
      (implies (and (integerp nterms)
                    (<= 0 nterms))
               (equal (* (sumlist
                           (taylor-exp-list nterms 0 x))
                          (sumlist
                           (taylor-exp-list nterms 0 y)))
                       (sumlist
                        (exp-x-*-exp-y-n x y 0
                                         (1- nterms))))))
```

Clearly, the functions `exp-x-y-k-n` and `exp-x-*-exp-y-n` are very similar. If it is the case that their difference is *i-small* for arbitrary *i-large* values of n, then it will follow that the Taylor approximation of e^{x+y} is *i-close* to the product of the approximations for e^x and e^y and therefore that $e^{x+y} = e^x \cdot e^y$. This difference can be computed using the following function:

```
    (defun prod-sum-delta (x y i n)
      (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
          (cons (* (sumlist
                     (taylor-exp-list i (1+ (- n i)) x))
                    (taylor-exp-term y i))
                (prod-sum-delta x y (1+ i) n))
        nil))
```

This function can be simplified by pushing the term $\frac{y^i}{i!}$ into the inner sum. The result is the following definition:

```
(defun prod-sum-delta-2 (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (append (mult-scalar
                (taylor-exp-list i (1+ (- n i)) x)
                (taylor-exp-term y i))
               (prod-sum-delta-2 x y (1+ i) n))
    nil))
```

The function `mult-scalar` simply multiplies all elements of a list by the given scalar value. To simplify matters, it is convenient to pad the inner sums with zeros. The function `taylor-exp-list-3` returns the same values as `taylor-exp-list`, but it adds a zero in place of all the $\frac{x^i}{i!}$ terms for i below a given value:

```
(defun taylor-exp-list-3 (nterms counter llimit x)
  (if (or (zp nterms)
          (not (integerp counter))
          (< counter 0))
      nil
    (cons (if (< counter llimit)
               0
             (taylor-exp-term x counter))
          (taylor-exp-list-3 (1- nterms)
                             (1+ counter)
                             llimit
                             x))))
```

The connection between the functions `taylor-exp-list` and `taylor-exp-list-3` is demonstrated by the following theorem; a partial sum of `taylor-exp-list` can be replaced by a full — that is, starting at index 0 — sum using `taylor-exp-list-3`:

```
(defthm taylor-exp-list-=-taylor-exp-list-3
  (implies (and (integerp counter)
                (integerp nterms)
                (<= 0 counter)
                (<= 0 nterms))
           (equal (sumlist
                   (taylor-exp-list nterms counter x))
                  (sumlist
                   (taylor-exp-list-3 (+ nterms counter)
                                      0
                                      counter
                                      x)))))
```

This leads to a redefinition of `prod-sum-delta`, using `taylor-exp-list-3` instead of `taylor-exp-list`:

```
(defun prod-sum-delta-3 (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (append (mult-scalar
               (taylor-exp-list-3 (1+ n)
                                  0
                                  (1+ (- n i))
                                  x)
               (taylor-exp-term y i))
              (prod-sum-delta-3 x y (1+ i) n))
    nil))
```

A similar process defines the function `exp-x-*-exp-y-n-2` which pushes the the term $\frac{y^i}{i!}$ into the inner sum:

```
(defun exp-x-*-exp-y-n-2 (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (append (mult-scalar (taylor-exp-list (1+ n) 0 x)
                           (taylor-exp-term y i))
              (exp-x-*-exp-y-n-2 x y (1+ i) n))
    nil))
```

Clearly, this function is simply a different version of `exp-x-*-exp-y-n`.

It remains to show that `prod-sum-delta-3` is *i-small*. Intuitively, this is possible by showing that `prod-sum-delta-3` is bounded by an *i-small* number. Recall, the value of `prod-sum-delta-3` can be thought of as the sum of the terms $t_{i,j} = \frac{x^i}{i!} \cdot \frac{y^j}{j!}$ in an $n \times n$ matrix that lie above the diagonal; that is, the some of those terms $t_{i,j}$ for which $i + j > n$. For *i-large* values of n, the sum of all the $t_{i,j}$ terms is *i-close* to $e^x \cdot e^y$. But if n is *i-large*, so is $n/2$, and hence the sum of all the $t_{i,j}$ terms is *i-close* to the sum of just the $t_{i,j}$ terms for which $i < n/2$ and $j < n/2$. In other words, the sum of all the terms $t_{i,j}$ with $i \geq n/2$ or $j \geq n/2$ must be *i-small*, and these terms include all the terms in `prod-sum-delta-3`. It is tempting, therefore, to conclude that the sum `prod-sum-delta-3` is less than the sum of all the terms $t_{i,j}$ with $i \geq n/2$ or $j \geq n/2$ and therefore *i-small*. However, for this crucial step to be true, it is necessary to ensure that all the $t_{i,j}$ are non-negative reals. The way to do this is to add up not the $t_{i,j}$ terms themselves, but their norm. What remains is to verify the argument outlined here with the sum of the norm of the $t_{i,j}$ terms.

The terms $t_{i,j}$ for which $i > n/2$ or $j > n/2$ can be collected as follows:

```
(defun exp-x-*-exp-y-n-3 (x y i n)
  (if (and (integerp i) (integerp n) (<= 0 i) (<= i n))
      (append (mult-scalar
               (if (< i (next-integer (/ n 2)))
                   (taylor-exp-list-3
                    (1+ n)
                    0
                    (next-integer (/ n 2))
                    x)
                 (taylor-exp-list-3 (1+ n) 0 0 x))
               (taylor-exp-term y i))
              (exp-x-*-exp-y-n-3 x y (1+ i) n))
    nil))
```

It is expected that for *i-large* values of n, the sum of the terms in `exp-x-*-exp-y-n-3` is *i-small*. To see this, consider the `sumlist-norm` of `taylor-exp-list-3`. Recall that `taylor-exp-list-3` returns a list generated by replacing a prefix of the analogous `taylor-exp-list` with zeros. It follows, therefore, that the sum of the norm of the terms in `taylor-exp-list-3` is the difference of the sum of the norm of the terms in the analogous `taylor-exp-list` minus the norm of the prefix replaced.

```
    (defthm sumlist-norm-taylor-exp-list
      (implies (and (integerp m) (<= 0 m)
                    (integerp i) (<= 0 i)
                    (integerp n) (<= m n))
               (equal (sumlist-norm
                       (taylor-exp-list-3 n i m x))
                      (- (sumlist-norm
                          (taylor-exp-list n i x))
                         (sumlist-norm
                          (taylor-exp-list (- m i)
                                           i
```

```
                                                x))))))
```

From this lemma, it is easy to see that the `sumlist-norm` of `exp-x-*-exp-y-n-3` is simply the difference of the `sumlist-norm` of the `exp-x-*-exp-y-n-2` given values of n and $n/2$:

```
(defthm sumlist-norm-exp-x-*-exp-y-n-3
  (implies (and (integerp i) (integerp n)
                (<= 0 i) (<= 2 n))
           (equal (sumlist-norm
                   (exp-x-*-exp-y-n-3 x y i n))
                  (- (sumlist-norm
                      (exp-x-*-exp-y-n-2 x y i n))
                     (sumlist-norm
                      (exp-x-*-exp-y-n-2
                       x
                       y
                       i
                       (1- (next-integer (/ n 2)))))))))
```

The lemma `exp-x-*-exp-y-n-=-exp-x-n-*-exp-y-n` shows that the `sumlist` of a `exp-x-*-exp-y-n-2` is the product of the separate Taylor series for e^x and e^y. It is an important lemma that this claim holds when `sumlist-norm` is used instead of `sumlist`:

```
(defthm sumlist-norm-exp-x-*-exp-y-n-2
  (implies (and (integerp i) (integerp n)
                (<= 0 i) (<= 0 n))
           (equal (sumlist-norm
                   (exp-x-*-exp-y-n-2 x y i n))
```

```
                  (* (sumlist-norm
                      (taylor-exp-list (1+ n) 0 x))
                     (sumlist-norm
                      (taylor-exp-list (- (1+ n) i)
                                       i
                                       y))))))
```

It is now a simple matter to verify that `exp-x-*-exp-y-n-3` is *i-small*:

```
(defthm sumlist-norm-exp-x-*-exp-y-n-3-small
  (implies (and (integerp n) (<= 2 n)
                (i-limited x) (i-limited y) (i-large n))
           (i-small (sumlist-norm
                     (exp-x-*-exp-y-n-3 x y 0 n)))))
```

To show that `prod-sum-delta-3` is also *i-small*, it is only necessary to show that it is bounded by `exp-x-*-exp-y-n-3`:

```
(defthm prod-sum-delta-3-seq-<=-exp-x-*-exp-y-n-3
  (implies (<= 2 n)
           (seq-norm-<= (prod-sum-delta-3 x y i n)
                        (exp-x-*-exp-y-n-3 x y i n))))
```

At this time, it is trivial to conclude that the `sumlist-norm` of `prod-sum-delta` is *i-small*, and hence so is its `sumlist`. What this means is that the difference between the Taylor approximation of e^{x+y} and the product of the Taylor approximations to e^x and e^y is *i-small*:

```
(defthm expt-x-*-expt-y-n---exp-x-y-k-n-small
  (implies (and (integerp nterms) (<= 0 nterms)
                (i-limited x) (i-limited y)
                (i-large nterms))
           (i-small (- (* (sumlist
                           (taylor-exp-list nterms 0 x))
                          (sumlist
                           (taylor-exp-list nterms 0
                                                   y)))
                       (sumlist
                        (taylor-exp-list nterms
                                         0
                                         (+ x y)))))))
```

In turn, this means that the two sums are *i-close* to each other, and hence they have the same *standard-part*. To conclude the proof, it is only necessary to use `defthm-std` to transfer the proof of the *i-large* Taylor sums to the actual exponential function:

```
(defthm-std exp-sum
  (implies (and (acl2-numberp x)
                (acl2-numberp y))
           (equal (acl2-exp (+ x y))
                  (* (acl2-exp x) (acl2-exp y)))))
```

5.3 The Continuity of the Exponential Function

The continuity of e^x follows almost directly from the theorem `exp-sum`. A function f is continuous if given any standard point x and *i-small* number ϵ, $f(x+\epsilon)$ is *i-close* to $f(x)$. Consider $e^{x+\epsilon}$. This is equal to $e^x \cdot e^\epsilon$, so it is *i-close* to e^x if e^ϵ is *i-close* to 1.

So it is sufficient to show that for *i-small* ϵ, e^ϵ is *i-close* to 1. Consider the Taylor approximation of $e^\epsilon = 1 + \epsilon + \frac{\epsilon^2}{2!} + \cdots$. It is clear that the sum $\epsilon + \frac{\epsilon^2}{2!} + \cdots$ must be *i-small*. The trick is to show that the terms $\sum_{i=2}^{n} \frac{\epsilon^i}{i!}$ are bounded by $\sum_{i=2}^{n} \frac{\epsilon}{2^{i-1}}$. The latter sum can be computed, since it is the sum of a geometric sequence. In particular, this sum must be less than ϵ. Therefore, the Taylor expansion of e^ϵ is within 2ϵ of 1; that is to say, it is *i-close* to 1.

The first step is to show that for *i-small* ϵ, $\epsilon^n \leq \epsilon$:

```
(defthm lemma-1
  (implies (and (< (norm x) 1)
                (integerp n)
                (<= 2 n))
           (<= (norm (expt x n)) (norm x))))
```

Moreover, the factorial terms $n!$ are larger than 2^{n-1}:

```
(defthm lemma-2
  (implies (and (integerp n)
                (<= 2 n))
           (<= (norm (expt-2-n (+ -1 n)))
               (norm (factorial n)))))
```

In this theorem, the function `expt-2-n` computes the value of 2^n. Together, these theorems show how the magnitude of the terms $\frac{\epsilon^i}{i!}$ can be bounded by $\frac{\epsilon}{2^{i-1}}$:

```
(defthm lemma-4
  (implies (and (< (norm x) 1)
                (integerp n)
                (<= 2 n))
           (<= (norm (taylor-exp-term x n))
               (* (norm x)
                  (/ (norm (expt-2-n (+ -1 n))))))))
```

With these theorems, it is possible to give a precise bound for the norm of the Taylor approximation of e^ϵ:

```
(defthm lemma-6
  (implies (and (< (norm x) 1)
                (integerp n)
                (<= 2 n))
           (<= (sumlist-norm
                (taylor-exp-list nterms n x))
               (* (norm x)
                  (sumlist-norm
                   (expt-2-n-list nterms n))))))
```

Here, the function `expt-2-n-list` returns the list of terms $\frac{1}{2^i}$, and the function `expt-2-n-list-norm` returns the sum of the norm of these terms. It is easy to see that the sum of the terms in `expt-2-n-list` can add up to no more than one:

```
(defthm sumlist-expt-2-n-list-norm-best
  (implies (and (not (zp nterms))
                (integerp n)
                (<= 2 n))
           (<= (sumlist (expt-2-n-list-norm nterms n))
               1)))
```

Combining this theorem with `lemma-6` finds a bound for all the terms in the sequence $\sum_{i=2}^{n} ||\frac{\epsilon^i}{i!}||$:

```
(defthm lemma-15
  (implies (and (< (norm x) 1)
                (not (zp nterms))
                (integerp n)
                (<= 2 n))
           (<= (sumlist-norm
                (taylor-exp-list nterms n x))
               (norm x))))
```

It is therefore possible to establish that the difference between the Taylor approximation of e^ϵ and 1 can be no more than twice ϵ:

```
(defthm lemma-28
  (implies (and (standard-numberp x)
                (< (norm x) 1))
           (<= (norm
                (standard-part (+ -1
                                  (sumlist
                                   (taylor-exp-list
                                    (i-large-integer)
                                    0
                                    x)))))
               (+ (norm x) (norm x)))))
```

From this, a simple application of the transfer principle shows that $e^\epsilon - 1$ is no more than 2ϵ:

```
(defthm-std lemma-30
  (implies (and (acl2-numberp x)
                (< (norm x) 1))
           (<= (norm (+ -1 (acl2-exp x)))
               (+ (norm x) (norm x)))))
```

There is a subtlety here, however. The transfer principle only applies to classical formulas; that is, it applies only to formulas with classical predicates and *standard* parameters. What this means is that it would be impossible to use the transfer principle to a formula that referred directly to ϵ, since this is a non-*standard* number. That is the reason that the hypothesis requires that $||x|| < 1$ rather than that x be *i-small*. The limitation of the transfer principle to classical formulas forces the use of this type of subterfuge often.

It is a simple matter to apply `lemma-30` to the case when x is *i-small*:

```
(defthm lemma-35
  (implies (and (acl2-numberp x)
                (i-small x))
           (i-small (+ -1 (acl2-exp x)))))
```

In these cases, it is possible to conclude that the norm of $1 - e^x$ is *i-small*, and so $1 - e^x$ is also *i-small*. Putting this result together with the theorem `exp-sum` yields the continuity of e^x:

```
(defthm exp-continuous-2
  (implies (and (standard-numberp x)
                (i-close x y))
           (i-close (acl2-exp x) (acl2-exp y))))
```

The function e^x will play an important role in the sequel. It will be used to define the trigonometric functions. Its properties, notably the theorem `exp-sum`, will be used to prove the usual trigonometric identities, such as $\sin(2x) = 2\sin(x)\cos(x)$. Moreover, the continuity of e^x will play a crucial role in the definition of π, which can be found as twice the value of the first positive zero of cosine. That such a zero exists is guaranteed by the intermediate value theorem, which applies only to continuous functions.

Chapter 6

Trigonometric Functions

This chapter develops a small part of the theory of trigonometry. The trigonometric functions themselves are defined using the exponential function. Notice that the resulting trigonometric functions are general complex functions; however, for real arguments, their values follow the familiar constraints. For example, it is possible to show that for real x, $\sin(x)$ and $\cos(x)$ are both real and $\sin^2(x) + \cos^2(x) = 1$. The definition of π is particularly interesting, since it derives from the continuity of the cosine function and the intermediate value theorem. Moreover, the theory of alternating series comes into play in showing that $\cos(0) = 1$ and $\cos(2) < 0$, and hence cosine has a zero between 0 and 2 — that zero is necessarily equal to $\pi/2$. A large part of trigonometry — the area concerned with trigonometric identities — is particularly well-suited to mechanical verification using a rewriting theorem prover. This chapter concludes with a demonstration of how ACL2 can prove many such identities.

6.1 Defining the Trigonometric Functions in ACL2

6.1.1 The Definition of Sine and Cosine

The definition of the trigonometric functions in ACL2 follows from the following theorems in analysis:

$$\begin{aligned} \sin(x) &= \frac{e^{ix} - e^{-ix}}{2i} \\ \cos(x) &= \frac{e^{ix} + e^{-ix}}{2} \end{aligned}$$

This suggests the following definitions in ACL2:

```
(defun acl2-sine (x)
  (/ (- (acl2-exp (* #c(0 1) x))
        (acl2-exp (* #c(0 -1) x)))
     #c(0 2)))

(defun acl2-cosine (x)
  (/ (+ (acl2-exp (* #c(0 1) x))
        (acl2-exp (* #c(0 -1) x)))
     2))
```

From sine and cosine, it is straightforward to define the remaining trigonometric functions:

```
(defmacro acl2-tangent (x)
  ‘(/ (acl2-sine ,x) (acl2-cosine ,x)))

(defmacro acl2-cotangent (x)
  ‘(/ (acl2-cosine ,x) (acl2-sine ,x)))

(defmacro acl2-secant (x)
  `(/ (acl2-cosine ,x)))

(defmacro acl2-cosecant (x)
  `(/ (acl2-sine ,x)))
```

The sine and cosine functions can also be defined from their Taylor approximation, as was the exponential function. In fact, it is possible to show that the two definitions are equivalent. Consider first this approximation to the Taylor series for $\sin(x)$:

```
(defun taylorish-sin-list (nterms counter sign x)
  (if (or (zp nterms)
          (not (integerp counter))
          (< counter 0))
      nil
    (if (nat-even-p counter)
        (cons 0 (taylorish-sin-list (1- nterms)
                                    (1+ counter)
                                    sign
                                    x))
      (cons (* sign
               (expt x counter)
               (/ (factorial counter)))
            (taylorish-sin-list (1- nterms)
                                (1+ counter)
                                (- sign)
                                x)))))
```

The function `nat-even-p` tests whether its argument is a natural even number. This sequence generates the complete Taylor series for $\sin(x) = 0 + x - 0 - \frac{x^3}{3!} + \cdots$. A similar function produces the Taylor sequence for $\cos(x)$. It is a simple matter to show that the function `taylorish-sin-list` generates the same results as the `acl2-sine` function:

```
(defthm taylorish-sin-valid
  (implies (standard-numberp x)
           (equal (acl2-sine x)
                  (standard-part
                   (sumlist
                    (taylorish-sin-list
                     (i-large-integer)
                     0
                     1
                     x))))))
```

One advantage of the Taylor definition of $\sin(x)$ is that it makes it immediately obvious that $\sin(x)$ is real for real values of x. Thus, it is possible to prove the following important theorems:

```
(defthm-std realp-sine
  (implies (realp x)
           (realp (acl2-sine x))))

(defthm-std realp-cosine
  (implies (realp x)
           (realp (acl2-cosine x))))
```

The Taylor series defined by `taylorish-cos-list` is of the form $1+0-\frac{x^2}{2!}-0+\frac{x^4}{4!}+\cdots$. If the zeros are eliminated, then this series is clearly alternating, so it is possible to estimate values of the cosine function. To this purpose, consider the following function, which simply removes the zeros from `taylorish-sin-list` and `taylorish-cos-list`:

```
(defun taylor-sincos-list (nterms counter sign x)
  (if (or (zp nterms)
          (not (integerp counter))
          (< counter 0))
      nil
    (cons (* sign
             (expt x counter)
             (/ (factorial counter)))
          (taylor-sincos-list (nfix (- nterms 2))
                              (+ counter 2)
                              (- sign)
                              x))))
```

It is easy to show that this function computes the same values as `taylor-sin-list` when the initial value of `counter` is odd and `taylor-cos-list` when the initial value of `counter` is even.

A particularly relevant value is $\cos(2)$. It is clear from the definition of cosine that $\cos(0)$ is equal to 1. From an analysis of the alternating sequence `taylor-sincos-list`, it will follow that $\cos(2) < 0$. According to the intermediate-value theorem, this implies the cosine function has a root between 0 and 2, say x_0. This root is unique, since it can be shown that for $0 < x < x_0$, $\cos(x) > 0$ and for $x_0 < x < 2 \cdot x_0$, $\cos(x) < 0$. Therefore, the value of x_0 is none other than $\pi/2$, and it serves to define π. But in order to do this, the theory of alternating series needs to be developed, as well as the intermediate value theorem.

6.1.2 Alternating Sequences

An alternating sequence is one whose terms meet two criteria:

- Successive terms alternate in sign.

- Terms decrease in magnitude.

A strict interpretation of these rules would disqualify a sequence consisting of zeros as an alternating sequence. It is convenient, however, to relax the restrictions so that tails of zero terms are ignored.

These more liberal properties can be easily defined in ACL2. Consider the first property:

```
(defun opposite-signs-p (x y)
  (or (= x 0)
      (= y 0)
      (equal (sign x) (- (sign y)))))


(defun alternating-sequence-1-p (lst)
  (if (null lst)
      t
    (if (null (cdr lst))
        t
      (and (opposite-signs-p (car lst) (cadr lst))
           (alternating-sequence-1-p (cdr lst))))))
```

The second property can be verified using the following function:

```
(defun alternating-sequence-2-p (lst)
  (if (null lst)
      t
    (if (null (cdr lst))
        t
      (and (or (and (equal (car lst) 0)
                    (equal (cadr lst) 0))
```

```
                (< (abs (cadr lst))
                   (abs (car lst))))
            (alternating-sequence-2-p (cdr lst))))))
```

Again, notice the special treatment of zeros. Taken together, these functions define an alternating sequence:

```
(defun alternating-sequence-p (lst)
  (and (alternating-sequence-1-p lst)
       (alternating-sequence-2-p lst)))
```

This function is true of `'(1 -1/2 1/4)` and false of both `'(1 1/2 1/4)` and `'(1 -1 1)`.

The reason alternating sequences are important is that the first element of such a sequence is an upper bound on the sum of all the elements. In particular, the following is possible to prove:

```
(defthm sumlist-alternating-sequence
  (implies (and (alternating-sequence-p x)
                (real-listp x)
                (not (null x)))
           (not (< (abs (car x)) (abs (sumlist x))))))
```

Using this theorem, it is possible to approximate the sum of an alternating sequence with as high degree of accuracy as required. It is only necessary to add the first elements of the sequence up until a term that is smaller in magnitude than the degree of accuracy desired.

6.1.3 The Intermediate Value Theorem

The remaining piece of mathematics needed before π can be defined is the intermediate value theorem. To prove this result in ACL2 requires developing a theory of continuous functions.

The concept of continuity can be captured in ACL2 using the `encapsulate` facility. Continuity is a second-order notion, but using `encapsulate` it is possible to derive the basic theorems and apply them later to arbitrary continuous functions.

The definition of continuity in non-standard analysis captures the intuitive notion very well. A function f is continuous at a point x if for all y *i-close* to x, $f(x)$ is *i-close* to $f(y)$. A function is continuous if it is continuous at all *standard* points. Moreover, if it is continuous at all points, *standard* or not, it is uniformly continuous. The definition of continuity can be specified as follows:

```
(encapsulate
 ((rcfn (x) t))

 (local (defun rcfn (x) x))

 (defthm rcfn-standard
   (implies (standard-numberp x)
            (standard-numberp (rcfn x))))

 (defthm rcfn-real
   (implies (realp x)
            (realp (rcfn x))))

 (defthm rcfn-continuous
   (implies (and (standard-numberp x)
                 (realp x)
                 (i-close x y)
                 (realp y))
            (i-close (rcfn x) (rcfn y)))))
```

Three constraints are placed on the function `rcfn`. First, it must return *standard* values for *standard* arguments, a condition is satisfied by all classical functions. Second, it must return real values for real arguments — `rcfn` stands for "Real Continuous Function." And third, it must satisfy the non-standard continuity constraint.

Recall from analysis that a continuous function is uniformly continuous on a closed and bounded interval. Now consider an arbitrary *i-limited* number x. Since x is *i-limited*, it follows that $x \in [-M, M]$ for some *standard* number M. In other words, x belongs to a closed and bounded interval, where `rcfn` is uniformly continuous. From the non-standard definition of continuity, it follows that `rcfn` is continuous at x, regardless of whether x is *standard* or not. This justifies the following theorem:

```
(defthm rcfn-uniformly-continuous
  (implies (and (i-limited x)
                (realp x)
                (i-close x y)
                (realp y))
           (i-close (rcfn x) (rcfn y))))
```

The proof of this theorem actually follows from considering the *standard-part* of x. This number is *standard* since x is *i-limited*. Moreover, y must be *i-close* to *standard-part*(x) since y is *i-close* to x and all *i-limited* numbers are *i-close* to their *standard-part*. Applying the continuity of `rcfn` at *standard-part*(x) twice, it follows that $rcfn($*standard-part*$(x))$ is *i-close* to both $rcfn(x)$ and $rcfn(y)$, hence $rcfn(x)$ must be *i-close* to $rcfn(y)$.

The derivation of the intermediate value theorem in non-standard analysis is very direct. Given a *standard* interval $[a, b]$ so that $rcfn(a) < z$ and $rcfn(b) > z$ for a *standard* real number z, it is possible to find a value $c \in [a, b]$ so that $rcfn(c) = z$ as follows. First, partition the interval $[a, b]$ into $\{a, a + \epsilon, a + 2\epsilon, \ldots, a + N\epsilon = b\}$, where N is a positive integer and $\epsilon = \frac{b-a}{N}$. Then, observe there must be a $k < N$ so that $rcfn(a + k\epsilon) < z$ while $rcfn(a+(k+1)\epsilon) \geq z$. Here continuity comes into play. If N is *i-large*, the number $a+k\epsilon$

is not necessarily *standard,* however it must be *i-limited* since it belongs to the *standard* interval $[a, b]$. Therefore, lemma `rcfn-uniformly-continuous` applies, and it is possible to conclude that $rcfn(a+k\epsilon)$, $rcfn(a+(k+1)\epsilon)$, and $rcfn($*standard-part*$(a+k\epsilon))$ are all *i-close* to each other. Since the choice of k ensures that $rcfn(a + k\epsilon) < z \leq rcfn(a+(k+1)\epsilon)$, it follows that z is also *i-close* to all these three values. Hence z is *i-close* to $rcfn($*standard-part*$(a + k\epsilon))$; since both numbers are *standard,* they must be equal to each other. The only remaining detail is to observe that *standard-part*$(a + k\epsilon) \in (a, b)$. This follows because $a \leq a + k\epsilon$ and so $a =$ *standard-part*$(a) \leq$ *standard-part*$(a + k\epsilon)$; similarly, $b \geq$ *standard-part*$(a + k\epsilon)$ — note that *standard-part*$(a + k\epsilon)$ is not equal to either a or b is guaranteed because $rcfn($*standard-part*$(a + k\epsilon)) = z$, $rcfn(a) < z$, and $rcfn(b) > z$.

The formalization of this argument in ACL2 begins with the definition of the following function, which finds the value of k above:

```
(defun find-zero-n (a z i n eps)
  (if (and (realp a)
           (integerp i)
           (integerp n)
           (< i n)
           (realp eps)
           (< 0 eps)
           (< (rcfn (+ a eps)) z))
      (find-zero-n (+ a eps) z (1+ i) n eps)
    (realfix a)))
```

Notice that `find-zero-n` is a classical function, so it is possible to use unrestricted induction to prove theorems about it.

The key properties of `find-zero-n` are easy to prove by induction. The following theorems demonstrate that `find-zero-n` does in fact return a suitable value for

k:

```
(defthm rcfn-find-zero-n-<-z
  (implies (and (realp a) (< (rcfn a) z))
           (< (rcfn (find-zero-n a z i n eps)) z)))

(defthm rcfn-find-zero-n+eps->=-z
  (implies (and (realp a)
                (integerp i)
                (integerp n)
                (< i n)
                (realp eps)
                (< 0 eps)
                (< (rcfn a) z)
                (< z (rcfn (+ a (* (- n i) eps)))))
           (<= z (rcfn (+ (find-zero-n a z i n eps)
                          eps)))))
```

Moreover, it is easy to prove that the value returned by `find-zero-n` is in the range $[a, b)$:

```
(defthm find-zero-n-lower-bound
  (implies (and (realp a) (realp eps) (< 0 eps))
           (<= a (find-zero-n a z i n eps)))))

 (defthm find-zero-n-upper-bound
   (implies (and (realp a)
                 (integerp i)
                 (integerp n)
                 (<= 0 i)
```

```
                  (<= i n)
                  (realp eps)
                  (< 0 eps))
             (<= (find-zero-n a z i n eps)
                 (+ a (* (- n i) eps))))
```

It is important that `find-zero-n` is a classical function, because the non-standard principle of induction in ACL2 is not powerful enough to prove these theorems. Take the theorem `rcfn-find-zero-n-<-z`, for example. The induction scheme is based on the variable n, so the non-standard principle of induction requires that the following formula be established:

```
(implies (and (not (standard-numberp n))
              (< (rcfn z) z))
         (< (rcfn (find-zero-n a z i n eps)) z))
```

However, the extra hypothesis does not directly contribute to a proof, so the proof attempt will fail.

Since `find-zero-n` returns a value in the range $[a, b)$, if a and b are *standard* it follows that `find-zero-n` returns an *i-limited* value. This justifies the following definition:

```
(defun-std find-zero (a b z)
  (if (and (realp a)
           (realp b)
           (realp z)
           (< a b))
      (standard-part
       (find-zero-n a
                    z
```

```
                                0
                                (i-large-integer)
                                (/ (- b a) (i-large-integer))))
                0))
```

Note that since the definition is using `defun-std`, it is accepted only if the body is *standard* for *standard* arguments, and this follows since the body takes the *standard-part* of the *i-limited* instance of `find-zero-n`. The function `find-zero` fills the parameters of `find-zero-n` to conform to the argument described above; in particular, it is in `find-zero` that it is guaranteed that the `eps` used in `find-zero-n` is *i-small*.

Given the properties of the classical function `find-zero-n`, it is easy to establish similar facts about the non-classical function `find-zero` by using the transfer principe. In particular, the following theorems are easy to prove:

```
(defthm-std rcfn-find-zero-<=-z
  (implies (and (realp a)
                (realp b)
                (< a b)
                (realp z)
                (< (rcfn a) z))
           (<= (rcfn (find-zero a b z)) z)))


 (defthm-std rcfn-find-zero->=-z
   (implies (and (realp a)
                 (realp b)
                 (< a b)
                 (realp z)
                 (< (rcfn a) z)
                 (< z (rcfn b)))
            (<= z (rcfn (find-zero a b z)))))

(defthm-std find-zero-lower-bound
  (implies (and (realp a) (realp b) (realp z)
                (< a b))
           (<= a (find-zero a b z))))

(defthm-std find-zero-upper-bound
  (implies (and (realp a) (realp b) (realp z)
                (< a b))
           (<= (find-zero a b z) b)))
```

To prove `rcfn-find-zero->=-z` it is sufficient to observe that `(find-zero a b z)` and `(+ (find-zero a b z) (/ (- b a) (i-large-integer)))` must be *i-close* since `(/ (- b a) (i-large-integer))` is *i-small*. The result then follows from the continuity of `rcfn` and the lemma `rcfn-find-zero-n+eps->=-z`. These four theorems taken together result in the intermediate value theorem:

```
    (defthm intermediate-value-theorem
      (implies (and (realp a)
                    (realp b)
                    (realp z)
                    (< a b)
                    (< (rcfn a) z)
                    (< z (rcfn b)))
               (and (realp (find-zero a b z))
                    (< a (find-zero a b z))
                    (< (find-zero a b z) b)
                    (equal (rcfn (find-zero a b z))
                        z))))
```

The theorem above assumes the function `rcfn` takes on a higher value at b than at a. If the opposite is true, the theorem can not be used directly to establish that `rcfn` must have an intermediate value. However, since the function `rcfn` is a constrained function, it is possible to prove a version of the intermediate value theorem for functions with a higher value at a than b. The first step is to define a new function for finding the value k:

```
(defun find-zero-n-2 (a z i n eps)
  (if (and (realp a)
           (integerp i)
           (integerp n)
           (< i n)
           (realp eps)
           (< 0 eps)
           (< z (rcfn (+ a eps))))
      (find-zero-n-2 (+ a eps) z (1+ i) n eps)
    (realfix a)))

(defun-std find-zero-2 (a b z)
  (if (and (realp a)
           (realp b)
           (realp z)
           (< a b))
      (standard-part
       (find-zero-n-2 a
                      z
                      0
                      (i-large-integer)
```

```
                         (/ (- b a) (i-large-integer))))
        0))
```

To accept the function `find-zero-2` it is necessary to prove that its body returns *standard* values for *standard* arguments. This can be done by functionally instantiating the analogous lemma for `find-zero-n`.

The second version of the intermediate value theorem can be proved as follows:

```
(defthm intermediate-value-theorem-2
  (implies (and (realp a)
                (realp b)
                (realp z)
                (< a b)
                (< z (rcfn a))
                (< (rcfn b) z))
           (and (realp (find-zero-2 a b z))
                (< a (find-zero-2 a b z))
                (< (find-zero-2 a b z) b)
                (equal (rcfn (find-zero-2 a b z))
                       z)))
  :hints (("Goal"
           :use ((:instance
                  (:functional-instance
                   intermediate-value-theorem
                   (rcfn (lambda (x) (- (rcfn x))))
                   (find-zero (lambda (a b z)
                                (find-zero-2 a b
                                             (- z))))
                   (find-zero-n (lambda (a z i n
                                                       eps)
                                             (find-zero-n-2
                                              a (- z) i n eps))))
                            (z (- z))
                            ))
                     :in-theory
                     (disable intermediate-value-theorem))))
```

Notice how the proof of the theorem instantiates `intermediate-value-theorem` with the negatives of `rcfn` and `z`.

6.1.4 π for Dessert

Recall the function `taylor-sincos-list` is defined as follows:

```
(defun taylor-sincos-list (nterms counter sign x)
  (if (or (zp nterms)
          (not (integerp counter))
          (< counter 0))
      nil
    (cons (* sign
             (expt x counter)
             (/ (factorial counter)))
          (taylor-sincos-list (nfix (- nterms 2))
                              (+ counter 2)
                              (- sign)
                              x))))
```

It is easy to see that `taylor-sincos-list` satisfies the first alternating sequence property — that is, successive elements of the list alternate in sign.

```
(defthm alternating-sequence-1-p-taylor-sincos
  (implies (and (realp sign)
                (realp x))
           (alternating-sequence-1-p
            (taylor-sincos-list nterms
                                counter
                                sign
                                x))))
```

If it can be shown that it also satisfies the second criteria for alternating sequences, namely that successive terms decrease in magnitude, it will be possible to estimate values for sine and cosine.

However, showing that `taylor-sincos-list` satisfies the second alternating sequence property is more difficult. The problem is similar to the one found in chapter 5, where it was desired to show that the Taylor approximation to e^x was bounded by a geometric sequence. The problem is that the condition is true, but only after eliminating a suitable prefix of the series.

Since only the value of $\cos(2)$ is needed for the definition of π, a weaker theorem will be sufficient. All that must be shown is that `taylor-sincos-list` satisfies the second alternating sequence property when x is equal to 2. That sequence begins with 1, $-\frac{2^4}{4!} = -2$, $\frac{2^6}{6!} = \frac{4}{45} \ldots$. It is easy to see that the property holds after the initial 1 is removed:

```
(defthm alternating-sequence-2-p-taylor-sincos-2
  (implies (and (realp sign)
                (not (equal sign 0))
                (integerp counter)
                (integerp nterms)
                (<= 0 nterms)
```

```
                  (<= 2 counter)
                  (realp x)
                  (< 0 x)
                  (<= x 2))
             (alternating-sequence-2-p
              (taylor-sincos-list nterms
                                  counter
                                  sign
                                  x))))
```

In particular, this implies that for $x = 2$ `taylor-sincos-list` is an alternating sequence, after the initial term is removed.

```
(defthm alternating-sequence-p-taylor-sincos-2
  (implies (and (integerp nterms)
                (<= 0 nterms))
           (alternating-sequence-p
            (taylor-sincos-list nterms 4 1 2))))
```

The important point is that the sum of the elements after the first two terms in the expansion of $\cos(2)$ can be no more than $2/3$. This is a straightforward application of the lemma `sumlist-alternating-sequence` and the simple expansion of the first few elements of `taylor-sincos-list` for $x = 2$.

```
(defthm remainder-taylor-cos-2
  (implies (and (integerp nterms)
                (< 0 nterms))
           (<= (abs (sumlist
                     (taylor-sincos-list nterms 4 1 2)))
               2/3)))
```

It is thus possible to conclude that the Taylor expansion of $\cos(2)$ must be smaller than $-\frac{1}{3} = 1 - 2 + \frac{2}{3}$:

```
(defthm sumlist-taylor-cos-2-negative
  (implies (and (integerp nterms)
                (<= 4 nterms))
           (<= (sumlist
                (taylor-sincos-list nterms 0 1 2))
               -1/3)))
```

Taking *standard-part* on both sides of the inequality establishes that $\cos(2) \leq -\frac{1}{3}$:

```
(defthm acl2-cos-2-negative-lemma
  (<= (acl2-cosine 2) -1/3))
```

The stage is now set for the definition of π; both $\cos(0) = 1 > 0$ and $\cos(2) \leq -\frac{1}{3} < 0$ are established. It remains only to define the function `find-zero-cos-2` that can be used to invoke the second version of the intermediate value theorem:

```
(defun find-zero-cos-n-2 (a z i n eps)
  (if (and (realp a)
           (integerp i)
           (integerp n)
           (< i n)
           (realp eps)
           (< 0 eps)
           (< z (acl2-cosine (+ a eps))))
      (find-zero-cos-n-2 (+ a eps) z (1+ i) n eps)
    (realfix a)))


(defun-std find-zero-cos-2 (a b z)
  (if (and (realp a)
           (realp b)
           (realp z)
           (< a b))
        (standard-part
         (find-zero-cos-n-2 a
                            z
                            0
                            (i-large-integer)
                            (/ (- b a)
                               (i-large-integer))))
    0))
```

Note that the definition of `find-zero-cos-2` is not accepted until its body is shown to return *standard* values for *standard* arguments. This lemma can be proved by functionally instantiating the analogous theorem for `find-zero-2`.

The continuity of `acl2-cosine` can be easily deduced from the continuity of e^x, proved in chapter 5.

```
(defthm cosine-continuous
  (implies (and (standard-numberp x)
                (i-close x y))
           (i-close (acl2-cosine x)
                    (acl2-cosine y))))
```

The intermediate value theorem can now be applied to the function `acl2-cosine`, yielding the following theorem:

```
(defthm find-zero-cosine
  (and (realp (find-zero-cos-2 0 2 0))
       (< 0 (find-zero-cos-2 0 2 0))
       (< (find-zero-cos-2 0 2 0) 2)
       (equal (acl2-cosine (find-zero-cos-2 0 2 0)) 0)))
```

The only remaining detail is the actual definition of π.

```
    (defun acl2-pi ()
      (* 2 (find-zero-cos-2 0 2 0)))
```

It is immediate from the definition that `acl2-pi` is a real number between 0 and 4.

The possibility remains that the value of π defined is not the usual one. This would happen if `find-zero-cos-2` returned a zero of cosine that was not $\pi/2$. In reality this can not be; it is a basic fact of trigonometry that the cosine function has no other roots between 0 and 2. However, this fact has yet to be established in ACL2. Subsequent sections will show ACL2 proofs that explore the sign of the sine and cosine functions in the different quadrants, as sectioned off by `acl2-pi`. This will establish that `acl2-pi` can be no other than the usual value of π. Moreover, a much smaller interval containing π, equivalently a better approximation to π, will be presented in section 6.3.1.

6.2 Basic Trigonometric Identities

Many basic trigonometric identities are easy to prove in ACL2. The formulas for $\sin(x+y)$ and $\cos(x+y)$ are particularly useful. They can be proved directly from the definition of sine and cosine in terms of the e^x function. Here, the important lemma is $e^{x+y} = e^x \cdot e^y$, which was proved in chapter 5.

```
    (defthm sine-of-sums
      (equal (acl2-sine (+ x y))
             (+ (* (acl2-sine x) (acl2-cosine y))
                (* (acl2-cosine x) (acl2-sine y)))))
```

```
(defthm cosine-of-sums
  (equal (acl2-cosine (+ x y))
         (- (* (acl2-cosine x) (acl2-cosine y))
            (* (acl2-sine x) (acl2-sine y)))))
```

Similarly, the familiar identity $\sin^2(x) + \cos^2(x) = 1$ follows from the definitions of sine and cosine and a little algebra:

```
(defthm sin**2+cos**2
  (equal (+ (* (acl2-sine x) (acl2-sine x))
            (* (acl2-cosine x) (acl2-cosine x)))
         1))
```

Other simple theorems include $\sin(-x) = -\sin(x)$ and $\cos(-x) = \cos(x)$:

```
(defthm sin-uminus
  (equal (acl2-sine (- x))
         (- (acl2-sine (fix x)))))


(defthm cos-uminus
  (equal (acl2-cosine (- x))
         (acl2-cosine (fix x))))
```

Section 6.1.4 showed how the Taylor approximation to $\cos(2)$ is an alternating sequence. In fact, `alternating-sequence-p-taylor-sincos-2` can be strengthened to include all values of $x \in [0, 2]$. An analysis of the first few elements of the Taylor approximation to $\sin(x)$ for these x shows that the sum must be larger than $x - \frac{x^3}{3!} \geq 0$. Therefore, for $x \in [0, 2]$, $\sin(x) \geq 0$:

```
(defthm-std acl2-sin-x-positive
  (implies (and (realp x)
                (< 0 x)
                (<= x 2))
           (<= 0 (acl2-sine x))))
```

In particular, $\sin(\pi/2) \geq 0$. Since $\cos(\pi/2) = 0$ by definition, `sin**2+cos**2` implies that $\sin(\pi/2) = 1$:

```
(defthm sine-pi/2
  (equal (acl2-sine (* (acl2-pi) 1/2)) 1))
```

The theorems `sine-of-sums` and `cosine-of-sums` can now be used to find the value of sine and cosine at the cardinal points. In particular, the following theorems are all easy consequences of the above:

```
(defthm sine-0
  (equal (acl2-sine 0) 0))
(defthm cosine-0
  (equal (acl2-cosine 0) 1))
(defthm sine-pi
  (equal (acl2-sine (acl2-pi)) 0))
(defthm cosine-pi
  (equal (acl2-cosine (acl2-pi)) -1))
(defthm sine-3pi/2
  (equal (acl2-sine (* (acl2-pi) 3/2)) -1))
(defthm cosine-3pi/2
  (equal (acl2-cosine (* (acl2-pi) 3/2)) 0))
(defthm sine-2pi
  (equal (acl2-sine (* (acl2-pi) 2)) 0))
(defthm cosine-2pi
  (equal (acl2-cosine (* (acl2-pi) 2)) 1))
```

Together with `sine-of-sums` and `cosine-of-sums`, these theorems prove a number of famous identities, such as $\sin(x+\pi/2) = \sin(x)\cos(\pi/2)+\cos(x)\sin(\pi/2) = \cos(x)$:

```
(defthm sin-pi/2+x
  (equal (acl2-sine (+ (* (acl2-pi) 1/2) x))
         (acl2-cosine x)))
```

Similar theorems can be found for the other multiples of $\pi/2$.

For arbitrary angles between 0 and 2π, it is not possible to find exact values of sine and cosine[1], but it is possible to find their sign.

The following theorem is simply a weaker version of `acl2-sin-x-positive`:

```
(defthm sine-positive-in-0-pi/2
  (implies (and (realp x)
                (< 0 x)
                (< x (* (acl2-pi) 1/2)))
           (< 0 (acl2-sine x))))
```

It demonstrates that $\sin(x)$ is positive in the first quadrant. To show that $\cos(x)$ is also positive in this quadrant follows from the fact that $\cos(x) = \sin(\pi/2 - x)$. So in the first quadrant, sine and cosine have the same sign:

```
(defthm cosine-positive-in-0-pi/2
  (implies (and (realp x)
                (< 0 x)
                (< x (* (acl2-pi) 1/2)))
           (< 0 (acl2-cosine x))))
```

[1]However, it is possible to find arbitrarily close approximations to the sine and cosine functions, as section 6.3.2 demonstrates.

Note, this theorems ensures that $\pi/2$ is the first positive root of the cosine function. This provides a justification in ACL2 that the value of `acl2-pi` is in fact equal to π.

In the second quadrant, $\sin(x) > 0$ while $\cos(x) < 0$. This follows from the formulas for $\sin(x + \pi/2)$ and $\cos(x + \pi/2)$ and the corresponding theorems for sine and cosine in the first quadrant.

```
(defthm sine-positive-in-pi/2-pi
  (implies (and (realp x)
                (< (* (acl2-pi) 1/2) x)
                (< x (acl2-pi)))
           (< 0 (acl2-sine x))))


(defthm cosine-negative-in-pi/2-pi
  (implies (and (realp x)
                (< (* (acl2-pi) 1/2) x)
                (< x (acl2-pi)))
           (< (acl2-cosine x) 0)))
```

Similar theorems are readily achieved for angles in the other quadrants.

For angles not in the range $[0, 2\pi]$, the theorems above can still be applied. It is only necessary to normalize the angle into the range $[0, 2\pi]$. This can be justified using the following theorem:

```
(defthm sin-2npi
  (implies (integerp n)
           (equal (acl2-sine (* (acl2-pi) 2 n))
                  0)))
```

It is easy to derive the equivalent theorem about cosine.

The double-angle rules are easy to derive from the theorems `sine-of-sums` and `cosine-of-sums`. In particular, the following are easy to prove:

```
(defthm sine-2x
  (implies (syntaxp (not (equal x '0)))
           (equal (acl2-sine (* 2 x))
                  (* 2 (acl2-sine x) (acl2-cosine x)))))

(defthm cosine-2x
  (implies (syntaxp (not (equal x '0)))
           (equal (acl2-cosine (* 2 x))
                  (+ (* (acl2-cosine x) (acl2-cosine x))
                     (- (* (acl2-sine x)
                           (acl2-sine x)))))))
```

These theorems can be used to find the half-angle formulas:

```
(defthm sine**2-half-angle
  (equal (* (acl2-sine (* 1/2 x))
            (acl2-sine (* 1/2 x)))
         (/ (- 1 (acl2-cosine x)) 2)))

(defthm cosine**2-half-angle
  (equal (* (acl2-cosine (* 1/2 x))
            (acl2-cosine (* 1/2 x)))
         (/ (+ 1 (acl2-cosine x)) 2)))
```

Note, square roots are not taken on both sides because there is no guarantee the sign of the left-hand sides is positive. A wonderful consequence of these formulas is that it makes it possible to find the sine and cosine of more angles, such as $\pi/4$. In fact, the following theorems are trivial corollaries of the above:

```
(defthm sine-pi/4
  (equal (acl2-sine (* (acl2-pi) 1/4))
         (acl2-sqrt 1/2)))


(defthm cosine-pi/4
  (equal (acl2-cosine (* (acl2-pi) 1/4))
         (acl2-sqrt 1/2)))
```

The remaining well-loved angles are $\pi/3$ and $\pi/6$. It is possible to find their sine and cosine using a similar trick, only a triple-angle formula is needed. This can be derived with some algebra:

```
(defthm sine-3x
  (implies (syntaxp (not (equal x ’0)))
           (equal (acl2-sine (* 3 x))
                  (- (* 3 (acl2-sine x))
                     (* 4
                        (acl2-sine x)
                        (acl2-sine x)
                        (acl2-sine x))))))
```

From this formula it follows that $\sin(\pi/3) = \sqrt{3}/2$. The remaining values of sine and cosine of $\pi/3$ and $\pi/6$ are almost immediate.

```
(defthm sine-pi/3
  (equal (acl2-sine (* (acl2-pi) 1/3))
         (/ (acl2-sqrt 3) 2)))


(defthm cosine-pi/3
  (equal (acl2-cosine (* (acl2-pi) 1/3)) 1/2))
```

```
(defthm sine-pi/6
  (equal (acl2-sine (* (acl2-pi) 1/6)) 1/2))

(defthm cosine-pi/6
  (equal (acl2-cosine (* (acl2-pi) 1/6))
         (/ (acl2-sqrt 3) 2)))
```

The study of trigonometry involves the proof of many amusing identities, some of them of dubious value. The proofs proceed more or less in the same manner: one term is repeatedly simplified until it looks identical to another. This style of proof seems particularly well-suited to ACL2 and its rewriting engine.

To illustrate the point, consider some identities from [29], a standard trigonometry textbook. A good place to start is with the identity $3\cos^4(x) + 6\sin^2(x) = 3 + 3\sin^4(x)$. This fact can be easily verified in ACL2:

```
(defthm identity-1
  (equal (+ (* 3 (expt (acl2-cosine x) 4))
            (* 6 (expt (acl2-sine x) 2)))
         (+ 3
            (* 3 (expt (acl2-sine x) 4)))))
```

To do so requires a few illustrative rewrite lemmas. First, notice the theorem has several exponents of sine and cosine. All the trigonometric theorems defined above are in terms of products of sine and cosine, so it is best to do away with the exponents. A few rules such as the following to do the trick:

```
(defthm expt-2
  (equal (expt x 2) (* x x)))
```

The key lemma converts $3\cos^4(x)$ into $3(1 - \sin^2(x))^2$. This step can be suggested to ACL2 by introducing the following lemma:

```
(defthm lemma-1
  (equal (* 3
            (acl2-cosine x) (acl2-cosine x)
            (acl2-cosine x) (acl2-cosine x))
         (* 3
            (- 1 (* (acl2-sine x) (acl2-sine x)))
            (- 1 (* (acl2-sine x) (acl2-sine x))))))
```

The remainder of the proof is automatically handled by ACL2.

Another identity along this vein is the following:

```
(defthm identity-2
  (equal (+ (expt (acl2-secant x) 2)
            (expt (acl2-tangent x) 2))
         (- (expt (acl2-secant x) 4)
            (expt (acl2-tangent x) 4))))
```

The only needed lemmas are the rewrite rule favoring $(x^2 + y^2)(x^2 - y^2)$ over $x^4 - y^4$ and a technical algebraic lemma.

No extra lemmas are needed to prove the following identity:

```
(defthm identity-3
  (implies (not (equal (acl2-cosine x) 0))
           (equal (/ (+ (acl2-sine x)
                        (acl2-cotangent x))
                     (acl2-cosine x))
                  (+ (acl2-tangent x)
                     (acl2-cosecant x)))))
```

It is worth noting that trigonometry texts routinely omit the hypothesis above. It is simply understood that the equalities are required to hold only when both terms are defined.

A fourth identity states that $\frac{\sin(x)}{1+\cos(x)} + \frac{1+\cos(x)}{\sin(x)} = 2\csc(x)$. To establish this result, it is necessary to show that $1 + \cos(x)$ is not equal to 0 when x is real and $\sin(x) \neq 0$. It is also necessary to prove a simple algebraic lemma, and then it is possible to prove the identity:

```
(defthm identity-4
  (implies (realp x)
           (equal (+ (/ (acl2-sine x)
                        (+ 1 (acl2-cosine x)))
                     (/ (+ 1 (acl2-cosine x))
                        (acl2-sine x)))
                  (* 2 (acl2-cosecant x)))))
```

A final example is possible to prove after some basic algebraic lemmas are proved:

```
(defthm identity-5
  (implies (realp x)
           (equal (/ (acl2-cotangent x)
                     (- (acl2-cosecant x) 1))
                  (/ (+ (acl2-cosecant x) 1)
                     (acl2-cotangent x)))))
```

6.3 Computations with the Trigonometric Functions

One of the virtues of Nqthm and ACL2 is that they are *computational* logics. That is, functions defined in them can be directly executed. However, functions defined using the non-standard primitives of ACL2(r) lose this important property. For example, the value of `(i-large-integer)` is not known. But it is still possible to perform useful computations in ACL2(r). This section illustrates two approaches. First, ACL2 is used to verify the value of π to 20 decimal places. The computational aspect of ACL2 is evident here, especially when the size of the intermediate terms is taken into account. Second, it is shown how ACL2 can be used to define arbitrarily close approximations to the sine and cosine functions, similar to the `iter-sqrt` approximation to square root presented in chapter 2.

6.3.1 Estimating π

This section will illustrate how ACL2 can carry out trigonometric computations by using ACL2 to verify an approximation to π. Recall that π was defined by finding the value of $x \in [0, 2]$ for which $\cos(x) = 0$. It followed immediately that π was a real number between 0 and 4. Since $\sin(\pi) = 0$ and it can be shown that $\sin(x) > 0$ for $x \in (0, 2)$, it is possible to prove that $2 \leq \pi < 4$. The challenge is to find a better estimate for π, and that will require that several trigonometric computations be carried out with a high degree of accuracy.

The `math.h` header file that ships with HPUX 10.20 declares the following approximation for π:

```
#  define M_PI                3.14159265358979323846
```

ACL2 does not support the decimal notation, so the value of `M_PI` must be defined as follows:

```
(defun m-pi ()
  314159265358979323846/100000000000000000000)
```

The following theorems will verify that this is, in fact, a good approximation.

The first question is whether `m-pi` is too small or too large. This question can be decided by ACL2. Consider the Taylor series for $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$. Since this is an alternating sequence, the sign of $\cos(x)$ can be determined by finding an index i so that, for example, $1 - \frac{x^2}{2!} + \ldots + \frac{x^i}{i!}$ is negative and so is $-\frac{x^{i+2}}{(i+2)!}$. It is particularly easy to determine if a specific index i satisfies this requirement. In ACL2, this test can be performed using the following function:

```
(defun cosine-clearly-negative (x nterms)
  (and (< (sumlist (taylor-sincos-list nterms 0 1 x)) 0)
       (< (car (taylor-sincos-list 2
                                   nterms
                                   (if (evenp
                                        (/ nterms 2))
                                       1
                                     -1)
                                   x))
          0)))
```

A similar function checks whether the series is clearly positive, by inspecting the suggested index.

These two functions can be used to determine whether the cosine of `m-pi` is positive or negative. What is needed is simply to check if the index 1 settles the question or not. If not, then it is necessary to check whether the issue is resolved by index 2, and so on. This process can be mechanized in ACL2, using the following function:

```
(defun cosine-clear-sign (x nterms)
  (declare (xargs :mode :program))
  (if (cosine-clearly-negative x nterms)
      (cons 'negative nterms)
    (if (cosine-clearly-positive x nterms)
        (cons 'positive nterms)
      (cosine-clear-sign x (+ nterms 2)))))
```

There is an interesting problem with this function: its termination is not immediately obvious. For example, consider what happens when the value of x is equal to $\pi/2$. The function would continue expanding the Taylor series, trying to find an initial sequence that determines $\cos(\pi/2)$ as definitely positive or definitely negative. There is no such sequence since the value of $\cos(\pi/2)$ is zero, and so the function will recurse forever. Recall, ACL2 does not accept the definition of a function, unless it can prove that the function terminates on all inputs, so it should not accept the definition of `cosine-clear-sign`. However, ACL2 allows the introduction of arbitrary functions, as long as they are declared in `:program` mode, which means ACL2 does not introduce any axioms about the function, so it can not prove any theorems about it. This is a perfect use of this feature, since the interest is not in reasoning about `cosine-clear-sign`, but rather in executing it to find the value of certain constants, namely an index large enough to decide whether the cosine of half of `m-pi` is positive or negative. Running `cosine-clear-sign` on `(/ m-pi 2)` with an initial index of 0 returns `'(POSITIVE . 28)`. It follows that `m-pi` is a lower bound on π.

If the `math.h` header file is to be believed, then changing the last digit of `m-pi` from a 6 to a 7 should result in an upper bound for π. In particular, ACL2 should be able to verify that the following is larger than π:

```
(defun m-pi+eps ()
  31415926535897932384 7/10000000000000000000)
```

Running `cosine-clear-sign` on this number returns `'(NEGATIVE . 26)`, which confirms that its cosine is negative and that the sum of all terms in the Taylor series up to the $\frac{x^{26}}{26!}$ is required to prove this fact.

It is easy to overlook an important fact. The term $\frac{x^{26}}{26!}$ where $x \approx \pi/2$ is very close to zero. The only reason why these computations can be carried out with confidence is that ACL2 uses infinite precision arithmetic on the rationals. In particular, it does not assume the numerator of a rational is less than 2^{31} or some other arbitrary limit. That the function `cosine-clear-sign` can be written is a testament to the computational power of ACL2.

To prove that π lies between `m-pi` and `m-pi+eps` is relatively easy. The first step is to prove that the cosine of half of `m-pi` is positive and that of half `m-pi+eps` is negative. This formalizes the intuition behind `cosine-clear-sign`. Next, observe that half of `m-pi` must be in the first quadrant, since its cosine is positive and it must be between 0 and π. Similarly, `m-pi+eps` must be in the second quadrant. But then $\pi/2$ is between half of `m-pi` and half of `m-pi+eps`. Equivalently, π is between `m-pi` and `m-pi+eps`.

Formalizing `cosine-clear-sign` requires that the Taylor series for $\cos(x)$ be split into a prefix and a suffix so that the sum of each part has the same sign. According to the results of running `cosine-clear-sign`, the split for `m-pi` should be performed after the $\frac{x^{28}}{28!}$ term. This split can be justified according to the following ACL2 theorem:

```
(defthm taylor-sincos-list-split-for-m-pi
  (equal (taylor-sincos-list (i-large-integer)
                             0
                             1
                             (* (m-pi) 1/2))
         (append (taylor-sincos-list 28 0 1
                                     (* (m-pi) 1/2))
                 (taylor-sincos-list (-
                                      (i-large-integer)
                                      28)
                                     28
                                     1
                                     (* (m-pi) 1/2)))))
```

A similar theorem finds the appropriate split for `m-pi+eps`.

The important thing about this split is that the sum of each term is positive for `m-pi` and negative for `m-pi+eps`. The sum of the first part can be directly computed by ACL2, so it can easily prove that it is positive in the case of `m-pi`:

```
(defthm taylor-sincos-list-prefix-m-pi->-0
  (> (sumlist (taylor-sincos-list 28 0 1
                                  (* (m-pi) 1/2))) 0))
```

Similarly, the sum of the first part for `m-pi+eps` must be negative.

The sum of the second part can not be computed directly, since the value of `i-large-integer` is unspecified. However, since `taylor-sincos-list` is an alternating sequence, the sum can be bounded by the first element in the sequence. The value of this first element can be computed directly in ACL2. Therefore, it can prove that the sum of the second part is positive for `m-pi`:

```
(defthm taylor-sincos-list-postfix-m-pi->-0
  (> (sumlist (taylor-sincos-list (- (i-large-integer)
                                     28)
                                  28
                                  1
                                  (* (m-pi) 1/2)))
     0))
```

A similar theorem applies to `m-pi+eps`.

This demonstrates that the sum of the first `i-large-integers` of the Taylor series for cosine of `m-pi` is positive. The value of cosine of `m-pi` is the *standard-part* of this Taylor sum. However, just because the sum is positive does not mean its *standard-part* is positive. To conclude that, it is necessary to observe that the *standard-part* of the sum is equal to the sum of the *standard-part* of each of the two parts. This follows since the two parts are *i-limited*. Moreover, since the first part of the sum is only adding an *i-limited* number of *standard* terms, it must be *standard* so it is its own *standard-part*. In other words, the *standard-part* of the first is positive. Since the *standard-part* of the second sum can not be negative, it follows that the entire sum is positive. A similar argument establishes that the cosine of `m-pi+eps` is negative.

The first necessary lemma is that the sum of the first part is *standard*. This follows from the fact that the sum of any *i-limited* prefix of the Taylor series is *standard*, since it is adding an *i-limited* number of *standard* values:

```
(defthm taylor-sincos-list-standard
   (implies (and (standard-numberp sign)
                 (integerp counter)
                 (<= 0 counter)
                 (standard-numberp counter)
                 (integerp nterms)
                 (<= 0 nterms)
                 (standard-numberp nterms)
                 (standard-numberp x))
            (standard-numberp
             (sumlist (taylor-sincos-list nterms
                                          counter
                                          sign
                                          x)))))
```

Also necessary is the fact that the sum of the second part is *i-limited*. Since the Taylor series is alternating, this follows directly.

```
(defthm taylor-sincos-list-postfix-limited
  (i-limited
   (sumlist (taylor-sincos-list (+ (i-large-integer)
                                   -28)
                                28
                                1
                                (* (m-pi) 1/2)))))
```

It is an immediate corollary of the theorems above that the cosine of half of `m-pi` must be positive.

```
(defthm cosine-m-pi/2->-0
  (> (acl2-cosine (* (m-pi) 1/2)) 0))
```

In turn, this forces half of `m-pi` to lie in the first quadrant, or equivalently `m-pi` to be less than π. An analogous theorem proves that `m-pi+eps` is larger than π. This results in the following tight estimate for π:

```
(defthm pi-tight-bound
  (and (< (m-pi) (acl2-pi))
       (< (acl2-pi) (m-pi+eps))
       (<= (abs (- (m-pi) (m-pi+eps)))
           1/100000000000000000000)))
```

As the theorem states, the estimate for π is correct up to 20 significant digits. Moreover, the computation described in this section can obviously be modified to yield an arbitrary degree of precision.

Kaufmann used a similar technique to compute the value of $\sin(1/2)$ with an accuracy of $\pm 1/645120$ [36]. That proof, limited by the numeric system of the contemporary ACL2, served to motivate and guide the introduction of non-standard analysis into ACL2 as described in this thesis.

6.3.2 Approximating sine and cosine

This section presents an epsilon approximation scheme for the sine and cosine functions. The basic idea is simple. Recall that sine and cosine are defined in terms of the exponential function. Moreover, these functions have been proved equal to their usual Taylor series expansion:

```
(defthm taylor-sin-valid
  (implies (standard-numberp x)
           (equal (acl2-sine x)
                  (standard-part
                   (sumlist (taylor-sincos-list
                             (1- (i-large-integer))
                             1
                             1
                             x))))))

(defthm taylor-cos-valid
  (implies (standard-numberp x)
           (equal (acl2-cosine x)
                  (standard-part
                   (sumlist (taylor-sincos-list
                             (i-large-integer)
                             0
                             1
                             x))))))
```

Since the Taylor series is an alternating series, it is possible to approximate its sum by adding a prefix of it. Moreover, the error of the approximation is bounded by the first element of the suffix not included in the approximate sum. To find the value of $\sin(x)$ within an error bound of $\pm\epsilon$, it is only necessary to find a term $x^n/n!$ of the Taylor approximation to $\sin(x)$ such that $|x^n/n!| < \epsilon$.

The first step is to show that `taylor-sincos-list` returns an alternating sequence. Recall from section 6.1.4 that `taylor-sincos-list` is alternating for all x in the range $x \in (0, 2]$. To generalize this theorem to all values of x, it is necessary to divide `taylor-sincos-list` into an *i-limited* prefix and an alternating suffix. A possible split places the first $\lceil x \rceil$ elements in the prefix and the remaining terms in the suffix. A similar approach was taken in chapter 5 when the function e^x was introduced.

That the sum of the prefix is *i-limited* can be proved with the following theorem:

```
(defthm limited-taylor-sincos-list
  (implies (and (i-limited nterms)
                (i-limited counter)
                (standard-numberp sign)
                (i-limited x))
           (i-limited (sumlist
                       (taylor-sincos-list nterms
                                           counter
                                           sign
                                           x)))))
```

This theorem is true because the sum of a *i-limited* number of *i-limited* numbers is *i-limited*.

To prove that the suffix is alternating, it is almost sufficient to show that $x^n/n! < x^{n+2}/(n+2)!$ when $|x| < n$. Care must be taken to account for the cases when x is zero.

```
(defthm alternating-sequence-2-p-taylor-sincos
  (implies (and (realp x)
                (realp sign)
                (integerp counter)
                (< (abs x) counter))
           (alternating-sequence-2-p
            (taylor-sincos-list nterms
                                counter
                                sign
                                x))))
```

Using the properties of alternating series, it is now possible to find an upper bound for the sum of the suffix:

```
(defthm sumlist-taylor-sincos-list-bound
  (implies (and (realp x)
                (realp sign)
                (integerp counter)
                (< (abs x) counter))
           (<= (abs (sumlist (taylor-sincos-list nterms
                                                 counter
                                                 sign
                                                 x)))
               (abs (taylor-sin-term sign counter x)))))
```

The function `taylor-sin-term` simply returns an element of the Taylor expansion for sine or cosine:

```
(defun taylor-sin-term (sign counter x)
  (* sign
     (expt x counter)
     (/ (factorial counter))))
```

What remains is to find a large enough n so that $|x|^n/n! < \epsilon$ for an arbitrary choice of x and $\epsilon > 0$. To find such an n, first observe that for an integer m less than n, $|x|^n/n!$ can be written as $\prod_{i=1}^{m} \frac{|x|}{i} \cdot \prod_{i=m+1}^{n} \frac{|x|}{i}$ and so $|x|^n/n! < M \cdot \left(\frac{|x|}{m}^{n-m}\right)$ where $M = \prod_{i=1}^{m} \frac{|x|}{i}$. If m is chosen so that $|x|/m \leq 1/2$, it follows that $|x|^n/n! < \epsilon$ if $\frac{|x|}{m}^{n-m} < 1/2^{n-m} < \epsilon/M$. So the problem is simply to find an n large enough so that $1/2^{n-m} < \epsilon/M$ for some $\epsilon/M > 0$. The function `guess-num-iters` introduced in chapter 2 can be used to find such an n.

The approximations to sine and cosine can be defined as follows:

```
(defun sine-approx (x eps)
  (sumlist
   (taylor-sincos-list (n-for-sincos x eps)
                       1
                       1
                       x)))

(defun cosine-approx (x eps)
  (sumlist
   (taylor-sincos-list (n-for-sincos x eps)
                       0
                       1
                       x)))
```

The function `n-for-sincos` simply finds a suitable n given x and eps. It finds this value as suggested by the argument above.

The remainder of the proof is simple. For *standard* values of `x` and `eps`, `(n-for-sincos x eps)` is *i-limited* and therefore less than `(i-large-integer)`. So the Taylor series for sine or cosine can be split after `(n-for-sincos x eps)` terms as follows:

```
(defthm standard-part-sumlist-taylor-sincos-list-split-2
  (implies (and (integerp nterms)
                (<= 0 nterms)
                (i-large nterms)
                (realp x) (standard-numberp x)
                (realp eps) (< 0 eps) (< eps 1)
                (standard-numberp eps)
                (equal n (n-for-taylor-sin-term x eps)))
           (equal (standard-part
                   (sumlist (taylor-sincos-list nterms
                                                1
                                                1
                                                x)))
                  (+ (sumlist
                      (taylor-sincos-list n 0 1 x))
                     (standard-part
                      (sumlist
                       (taylor-sincos-list (- nterms n)
                                           (1+ n)
                                           (if (evenp
                                                (/ n 2))
                                               1
                                             -1)
                                           x))))))
```

The function `n-for-taylor-sin-term`, used by `n-for-sincos`, returns a suitably large value of n but only for $\epsilon < 1$. The function `n-for-sincos` adjusts the value of ϵ if necessary before passing it to `n-for-taylor-sin-term`. The first term of this sum is equal to `(sine-approx x eps)`. The second sum can be bounded by its first element, and the value of `n` is chosen so that this element is less than ϵ. This proves the correctness of the approximation function `sine-approx`:

```
(defthm-std sine-approx-valid
  (implies (and (realp x)
                (realp eps) (< 0 eps))
           (< (abs (- (acl2-sine x)
                      (sine-approx x eps)))
```

```
                    eps)))
```

A similar argument shows `cosine-approx` is an epsilon approximation scheme for cosine.

It should be emphasized that the `sine-approx` and `cosine-approx` functions are executable. For example, in section 6.1.4 it was shown that the cosine of 2 is negative, but its value was left undetermined. Using `cosine-approx`, it is possible to find estimates to $\cos(2)$ with as much accuracy as desired. ACL2 evaluates `(cosine-approx 2 1/1000)` to `-265715113/638512875` or `-0.4161`. This number is guaranteed to be within $1/1000$ of the actual value of $\cos(2)$, as can be verified with a calculator.

This chapter began by introducing the sine and cosine functions into ACL2. However, the previous results show that significantly more has been accomplished. ACL2 has a fair amount of trigonometric knowledge, and with some help and guidance it is able to prove quite interesting trigonometric facts and approximate complex trigonometric expressions.

Chapter 7

Powerlists in ACL2

This chapter makes a sharp departure from the earlier ones. It is time for the spotlight to move away from the real numbers and non-standard analysis and into the world of data structures.

In [46], Misra introduced the powerlist data structure, which is well suited to express recursive, data-parallel algorithms. Misra and others have shown how powerlists can be used to prove the correctness of several algorithms. Naturally, this success has encouraged some researchers to pursue automated proofs of theorems about powerlists[32, 33, 34]. This chapter shows how such theorems can be proved in ACL2.

7.1 Regular Powerlists

Misra defines powerlists as follows. For any scalar x, the object $\langle x \rangle$ is a singleton powerlist. If x and y are "similar" powerlists — that is, they have the same number of elements, and corresponding elements are similar — the new powerlists $x \mid y$ and $x \bowtie y$, called the tie and zip of x and y, respectively, can be constructed. The powerlist $x \mid y$ consists of all elements of x followed by the elements of y. In contrast, $x \bowtie y$ contains the elements of x interleaved with the elements of y. Since tie and zip are defined only for similar powerlists, all powerlists are of length 2^n for some integer n, and moreover all elements of a powerlist are similar to each other. In the sequel, a more general notion of powerlist will be introduced. To avoid confusion, powerlists adhering to Misra's original criteria will be referred to as "regular" powerlists.

For example, $\langle 1\rangle$, $\langle 1,2\rangle$, $\langle 3,4\rangle$, $\langle 1,2,3,4\rangle$ and $\langle 1,3,2,4\rangle$ are all powerlists. Moreover, $\langle 1,2\rangle \mid \langle 3,4\rangle = \langle 1,2,3,4\rangle$ and $\langle 1,2\rangle \bowtie \langle 3,4\rangle = \langle 1,3,2,4\rangle$.

The theory of powerlists is given by the following axioms (laws in [46]):

L0. For singleton powerlists $\langle x\rangle$ and $\langle y\rangle$, $\langle x\rangle \mid \langle y\rangle = \langle x\rangle \bowtie \langle y\rangle$.

L1a. For any non-singleton powerlist X, there are similar powerlists L, R so that $X = L \mid R$.

L1b. For any non-singleton powerlist X, there are similar powerlists O, E so that $X = O \bowtie E$.

L2a. For singleton powerlists $\langle x\rangle$ and $\langle y\rangle$, $\langle x\rangle = \langle y\rangle$ iff $x = y$.

L2b. For powerlists $X_1 \mid X_2$ and $Y_1 \mid Y_2$, $X_1 \mid X_2 = Y_1 \mid Y_2$ iff $X_1 = Y_1$ and $X_2 = Y_2$.

L2c. For powerlists $X_1 \bowtie X_2$ and $Y_1 \bowtie Y_2$, $X_1 \bowtie X_2 = Y_1 \bowtie Y_2$ iff $X_1 = Y_1$ and $X_2 = Y_2$.

L3. For powerlists X_1, X_2, Y_1, and Y_2, $(X_1 \mid X_2) \bowtie (Y_1 \mid Y_2) = (X_1 \bowtie Y_1) \mid (X_2 \bowtie Y_2)$.

It is possible to find a smaller set of axioms to characterize powerlists. For example, law *L3* can be used to define zip in terms of tie, so that only tie remains as an undefined term.

What makes powerlists so special is that they provide a notation in which data-parallel, recursive algorithms can be expressed naturally. Consider the function $s(x)$ that adds up all the elements in the powerlist x. It can be defined as follows:

$$\begin{aligned} s(\langle x\rangle) &= x \\ s(x \mid y) &= s(x) + s(y) \end{aligned}$$

Notice that the recursive computation is recursing twice, once on each half of $x \mid y$. The result is that the partial sums $s(x)$ and $s(y)$ can be computed in parallel, resulting in the classic addition binary tree implementation. This stands in sharp contrast to the function $t(x)$ which adds the elements of the *list* (not powerlist) x:

$$\begin{aligned} t([x]) &= x \\ t(a.x) &= a + t(x) \end{aligned}$$

This time, notice how the computation is recursing only once, on a list that is almost as long as the original list. The resulting computation tree is actually a linear tree as deep as the input list is long. Clearly, the implied computation is inherently serial; it does not lend itself to a parallel implementation.

Another benefit of powerlists is that the properties of powerlist functions can be verified algebraically, proceeding from the powerlist axioms *L0* through *L3*. Consider a different function that also computes the sum of the elements in a powerlist:

$$\begin{aligned} u(\langle x \rangle) &= x \\ u(x \bowtie y) &= u(x) + u(y) \end{aligned}$$

It is possible to prove using a simple induction scheme that $u(x)$ is identical to $s(x)$. Such proofs will be presented in the following sections.

7.2 Defining Powerlists in ACL2

A Naive Representation of Powerlists

Choosing the right representation of powerlists in ACL2 is not trivial. One immediate stumbling block is that ACL2 does not support partial functions, so the definitions of $\mid$ and $\bowtie$ must do *something* for non-similar powerlists, and in fact for non-powerlist operands. A first approach might represent powerlists in ACL2 as lists of length 2^n. The function `tie` would take two powerlists and, if they are of equal length, return their concatenation, otherwise a special error powerlist (e.g., `nil`). A similar definition would work for `zip`. Such an approach is taken in [33], where partial constructors play a central role.

There are a few problems with taking this approach in ACL2. First of all, each time a `tie` or `zip` operation is made, it must be proved that the arguments are of equal length. These proof obligations can become expensive at best, and disastrous if they prevent term simplification. Moreover, the proof obligations propagate into all theorems concerning `tie` and `zip`; these obligations can place a large burden on the ACL2 rewriter. The second problem is that since ACL2 does not support function definitions over terms, powerlist functions such as

$$\begin{aligned} rev(\langle x\rangle) &= x \\ rev(x \mid y) &= rev(y) \mid rev(x) \end{aligned}$$

need to be turned into the form

$$rev(X) = \begin{cases} X & \text{if } X \text{ is a singleton} \\ rev(right(X)) \mid rev(left(X)) & \text{otherwise} \end{cases}$$

where the functions $left$ and $right$ are defined so that $left(X) \mid right(X) = X$. But defining these functions in ACL2 — more germanely, reasoning about them — is not simple. Intuitively, the problem is that to compute $left(X)$, it is necessary to count the elements of X, divide by two, then walk back through the elements of X and return half of them. Reasoning about all these steps is necessary in every function invocation, so the overhead quickly overwhelms the prover.

Another problem with this approach is that an explicit definition of `zip` and `tie` leaves open the possibility that the theory developed depends on the particular definitions of `zip` and `tie` used, not just on the powerlist laws as defined by Misra. Simply showing that the ACL2 functions `zip` and `tie` satisfy the axioms is not sufficient, since other properties of the specific functions `zip` and `tie` may be used in the proof of some theorem.

A Better Representation of Powerlists

The observations above suggest an alternative approach. Instead of representing powerlists as lists, represent them as binary trees, e.g., `cons` trees. Moreover, remove the restriction that `tie` and `zip` only apply to similar powerlists. The operation `tie` is now replaced by a simple `cons` and `left` and `right` can be defined in terms of `car` and `cdr`. The definition of `zip` requires a recursive function, very similar to the one used when representing powerlists as lists. The result of this representation is that reasoning about powerlists requires much less overhead than before; however, the representation allows objects that were previously not recognized as powerlists, for example $\langle 1.\langle 2.3\rangle\rangle$, where dotted notation is used to emphasize the structural nature of the representation. In the sequel, the term "powerlists" will be used to refer to arbitrary "dotted-pair" powerlists as above. The term "regular powerlists" is reserved to powerlists satisfying Misra's original constraints.

The generalized notion of powerlists allows the expression of some algorithms which can not be stated in traditional powerlist theory, for example insertion sort. On the other hand, it presents some new problems. First, it does not retain a 1-to-1 correspondence between linear lists and powerlists. For example, the list $(1, 2, 3, 4)$ can be viewed as either of the powerlists $\langle 1.\langle 2.\langle 3.4\rangle\rangle\rangle$ or $\langle\langle 1.2\rangle.\langle 3.4\rangle\rangle$. This is not too troubling, because the theorems presented here will be true of either powerlist representation. Naturally, in parallel processing applications, it is best to choose the powerlist with the smallest maximal branch height. The choice, however, is made in the translation from lists to powerlists, not in the powerlist theory. A second problem is that the operational semantics of certain functions may not carry over to generalized powerlists. For example, the operational semantics of zip is that it interleaves the elements from its two powerlist arguments. This is clearly not possible if the arguments have different lengths. The functions defined here match the operational semantics only for regular powerlists, but they retain the relevant algebraic properties for all powerlists. For example, the `zip` operator interleaves the elements from its two arguments when these are regular and similar to each other. Furthermore, for all powerlists, `zip` obeys the algebraic properties stated in laws *L1b*, *L2c*, and *L3*.

The choice to use generalized powerlists was made taking these tradeoffs into account. Similar tradeoffs can be found in other approaches to generalized powerlists, such as Kornerup's parlists [44].

It is important that the resulting theory is nevertheless faithful to the original theory. That is, the original axioms of `zip` and `tie` hold in the new theory. At the very least, it is important to ensure that the theorems about regular powerlists are precisely those of Misra's theory. To do so requires examining each of Misra's powerlist axioms in turn.

Observe, since the scalar powerlist $\langle x \rangle$ is simply represented as x in this scheme, law *L2a* is trivially true. A drawback of this approach is that nested powerlists are not implicitly allowed, e.g., $\langle\langle 1,2\rangle,\langle 3,4\rangle\rangle$ is indistinguishable from $\langle 1,2,3,4\rangle$ in this representation. Where nested powerlists are needed, e.g., for matrices, an explicit $nest$ operator must be used, as in $\langle nest(\langle 1,2\rangle), nest(\langle 3,4\rangle)\rangle$.

The Tie Constructor

The actual implementation begins with the definition of the data type powerlists. Recall the intent of clearly separating the logical properties of powerlists from any special properties deriving from the specific definitions of `tie` and `zip`. This separation can be cleanly enforced by using the ACL2 `encapsulate` primitive to introduce `tie` as a constrained function. Consider the following partial encapsulate:

```
(encapsulate
 ((powerlist (car cdr) t)
  (powerlist-p (powerlist) t)
  (powerlist-car (powerlist) t)
  (powerlist-cdr (powerlist) t))

 (local
  (defun powerlist (car cdr)
    (cons 'powerlist (cons car (cons cdr nil)))))

 (local
  (defun powerlist-p (powerlist)
    (and (consp powerlist) (consp (cdr powerlist))
         (consp (cdr (cdr powerlist)))
         (null (cdr (cdr (cdr powerlist))))
         (eq (car powerlist) 'powerlist))))

 (local
  (defun powerlist-car (powerlist)
    (car (cdr powerlist))))

 (local
  (defun powerlist-cdr (powerlist)
    (car (cdr (cdr powerlist)))))

 ...
 )
```

This exports the signatures of the powerlist constructor `powerlist`, the powerlist recognizer `powerlist-p`, and the destructors `powerlist-car` and `powerlist-cdr`, while completely hiding the details of the implementation. Note, the function `powerlist` is intended to correspond to the tie operation. To emphasize this fact, it is convenient to explicitly define `p-tie` as a macro:

```
(defmacro p-tie (x y)
  `(powerlist ,x ,y))
```

Similarly, the macros `p-untie-l` and `p-untie-l` can stand for `powerlist-car` and `powerlist-cdr`, respectively. These destructors will be referred to as the "left" and "right half" of a powerlist.

What remains is to present sufficient constraints on these functions so that they satisfy Misra's powerlist laws. For example, `powerlist-p` should be true of an object constructed with `powerlist`. Moreover, `powerlist-car` and `powerlist-cdr` should behave as inverses to `powerlist`:

```
(encapsulate
 ...

 (defthm defs-powerlist-p-powerlist
   (equal (powerlist-p (powerlist car cdr)) t))

 (defthm defs-read-powerlist
   (and (equal (powerlist-car (powerlist car cdr))
               car)
        (equal (powerlist-cdr (powerlist car cdr))
               cdr)))

 (defthm defs-eliminate-powerlist
   (implies (powerlist-p powerlist)
            (equal (powerlist (powerlist-car powerlist)
                              (powerlist-cdr powerlist))
                   powerlist)))
 )
```

Because powerlists are recursive data structures, it is no surprise that functions will be defined recursing on `powerlist-car` and `powerlist-cdr`. So it is also necessary to prove that such recursions are justified; i.e., that there is a well-founded measure that is reduced by invocations to `powerlist-car` and `powerlist-cdr`:

```
(encapsulate
 ...

 (defthm untie-reduces-count
   (implies (powerlist-p x)
            (and (< (acl2-count (powerlist-car x))
                    (acl2-count x))
                 (< (acl2-count (powerlist-cdr x))
                    (acl2-count x)))))
 )
```

Because `untie-reduces-count` is used so often in inductive definitions, it is worthwhile to make it a `:builtin` rule in ACL2. This requires only that the theorem is stated in precisely the same way that ACL2's definition procedure phrases the recursive termination proof obligation. The easiest way to do so is simply to cut and paste the appropriate goal from a run of ACL2. Finally, it is necessary to show some objects that are guaranteed not to be powerlists, for example numbers and booleans:

```
(encapsulate
 ...

 (defthm nesting-powerlists
   (equal (powerlist-p (list ’nest x)) nil))

 (defthm powerlist-non-boolean
   (implies (powerlist-p x)
            (not (booleanp x))))
 (defthm boolean-non-powerlist
   (implies (booleanp x)
            (not (powerlist-p x))))

 (defthm powerlist-non-numeric
   (implies (powerlist-p x)
            (not (acl2-numberp x))))

 (defthm numeric-non-powerlist
   (implies (acl2-numberp x)
            (not (powerlist-p x))))
 )
```

Notice the theorem `nesting-powerlists`. Recall that in this representation it is impossible to represent nested powerlists without an explicit nesting operator. The theorem `nesting-powerlists` guarantees that objects of the form `'(nest` X`)` can not possibly be powerlists, hence a powerlist x can be nested inside another powerlist by placing it in the term `'(nest` x`)`.

The Zip "Constructor"

It is now possible to define the function `p-zip` which implements the zip "constructor":

```
(defun p-zip (x y)
  (if (and (powerlist-p x) (powerlist-p y))
      (p-tie (p-zip (p-untie-l x) (p-untie-l y))
             (p-zip (p-untie-r x) (p-untie-r y)))
    (p-tie x y)))
```

Note how the definition of `p-zip` mirrors *L0* and *L3*, hence these axioms are satisfied by these definition of `p-tie` and `p-zip`. In order to accept definitions based on `p-zip`, the functions `p-unzip-l` and `p-unzip-r`, analogous to `p-untie-l` and `p-untie-r`, need to be defined. This can be done as follows:

```
(defun p-unzip-l (x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (if (powerlist-p (p-untie-r x))
              (p-tie (p-unzip-l (p-untie-l x))
                     (p-unzip-l (p-untie-r x)))
            (p-untie-l x))
        (p-untie-l x))
    x))

(defun p-unzip-r (x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (if (powerlist-p (p-untie-r x))
              (p-tie (p-unzip-r (p-untie-l x))
                     (p-unzip-r (p-untie-r x)))
            (p-untie-r x))
        (p-untie-r x))
    nil))
```

Note, these functions provide the equivalent to Misra's law *L1b*. It is possible to prove the validity of recursion based on `p-zip`, using a theorem similar to `untie-reduces-count`.

Notice that p-unzip-l and p-unzip-r return every other element of a regular powerlist x. If the elements of x are indexed from 1, (p-unzip-l x) returns the odd-indexed elements,and (p-unzip-r x) the even-indexed ones. Hence, in the sequel p-unzip-l and p-unzip-r will be referred to as the odd- and even-indexed elements of x, respectively. They will also be referred to as the "left unzip" and "right unzip" of x.

The definitions of p-unzip-l and p-unzip-r were carefully constructed so that the following theorems are all true:

```
(defthm zip-unzip
  (implies (powerlist-p x)
           (equal (p-zip (p-unzip-l x)
                         (p-unzip-r x))
                  x)))

(defthm unzip-l-zip
  (equal (p-unzip-l (p-zip x y)) x))

(defthm unzip-r-zip
  (equal (p-unzip-r (p-zip x y)) y))
```

These three theorems prove the equivalent of law *L2c* for the ACL2 powerlists. On an implementation note, zip-unzip should be an :elim rule so that ACL2 can use it to eliminate the destructors p-unzip-l and p-unzip-r in favor of p-zip, in the same way it removes car and cdr and replaces them using cons.

7.2.1 Similar Powerlists

At this point, the definitions of p-tie and p-zip are known to satisfy all of Misra's powerlist axioms, except for the notion of similarity. Laws *L1a* and *L1b* claim that the p-untie-l and p-untie-r of a powerlist are similar, i.e. of the same length, as are its p-unzip-l and p-unzip-r. This is certainly not the case with ACL2 powerlists, since powerlists are not required to be of length 2^n. However, it is possible to add conditions, namely that the given powerlists be regular, that make these theorems true. Later, these regularity conditions will surface as hypotheses in some of the example theorems proved.

As in [46], two powerlists are defined as similar if they have the same tie-tree structure:

```
(defun p-similar-p (x y)
  (if (powerlist-p x)
      (and (powerlist-p y)
           (p-similar-p (p-untie-l x) (p-untie-l y))
           (p-similar-p (p-untie-r x) (p-untie-r y)))
    (not (powerlist-p y))))
```

It follows immediately that p-similar-p is an equivalence relation. This proves useful, because ACL2 can use this fact in its generic "equivalence" reasoning, hence making theorems about p-similar-p easier to prove.

The next task is to show how p-similar-p powerlists behave in conjunction with the constructors and destructors based on p-tie and p-zip. These theorems are trivial for regular powerlists, since powerlists are similar if and only if they have the same length. Moreover, both zip and tie double the length of a powerlist, and unzip and untie halve it.

Things are a little murkier in the case of general powerlists; this lost simplicity is the price paid for not using a regular data structure as suggested by Misra. For starters, it is easy to prove theorems about the destructors, such as the following:

```
(defthm unzip-l-similar
  (implies (p-similar-p x y)
           (p-similar-p (p-unzip-l x) (p-unzip-l y))))
```

The analogous theorems for `p-unzip-r` as well as for `p-untie` are also trivial. These theorems will be used most often in proving the antecedent of an inductive hypothesis. For example, with the goal

```
    (implies (p-similar-p x y)
             (P x y))
```

where property `P` is defined in terms of `p-zip`, the following subgoal is likely to be generated by induction:

```
    (implies (and (powerlist-p x)
                  (p-similar-p x y)
                  (implies (p-similar-p (p-unzip-l x)
                                        (p-unzip-l y))
                           (P (p-unzip-l x) (p-unzip-l y)))
                  (implies (p-similar-p (p-unzip-r x)
                                        (p-unzip-r y))
                           (P (p-unzip-r x) (p-unzip-r y))))
             (P x y))
```

At this point, `unzip-l-similar` can be used to establish that `(P (p-unzip-l x) (p-unzip-l y))` is true and the proof can proceed. Since this is the intended use, these theorems can be turned into `:forward-chaining` rules, which are triggered before ACL2's general rewriting engine (and hence provide a modest performance improvement).

Remaining are the constructors `p-tie` and `p-zip`. It should be possible to prove that when a powerlist is zipped (tied) to one of two similar powerlists, the result is similar to when it is zipped (tied) to the other. ACL2 provides a general way to reason about this type of theorem, namely congruence rewriting. With congruence rewriting, ACL2 will deduce `(p-zip x1 y)` is similar to `(p-zip x2 y)` when `x1` is similar to `x2`, and similarly that `(p-zip x y1)` and `(p-zip x y2)` are similar when `y1` and `y2` are. Congruence rules can be defined in ACL2 as follows:

```
    (defcong p-similar-p p-similar-p (p-zip x y) 1)
    (defcong p-similar-p p-similar-p (p-zip x y) 2)
```

The `defcong` event is simply syntactic sugar for the following rules:

```
    (defthm p-similar-p-implies-p-similar-p-p-zip-1
      (implies (p-similar-p x x-equiv)
               (p-similar-p (p-zip x y)
                            (p-zip x-equiv y)))
      :rule-classes (:congruence))

    (defthm p-similar-p-implies-p-similar-p-p-zip-2
      (implies (p-similar-p y y-equiv)
               (p-similar-p (p-zip x y)
                            (p-zip x y-equiv)))
      :rule-classes (:congruence))
```

7.2.2 Regular Powerlists

Another useful property of powerlists is `p-regular-p` which is true of a perfectly balanced powerlist, that is, one which corresponds to the theory in [46]. This condition is more expensive to verify than `p-similar-p`, because it requires passing information from one half of the powerlist to the other, i.e., not only must the left and right halves of the powerlist be regular, their depth must be the same. Rather than explicitly reasoning about depth, it is convenient to use `p-similar-p`, since several theorems about it have already been established. The result is the following definition:

```
    (defun p-regular-p (x)
      (if (powerlist-p x)
          (and (p-similar-p (p-untie-l x) (p-untie-r x))
               (p-regular-p (p-untie-l x))
               (p-regular-p (p-untie-r x)))
        t))
```

Note that both the similarity and regularity conditions of the definition are required to restrict powerlists to be of length 2^n. For example, if the similarity condition were left out, $\langle 1.\langle\langle 2.3\rangle.\langle 4.5\rangle\rangle\rangle$ would be considered regular. Likewise, if the regularity conditions were left out, the powerlist $\langle\langle 1.\langle 2.3\rangle\rangle.\langle 4.\langle 5.6\rangle\rangle$ would be considered regular. It will be seen later that both regularity conditions are not needed in the definition.

As was the case with `p-similar-p`, it is necessary to show how `p-regular-p` interacts with the constructors and destructors of `p-tie` and `p-zip`. This results in the following type of theorem:

```
    (defthm unzip-regular
      (implies (p-regular-p x)
               (and (p-regular-p (p-unzip-l x))
                    (p-regular-p (p-unzip-r x)))))
```

The converse theorem, about the constructor functions requires an extra hypothesis, namely that the powerlists to be tied or zipped be similar. This is the formal equivalent of the restriction that $|$ and $\bowtie$ only apply to powerlists of the same length. The theorem can be stated as follows:

```
    (defthm zip-regular
      (implies (and (p-regular-p x)
                    (p-similar-p x y))
               (p-regular-p (p-zip x y))))
```

Another group of theorems explores the interaction between `p-regular-p` and `p-similar-p` powerlists. For example, the unzips and unties of regular powerlists are similar:

```
(defthm regular-similar-unzip-untie
  (implies (and (powerlist-p x)
                (p-regular-p x))
           (and (p-similar-p (p-unzip-l x)
                             (p-unzip-r x))
                (p-similar-p (p-unzip-l x)
                             (p-untie-l x))
                (p-similar-p (p-unzip-r x)
                             (p-untie-r x)))))
```

This particular theorem provides the missing similarity assertion of laws *L1a* and *L1b*.

Similar theorems, such as a powerlist similar to a regular powerlist is also regular, can be easily proved. This particular theorem justifies the removal of one of the recursive `p-regular-p` instances in the definition of `p-regular-p`. It is probably best to leave the definition as is, because of symmetry and also because having the extra condition immediately available may be useful when `p-regular-p` is found as a hypothesis in a theorem.

The notion `p-similar-p` appears much more useful than `p-regular-p`, since similarity ensures that a function taking more than one argument can recurse on one of the arguments and still visit all the nodes of the other argument, e.g., for pairwise addition of powerlists. In fact, the main use of `p-regular-p` is to show that two powerlists are similar. This occurs when a single powerlist is split and a function applied to the two halves. It also occurs when two powerlists are traversed in a non-standard ordering, e.g., by splitting them into left and right halves and then combining the left half of one with the right half of the other or by splitting with unzip and combining with tie. In these cases, the `p-regular-p` condition ensures that all of the pieces that can be split are `p-similar-p` to each other, making it possible to use whatever function of two lists should process them.

7.2.3 Functions on Powerlists

When working with powerlists, many similar functions, usually small and incidental to the main theorem, are encountered. For example, it may be necessary to add all the elements of a powerlist, or find their minimum or maximum, etc. It may also be necessary to take two powerlists and return their pairwise sum, product, etc. Moreover, often similar theorems about these functions need to be proved, such as the sum (maximum, minimum) of the sum (maximum, minimum) of two powerlists is the same as the sum (maximum, minimum) of their zip. This is a perfect opportunity to use ACL2's encapsulation primitive to prove the appropriate theorem schemas, which can later be instantiated with specific functions in mind.

To illustrate the approach, consider the following encapsulation:

```
(encapsulate
 ((fn1       (x)   t)
  (fn2-accum (x y) t)
  (equiv     (x y) t))

 (local (defun fn1       (x)   (fix x)))
 (local (defun fn2-accum (x y) (+ (fix x) (fix y))))
 (local (defun equiv     (x y) (equal x y)))

 (defthm fn1-scalar
   (implies (not (powerlist-p x))
            (not (powerlist-p (fn1 x)))))

 (defthm fn2-accum-commutative
   (equiv (fn2-accum x y) (fn2-accum y x)))
 (defthm fn2-accum-associative
   (equiv (fn2-accum (fn2-accum x y) z)
          (fn2-accum x (fn2-accum y z))))

 (defcong equiv equiv (fn2-accum x y) 1)
 (defcong equiv equiv (fn2-accum x y) 2)

 (defequiv equiv))
```

This defines `fn1` as a scalar function, `fn2-accum` as an associative-commutative binary function, and `equiv` as an equivalence relation. Outside of the encapsulation, nothing is known about the functions other than the constraints proved in the encapsulate. Hence, any theorems that can be proved about these functions could also be proved about any functions that satisfy the constraints. In effect, theorems about `fn1`, `fn2-accum`, and `equiv` are theorem schemas, which can be instantiated for any suitable function. This allows the basic proof pattern to be derived once and to be used in multiple instances thereafter.

As a motivating example, consider applying `fn1` to all the elements of a powerlist, e.g., squaring all values in a powerlist. Another example uses `fn2-accum` to accumulate all the values in the powerlist into an aggregate. Both of these functions can be defined in two obvious ways, namely recursing in terms of either `p-tie` or `p-zip`. Naturally, the result is expected to be the same, regardless of which way the function is defined. So for example, the following theorem should be established:

```
(defun a-zip-fn2-accum-fn1 (x)
  (if (powerlist-p x)
      (fn2-accum (a-zip-fn2-accum-fn1 (p-unzip-l x))
                 (a-zip-fn2-accum-fn1 (p-unzip-r x)))
    (fn1 x)))
```

```
(defun b-tie-fn2-accum-fn1 (x)
  (if (powerlist-p x)
      (fn2-accum (b-tie-fn2-accum-fn1 (p-untie-l x))
                 (b-tie-fn2-accum-fn1 (p-untie-r x)))
    (fn1 x)))

(defthm a-zip-fn2-accum-fn1-same-as-b-tie-fn2-accum-fn1
  (equiv (b-tie-fn2-accum-fn1 x)
         (a-zip-fn2-accum-fn1 x)))
```

At this point, it is not clear that anything important has been accomplished. After all, the abstract theorem proved seems a bit contrived. How often does one define a function first in terms of `p-zip`, then in terms of `p-tie`? If such duplicate definitions are avoided, say by arbitrarily choosing to define them in terms of `p-tie` always, the above is wasted effort.

The following intuition will suffice to explain why this effort is important. While simple functions, such as the above, are just as easily defined in terms of `p-tie` as `p-zip`, this is not the case for more complex functions. For example, consider the function `p-ascending-p` which is true for an ascending powerlist. This is much more easily expressed in terms of `p-tie`, since it is simpler to decide when the `p-tie` of two ascending powerlists is ascending than to decide when their `p-zip` is ascending. On the other hand the function `p-batcher-merge` discussed later is naturally expressed in terms of `p-zip`, since it works by successively merging the odd- and even-indexed elements of a powerlist. Naturally, when proving theorems about `p-ascending-p`, auxiliary functions should be defined in terms of `p-tie`, so that all the recursions open up in the same way. Such a function may find the minimum of a powerlist. But when reasoning about `p-batcher-merge`, the same functions are needed, only this time it is preferable to recurse in terms of `p-zip`, so that it “opens up” the same way in an inductive proof. What is left then is the glue to tie the two definitions of each auxiliary function together. This is an explicit instance of the theorem schema above.

In fact, it should be pointed out that the creation of these theorem schemas came as a direct result of having proved a seemingly endless stream of similar small theorems. It is these theorems that formed the basis of the theorem schema above; i.e., all these abstract theorems were constructed by "unifying" needed lemmas in one specific proof of another. To reinforce this, consider the accumulators above. The scalar function `fn1` seems unnecessary, as does the equivalence relation `equiv`. It would be simpler to state the theorems purely in terms of `fn2-accum` which is the binary operator that is being abstracted and `equal`. However, the forms above were suggested by the specific instances appearing in the examples. One such instance is `minimum` where the accumulator is the `min` function and `equiv` and `fn1` are the equality and identity functions, respectively. Another instance is `list-of-type` where the accumulator is the `and` function, `equiv` the `iff` function, and `fn1` a scalar `type-p` function.

Accepting for now that this effort is not wasted, consider the following useful theorems. As expected by now, a key series of lemmas show how the functions `a-zip-fn2-accum-fn1` and `b-tie-fn2-accum-fn1` behave with respect to the constructors and destructors of `p-tie` and `p-zip`; for example, the following theorems relate `b-tie-fn2-accum-fn1` to `p-zip`:

```
(defthm zip-b-tie-fn2-accum-fn1
  (equiv (b-tie-fn2-accum-fn1 (p-zip x y))
         (fn2-accum (b-tie-fn2-accum-fn1 x)
                    (b-tie-fn2-accum-fn1 y))))
(defthm unzip-b-tie-fn2-accum-fn1
  (implies (powerlist-p x)
           (equiv (fn2-accum
                   (b-tie-fn2-accum-fn1 (p-unzip-l x))
                   (b-tie-fn2-accum-fn1 (p-unzip-r x)))
```

```
                    (b-tie-fn2-accum-fn1 x))))
```

Both of these theorems are useful in establishing the antecedent of induction hypotheses.

ACL2 provides a special type of rule which seems tailor made for this purpose. The `:definition` rule type allows alternative definitions to be specified for a given function; e.g., it allows the function `fn2-accum-fn1` to be given a definition in terms of `p-tie` and one in terms of `p-zip`, provided their equivalence can be proved. The ACL2 rewriter decides which definition to use when expanding `fn2-accum-fn1` depending on the other terms in the theorem to be proved. So for example, if a theorem contains many instances of `p-zip`, the `p-zip` definition of `fn2-accum-fn1` would be chosen. While this approach seems promising, it resulted in a significant performance degradation, because the rewriter spent too much time deciding which definition to use. As a result, the approach outlined here appears best, at least to those who prefer guiding the theorem prover towards a particular proof rather than waiting for the prover to possibly discover deep proofs on its own.

7.3 Simple Examples

In this section, various examples from [46] are formalized in ACL2. This illustrates how the primitives defined in section 7.2 are sufficient for ACL2 to prove theorems about powerlists.

Start with the `p-reverse` function, which reverses a powerlist; e.g., `p-reverse` of $\langle\langle 1.2\rangle.\langle 3.4\rangle\rangle$ is $\langle\langle 4.3\rangle.\langle 2.1\rangle\rangle$. The definition, a straight transliteration from [46], is as follows:

```
(defun p-reverse (p)
  (if (powerlist-p p)
      (p-tie (p-reverse (p-untie-r p))
             (p-reverse (p-untie-l p)))
    p))
```

Similarly, `p-reverse-zip` can be defined, reversing in terms of `p-zip` instead of `p-tie`. ACL2 can immediately verify that `p-reverse` is its own inverse. That is, it trivially accepts the following theorem:

```
(defthm reverse-reverse
  (equal (p-reverse (p-reverse x)) x))
```

Before proving that `p-reverse` and `p-reverse-zip` are equal, however, the following lemma is needed:

```
(defthm reverse-zip
  (equal (p-zip (p-reverse x) (p-reverse y))
         (p-reverse (p-zip y x))))
```

This lemma, typical of both Nqthm and ACL2 lemmas, tells ACL2 how to "push" `p-zip` into a `p-reverse`. Given this lemma, ACL2 can now easily verify the following:

```
(defthm reverse-reverse-zip
  (equal (p-reverse-zip x) (p-reverse x)))
```

It is interesting to note that the theorem above does not depend on the structure of the powerlist `x`. Specifically, there is no requirement that `x` is regular.

The functions `p-rotate-right` and `p-rotate-left` which return a rotation of their arguments are easily defined in terms of `p-zip`; indeed their simplicity is a tribute to the `p-zip` constructor:

```
(defun p-rotate-right (x)
  (if (powerlist-p x)
      (p-zip (p-rotate-right (p-unzip-r x))
             (p-unzip-l x))
    x))
(defun p-rotate-left (x)
  (if (powerlist-p x)
      (p-zip (p-unzip-r x)
             (p-rotate-left (p-unzip-l x)))
    x))
```

Again, ACL2 can prove a number of theorems unassisted. For example, it can show that `p-rotate-right` and `p-rotate-left` are inverses with the following theorem:

```
    (defthm rotate-left-right
      (equal (p-rotate-left (p-rotate-right x)) x))
```

Notice again that the theorem remains true even for arbitrary powerlists, not just regular powerlists. ACL2 can also prove the analogous theorem where the powerlist is rotated to the left first.

In addition, ACL2 proves that zip and rotate "almost" commute. The operations are not strictly commutative, because the direction of the rotation needs to be reversed:

```
    (defthm reverse-rotate
      (equal (p-reverse-zip (p-rotate-right x))
             (p-rotate-left (p-reverse-zip x))))
```

This theorem can be used to prove the following "amusing identity" due to Misra:

```
    (defthm reverse-rotate-reverse-rotate
      (equal (p-reverse-zip
               (p-rotate-right
                (p-reverse-zip
                 (p-rotate-right x))))
             x))
```

Next, consider repeated shifts. The function `p-rotate-right-k` loops over `p-rotate-right` `k` times:

```
(defun p-rotate-right-k (x k)
  (if (zp k)
      x
    (p-rotate-right (p-rotate-right-k x (1- k)))))
```

A subtler definition shifts the odd-indexed and even-indexed elements by about half of `k`, then joins the result. This is given below:

```
(defun p-rotate-right-k-fast (x k)
  (if (powerlist-p x)
      (if (integerp (/ k 2))
          (p-zip (p-rotate-right-k-fast (p-unzip-l x)
                                        (/ k 2))
                 (p-rotate-right-k-fast (p-unzip-r x)
                                        (/ k 2)))
        (p-zip (p-rotate-right-k-fast (p-unzip-r x)
                                      (1+ (/ (1- k) 2)))
               (p-rotate-right-k-fast (p-unzip-l x)
                                      (/ (1- k) 2))))
    x))
```

ACL2 can prove the equality of these two functions, but only with a certain amount of help, partly because ACL2 has a hard time reasoning about the values in `k` above.

Another function suggested by Misra is the shuffle function, which rotates not the elements of a powerlist, but their indices, based on zero-indexing. For example, the low-order bit of the index becomes the high-order bit, and hence the even-indexed elements will appear at the front of the result. This function can be defined as follows:

```
(defun p-right-shuffle (x)
  (if (powerlist-p x)
      (p-tie (p-unzip-l x) (p-unzip-r x))
    x))
```

Particularly noteworthy is that `p-right-shuffle` mixes the `p-zip` destructors with the `p-tie` constructor. Once more, ACL2 is able to prove without assistance that `p-left-shuffle` and `p-right-shuffle` are inverses:

```
(defthm left-right-shuffle
  (equal (p-left-shuffle (p-right-shuffle x)) x))
```

Notice again that the theorem is true regardless of whether the powerlist `x` is regular. This is somewhat surprising given that the functions were defined precisely with a regular powerlist in mind.

Another interesting permutation function is `p-invert` which reverses the bit vector of the index of a powerlist. This function is used, for example, in the Fast Fourier Transform algorithm. It can be defined as follows:

```
(defun p-invert (x)
  (if (powerlist-p x)
      (p-zip (p-invert (p-untie-l x))
             (p-invert (p-untie-r x)))
    x))
```

Following [46], the following lemma can be proved:

```
(defthm invert-zip
  (equal (p-invert (p-zip x y))
         (p-tie (p-invert x) (p-invert y))))
```

It is interesting that this lemma, although typical of ACL2 lemmas, was actually needed in Misra's original hand proof. As in [46], ACL2 can now prove, without user intervention, that `p-invert` is its own inverse. Moreover, it can prove that `p-invert` and `p-reverse` commute:

```
(defthm invert-invert
  (equal (p-invert (p-invert x)) x))


(defthm invert-reverse
  (equal (p-invert (p-reverse x))
         (p-reverse (p-invert x))))
```

Finally, given an arbitrary binary function `fn2` (defined in an `encapsulate` similar to the one in section 7.2.3) applied pairwise to the elements of two lists, it can be shown that `p-invert` and `fn2` commute:

```
(defthm invert-zip-fn2
  (implies (p-similar-p x y)
           (equal (p-invert (a-zip-fn2 x y))
                  (a-zip-fn2 (p-invert x)
                             (p-invert y)))))
```

7.4 Sorting Powerlists

Consider the problem of sorting a powerlist. The specification of being sorted is as follows:

```
(defun p-sorted-p (x)
  (if (powerlist-p x)
      (and (p-sorted-p (p-untie-l x))
           (p-sorted-p (p-untie-r x))
           (<= (p-max-elem (p-untie-l x))
               (p-min-elem (p-untie-r x))))
    t))
```

where the functions `p-min-elem` and `p-max-elem` return the minimum and maximum elements of a powerlist respectively. The definition of `p-min-elem` is as follows:

```
(defun p-min-elem (x)
  (if (powerlist-p x)
      (if (<= (p-min-elem (p-untie-l x))
              (p-min-elem (p-untie-r x)))
          (p-min-elem (p-untie-l x))
        (p-min-elem (p-untie-r x)))
    (realfix x)))
```

For example, the `p-min-elem` of $\langle 3, 2, 4, 6\rangle$ is 2. Notice how `p-sorted-p` is most naturally expressed in terms of `p-tie`; in fact, it is not immediately obvious how an equivalent definition can be written in terms of `p-zip`. For this reason, `p-min-elem` in also defined in terms of `p-tie`, though it could just as easily have been defined in terms of `p-zip`. However, since it is likely that reasoning about `p-zip` will be needed in the future, theorems such as the following should prove useful:

```
(defthm min-elem-zip
  (equal (p-min-elem (p-zip x y))
         (if (<= (p-min-elem x) (p-min-elem y))
             (p-min-elem x)
           (p-min-elem y))))

(defthm min-elem-unzip
  (implies (powerlist-p x)
           (and (>= (p-min-elem (p-unzip-l x))
                    (p-min-elem x))
                (>= (p-min-elem (p-unzip-r x))
                    (p-min-elem x)))))
```

Both of these theorems are instances of generic theorems proved in section 7.2.3, so ACL2 does not need to perform added work in proving them (given an appropriate hint to instantiate the generic theorems). Moreover, since different sorting algorithms are likely to require similar theorems about `p-min-elem`, `p-sorted-p`, and so on, it pays to prove these up front. For example, it can be established once and for all that the minimum of a powerlist is no larger than its maximum. It can also be proved how `p-sorted` behaves in the presence of `p-zip`, etc.

Another requirement of a sorting algorithm is that it return a permutation of its argument. To ensure this, the following function can be used, returning the number of times a given argument appears in a powerlist:

```
(defun p-member-count (x m)
  (if (powerlist-p x)
      (+ (p-member-count (p-untie-l x) m)
         (p-member-count (p-untie-r x) m))
    (if (equal x m) 1 0)))
```

Again, it is useful to prove basic theorems about `p-member-count`, such as how it behaves with `p-zip`, since these lemmas will likely prove useful to any sorting algorithm.

In summary, a proposed sorting algorithm `p-sort` should satisfy the following theorems:

- `(p-sorted-p (p-sort x))`

-
  ```
  (equal (p-member-count (p-sort x) m)
         (p-member-count x m))
  ```

Of course, specific sorting routines may impose restrictions on the original powerlist `x`, e.g., a routine may only work with numeric lists.

7.4.1 Merge Sorting

Merge sort is a very natural parallel sorting algorithm. An abstract merge sort over powerlists can be defined as follows:

```
(defun my-merge-sort (x)
  (if (powerlist-p x)
      (p-merge (my-merge-sort (p-split-1 x))
               (my-merge-sort (p-split-2 x)))
    x))
```

The functions `p-merge`, and `p-split-1` and `p-split-2` instantiate specific merge sort algorithms. Classically, `p-merge` will be a complicated function and the split functions will be trivial. These functions and their relevant theorems can be encapsulated, and the correctness of this generic merge sort should be provable from the constraints on these functions. In particular, the following theorems should be established:

```
(defthm merge-sort-is-permutation
  (implies (p-sortable-p x)
           (equal (p-member-count (p-merge-sort x) m)
                  (p-member-count x m))))

(defthm merge-sort-sorts-input
  (implies (p-sortable-p x)
           (p-sorted-p (p-merge-sort x))))
```

The `p-sortable-p` goal permits merge algorithms that only work for a subclass of powerlists; the forthcoming Batcher merge, which only works for regular powerlists, is an example of such an algorithm.

In order to prove the theorems above, the following assumptions about the generic merge functions are needed:

```
(encapsulate
 ((p-sortable-p (x) t)
  (p-mergeable-p (x y) t)
  (p-split-1 (x) t)
  (p-split-2 (x) t)
  (p-merge (x y) t)
  (p-merge-sort (x) x))
 ...
 (defthm *obligation*-split-reduces-count
   (implies (powerlist-p x)
            (and (e0-ord-< (acl2-count (p-split-1 x))
                           (acl2-count x))
                 (e0-ord-< (acl2-count (p-split-2 x))
                           (acl2-count x)))))

 (defthm *obligation*-member-count-of-splits
   (implies (powerlist-p x)
            (equal (+ (p-member-count (p-split-1 x) m)
                      (p-member-count (p-split-2 x) m))
                   (p-member-count x m))))

 (defthm *obligation*-member-count-of-merge
   (implies (p-mergeable-p x y)
            (equal (p-member-count (p-merge x y) m)
                   (+ (p-member-count x m)
                      (p-member-count y m)))))

 (defthm *obligation*-sorted-merge
   (implies (and (p-mergeable-p x y)
                 (p-sorted-p x)
```

```
                    (p-sorted-p y))
               (p-sorted-p (p-merge x y))))

(defthm *obligation*-merge-sort
  (equal (p-merge-sort x)
         (if (powerlist-p x)
             (p-merge (p-merge-sort (p-split-1 x))
                      (p-merge-sort (p-split-2 x)))
           x)))

(defthm *obligation*-sortable-split
  (implies (and (powerlist-p x)
                (p-sortable-p x))
           (and (p-sortable-p (p-split-1 x))
                (p-sortable-p (p-split-2 x)))))

(defthm *obligation*-sortable-mergeable
  (implies (and (powerlist-p x)
                (p-sortable-p x))
           (p-mergeable-p (p-merge-sort
                           (p-split-1 x))
                          (p-merge-sort
                           (p-split-2 x))))))
```

Recall, however, that before ACL2 accepts such an `encapsulate` event, it must be given a witness function; that is, an implementation of such a merging scheme. The easiest route is to use a vacuous merger, i.e., by locally defining `p-sortable-p` to be `nil`.

7.4.2 Batcher Sorting

One of Batcher's sorting algorithms can be defined as follows [4]:

```
(defun p-batcher-sort (x)
  (if (powerlist-p x)
      (p-batcher-merge (p-batcher-sort (p-untie-l x))
                       (p-batcher-sort (p-untie-r x)))
    x))
```

The definition follows the pattern of the generic merging algorithm introduced in the previous section, using `p-batcher-merge` as the merge operation. Therefore, the correctness of this function can be verified simply by verifying that `p-batcher-merge` correctly merges two sorted lists.

The Batcher merge is given by the following:

```
(defun p-batcher-merge (x y)
  (if (powerlist-p x)
      (p-zip (p-min (p-batcher-merge (p-unzip-l x)
                                     (p-unzip-r y))
                    (p-batcher-merge (p-unzip-r x)
                                     (p-unzip-l y)))
             (p-max (p-batcher-merge (p-unzip-l x)
                                     (p-unzip-r y))
                    (p-batcher-merge (p-unzip-r x)
                                     (p-unzip-l y))))
    (p-zip (p-min x y) (p-max x y))))
```

The functions `p-min` and `p-max` return respectively the pairwise minimum and maximum of two powerlists. Since `p-zip` features prominently in the definition of `p-batcher-merge`, it is not surprising to find `p-min` and `p-max` similarly defined.

At first glance, the definition of `p-batcher-merge` looks straight-forward. Certainly, it seems that a straight-forward structural induction should be sufficient to prove all the properties about it one would wish. Howerver, there are two imposing challenges ahead. The first is that `p-batcher-merge` is defined in terms of `p-zip`, whereas the target predicate `p-sorted-p` is defined in terms of `p-tie`. This is especially troublesome because `p-batcher-merge` does not recurse evenly through its arguments. Notice in particular how the the *left* unzip of `x` is merged with the *right* unzip of `y`, and vice versa.

Consider first the proof of the following goal:

```
(equal (p-member-count (p-batcher-merge x y) m)
       (+ (p-member-count x m)
          (p-member-count y m)))
```

Since `p-min` and `p-max` operate on the pairwise points of `x` and `y`, it is reasonable to require that `x` and `y` be similar. Moreover, since `p-batcher-merge` is recursing on opposite halves of `x` and `y`, it is reasonable to expect the powerlists must also be regular. Moreover, it will be necessary to constrain the powerlist to contain only real numbers. This is because the ordering imposed by `p-max` is only well-defined over the reals. Of course, it will be necessary to prove the theorems that all intermediate results satisfy the structural requirements of the hypothesis; i.e., for similar `x` and `y` their `p-min` and `p-max` are also similar, etc.

The goal becomes the following:

```
(defthm member-count-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (p-member-count
                    (p-batcher-merge x y) m)
                  (+ (p-member-count x m)
                     (p-member-count y m)))))
```

To prove the above claim, it must be established that all the values of `x` and `y` can be found somewhere in their `p-min` and `p-max`. This can be proved generically; that is, it can be shown that the sum of any scalar function over `x` and `y` is unaffected by `p-min` and `p-max`:

```
(defthm a-zip-plus-fn1-of-min-max
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (+ (a-zip-plus-fn1 (p-max x y))
                     (a-zip-plus-fn1 (p-min x y)))
                  (+ (a-zip-plus-fn1 x)
                     (a-zip-plus-fn1 y)))))
```

Notice that this is an extension of the generic theorems defined in section 7.2.3 to include specific functions, such as `p-min` and `p-max`. The function `a-zip-plus-fn1` is an instance of `a-zip-fn2-accum-fn1`; it finds the sum of all the elements in a powerlist. From the generic lemmas proved in section 7.2.3, it follows that `a-zip-plus-fn1` is equivalent to `b-tie-plus-fn1`, which is analogous to `b-tie-fn2-accum-fn1`. Using this lemma, the similar result for `p-batcher-merge` follows easily:

```
(defthm a-zip-plus-fn1-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (a-zip-plus-fn1 (p-batcher-merge x y))
                  (+ (a-zip-plus-fn1 x)
                         (a-zip-plus-fn1 y)))))
```

By instantiating `fn1` with the pseudo-function `(lambda (x) (if (= x m) 1 0))` and using the equivalence of `a-zip-plus-fn1` and `b-tie-plus-fn1`, the theorem `member-count-of-merge` follows trivially.

Notice above how all the reasoning was done with respect to `p-zip`, and only the last step appealed to the equivalence of `p-member-count` as defined in terms of `p-zip` and `p-tie` to complete the proof.

It is time to tackle the question of when `p-batcher-merge` returns a sorted powerlist. The recursive step returns a powerlist of the form

```
(p-zip (p-min (p-batcher-merge X1 Y2)
              (p-batcher-merge X2 Y1))
       (p-max (p-batcher-merge X1 Y2)
              (p-batcher-merge X2 Y1)))
```

From the inductive hypothesis it will follow that both `(p-batcher-merge X1 Y2)` and `(p-batcher-merge X2 Y1)` are sorted. It is natural to ask, therefore, whether `(p-zip (p-min X Y) (p-max X Y))` is sorted, given sorted `X` and `Y`. Unfortunately, this is not the case, as the powerlists $\langle 1, 2\rangle$ and $\langle 3, 4\rangle$ demonstrate. The problem is that the `p-min` of 2 and 4 is 2, which is smaller than the `p-max` of 1 and 3. What is needed is an assurance that the elements of the lists are not only sorted independently, but that one lists does not "grow" too much faster than the other.

Consider $X = \langle x_1, x_2, x_3, x_4\rangle$ and $Y = \langle y_1, y_2, y_3, y_4\rangle$. The condition amounts to the following:

$$\max(x_i, y_i) \leq \min(x_j, y_j)$$

for all indices $i < j$. This condition automatically implies that X and Y are sorted. In ACL2, the required property can be defined as follows:

```
(defun p-interleaved-p (x y)
      (if (powerlist-p x)
          (and (powerlist-p y)
               (p-interleaved-p (p-untie-l x) (p-untie-l y))
               (p-interleaved-p (p-untie-r x) (p-untie-r y))
               (<= (p-max-elem (p-untie-l x))
                   (p-min-elem (p-untie-r x)))
               (<= (p-max-elem (p-untie-l x))
                   (p-min-elem (p-untie-r y)))
               (<= (p-max-elem (p-untie-l y))
                   (p-min-elem (p-untie-r x)))
               (<= (p-max-elem (p-untie-l y))
                   (p-min-elem (p-untie-r y))))
        (not (powerlist-p y))))
```

For example, the powerlists $\langle 1, 2\rangle$ and $\langle 3, 4\rangle$ do not satisfy `p-interleaved-p`, but the powerlists $\langle 1, 4\rangle$ and $\langle 2, 3\rangle$ do.

If `(p-interleaved-p x y)` is true, `(p-zip (p-min x y) (p-max x y))` is sorted. Intuitively, this is a simple result. In the example above, the first two elements of Z will be x_1 and y_1, in ascending order. Moreover, the hypothesis assures us these two numbers are the smallest of the x_j and y_j for $j \geq 2$. A similar argument will work for x_2 and y_2, and so on.

To prove the claim in ACL2, it is necessary to reason about the interaction of `p-min` and `p-min-elem`, as well as their `max` counterparts. Since `p-min` is defined in terms of `p-zip` and `p-min-elem` in terms of `p-tie`, it is easier to prove this theorem in terms of a single recursive scheme, say `p-tie` and then use the bridging lemmas to prove the result:

```
    (defthm zip-min-max-sorted-if-interleaved
      (implies (and (p-interleaved-p x y)
                    (p-similar-p x y)
                    (p-number-list x)
                    (p-number-list y))
               (p-sorted-p (p-zip (p-min x y)
                                  (p-max x y)))))
```

Again, it is easier to prove this theorem first for versions of `p-min` and `p-max` defined in terms of `p-tie` instead of `p-zip`, since `p-sorted-p` is defined in terms of `p-tie`.

It only remains to be shown that the recursive calls to `p-batcher-merge` return `p-interleaved-p` lists. That is, given sorted `X` and `Y`,

```
L1 = (p-batcher-merge (p-unzip-l X) (p-unzip-r Y))
L2 = (p-batcher-merge (p-unzip-r X) (p-unzip-l Y))
```

are `p-interleaved-p`. Intuitively, it is clear why this must be the case. It can be assumed that both `L1` and `L2` are sorted, since this fact will follow from the induction hypothesis. Any prefix of `L1` will have some values from `X` and some from `Y`, say i and j values respectively. Moreover, since `L1` has only odd-indexed elements of `X` and `L2` only the even-indexed elements of `X`, no prefix of `L1` can have more elements from `X` than the corresponding prefix of `L2`, and similarly for the elements from `Y`. For example, suppose that `L1` starts with x_1 and x_3, but the corresponding prefix of `L2` does not contain x_2. In this case, `L2` must start with y_1 and y_3, which means that $y_3 < x_2$, since `L2` is sorted and its prefix does not contain x_2. But, it follows from `L1` that $x_3 \leq y_2$, since `L1` is also sorted. Therefore, $x_3 \leq y_2 \leq y_3 < x_2$ and $x_3 < x_2$. But this is a contradiction, since X is sorted.

Formalizing the argument given above places a severe challenge on the powerlist paradigm, since the reasoning involves indices so explicitly, whereas powerlists do away with the index concept. In fact, the whole concept of "prefix" is strange, since these prefixes will by definition be irregular, and it has already been observed how `p-batcher-merge` requires regular arguments. This calls for a little subtlety. The "prefix" concept can be replaced with the following:

```
(defun p-member-count-<= (x m)
  (if (powerlist-p x)
      (+ (p-member-count-<= (p-untie-l x) m)
         (p-member-count-<= (p-untie-r x) m))
    (if (<= (realfix x) m) 1 0)))
```

This returns the number of elements in `x` which are less than or equal to `m`; that is, for an element `m` in `x`, it returns its (largest) index in `x`. With this notion, the argument involving the “prefix” of a powerlist can be formalized.

Consider expressions of the form

```
M1 = (p-member-count-<= (p-batcher-merge (p-unzip-l x)
                                         (p-unzip-r y))
                        m)
M2 = (p-member-count-<= (p-batcher-merge (p-unzip-r x)
                                         (p-unzip-l y))
                        m)
```

The following theorem shows how these terms can be simplified:

```
(defthm member-count-<=-of-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y))
           (equal (p-member-count-<=
                     (p-batcher-merge x y)
                     m)
                  (+ (p-member-count-<= x m)
                     (p-member-count-<= y m)))))
```

This theorem justifies the removal of `p-batcher-merge` from the computation of `p-member-count-<=`. The following terms remain:

```
M1 = (+ (p-member-count-<= (p-unzip-l x) m)
        (p-member-count-<= (p-unzip-r y) m))
M2 = (+ (p-member-count-<= (p-unzip-r x) m)
        (p-member-count-<= (p-unzip-l y) m))
```

So the next step is to compare the `p-member-count-<=` of the `p-unzip-l` and `p-unzip-r` of a powerlist, specifically a *sorted* powerlist. Intuitively, these are expected to differ by no more than 1; moreover, since the `p-unzip-r` starts counting from the second position, its `p-member-count-<=` should be smaller than that of the `p-unzip-l`. This intuition suggests the following theorems:

```
(defthm member-count-<=-of-sorted-unzips-1
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
           (<= (p-member-count-<= (p-unzip-r x) m)
               (p-member-count-<= (p-unzip-l x) m))))

(defthm member-count-<=-of-sorted-unzips-2
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-sorted-p x))
           (<= (p-member-count-<= (p-unzip-l x) m)
               (1+ (p-member-count-<= (p-unzip-r x)
                                       m)))))
```

All these results can be combined into a single theorem stating that `M1` and `M2` differ by no more than 1:

```
(defthm member-count-<=-of-merge-unzips
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y))
           (let ((M1 (p-member-count-<= (p-batcher-merge
                                          (p-unzip-l x)
                                          (p-unzip-r y))
                                         m))
                 (M2 (p-member-count-<= (p-batcher-merge
                                          (p-unzip-r x)
                                          (p-unzip-l y))
                                         m)))
             (or (equal M1 M2)
                 (equal (1+ M1) M2)
                 (equal (1+ M2) M1)))))
```

The next step is to show that for non `p-interleaved-p` lists, there is some `m` so that the respective `p-member-count-<=` differ by more than 1. This `m` can be found by making a "cut" through the two lists at the precise spot where they fail the `p-interleaved-p` test. The following function performs such a "cut":

```
(defun interleaved-p-cutoff (x y)
  (if (and (powerlist-p x) (powerlist-p y))
      (cond ((< (p-min-elem (p-untie-r x))
                (p-max-elem (p-untie-l x)))
```

```
                    (p-min-elem (p-untie-r x)))
                   ((< (p-min-elem (p-untie-r x))
                       (p-max-elem (p-untie-l y)))
                    (p-min-elem (p-untie-r x)))
                   ((interleaved-p-cutoff (p-untie-l x)
                                          (p-untie-l y))
                    (interleaved-p-cutoff (p-untie-l x)
                                          (p-untie-l y)))
                   ((interleaved-p-cutoff (p-untie-r x)
                                          (p-untie-r y))
                    (interleaved-p-cutoff (p-untie-r x)
                                          (p-untie-r y))))
          nil))
```

When `x` and `y` are `p-interleaved-p`, the function `interleaved-p-cutoff` will return `nil`. In all other cases, it returns a valid choice of `m` as a counterexample in `member-count-<=-of-merge-unzips`. Thus, the `interleaved-p-cutoff` of $\langle 1,4\rangle$ and $\langle 2,3\rangle$ is `nil`, but that of $\langle 1,2\rangle$ and $\langle 3,4\rangle$ is 2.

The first observation can be verified with the following theorem:

```
(defthm interleaved-p-if-nil-cutoff
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (not (numericp
                      (interleaved-p-cutoff x y)))
                (not (numericp
                      (interleaved-p-cutoff y x))))
           (p-interleaved-p x y)))
```

In order to establish that `interleaved-p-cutoff` finds a valid counterexample when `x` and `y` are not `p-interleaved-p`, notice that `interleaved-p-cutoff` always returns an element of `x`, and furthermore for sorted `x` this value `m` is such that its "index" in `x` is at least one more than its "index" in `y`, since it must satisfy

```
(< (p-min-elem (p-untie-r x))
   (p-max-elem (p-untie-l y)))
```

for some corresponding subtree of `x` and `y`. In ACL2, this establishes the following theorem:

```
(defthm member-count-diff-2-if-interleaved-cutoff-sorted
  (implies (and (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y)
                (interleaved-p-cutoff x y))
           (< (1+ (p-member-count-<=
                   y
                   (interleaved-p-cutoff x y)))
              (p-member-count-<=
               x
               (interleaved-p-cutoff x y)))))
```

The counterexample needed by the lemmas `member-count-<=-of-merge-unzips` and `interleaved-p-if-nil-cutoff` can be found using this theorem. The preceding results can be summarized as follows:

```
(defthm inner-batcher-merge-call-is-interleaved-p
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y)
                (p-sorted-p
                 (p-batcher-merge (p-unzip-l x)
                                  (p-unzip-r y)))
                (p-sorted-p
                 (p-batcher-merge (p-unzip-r x)
                                  (p-unzip-l y))))
           (p-interleaved-p
            (p-batcher-merge (p-unzip-l x)
                             (p-unzip-r y))
            (p-batcher-merge (p-unzip-r x)
                             (p-unzip-l y)))))
```

From this point, the remainder of the proof is almost propositional. The inductive case of the correctness of `batcher-merge` follows directly from the lemma `inner-batcher-merge-call-is-interleaved-p`. Notice that the inductive hypothesis shares the antecedent of `inner-batcher-merge-call-is-interleaved-p`.

```
(defthm recursive-batcher-merge-is-sorted
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
```

```
               (p-sorted-p x)
               (p-sorted-p y)
               (p-sorted-p
                (p-batcher-merge (p-unzip-l x)
                                 (p-unzip-r y)))
               (p-sorted-p
                (p-batcher-merge (p-unzip-r x)
                                 (p-unzip-l y))))
          (p-sorted-p (p-batcher-merge x y))))
```

The main result, establishing the correctness of Batcher merging, is an almost immediate corollary of the above:

```
(defthm sorted-merge
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (p-number-list x)
                (p-number-list y)
                (p-sorted-p x)
                (p-sorted-p y))
           (p-sorted-p (p-batcher-merge x y))))
```

With the theorem above and the meta-theorems about merge sorts proved in section 7.4.1, the correctness of Batcher sorting can be easily established:

```
(defthm batcher-sort-is-permutation
  (implies (and (p-regular-p x)
                (p-number-list x))
           (equal (p-member-count (p-batcher-sort x) m)
                  (p-member-count x m))))
```

```
(defthm batcher-sort-sorts-inputs
  (implies (and (p-regular-p x)
                (p-number-list x))
           (p-sorted-p (p-batcher-sort x))))
```

These theorems are instances of the generic merge sorting theorems proved in section 7.4.1.

7.4.3 A Comparison with the Hand-Proof

It is instructive to compare the machine-verified proof of section 7.4.2 with the hand-proof provided in [46] and verified in [33].

The proof starts by defining the function z as follows:

$$\begin{aligned} z(\langle x\rangle) &= 1 \text{ if } x = 0,\ 0 \text{ otherwise} \\ z(p \bowtie q) &= z(p) + z(q) \end{aligned}$$

That is, $z(x)$ counts the number of zeros in x. Assuming that all powerlists range only over 0's and 1's, this yields the following characterization of sorted powerlists:

$$\begin{aligned} &sorted(\langle x\rangle) \\ sorted(p \bowtie q) &= sorted(p) \wedge sorted(q) \wedge 0 \le z(p) - z(q) \le 1 \end{aligned}$$

The 0-1 assumption completely characterizes the pairwise minimum and maximum of two sorted lists as follows:

$$\begin{aligned} min(x,y) &= x, \text{ if } sorted(x), sorted(y), \text{ and } z(x) \ge z(y) \\ max(x,y) &= y, \text{ if } sorted(x), sorted(y), \text{ and } z(x) < z(y) \end{aligned}$$

Moreover, the following key lemma can be established:

$$sorted(min(x,y) \bowtie max(x,y)) \quad \text{if } sorted(x), sorted(y), \text{ and } |z(x) - z(y)| \le 1$$

With some algebraic reasoning, this yields the main correctness result:

$$sorted(pbm(x,y)) \quad \text{if } sorted(x) \text{ and } sorted(y)$$

where pbm is the Batcher merge function on powerlists.

This proof is much simpler than that given in section 7.4.2, and that may be taken as an indication that ACL2 is ineffective in reasoning about powerlists. However, such a conclusion is premature. In fact, ACL2 can verify the reasoning given above without too much difficulty. But the end result would not be as satisfying as the main theorems proven in 7.4.2 for a number of reasons. First, the hand proof relies on the 0/1 principle, which states that any sorting algorithm based on comparing pairs of elements from a list will correctly sort an arbitrary list if it correctly sorts all lists consisting exclusively of zeros and ones. The formal proof in the powerlist logic proves the correctness only for lists of zeros and ones, and then it uses the 0/1 principle to "lift" this proof to the arbitrary case. But the 0/1 principle is certainly not obvious; if anything, it is more surprising than the proof of Batcher merge itself. For instance, proving the 0/1 principle in ACL2 would be extremely difficult, if not impossible, because the principle references all possible sorting functions based on comparing input pairs, and there is no apparent way to express this notion in the logic of ACL2.

Another problem with the hand proof is that the definition of $sorted$ used is not the same as the "standard" definition of a sorted list. It is *only* true for lists of 0's and 1's, and it is not immediately clear how this property compares to the usual notion of sorted lists. The definition supplied, however, is extremely useful since it is based on `zip` instead of `tie`, so it works more naturally with the definition of Batcher merge. However, the proof of the equivalence of the two definitions is missing. This is especially important if Batcher sorting were being used as part of a more complex function, e.g. a search routine, since the key property required in the complex function — that Batcher sort correctly sorts its input — has not been established yet.

In fact, it is fair to say that the hand proof presents a mixture of formal reasoning and informal arguments. Such a mixture is extremely convenient when generating the proof by hand, but it can also be the source of subtle errors, such as the failure to identify needed hypotheses.

7.5 Prefix Sums of Powerlists

Prefix sums appear in many applications, e.g., arithmetic circuit design. For a powerlist $X = \langle x_1, x_2, \ldots, x_n \rangle$, its prefix sum is given by $ps(X) = \langle x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n \rangle$. The operator $\oplus$ is an arbitrary binary operator; for the purposes of this section, it can be assumed to be associative and to have a left-identity 0.

The functions `bin-op` and `left-zero` encapsulate the binary operator $\oplus$ and its left identity, respectively. By using ACL2's `encapsulate`, the following theorems are all really theorem schemas which can be instantiated with any suitable operator, e.g, `plus`, `and`, `min`, etc. The required axioms are as follows:

```
(encapsulate
 ((domain-p (x) t)
  (bin-op (x y) t)
  (left-zero () t))

 (defthm booleanp-domain-p
   (booleanp (domain-p x)))

 (defthm scalar-left-zero
   (domain-p (left-zero)))

 (defthm domain-powerlist
   (implies (domain-p x)
            (not (powerlist-p x))))
 (defthm left-zero-identity
   (implies (domain-p x)
            (equal (bin-op (left-zero) x) x)))

 (defthm bin-op-assoc
   (equal (bin-op (bin-op x y) z)
          (bin-op x (bin-op y z))))

 (defthm scalar-bin-op
   (domain-p (bin-op x y)))
)
```

The function `domain-p` recognizes the intended domain which is required to be scalar, i.e. non-powerlist. The function `p-domain-list-p` extends `domain-p` over powerlists; i.e., it recognizes powerlists of `domain-p` elements. Note that the second argument to `bin-op` is required to be `domain-p` in `left-zero-identity`, but that `domain-p` is not a requirement of `bin-op-assoc`, and furthermore that `domain-p` is always true of the result of `bin-op`. This turns out to be important, in that ACL2 defines many binary operators that meet these requirements precisely. Moreover, at least one of these theorems needs to have `domain-p` as a hypothesis. For example, if the hypothesis is removed from `left-zero-identity`, then for an arbitrary powerlist x, it would follow that $0 \oplus x = x$ and so $\oplus$ would not always return a scalar.

There is a natural definition of prefix sums in terms of indices. That is, entry y_j in the prefix sum of X is equal to the sum of all the x_i up to x_j. However, this definition does not extend nicely to powerlists, since the two halves of a prefix sum are not themselves prefix sums. The trick is to generalize prefix sums to allow an arbitrary value to be added to the first element, in a manner analogous to a carry-in bit. This leads to the following definitions:

```
(defun p-prefix-sum-aux (prefix x)
  (if (powerlist-p x)
      (p-tie (p-prefix-sum-aux prefix (p-untie-l x))
             (p-prefix-sum-aux (p-last (p-prefix-sum-aux
                                         prefix
                                         (p-untie-l x)))
                               (p-untie-r x)))
    (bin-op prefix x)))
(defmacro p-prefix-sum (x)
  ‘(p-prefix-sum-aux (left-zero) ,x))
```

where `p-last` returns the last element of a powerlist. Most of the following theorems will be about `p-prefix-sum-aux`, though a few will have to be proved exclusively for `p-prefix-sum`. Note, `p-prefix-sum-aux` could have been defined to pass the sum of the left half of `x` instead of the last element of the left prefix sum. The current definition was preferred, simply because it is closer to the usual way the author computes prefix sums.

7.5.1 Simple Prefix Sums

The definition of `p-prefix-sum` is inherently sequential. In this section, it is shown that the following definition, more amenable to a parallel implementation, is equivalent:

```
(defun p-simple-prefix-sum (x)
  (if (powerlist-p x)
      (let ((y (p-add (p-star x) x)))
        (p-zip (p-simple-prefix-sum (p-unzip-l y))
               (p-simple-prefix-sum (p-unzip-r y))))
    x))
```

The function `p-add` returns the pairwise sum of two powerlists, and `p-star` shifts a powerlist to the right, prefixing the result with `left-zero`:

```
(defun p-star (x)
  (if (powerlist-p x)
      (p-zip (p-star (p-unzip-r x)) (p-unzip-l x))
    (left-zero)))
(defun p-add (x y)
  (if (powerlist-p x)
      (p-zip (p-add (p-unzip-l x) (p-unzip-l y))
             (p-add (p-unzip-r x) (p-unzip-r y)))
    (bin-op x y)))
```

An immediate problem is that ACL2 does not accept the definition given above for `p-simple-prefix-sum`. The difficulty is that the definition recurses with `x` changing to `(p-unzip-l (p-add (p-star x) x))` and the latter term is not obviously "smaller" than `x`. Therefore, ACL2 can not prove that the recursive definition is well-founded. To circumvent this, a new "measure" on powerlists is needed, one that is reduced when `(p-unzip-l (p-add (p-star x) x))` is substituted for `x`:

```
(defun p-measure (x)
  (if (powerlist-p x)
      (+ (p-measure (p-unzip-l x))
         (p-measure (p-unzip-r x)))
    1))
```

The measure, in effect, counts the number of elements in a powerlist. Intuitively, `p-star` and `p-add` should not affect the measure of a powerlist, while `p-unzip-l` and `p-unzip-r` should halve it. The first observation can be verified with the following pair of theorems:

```
    (defthm measure-star
      (equal (p-measure (p-star x)) (p-measure x)))

    (defthm measure-add
      (<= (p-measure (p-add x y)) (p-measure x)))
```

The second observation does not need an explicit lemma, because ACL2 can tell from the definition of `p-measure` that both `p-unzip-l` and `p-unzip-r` reduce the `p-measure` by at least 1. This means that when `p-simple-prefix-sum` is introduced, ACL2 needs a hint to use `p-measure` to prove that the definition is well-founded, and then it is able to accept the definition.

Now consider the correctness of `p-simple-prefix-sum`. The definition of this function suggests two possible approaches: explore the powerlist given by `(p-add (p-star x) x)`, or consider the `unzip` of the prefix sum of `x`. The first approach seems more promising. Recall that `p-star` shifts its argument to the right, and that `p-add` returns a pairwise sum. Thus, for `x` given by

$$X \quad = \quad \langle x_1, x_2, x_3, \ldots, x_n \rangle$$

`(p-add (p-star x) x)` is

$$Y \quad = X^* \oplus X = \quad \langle x_1, x_1 \oplus x_2, x_2 \oplus x_3, \cdots, x_{n-1} \oplus x_n \rangle$$

Taking the `p-unzip` of this powerlist, gives the following:

$$Y_1 \quad = \quad \langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$$
$$Y_2 \quad = \quad \langle x_1 \oplus x_2, x_3 \oplus x_4, \ldots, x_{n-1} \oplus x_n \rangle$$

where $Y = Y_1 \bowtie Y_2$. It is clear now that indeed the prefix sum of Y_1 yields precisely the odd-indexed elements of the prefix sum of X and, similarly, the prefix sum of Y_2 yields the even-indexed elements. Intuitively, this verifies the correctness of `p-simple-prefix-sum`. To formalize the argument, it will be convenient to think of Y_1 and Y_2 not as components of Y, but as two separate lists in their own right. This removes the awkward reference to `p-unzip` and allows the derivation of Y_1 and Y_2 in a way more amenable to reasoning about `p-prefix-sum`. Consider this new characterization of Y_2:

```
    (defun p-add-right-pairs (x)
      (if (powerlist-p x)
          (if (powerlist-p (p-untie-l x))
              (p-tie (p-add-right-pairs (p-untie-l x))
                     (p-add-right-pairs (p-untie-r x)))
            (bin-op (p-untie-l x) (p-untie-r x)))
        x))
```

This redefinition of Y_2 is especially useful because it is in terms of `p-tie`, not `p-zip`, so it will be easier to reason about its `p-prefix-sum`. To begin with, it is trivial to characterize the prefix sum of the `p-add-right-pairs` of a two-element powerlist — note that a two-element powerlist is the natural base case for an induction, since `p-add-right-pairs` is only reasonable over non-singleton arguments. In particular, it can be proved that for a powerlist $X = \langle x_1, x_2 \rangle$, both the prefix sum of its `p-add-right-pairs` and the right unzip of its prefix sum are equal to $x_1 \oplus x_2$:

```
    (defthm prefix-sum-p-add-right-pairs-base
      (implies (and (domain-p val)
                    (powerlist-p x)
                    (not (powerlist-p (p-untie-l x)))
                    (p-regular-p x)
                    (p-domain-list-p x))
               (and (equal (p-prefix-sum-aux
                            val
                            (p-add-right-pairs x))
                           (bin-op val
                                   (bin-op (p-untie-l x)
                                           (p-untie-r x))))
                    (equal (p-unzip-r
                            (p-prefix-sum-aux val x))
                           (bin-op val
                                   (bin-op (p-untie-l x)
                                           (p-untie-r x))))
               )))
```

The definition of `p-prefix-sum` uses the last element of the left prefix sum to compute the right prefix sum. This suggests the following important lemma:

```
    (defthm p-last-p-prefix-sum-p-add-right-pairs
      (implies (and (domain-p val)
                    (p-regular-p x)
                    (p-domain-list-p x))
               (equal (p-last (p-prefix-sum-aux
                                val
                                (p-add-right-pairs x)))
                      (p-last (p-prefix-sum-aux val x)))))
```

This provides an important bridge in any induction involving `p-prefix-sum-aux` of `p-add-right-pairs`. Such an induction can establish that the prefix sum of `p-add-right-pairs` computes the right unzip of the prefix sum of a powerlist:

```
    (defthm prefix-sum-p-add-right-pairs
      (implies (and (domain-p val)
                    (powerlist-p x)
                    (p-regular-p x)
                    (p-domain-list-p x))
               (equal (p-prefix-sum-aux
                       val
                       (p-add-right-pairs x))
                      (p-unzip-r
                       (p-prefix-sum-aux val x)))))
```

The second half of the proof is similar. Consider `p-add-left-pairs`, which is a new characterization of $Y_1 = \langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$:

```
(defun p-add-left-pairs (val x)
  (if (powerlist-p x)
      (if (powerlist-p (p-untie-l x))
          (p-tie (p-add-left-pairs val (p-untie-l x))
                 (p-add-left-pairs (p-last
                                    (p-untie-l x))
                                   (p-untie-r x)))
        (bin-op val (p-untie-l x)))
    (bin-op val x)))
```

Unfortunately, the function `p-add-left-pairs` is considerably more complicated than `p-add-right-pairs`. The reason is that in `p-add-right-pairs` there was no need for the left half of the computation to pass any information over to the right half; i.e., the two recursive calls were completely independent of each other. The net effect is that reasoning about `p-add-left-pairs` is much more difficult than reasoning about `p-add-right-pairs`. However, there is a simple way around this. Consider the powerlist $X = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ again. Shifting this powerlist yields $X' = \langle 0, x_1, x_2, x_3, \ldots, x_{n-1} \rangle$, and the `p-add-right-pairs` of this shifted powerlist is $\langle x_1, x_2 \oplus x_3, \ldots, x_{n-2} \oplus x_{n-1} \rangle$, which is precisely the same as the `p-add-left-pairs` of X. Moreover, it is clear that the prefix sum of X and the prefix sum of X' are related; specifically, the prefix sum of the shift is the shifted prefix sum. If this intuition can be formalized, the theorem about `p-`

add-right-pairs can be used to prove the analogous theorem about p-add-left-pairs, without having to reason about p-add-left-pairs at all.

Since both p-add-left-pairs and p-add-right-pairs are defined in terms of p-tie, it is convenient to redefine p-shift in terms of p-tie, rather than using the equivalent function p-star:

```
(defun p-shift (val x)
  (if (powerlist-p x)
      (p-tie (p-shift val (p-untie-l x))
             (p-shift (p-last (p-untie-l x))
                      (p-untie-r x)))
    val))
```

Consider the claim that the prefix sum and shift operators commute. This can be verified by the following theorem:

```
(defthm p-prefix-sum-p-shift
  (implies (and (domain-p c1)
                (domain-p c2)
                (p-domain-list-p x))
           (equal (p-prefix-sum-aux c1 (p-shift c2 x))
                  (p-shift (bin-op c1 c2)
                           (p-prefix-sum-aux
                            (bin-op c1 c2)
                            x)))))
```

The proof of this theorem requires a subtle induction scheme. In particular, to conclude the theorem, the following two partial prefix sums need to be considered:

```
PS1 = (p-prefix-sum-aux c1 (p-shift c2 (p-untie-l x)))
PS2 = (p-prefix-sum-aux (p-last PS1)
                        (p-shift (p-last (p-untie-l x))
                                 (p-untie-r x)))
```

In the second instance, the term `(bin-op c1 c2)` in the theorem becomes

```
    (bin-op (p-last (p-prefix-sum-aux
                     c1
                     (p-shift c2 (p-untie-l x))))
            (p-last (p-untie-l x)))
```

which using the inductive hypothesis is equal to the following:

```
    (bin-op (p-last (p-shift (bin-op c1 c2)
                             (p-prefix-sum-aux
                              (bin-op c1 c2)
                              (p-untie-l x))))
            (p-last (p-untie-l x)))
```

This term can be simplified into

```
    (p-last (p-prefix-sum-aux (bin-op c1 c2) (p-untie-l x)))
```

using the following technical lemma:

```
    (defthm binop-last-shift-prefix-sum
      (implies (domain-p c)
               (equal (bin-op (p-last
                                (p-shift
                                 c (p-prefix-sum-aux c x)))
                               (p-last x))
                       (p-last (p-prefix-sum-aux c x)))))
```

This simplification is the key step in the proof.

Having established that prefix sum and shift commute, it is now possible to return to `p-add-left-pairs`. In particular, an instance of `p-add-left-pairs` can be converted into the `p-add-right-pairs` of a shifted powerlists as follows:

```
(defthm p-add-left-pairs->p-add-right-pairs-p-shift
  (implies (and (domain-p val)
                (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x))
           (equal (p-add-left-pairs val x)
                  (p-add-right-pairs (p-shift val x)))))
```

It is now trivial to establish that

```
(p-prefix-sum-aux val (p-add-left-pairs val2 x))
```

is equal to

```
(p-prefix-sum-aux val
                  (p-add-right-pairs (p-shift val2 x)))
```

and hence to

```
(p-unzip-r (p-prefix-sum val (p-shift val2 x)))
```

and

```
(p-unzip-r (p-shift (bin-op val val2)
                    (p-prefix-sum (bin-op val val2) x)))
```

To complete the proof, the following technical lemma is required to convert the right unzip of a shift to the left unzip of the powerlist:

```
(defthm p-unzip-r-p-shift
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (not (powerlist-p val)))
           (equal (p-unzip-r (p-shift val x))
                  (p-unzip-l x))))
```

Putting it all together gives the needed characterization of the prefix sum of the `p-add-left-pairs` of a powerlist:

```
(defthm prefix-sum-p-add-left-pairs
  (implies (and (p-regular-p x)
                (p-domain-list-p x)
                (powerlist-p x)
                (domain-p val1)
                (domain-p val2))
           (equal (p-prefix-sum-aux val1
                                    (p-add-left-pairs
                                     val2 x))
                  (p-unzip-l
                   (p-prefix-sum-aux (bin-op val1 val2)
                                     x)))))
```

This is an important moment, because `prefix-sum-p-add-left-pairs` and `prefix-sum-p-add-right-pairs` together give a characterization of the *unzips* of `p-prefix-sum`. Thus, the original definition of `p-prefix-sum`, which was inherently sequential, has been replaced with an independent characterization of its unzips; this will make it much easier to prove the correctness of `p-simple-prefix-sum`.

However, `p-simple-prefix-sum` is defined in terms of `p-star` and `p-add`, and the new characterization uses `p-add-left-pairs` and `p-add-right-pairs`.

The next step is to show how these are related. To start with, consider alternative definitions of `p-star` and `p-add` using `tie` instead of `zip`; this will make it easier to reason about them and `p-add-left-pairs`/`p-add-right-pairs` together. Recall that `p-star` performs a shift operation and `p-add` a pairwise addition. The function `p-shift` has already been defined. Pairwise addition can be defined as follows:

```
(defun p-add-tie (x y)
  (if (powerlist-p x)
      (p-tie (p-add-tie (p-untie-l x) (p-untie-l y))
             (p-add-tie (p-untie-r x) (p-untie-r y)))
    (bin-op x y)))
```

ACL2 can quickly prove the equivalence of these definitions with the original ones. The following theorem is particularly useful in the current context:

```
(defthm add-star-add-tie-shift
  (implies (and (p-regular-p x)
                (p-similar-p x y))
           (equal (p-add (p-star x) y)
                  (p-add-tie (p-shift (left-zero) x)
                             y))))
```

Using `p-shift` and `p-add-tie`, it can now be proved how `p-add-left-pairs` and `p-add-right-pairs` are constructed in the definition of `p-simple-prefix-sum`:

```
(defthm zip-add-left-pairs-add-right-pairs
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x))
           (equal (p-zip (p-add-left-pairs (left-zero)
                                            x)
```

```
                             (p-add-right-pairs x))
                      (p-add (p-star x) x))))
```

At this point, the proof is almost complete. The term

```
(p-add (p-star x) x)
```

can be rewritten as

```
(p-add-tie (p-shift (left-zero) x) x)
```

Moreover, this term can be unzipped into the two terms

```
(p-add-left-pairs (left-zero) x)
(p-add-right-pairs x)
```

Finally, the prefix sum of these terms can be zipped back together to get the prefix sum of `x`. Taken together, this establishes the correctness of `p-simple-prefix-sum`:

```
(defthm simple-prefix-sum-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list-p x))
           (equal (p-simple-prefix-sum x)
                  (p-prefix-sum x))))
```

7.5.2 Ladner-Fischer Prefix Sums

[46] verifies another algorithm for computing prefix sums, this one due to Ladner and Fischer [45]:

```
(defun p-lf-prefix-sum (x)
  (if (powerlist-p x)
      (let ((y (p-lf-prefix-sum
                (p-add (p-unzip-l x) (p-unzip-r x)))))
        (p-zip (p-add (p-star y) (p-unzip-l x)) y))
    x))
```

The complexity of this algorithm is what justifies the previous usage of the name `p-simple-prefix-sum`!

As was the case with `p-simple-prefix-sum`, it is worthwhile to consider the correctness of the left and right unzips separately. The right unzip is immediate:

```
(defthm unzip-r-lf-prefix-sum
  (implies (and (powerlist-p x)
                (p-regular-p x)
                (p-domain-list-p x)
                (equal
                 (p-lf-prefix-sum (p-add (p-unzip-l x)
                                         (p-unzip-r x)))
                 (p-prefix-sum (p-add (p-unzip-l x)
                                      (p-unzip-r x)))))
           (equal
            (p-lf-prefix-sum (p-add (p-unzip-l x)
                                    (p-unzip-r x)))
            (p-unzip-r (p-prefix-sum x)))))
```

It is only necessary to recognize that

```
(p-add (p-unzip-l x) (p-unzip-r x))
```

is the same as `(p-add-right-pairs x)`. The rest follows from the lemmas proved in the previous section.

The left unzip is a little more subtle. It is equal to

```
(p-add (p-star (p-prefix-sum (p-add (p-unzip-l x)
                                    (p-unzip-r x))))
       (p-unzip-l x))
```

which can be reduced to

```
(p-add (p-star (p-unzip-r (p-prefix-sum x)))
       (p-unzip-l x))
```

This can be simplified further using the following trivial lemma:

```
(defthm unzip-l-star
  (equal (p-unzip-l (p-star x)) (p-star (p-unzip-r x))))
```

The simplified term is

```
(p-add (p-unzip-l (p-star (p-prefix-sum x)))
       (p-unzip-l x))
```

which should further simplify to

```
(p-unzip-l (p-prefix-sum x))
```

The ubiquity of `p-unzip-l` in the terms above suggest a natural generalization, which is provable by ACL2:

```
(defthm add-star-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list-p x))
           (equal (p-add (p-star (p-prefix-sum x)) x)
                  (p-prefix-sum x))))
```

This theorem, called the "Defining Equation" in [46], plays a key role in the hand proof. It will be revisited in section 7.5.3.

With the results above, it is now easy to establish that `p-lf-prefix-sum` equals `p-prefix-sum`, thus demonstrating its correctness:

```
(defthm lf-prefix-sum-prefix-sum
  (implies (and (p-regular-p x)
                (p-domain-list-p x))
           (equal (p-lf-prefix-sum x)
                  (p-prefix-sum x))))
```

7.5.3 Comparing with the Hand-Proof Again

As was the case with Batcher sorting, the hand proof given in [46] is much simpler than the machine-verified proof given above for the correctness of the prefix sum algorithms. Part of the reason is that in [46] the proof begins in media res, as it were. Instead of providing a constructive definition, the prefix sum $ps(x)$ of a powerlist x is defined as the solution to the following "defining equation":

$$z = z^* \oplus x$$

This equation is verified by the theorem `add-star-prefix-sum`.

The hand proof proceeds by applying the defining equation to derive formulas for the left and right unzip of a prefix sum. Specifically, the derivation yields the Ladner-Fischer scheme. From there, it is shown how this scheme can be algebraically simplified to yield the simple prefix sum algorithm.

However, as section 7.5.2 testifies, establishing the correctness of the defining equation requires a fair amount of effort, and once it is established the remainder of the Ladner-Fischer proof is relatively simple.

The extra difficulty observed in the previous sections is a direct result of insisting that the specifications, i.e., defining axioms, be constructive and readily accepted. This insistence is necessary in the context of machine verification, where faith in a mechanically verified proof should not be undermined by the necessity for a large unstated theory which has only been verified by human hands.

Moreover, requiring that correctness be established with respect to generally accepted specifications is a necessity if the proof is to be used in part of a larger project. For example, prefix sums appear in many applications, so it is not surprising to find a prefix sum computation in the middle of a complex algorithm. However, in establishing the correctness of the embedding algorithm, the important property of prefix sums is that prefix sum of $X = \langle x_1, x_2, \ldots, x_n\rangle$ is in fact $ps(X) = \langle x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n\rangle$. An equivalent correctness result, such as the defining equation above, will not help immediately. In the next section, an example of an embedded prefix sum is presented.

7.5.4 L'agniappe: A Carry-Lookahead Adder

Powerlists have been used to represent n-bit registers and to reason about arithmetic operations on them[1, 34]. This section outlines a proof of correctness for a carry-lookahead adder, using the correctness of a parallel prefix sum algorithm, i.e., the Ladner-Fischer scheme.

The "ripple-carry" or "schoolbook" algorithm for adding two n-bit registers is inherently sequential. Beginning with the least-significant bit, the algorithm progresses by adding corresponding bits. In so doing, it generates the carry bit for the next significant bit, and so on. This algorithm serves as a specification for n-bit register addition.

```
(defun adder-fa (x y cin)
  (if (zp cin)
      (cons (if (equal (zp x) (zp y)) 0 1)
            (if (or    (zp x) (zp y)) 0 1))
    (cons (if (equal (zp x) (zp y)) 1 0)
          (if (and   (zp x) (zp y)) 0 1))))

(defun adder-rc (x y cin)
  (if (powerlist-p x)
      (let ((left (adder-rc (p-untie-l x)
                            (p-untie-l y)
                            cin)))
        (let ((right (adder-rc (p-untie-r x)
                               (p-untie-r y)
                               (cdr left))))
          (cons (p-tie (car left)
                       (car right))
                (cdr right))))
    (adder-fa x y cin)))
```

The function `adder-fa` models a full-adder or 1-bit adder. It returns two values, the sum of the input bits and the generated carry bit. Similarly, `adder-rc` adds a pair of powerlists with a given input carry bit. It returns two values, the sum of the powerlists and the generated carry-out bit.

The carry-lookahead adder uses the following observation. If it were only possible to compute all the carry bits a priori, the result of adding two n-bit registers could be computed in a single parallel step (using n 1-bit adders). Moreover, given inputs $X = x_n x_{n-1} \ldots x_1$ and $Y = y_n y_{n-1} \ldots y_1$, the carry vector $C = c_n c_{n-1} \ldots c_1$ can be computed as follows. Consider c_j. If x_j and y_j are both 0, then c_j must also be 0. Moreover, if x_j and y_j are both 1, then c_j is equal to 1. In the remaining cases, c_j is equal to c_{j-1}, where c_0 is the original carry bit.

The essential remaining point is that this computation is actually a prefix sum over an associative operator with left-identity. The prefix sum runs over the domain $\{0, 1, p\}$ with intuitive meaning of *no-carry*, *carry*, and *propagate carry* respectively. In constant time, the carry bit for c_i can be estimated as either 0, 1, or p, depending on whether x_i and y_i are both 0, both 1, or otherwise. The prefix sum over this vector of the operator $\odot$ with $x \odot 0 = 0$, $x \odot 1 = 1$ and $x \odot p = x$ will generate the required carry bits. It is easily seen that the operator $\odot$ is associative, with left-identity p. This informal argument, as described for example in [15], can be made precise in ACL2.

`Local-carry-vector` computes the first pass of the carry-lookahead computation, generating values of either `0`, `1`, or `nil` for the carry bit:

```
(defun local-carry (x y)
  (if (equal (zp x) (zp y))
      (if (zp x) 0 1)
    nil))

(defun local-carry-vector (x y)
  (if (powerlist-p x)
      (p-tie (local-carry-vector (p-untie-l x)
                                 (p-untie-l y))
             (local-carry-vector (p-untie-r x)
                                 (p-untie-r y)))
    (local-carry x y)))
```

The carry-lookaheads can be computed by taking the prefix sum of this powerlist:

```
(defun prop-carry (cin local-carry)
  (if (null cin)
      (if (null local-carry)
          local-carry
        (if (zp local-carry) 0 1))
    (if local-carry
        (if (zp local-carry) 0 1)
      (if (zp cin) 0 1))))

(defun prop-carry-vector (cin lcv)
  (if (powerlist-p lcv)
      (p-tie (prop-carry-vector cin (p-untie-l lcv))
             (prop-carry-vector
              (p-last (prop-carry-vector
                       cin
                       (p-untie-l lcv)))
              (p-untie-r lcv)))
    (prop-carry cin lcv)))
```

To define the carry-lookahead adder, one more auxiliary function is needed. The function `pairwise-adder` computes the pointwise sum of *three* powerlists, two input powerlists and a powerlist of carry bits:

```
(defun pairwise-adder (x y c)
  (if (powerlist-p x)
      (p-tie (pairwise-adder (p-untie-l x)
                             (p-untie-l y)
                             (p-untie-l c))
             (pairwise-adder (p-untie-r x)
                             (p-untie-r y)
                             (p-untie-r c)))
    (car (adder-fa x y c))))
```

The carry-lookahead function can now be defined as follows:

```
(defun adder-cla-slow (x y cin)
  (let ((carry-vector
         (prop-carry-vector nil
                            (p-shift cin
                             (local-carry-vector x
                                                 y)))))
    (cons (pairwise-adder x y carry-vector)
          (prop-carry cin
                      (prop-carry (p-last carry-vector)
                                  (p-last
                                   (local-carry-vector
                                    x y)))))))
```

This function is “slow,” because it uses a linear computation for the prefix sum. This is deliberate. The immediate goal is to establish that this function performs the same computation as the ripple-carry adder, and that is more easily accomplished with the sequential version of prefix sum. Replacing the prefix sum computation by a parallel implementation can later be justified using the theorems about the correctness of prefix sum proved in sections 7.5.1 and 7.5.2.

Verifying `adder-cla-slow` is surprisingly easy. It is convenient to redefine `adder-cla-slow` in a way that makes it look more similar to the ripple-carry adder:

```
(defun adder-cla-slow-good (x y cin)
  (let ((carry-vector
         (prop-carry-vector cin
                            (local-carry-vector x y))))
    (cons (pairwise-adder x y
                          (p-shift cin carry-vector))
          (p-last carry-vector))))
```

These functions can be easily shown to be equivalent. Moreover, an instance of `adder-cla-slow-good` can be transformed into an instance of `adder-rc`; therefore, the following theorem can be established:

```
(defthm adder-cla-slow-adder-rc
      (implies (and (p-similar-p x y)
                    (bit-p cin)
                    (bit-nil-p x)
                    (bit-nil-p y))
               (equal (adder-cla-slow x y cin)
                      (adder-rc x y cin))))
```

The functions `bit-p` and `bit-nil-p` test that a powerlist is composed exclusively of zeros and ones or exclusively of zeros, ones, and nils, respectively.

Now consider a "fast" version of carry-lookahead. The only sequential step in `adder-cla-slow` is the prefix-sum computation in `prop-carry-vector`. This can be replaced with a Ladner-Fischer scheme as follows:

```
    (defun cla-star (x)
      (if (powerlist-p x)
          (p-zip (cla-star (p-unzip-r x)) (p-unzip-l x))
        nil))

    (defun cla-add (x y)
      (if (powerlist-p x)
          (p-zip (cla-add (p-unzip-l x) (p-unzip-l y))
                 (cla-add (p-unzip-r x) (p-unzip-r y)))
        (prop-carry x y)))

    (defun carry-look-ahead (x)
      (if (powerlist-p x)
          (let ((y (carry-look-ahead
                    (cla-add (p-unzip-l x) (p-unzip-r x)))))
            (p-zip (cla-add (cla-star y) (p-unzip-l x)) y))
        x))

    (defun adder-cla (x y cin)
      (let ((carry-vector (carry-look-ahead
                            (p-shift cin
                                     (local-carry-vector
                                      x y)))))
        (cons (pairwise-adder x y carry-vector)
              (prop-carry cin (prop-carry
                                (p-last carry-vector)
                                (p-last
                                 (local-carry-vector
                                  x y)))))))
```

The key observation is that `carry-look-ahead` is a faithful redefinition of `prop-carry-vector`. Thus, the following theorem can be proved simply by instantiating the generic theorems about prefix sums proved in section 7.5.2:

```
    (defthm carry-look-ahead-prop-carry-vector
      (implies (and (p-regular-p x)
                    (bit-nil-list-p x))
               (equal (carry-look-ahead x)
                      (prop-carry-vector nil x))))
```

Notice the requirement that `x` is a `p-regular-p` powerlist. This hypothesis is needed in the correctness proof of the Ladner-Fischer scheme. With this lemma, it is easy to establish that `adder-cla` computes the same value as `adder-cla-slow`:

```
    (defthm adder-cla-adder-cla-slow
      (implies (and (p-regular-p x)
                    (p-similar-p x y)
                    (bit-nil-p cin))
               (equal (adder-cla x y cin)
                      (adder-cla-slow x y cin))))
```

In turn, this justifies the correctness of the carry-lookahead algorithm:

```
(defthm adder-cla-adder-rc
  (implies (and (p-regular-p x)
                (p-similar-p x y)
                (bit-p cin)
                (bit-nil-p x)
                (bit-nil-p y))
           (equal (adder-cla x y cin)
                  (adder-rc x y cin))))
```

This formal proof follows the informal argument rather closely. That is, the hardest step in the proof is the establishment that the prefix sum computation — based on a linear algorithm similar to `p-prefix-sum-aux` — actually computes the correct carry vector. Both formal and informal proofs are made simpler by the fact that the linear prefix sum algorithm is very similar to the ripple-carry adder algorithm. This would not be the case, of course, with a more complex version of prefix sum, e.g., one based on the Ladner-Fischer scheme, or with an abstract definition of prefix sum, such as the "defining equation" described in section 7.5.3. However, once the basic correctness results are established, it is trivial to extend this result to a carry-lookahead algorithm based on a fast prefix sum: the "hard" part of the proof is a simple instance of the generic theorems proved in section 7.5.

It is encouraging that the formal proof for carry-lookahead was so easy to establish — it took no more than a single session with ACL2. This illustrates the power of the powerlist formalism in general, the specific powerlist formalization presented in section 7.2, and the usefulness of mechanically establishing correctness results with respect to “natural” specifications, as emphasized in sections 7.4 and 7.5.

Chapter 8

The Fast Fourier Transform

In chapters 3 through 6 notions from non-standard analysis were introduced into ACL2. This culminated in the development of a basic theory of trigonometry in ACL2. Chapter 7 departed from the world of real numbers into the world of data structures. Specifically, it developed the theory of powerlists, an aggregate data structure ideally suited to the expression of recursive, data-parallel algorithms.

These two themes are merged in this chapter, where powerlists are used to define and verify the correctness of the Fast Fourier Transform (FFT) algorithm. The algorithm is defined in terms of the trigonometric functions, and their properties play an important role in the correctness of the FFT.

8.1 The Fast Fourier Transform

The Fourier transform of a real or complex vector $P = (p_1, p_2, p_3, \ldots, p_n)$ is defined as $FT(P) = (\overline{P}(w_n), \overline{P}(w_n^2), \overline{P}(w_n^3), \ldots, \overline{P}(w_n^n))$, where w_n is the n^{th} principal root of 1, and $\overline{P}$ is the polynomial constructed from P by $\overline{P}(x) = \sum_{i=1}^{n} p_i \cdot x^{i-1}$.

Naively, the Fourier Transform of P can be computed in n^2 sequential steps, by evaluating $\overline{P}(x)$ at each of the n powers of w_n. This naive implementation can serve as a formal specification.

The Fourier Transform can be succintly defined in the notation of powerlists. Following [46], consider the function ep which evaluates a polynomial P pointwise at a vector V:

$$\begin{aligned}
\langle x\rangle \; ep \; v &= \langle x\rangle && (8.1)\\
(p \bowtie q) \; ep \; v &= p \; ep \; v^2 + v \cdot (q \; ep \; v^2) && (8.2)\\
p \; ep \; (u \mid v) &= (p \; ep \; u) \mid (p \; ep \; v) && (8.3)
\end{aligned}$$

Note that in the case $\langle x\rangle \; ep \; (u \mid v)$ the computation can proceed using either rule 8.3 or rule 8.1. Unfortunately, this will result in different answers. Thus, it is tacitly assumed for now that rule 8.1 is disabled while rule 8.3 is applicable. Observe, this is the only inconsistency as long as the arithmetic operators used in rule 8.2 are assumed to apply pointwise to vectors. This inconsistency will be resolved in the next section, when the definitions are formally specified in ACL2. The Fourier Transform can now be defined simply as

$$FT(p) = p \; ep \; W_n \qquad (8.4)$$

where n is the length of p and $W_n = (w_n, w_n^2, \ldots, w_n^n)$.

The Fast Fourier Transform (FFT) is an algorithm which evaluates the Fourier transform in $O(n \log n)$ sequential steps by using the special properties of the vector of powers of w_n. In particular, let $W_n = (w_n, w_n^2, \ldots, w_n^n)$, for n a power of two greater than one. Then, W_n can be written as follows:

$$\begin{aligned}
W_n &= u \mid -u\\
W_{n/2} &= u^2
\end{aligned}$$

The first property is true because w_n is the n^{th} principal root of 1, so $w_n^n = 1$ and therefore $w_n^{n/2} = -1$ (since w_n is the n^{th} *principal* root and $n/2$ is an integer less than n — recall that $n > 1$ is a power of two — $w_n^{n/2} \neq 1$). For any $n/2 < k \leq n$, $w_n^k = w_n^{n/2} \cdot w_n^{k-n/2} = -w_n^{k-n/2}$. The second property is true because the first $n/2$ values of W_n are the first $n/2$ powers of the n^{th} principal root of 1. These are precisely the (principal) square roots of the $n/2$ powers of the $(n/2)^{\text{th}}$ principal root of 1, that is, $W_{n/2}$.

Since only lists with length equal to a power of 2 are relevant in this context, it is convenient to define $\overline{W}_N = W_{2^N}$. Using this notation, the properties above can be rewritten as follows:

$$\begin{aligned} \overline{W}_n &= u \mid -u \\ \overline{W}_{n-1} &= u^2 \end{aligned}$$

These new characterizations are more amenable to induction.

The Fast Fourier Transform can be derived as follows. For singleton powerlists, it is clear that

$$\begin{aligned} FT(\langle x \rangle) &= \langle x \rangle \; ep \; \overline{W}_0 && (8.5) \\ &= \langle x \rangle && (8.6) \end{aligned}$$

Since $\overline{W}_0$ is a singleton (equal to 1), rule 8.1 of the definition of ep can be used to evaluate the term. For a powerlist of length $2^N > 1$, it follows that

$$\begin{aligned} FT(p \bowtie q) &= (p \bowtie q) \; ep \; \overline{W}_N && (8.7) \\ &= (p \bowtie q) \; ep \; (u \mid -u) && (8.8) \\ &= ((p \bowtie q) \; ep \; u) \mid ((p \bowtie q) \; ep \; -u) && (8.9) \\ &= (p \; ep \; u^2 + u \cdot (q \; ep \; u^2)) \mid (p \; ep \; u^2 - u \cdot (q \; ep \; u^2)) && (8.10) \\ &= (p \; ep \; \overline{W}_{N-1} + u \cdot (q \; ep \; \overline{W}_{N-1})) \mid \\ &\quad\; (p \; ep \; \overline{W}_{N-1} - u \cdot (q \; ep \; \overline{W}_{N-1})) && (8.11) \\ &= (FT(p) + u \cdot FT(q)) \mid (FT(p) - u \cdot FT(q)) && (8.12) \end{aligned}$$

Using these results, we can derive the Fast Fourier Transform as follows:

$$FFT(\langle x \rangle) = \langle x \rangle \tag{8.13}$$

$$FFT(p \bowtie q) = (FFT(p) + u \cdot FFT(q)) \mid (FFT(p) - u \cdot FFT(q)) \tag{8.14}$$

where the vector u contains the first $2^N/2$ elements of $\overline{W}_N$, and 2^N is the length of $p \bowtie q$. It is clear that $FFT(p \bowtie q)$ can be computed in $O(2^N)$ time given $FFT(p)$ and $FFT(q)$. Thus, it can be computed in $O(N2^N)$ (sequential) time, which is $O(n \log n)$ time, where $n = 2^N$ is the length of $p \bowtie q$. By unraveling the recursive calls, it is possible to synthesize a parallel circuit to implement the FFT. This requires $O(n \log n)$ computation nodes and $O(\log n)$ depth[15].

8.2 Verifying the Fast Fourier Transform in ACL2

In this section, Misra's hand proof of the correctness of the FFT, presented in section 8.1, is translated into ACL2. Begin by translating the function *ep* into ACL2. Recall, the definition of P *ep* V was non-deterministic: it was possible to recurse based on the polynomial P or the vector V. The ambiguity is resolved in favor of recursing on the vector V. The development is simplified if *ep* is split into the functions `eval-poly` and `eval-poly-at-point`. Their definitions are straightforward:

```
(defun eval-poly-at-point (p x)
  (if (powerlist-p p)
      (+ (eval-poly-at-point (p-unzip-l p) (* x x))
         (* x (eval-poly-at-point (p-unzip-r p)
                                   (* x x))))
    (fix p)))

(defun eval-poly (p x)
  (if (powerlist-p x)
      (p-tie (eval-poly p (p-untie-l x))
             (eval-poly p (p-untie-r x)))
    (eval-poly-at-point p x)))
```

The term `(fix p)` is used in the definition of `eval-poly-at-point` instead the simpler `p` so that the value returned by `eval-poly-at-point` is numeric even when `p` is not. This preserves the ACL2 tradition that treats all non-numeric arguments to a numeric function as zero and forces numeric functions *always* to return a numeric value.

The correctness proof uses the definition of ep not only over points, but also over vectors. In particular, the step

$$((p \bowtie q)\ ep\ u) \mid ((p \bowtie q)\ ep\ -u) \tag{8.9}$$

$$= (p\ ep\ u^2 + u \cdot (q\ ep\ u^2)) \mid (p\ ep\ u^2 - u \cdot (q\ ep\ u^2)) \tag{8.10}$$

uses polynomial versions of the arithmetic operators. ACL2 reserves the arithmetic operators for numbers only; in fact, $x \cdot 1$ is equal to zero for all non-numeric arguments x, including vectors represented as powerlists. It is therefore necessary to define a set of "arithmetic" operators over powerlists: `p-+`, `p--`, and `p-*` for pairwise addition, subtraction and multiplication, respectively. Using these operators it is possible to rewrite the polynomial evaluation over vectors with the following lemma:

```
(defthm eval-poly-lemma
  (implies (powerlist-p p)
           (equal (eval-poly p x)
                  (p-+ (eval-poly (p-unzip-l p)
                                  (p-* x x))
                       (p-* x (eval-poly (p-unzip-r p)
                                         (p-* x x)))))))
```

The theorem `eval-poly-lemma` is almost sufficient to prove (8.10). However, (8.10) also uses properties of $-u$, such as $(-u)^2 = u^2$. To prove these facts in ACL2, it is necessary to introduce unary minus on powerlists and prove some basic lemmas about its interaction with the other arithmetic operators:

```
(defthm p-*-p-unary--
  (equal (p-* (p-unary-- x) y)
         (p-unary-- (p-* x y))))


(defthm p-*-p-unary--x-p-unary--y
  (implies (p-similar-p x y)
           (equal (p-* (p-unary-- x) (p-unary-- y))
                  (p-* x y))))
```

The first theorem is simple enough, stating how a unary minus in the first argument of a product can be factored out of the product. The second theorem seems odd, because of the `p-similar-p` requirement. It is needed because the function `p-*` is defined in terms of the structure of the first argument, so it is possible that *y* will "run out" of terms before *x* does, in which case `p-*` will recurse using the `p-untie-l` and `p-untie-r` of a non-powerlist object. Neither `p-untie-l` nor `p-untie-r` guarantee a particular value when applied to a non-powerlist; in fact, it is possible to find implementations of the powerlist constraints that would invalidate the theorem without the `p-similar-p` hypothesis.

The heuristics of ACL2 exploit rewrite rules without any hypotheses, so-called simplification rules. The theorem `p-*-p-unary--x-p-unary--y` has an important specialization that can be written in this fashion, namely when `x` is equal to `y`. Since all powerlists are similar to themselves, the hypothesis can be removed in this special case:

```
(defthm p-*-p-unary--x-p-unary--x
  (equal (p-* (p-unary-- x) (p-unary-- x))
         (p-* x x)))
```

Note, a given term may rewrite using any one of the above rules. Certainly, if the last rewrite rule applies, so will the two earlier ones. It is important, therefore, that the rules be given to ACL2 *in this order*. That is, the most specific rules should be given last, since the more recent rules are tried first.

The only remaining rule deals with unary minus and addition:

```
(defthm p-+-p-unary--
  (implies (p-similar-p x y)
           (equal (p-+ x (p-unary-- y))
                  (p-- x y))))
```

As before, the similarity requirement can not be relaxed.

It is time to attempt proving the following theorem, justifying steps 8.9 and 8.10 of the proof:

```
(defthm eval-poly-u-unary---u
  (implies (powerlist-p x)
           (equal (eval-poly x (p-tie u (p-unary-- u)))
                  (p-tie (p-+ (eval-poly (p-unzip-l x)
                                         (p-* u u))
                              (p-* u
                                   (eval-poly
                                    (p-unzip-r x)
                                    (p-* u u))))
                         (p-- (eval-poly (p-unzip-l x)
                                         (p-* u u))
                              (p-* u
                                   (eval-poly
                                    (p-unzip-r x)
                                    (p-* u u))))))))
```

Unfortunately, this proof attempt fails, because the ACL2 rewriter will not use the rewrite rules about unary minus, as it can not relieve the similarity hypothesis. For example, part of the proof requires `(eval-poly x (p-* u u))` to be similar to `(p-* u (eval-poly y (p-* u u)))` where `x` and `y` are similar to each other. While true, this fact is not obvious to the ACL2 rewriter, and hence the rewrite rule taking $(x\ ep\ u^2) + (-(u \cdot y\ ep\ u^2))$ to the simpler $(x\ ep\ u^2) - (u \cdot y\ ep\ u^2)$ is not applied.

There are two solutions to this problem. The first is to add a number of rules to help ACL2 determine when two objects are similar. This approach is successful, but it results in a large number of tedious lemmas. ACL2 provides a more immediate approach — "forcing." Essentially, ACL2 allows a hypothesis to be marked as "forceable," which means that it is assumed true by the rewriter, allowing the proof to proceed. At the end of the proof, the forced hypotheses are tackled using the full power of the theorem prover, not just the rewriter. To take advantage of this, the similarity conditions are marked as forceable in the theorems `p-*-p-unary--x-p-unary--y` and `p-+-p-unary--`. At this point, ACL2 proves `eval-poly-u-unary---u` without a problem.

Why not simply force all the hypotheses, allowing the theorem prover to proceed at blinding speed, only to discard those pesky hypotheses at a later time? Because, if a rewrite rule with a false forced hypothesis is used, the proof attempt will subsequently fail — even if some *other* rewrite rule could have been applied at that time. This means that one should never force a hypothesis that is not expected to be "always" true, where by "always" is meant in the terms that the theorem prover will encounter. In the current context, only similar powerlists are encountered, so the `p-similar-p` hypothesis is a good candidate for forcing. There is a second caveat, however. In the forcing round, ACL2 does not restore *all* the facts that were available when the forced rewrite rule was used. In particular, it is possible for ACL2 to "drop" a hypothesis that will be needed when ACL2 attempts to prove the forced hypothesis.

Consider now the lists $\overline{W}_n$. The only properties of this function that are actually needed are the following:

$$\overline{W}_n \quad = \quad u \mid -u$$

$$\overline{W}_{n-1} \quad = \quad u^2$$

Since the function $\overline{W}_n$ is quite complicated, involving powers of the principal $(2^n)^{\text{th}}$ power of 1, it is advantageous to pursue the proof at an abstract level, where the only known properties are the ones stated above. As was the case in similar cases earlier, the ACL2 `encapsulate` primitive serves this purpose:

```
(encapsulate
 ((p-omega (n) t)
  (p-omega-sqrt (n) t))

 (local
  (defun p-omega (n)
    (if (zp n)
        0
      (p-tie (p-omega (1- n)) (p-omega (1- n))))))

 (local
  (defun p-omega-sqrt (n)
    (p-omega n)))

 (local
  (defthm p-unary---omega
    (equal (p-unary-- (p-omega n))
           (p-omega n))))

 (defthm numberp-omega-0
   (acl2-numberp (p-omega 0)))
 (defthm p-omega->tie-minus
   (implies (not (zp n))
            (equal (p-omega n)
                   (p-tie (p-omega-sqrt (1- n))
                          (p-unary--
                           (p-omega-sqrt (1- n)))))))

 (defthm p-omega-sqrt**2
   (equal (p-* (p-omega-sqrt n)
               (p-omega-sqrt n))
          (p-omega n)))
 )
```

The local theorem `p-unary---omega` is used to help ACL2 prove that the specified constraints are satisfiable. It is only true for the local definition of `p-omega` used, specifically the zero vector. Note also, it was necessary to define a specific function for u, since there is a *different* u for each value of n. This function is called `p-omega-sqrt`, as suggested by the last constraint.

The following theorem, justifying Misra's proof through step 8.10, can now be verified:

```
(defthm eval-poly-omega-n
  (implies (and (powerlist-p x)
                (not (zp n)))
           (let ((n1 (1- n)))
            (equal (eval-poly x (p-omega n))
                   (p-tie (p-+ (eval-poly (p-unzip-l x)
                                          (p-omega n1))
                               (p-* (p-omega-sqrt n1)
                                    (eval-poly
                                     (p-unzip-r x)
                                     (p-omega n1))))
                          (p-- (eval-poly (p-unzip-l x)
                                          (p-omega n1))
                               (p-* (p-omega-sqrt n1)
                                    (eval-poly
                                     (p-unzip-r x)
                                     (p-omega n1))))))))
 )
```

Proving this theorem requires a hint to encourage ACL2 to use the rule converting `(p-omega n)` into its $u \mid -u$ equivalent, so that `eval-poly-u-unary---u` can apply. Also needed are hints to keep ACL2 from considering lemmas relating to several functions. This is because the intermediate terms are so large they contain many function applications which ACL2 would normally consider further — unfortunately, once ACL2 starts going down that path, it loses the special structure of the theorem that allows a simple proof. It is rare that one needs to override the ACL2 heuristics quite so much, but it is vital that such overriding is possible.

At this point, it is almost possible to prove the main result. However, at this stage, the reasoning is overly general since it deals with *any* sequence of powers of roots of 1. It is not restricted to the specific sequence with as many elements as required by the Fourier Transform. To do so, it is necessary to reason about the length of a list, or better yet, about the logarithm of its length, i.e., its depth as a binary tree. This yields the following lemma:

```
(defun p-depth (x)
  (if (powerlist-p x)
      (1+ (p-depth (p-untie-l x)))
    0))
```

```
(defthm eval-poly-omega-depth
  (let* ((n (p-depth x))
         (n1 (1- n)))
   (implies (powerlist-p x)
            (equal (eval-poly x (p-omega n))
                   (p-tie (p-+ (eval-poly (p-unzip-l x)
                                          (p-omega n1))
                               (p-* (p-omega-sqrt n1)
                                    (eval-poly
                                     (p-unzip-r x)
                                     (p-omega n1))))
                          (p-- (eval-poly (p-unzip-l x)
                                          (p-omega n1))
                               (p-* (p-omega-sqrt n1)
                                    (eval-poly
                                     (p-unzip-r x)
                                     (p-omega n1))))))))
 )
```

To complete the proof, it is necessary to define the Fourier Transform in ACL2:

```
(defun p-ft-omega (x)
  (eval-poly x (p-omega (p-depth x))))
```

The main correctness result of the FFT simply extends `eval-poly-omega-depth` into `p-ft-omega`. However, this requires reasoning about the `p-depth` of p, given the `p-depth` of $p \bowtie q$. The following technical lemma can be used to simplify those terms:

```
(defthm p-depth-unzip
  (implies (and (powerlist-p x)
                (p-regular-p x))
           (and (equal (p-depth (p-unzip-l x))
                       (1- (p-depth x)))
                (equal (p-depth (p-unzip-r x))
                       (1- (p-depth x))))))
```

It may be surprising that this is the only theorem that requires the powerlist x to be regular.

Finally, it is possible to prove the main result given in the hand-proof of section 8.1:

```
(defthm ft-omega-lemma
  (implies (and (powerlist-p x)
                (p-regular-p x))
           (equal (p-ft-omega x)
                  (p-tie (p-+ (p-ft-omega (p-unzip-l x))
                              (p-* (p-omega-sqrt
                                     (1- (p-depth x)))
                                    (p-ft-omega
                                     (p-unzip-r x))))
                         (p-- (p-ft-omega (p-unzip-l x))
                              (p-* (p-omega-sqrt
                                     (1- (p-depth x)))
                                    (p-ft-omega
                                     (p-unzip-r x))))))))
```

To complete the proof, it is only necessary to introduce the ACL2 version of the Fast Fourier Transform:

```
(defun p-fft-omega (x)
  (if (powerlist-p  x)
      (p-tie (p-+ (p-fft-omega (p-unzip-l x))
                  (p-* (p-omega-sqrt (1- (p-depth x)))
                       (p-fft-omega (p-unzip-r x))))
             (p-- (p-fft-omega (p-unzip-l x))
                  (p-* (p-omega-sqrt (1- (p-depth x)))
                       (p-fft-omega (p-unzip-r x)))))
    (fix x)))
```

Note, again, the use of `fix` to ensure `p-fft-omega` always returns a numeric result. This is *required* here because of it is the way `eval-poly` was defined.

The main theorem of this section equates the Fast Fourier Transform with the Fourier Transform:

```
(defthm fft-omega->ft-omega
  (implies (p-regular-p x)
           (equal (p-fft-omega x)
                  (p-ft-omega x))))
```

It is a direct corollary of `ft-omega-lemma`.

This proof is more general than necessary. It proves the correctness of an FFT-like algorithm for any polynomial evaluation at vectors satisfying the constraints on W_N. In the next section, this proof is refined by defining instances of `p-omega` and `p-omega-sqrt` in terms of complex exponentiation. These instances correspond to the traditional definition of the Fourier Transform, and the correctness result can be established directly by functional instantiation.

8.3 Specializing the ACL2 Proof

The previous section showed how the function

$$FT(x) = x \; ep \; \overline{W}_n$$

where n is the depth of x can be quickly computed for any family of vectors $\overline{W}_n$ such that

$$\begin{aligned} \overline{W}_n &= u \mid -u \\ \overline{W}_{n-1} &= u^2 \end{aligned}$$

for some u, possibly depending on n. The actual Fourier Transform uses powers of the $(2^n)^{\text{th}}$ principal root of 1 in place of $\overline{W}_n$. In this section, it is shown that this particular vector satisfies the needed properties.

The n^{th} principal root of 1 is given by the complex number $e^{2\pi i/n}$. Using the standard definition of complex exponentiation, this gives

$$\begin{aligned} w_n &= e^{2\pi i/n} \\ &= \cos(2\pi/n) + i\sin(2\pi/n) \end{aligned}$$

The properties of the vector $\overline{W}_n = (w_{2^n}, w_{2^n}^2, \ldots, w_{2^n}^{2^n})$ can be derived from the basic properties of sine, cosine, and π, as established in chapter 6. Specifically needed are the formulas for $\sin(x+y)$ and $\cos(x+y)$. Moreover, in order to establish that $W_n = u \mid -u$, the facts that $\sin\pi = 0$ and $\cos\pi = -1$ will also be required. Recall, all of these results are proved in chapter 6.

Consider the definition of $\overline{W}_n$ from $w_{2^n} = e^{2\pi i/2^n}$. This is one place where it would be simpler programmatically to process the elements of $\overline{W}_n$ serially than in parallel, i.e., where it would be easier to use linear lists than powerlists. The reason is that it is not simple to do a "for i from 1 to n" loop in powerlists, since their recursive structure is always a split down the middle. The solution is to think of the defining properties as a recurrence relation:

$$\begin{aligned} \overline{W}_n &= \sqrt{\overline{W}_{n-1}} \,\Big|\, -\sqrt{\overline{W}_{n-1}} \\ \overline{W}_0 &= 1 \end{aligned}$$

This gives a recurrence relation for the exponents as follows:

$$E_n = E_{n-1}/2 \mid (E_{n-1}/2 + \pi)$$

$$E_0 = 2\pi$$

where the arithmetic operators are defined over pointwise powerlists. Note, E_0 is defined as 2π instead of the more natural $E_0 = 0$, so that E_1 is $(\pi, 2\pi)$, not $(0, \pi)$. From this definition, $\overline{W}_n$ can be derived as e^{iE_n}.

The needed scalar operators are easy to define:

```
(defun p-halve (x)
  (if (powerlist-p x)
      (p-tie (p-halve (p-untie-l x))
             (p-halve (p-untie-r x)))
    (/ x 2)))

(defun p-offset (x p)
  (if (powerlist-p p)
      (p-tie (p-offset x (p-untie-l p))
             (p-offset x (p-untie-r p)))
    (+ x p)))

(defun p-exponents (n)
  (if (zp n)
      (* 2 (acl2-pi))
    (let ((sqrt-expnts (p-halve (p-exponents
                                  (1- n)))))
      (p-tie sqrt-expnts
             (p-offset (acl2-pi) sqrt-expnts)))))

(defun complex-expt (x)
  (complex (acl2-cosine x) (acl2-sine x)))
```

```
(defun p-complex-expt (x)
  (if (powerlist-p x)
      (p-tie (p-complex-expt (p-untie-l x))
             (p-complex-expt (p-untie-r x)))
    (complex-expt x)))
```

It is now possible to define the functions `p-expt-omega` and `p-expt-omega-sqrt` which generate $\overline{W}_n$ and $\sqrt{\overline{W}_n}$, respectively.

```
(defun p-expt-omega (n)
  (p-complex-expt (p-exponents n)))


(defun p-expt-omega-sqrt (n)
  (p-complex-expt (p-halve (p-exponents n))))
```

It must be shown that these functions satisfy all the constraints associated with `p-omega` and `p-omega-sqrt`. Begin with the simplest constraint, namely that $\overline{W}_0$ is real:

```
(defthm numberp-expt-omega-0
  (realp (p-expt-omega 0)))
```

The next constraint is that `p-expt-omega-sqrt` is the pairwise square root of `p-expt-omega`. To show this requires the fact that $e^{x/2}e^{x/2} = e^x$. This can be proved in ACL2 with the following theorem:

```
(defthm complex-expt-/-2
  (implies (realp x)
           (equal (* (complex-expt (* 1/2 x))
                     (complex-expt (* 1/2 x)))
                  (complex-expt x))))
```

ACL2 needs hints to generate good instances of the sine of sums and cosine of sums axioms.

The next step is to generalize the lemma `complex-expt-/-2` to all powerlists:

```
(defthm p-complex-expt-halve
  (implies (p-acl2-realp-list x)
           (equal (p-* (p-complex-expt (p-halve x))
                       (p-complex-expt (p-halve x)))
                  (p-complex-expt x))))
```

The function `p-acl2-realp-list` verifies that a powerlist is composed exclusively of real numbers. This hypothesis is needed, because `complex-expt-/-2` requires `x` to be a real.

With this new rule, it is easy to prove the next constraint required, namely that `p-expt-omega-sqrt` is the square root of `p-expt-omega`:

```
(defthm p-expt-omega-sqrt**2
  (equal (p-* (p-expt-omega-sqrt n)
              (p-expt-omega-sqrt n))
         (p-expt-omega n)))
```

The final constraint deals with unary minus. The following lemma is required:

```
(defthm complex-expt-offset-pi
  (implies (p-acl2-realp-list expnts)
           (equal (p-complex-expt (p-offset (acl2-pi)
                                            expnts))
                  (p-unary-- (p-complex-expt expnts)))))
```

This follows from the facts that $e^{x+y} = e^x e^y$ and $e^{i\pi} = -1$ — Euler's beautiful identity. ACL2 can then immediately extend this result to powerlists, which is the third and last constraint on `p-omega` and `p-omega-sqrt`:

```
(defthm p-expt-omega->tie-minus
  (implies (not (zp n))
           (equal (p-expt-omega n)
                  (p-tie (p-expt-omega-sqrt (1- n))
                         (p-unary--
                          (p-expt-omega-sqrt
                           (1- n))))))))
```

What this means is that the theorems proved in section 8.2 about the Fast Fourier Transform can now be instantiated with `p-expt-omega` and `p-expt-omega-sqrt`. First, the new specific versions of the Fourier Transform and Fast Fourier Transform based on the trigonometric version of W_n need to be defined:

```
(defun p-ft-expt-omega (x)
  (eval-poly x (p-expt-omega (p-depth x))))


(defun p-fft-expt-omega (x)
  (if (powerlist-p x)
      (p-tie (p-+ (p-fft-expt-omega (p-unzip-l x))
                  (p-* (p-expt-omega-sqrt
                        (1- (p-depth x)))
                       (p-fft-expt-omega
                        (p-unzip-r x))))
             (p-- (p-fft-expt-omega (p-unzip-l x))
                  (p-* (p-expt-omega-sqrt
                        (1- (p-depth x)))
                       (p-fft-expt-omega
                        (p-unzip-r x)))))
    (fix x)))
```

ACL2 immediately verifies that the new definition of the FFT correctly computes the Fourier Transform:

```
(defthm fft-expt-omega->ft-expt-omega
  (implies (p-regular-p x)
           (equal (p-fft-expt-omega x)
                  (p-ft-expt-omega x))))
```

As with all uses of meta-theorems, a hint is required to prove this theorem by instantiating `fft-omega-correctness`. Theorem `fft-expt-omega->ft-expt-omega` justifies the use of the Fast Fourier Transform to compute the Fourier Transform. The proof required various properties of the trigonometric functions. It is relevant to note that the properties used need not necessarily have been true of approximations to the functions $\sin(x)$, $\cos(x)$, and π. This emphasizes the usefulness of extending ACL2 to incorporate the transcendental functions.

It is worth remarking on a subtle benefit derived by the use of `encapsulate` above. By using `encapsulate` it was possible to split the proof of the correctness of the FFT into two parts. The first part dealt exclusively with abstract properties that are sufficient to prove the correctness of the FFT family of algorithms. The second part that the FFT belonged to this family. The advantage of this split is that the focus of ACL2 is narrowed at each step. Because of this, the proof of each of the two halves is vastly simplified, since it takes place over a different set of functions, i.e. vector arithmetic on the first half and trigonometry on the second. This restricts ACL2's search space in looking for a proof, and this increases its chance of finding one. This usage of `encapsulate` is very useful when building large theories.

Chapter 9

Conclusion

The main goal of this thesis was the modification of ACL2 to reason about the irrational numbers using non-standard analysis. A second goal was establishing that ACL2 provides a powerful vehicle for reasoning about real-valued algorithms. The results presented in the earlier chapters demonstrate that both of these goals have been achieved. Moreover, the techniques described in this thesis should be valuable to other researchers. Some may wish to use the modified ACL2 to prove more theorems involving the real numbers, possibly in the correctness specification of a floating-point algorithm or circuit. In addition, those who decide to include non-standard analysis in their own theorem provers may find the approach presented here applicable.

A special consideration when presenting mechanized proofs is how much detail to provide. Too much detail can obscure the overall direction of the proof because it inundates the reader with a plethora of lemmas, some trivial, some deeply technical. Often, the need for these lemmas is obvious only to someone with experience in automated theorem proving. On the other hand, too little detail can mislead the reader into thinking that most of the reasoning was mechanized, giving a false impression of the state of the art in automated theorem proving. The presentation in this document tried to strike a balance between these two extremes, but it was biased towards simplifying the presentation, at the expense of ig-

Category	**Definitions**	**Theorems**
Non-Standard Analysis	19	270
The Square Root Function	11	145
The Exponential Function	62	356
The Trigonometric Functions	38	376
Powerlists	120	462
The Fast Fourier Transform	24	71
Miscellaneous Lemmas	12	85

Table 9.1: An Estimate of the Proof Effort

noring some needed lemmas. However, before embarking on a similar project, the reader is encouraged to make a fair estimate of the actual effort involved. The best way to make that estimate is to browse through the source code in the accompanying CD-ROM. A rougher estimate of the effort can be gleaned from table 9.1.

The research presented here can be continued in a number of possible directions. Already under development by users of the revised ACL2 is a library of results from real analysis. This will include such results as the mean value theorem, the chain rule, and the fundamental theorem of calculus.

Recall, the introduction of non-standard functions in ACL2 was limited to non-recursive functions. A useful area for future work will be relaxing this restriction. This will present many challenges. For example, consider the following function:

```
(defun smallest-ns-natural (n accum)
  (if (or (zp n) (standard-numberp n))
      accum
    (smallest-ns-natural (1- n) n)))
```

Were this definition accepted, `(smallest-ns-natural N N)` could be used to denote the first non-*standard* natural, given an arbitrary *i-large* integer `N`. This would lead to an immediate contradiction. However, consider the following theorem:

```
(defun limited-list-p (l)
  (if (null l)
      t
    (and (i-limited (car l))
         (limited-list-p (cdr l)))))

(defthm sum-of-limited-numbers
  (implies (and (limited-list-p l)
                (i-limited (length l)))
           (i-limited (sumlist l))))
```

This theorem can not be proved in the current ACL2, because the definition of `limited-list-p` is not accepted since it uses recursion on a non-classical formula. However, the theorem is clearly true, and it is of obvious utility. This motivates the admission of `limited-list-p` while excluding `smallest-ns-natural`. Can such a modification be made? Possibly. One observation that may be of use is the following: the measure in `smallest-ns-natural` is the paramater `n`, which takes a non-*standard* value in `(smallest-ns-natural N N)` above. If the measure in `limited-list-p` is the length of `l`, then the definition can be usefully restricted to lists of *i-limited* length.

The modifications to ACL2 described in this thesis introduced the notion of non-standard numbers into ACL2. However, this did not affect the other ACL2 data types. It may be possible to define the notion of a non-standard ACL2 object, possibly including lists and atoms.

The formalization of powerlists can also lead to some interesting new directions. As was mentioned in chapter 7, Kornerup generalized powerlists to include irregular powerlists. However, his generalization is different than the one presented here. It may be interesting to reconcile the two of them. On a different note, many algorithms have been formalized in the language of powerlists, and these formalizations can be verified using ACL2. Particularly challenging will be algorithms involving nested powerlists.

A final direction should be mentioned. Russinoff has pointed the way to the verification of floating-point approximations to some irrational functions [54, 55]. The next step should be a mechanical proof that a particular algorithm approximates a given trigonometric function. This will likely require the verification of a table of initial values for the approximation, perhaps using the verified approximation functions given in section 6.3.2. Moreover, the results will likely depend on other properties of the sine and cosine functions, such as $\sin'(x) = \cos(x)$. These results are within reach, yet they should prove handsomely rewarding.

Appendix A

A Simple Introduction to ACL2

The following is not a precise description of the ACL2 syntax or logic. Instead it is an informal description that should give the reader enough information to read the rest of this thesis. More complete descriptions can be found in [38, 37].

A term in ACL2 can be a number, atom, string, or pair. The syntax of numbers holds few surprises. Signed and unsigned integers are permitted, for example `3` and `-9`. It is worth noting that these integers can have arbitrary precision; that is, there is no requirement that the integers be less than 2^{31}. ACL2 also allows rational constants, written as *numerator/denominator*. For example, `-3/9` is an ACL2 constant equivalent to `-1/3`. Ratios also have infinite precision. Notice in particular that `-1/3` is not equal to the floating-point number `.333`. The last group of numeric constants recognized by ACL2 are the complex rationals. The complex rational $a + bi$ is written as `#C(`a b`)`. For example, `#C(0 1)` is the ACL2 constant for i.

The syntax for atoms is quite generous. Almost any non-numeric string of letters, digits, and symbols can be an atom, although parentheses should be avoided. Example atoms include `fred`, `f91`, `1+`, and `f->g`. Atoms stand for variables in ACL2. With the exception of a few atoms which always evaluate to themselves (specifically, `t` and `nil`), atoms have a value. It is an error to access an atom that has not been defined.

A string is simply a quoted sequence of characters, such as `"episode one"`.

Dotted pairs play a major role in the language. They are used to build all other data structures. Following the tradition of LISP, the symbol `nil` is very special. When it is the second element of a dotted pair, the pair is considered to be a list of the first element. For example, the pair `(1 . nil)` is the list consisting of the single element `1`, which is written as `(1)`. Longer lists are formed by taking the dotted pair of the leading element with a list containing the remaining elements. For example, the list `(1 2)` is really the pair `(1 . (2 . nil))`. Viewed this way, `nil` is not just an atom, it is also the empty list.

Lists are central in ACL2. Functions are invoked by putting together the function name and the arguments in a list. For example, pairs are constructed using the function `cons`, so the examples above can be written as `(cons 1 nil)` and `(cons 1 (cons 2 nil))`. The functions `car` and `cdr` select the first and second element of a pair respectively. So `(car (cons 1 (cons 2 nil)))` is `1` and `(cdr (cons 1 (cons 2 nil)))` is `(cons 2 nil)` or the list `(2)`.

This brings up a problem. The term `(cons 2 nil)` refers to a function invocation. Its value is whatever the function `cons` returns when given those two argument. However, the list `(2)` is a *constant* term. How is ACL2 to know when a list should be evaluated, as opposed to being treated as a constant? The answer is that constants must be explicitly quoted with a leading single quote. So the list above would be written as `'(2)`. The leading quote notifies ACL2 that the term to follow should be treated as a constant. This also works with atoms. Recall that atoms are treated as variables, so they are evaluated in that context. The value of the atom `fred` may be the list `'(2)`. But if the atom is quoted, it is not evaluated. Thus, `'fred` is the ACL2 constant with value `fred`.

Lists can also be constructed by using the function `list` which returns a list of all its arguments. For example, `(list 2 4 6)` is equal to `'(2 4 6)`. Another way of constructing lists or other arbitrary objects is to use the backquote notation. The backquote reverses the normal ACL2 quoting mechanism, so that all terms are considered constant, unless they are explicitly marked as evaluable. This device allows the structure of a term to be defined, leaving slots to be filled in. For example, consider the template `(`x y `(inv` x`) (inv` y`))`. This can be constructed using the following expression:

```
(list x y (list 'inv x) (list 'inv y))
```

However, this expression does not retain the intuitive appeal of the specified pattern. Using the backquote notation this problem is solved:

```
`(,x ,y (inv ,x) (inv ,y))
```

After encountering the backquote, ACL2 processes the following term as those it were a constant. Variables, subterms in general, are only evaluated if they are preceeded by a comma, as in `,x`. Thus, the subterm `(inv ,x)` is equal to `'(inv 3)` if the current value of `x` is `3`.

New functions can be defined in ACL2 using the function `defun`. Consider the following definition of the function 2^n:

```
(defun 2**n (n)
  (if (equal n 0)
      1
    (* 2 (2**n (1- n)))))
```

Notice the syntax of the definition. The function `defun` accepts three arguments. The first argument is the name of the function being defined — this is one of the few places where an atom occurs without being evaluated, so that it is `2**n` that is being defined, not whatever value `2**n` has. The second argument is a list containing the formal arguments for the new function. In this case, the only argument is `n`. The third argument is the body of the function. It introduces a few new functions; such as `if` with "test," "then," and "else" arguments; `equal` which tests its two arguments for equality; `*` which returns the product of all its arguments; and `1-` which returns one less than its argument.

As written, the definition of `2**n` is not accepted by ACL2. When ACL2 examines a new function, it tries to prove that the function terminates for all values of its arguments. But the definition given above does not terminate for all arguments; consider the value of `(2**n -1)`, for example. A correct definition is the following:

```
(defun 2**n (n)
  (if (not (and (integerp n) (< 0 n)))
      1
    (* 2 (2**n (1- n)))))
```

This function correctly computes 2^n of all natural numbers n, and it returns `1` for all other arguments. This follows the ACL2 tradition of treating invalid arguments as `0` for all numeric functions. The new functions `and` and `not` have the expected semantics, as does `<`. The idiom `(not (and (integerp n) (< 0 n)))` is so common, that it is predefined in ACL2 as the function `zp`. Thus, the function above would be defined as follows:

```
(defun 2**n (n)
  (if (zp n)
      1
    (* 2 (2**n (1- n)))))
```

ACL2 also allows the introduction of theorems. The function `defthm` is used to specify a new theorem to the theorem prover. Consider the proof that `2**n` is a positive integer:

```
(defthm 2**n-positive-integer
  (and (integerp (2**n n))
       (< 0 (2**n n))))
```

The first argument to `defthm` is a name for the theorem being introduced. Subsequently, it is possible to give ACL2 hints, such as "use the theorem `2**n-positive-integer` here," or "`2**n-positive-integer` is not useful here." As you might expect, the function `integerp` tests whether its argument is an integer. Actually, the theorem above is proved automatically when the function `2**n` is introduced. ACL2 tries to guess the type of all defined functions, and it usually does a remarkable job.

Another simple theorem about `2**n` is that it is even for all integer values of n at least equal to 1. In ACL2, this can be stated as follows:

```
(defthm 2**n-even
  (implies (and (integerp n)
                (<= 1 n))
           (evenp (2**n n))))
```

This introduces some new functions whose meaning should be obvious from the context. The proof of this theorem requires the arithmetic library; it can be loaded with the following command:

```
(include-book "ACL2-DIR/books/arithmetic/top")
```

The directory *ACL2-DIR* should be replaced with the location of the locally installed ACL2 tree.

A more difficult theorem is that `2**n` is a 1-to-1 function. Consider the following theorem:

```
(defthm 2**n-1-to-1
  (implies (equal (2**n x) (2**n y))
           (equal x y)))
```

ACL2 is unable to prove this theorem, because it is false. Recall, the definition of `2**n` treated all "inappropriate" arguments as effectively equal to zero. For example, `(2**n nil)` is equal to `1`. It is impossible to guarantee that `x` and `y` are equal, but it is possible to prove that they are equivalent in the context of natural numbers.

ACL2 defines a number of functions that are equivalent to the identity function in a given context. For example, the function `fix` forces its argument to be numeric; if it is, it simply returns the argument, and otherwise it returns zero. Similarly, `nfix` forces its argument to be a natural number. A theorem that can be established is the following:

```
(defthm 2**n-1-to-1
  (implies (equal (2**n x) (2**n y))
           (equal (nfix x) (nfix y))))
```

However, the proof is not automatic. The theorem can be proved by an easy inductive argument. ACL2 has a sophisticated set of heuristics that allows it to pick good induction schemes in general, but those heuristics fail in this case, because ACL2 does not notice that it must reduce `x` and `y` simultaneously.

Fortunately, ACL2 allows the user to guide it by providing hints. In this case, a hint is needed to let ACL2 know the correct induction scheme. This is done by defining a function that recurses in the desired fashion and then telling ACL2 to use the induction scheme "suggested" by that function. The function is as follows:

```
(defun induct-hint (x y)
  (if (zp x)
      y
    (+ x (induct-hint (1- x) (1- y)))))
```

Notice, the specific value computed by this function is irrelevant. The only important thing is that each recursive call decrements both `x` and `y`. The base case suggested by this function is based on `x`, but this is not significant.

The theorem can now be proved as follows:

```
(defthm 2**n-1-to-1
  (implies (equal (2**n x) (2**n y))
           (equal (nfix x) (nfix y)))
```

```
      :hints (("Goal"
               :induct (induct-hint x y))))
```

This time ACL2 is able to prove the theorem without any difficulties whatsoever.

Besides `:induct`, ACL2 provides a wide variety of `:hints`. The most common is to suggest ACL2 use a particular instance of a previously proved theorem. Another useful `:hint` is to avoid using a particular theorem that is not relevant in the current proof. This thesis deliberately avoided displaying `:hints`, although they were needed in many of the theorems that were proved. The `:hint` above gives some of the flavor of working with ACL2.

ACL2 also allows the definition of macros. A macro is simply a template that is evaluated as soon as the expression is encountered, and its value syntactically replaces the macro in its context. For example, the following macro defines the successor function:

```
    (defmacro succ (s)
      ‘(cons ,s ,s))
```

Notice the use of the backquote to specify the body of the macro. Since the value of the macro is meant to be a piece of syntax that is substituted for the macro, it is common to write it as a pattern using the backquote notation.

A powerful feature of ACL2 is its `encapsulate` primitive. This allows a proof schema to be developed and then applied to a number of different functions. In the language of formal logic, it is used to justify the introduction of derived inference rules.

To understand why `encapsulate` is useful, consider the following. The function `even-list-p` can be defined to verify that all elements of a list are even. It should be possible to prove that any subset of this list also satisfies `even-list-p`, but the proof may not be trivial. Furthermore, suppose the function `odd-list-p` has also been defined. It would be nice if the analogous theorem about subsets could be proved automatically.

That is where encapsulate comes in. The functions `even-list-p` and `odd-list-p` are very similar; the only difference is that one is checking `evenp` while the other checks `oddp`. If the functions `evenp` and `oddp` can be replaced with an arbitrary boolean function `prop`, it should be possible to prove the main theorem about `prop` and then instantiate this theorem for `evenp` and `oddp` as required. Abstract functions are called *constrained* functions in ACL2.

The constrained function `prop` can be introduced as follows:

```
(encapsulate
 ((prop (x) t))

 (local
  (defun prop (x)
    (if x t nil)))

 (defthm booleanp-prop
   (booleanp (prop x))))
```

The second line of the `encapsulate` specifies that `prop` is the constrained function being introduced, that it expects a single argument `x`, and that it returns a single value. The next term in the `encapsulate` is the definition of `prop`. Even though `prop` is being introduced as a constrained function, ACL2 requires that a possible body be defined for it. This establishes that there is at least one function that satisfies all the constrains about `prop` and is important in preserving the soundness of ACL2 (see [39] for details). However, because the definition is placed inside a `local` term, it is not exported outside of the `encapsulate`. The third term in the `encapsulate` is a theorem about `prop`. This theorem is a constraint that must be satisfied by any candidate function that is offered as an instance of `prop`. The specific constraint is that `prop` is a boolean function. Like most dialects of Lisp, ACL2 considers the atoms `t` and `nil` to represent the boolean objects "true" and "false," respectively. Unfortunately, this gives yet a third meaning to `nil` — it is the atom which evaluates to `nil`, the empty list, and the boolean false. Notice how the body of `prop` is constructed to always return `t` or `nil`. You may also notice that the function `if` does not require that its first argument be a boolean; any non-`nil` value is considered equivalent to `t` in a boolean context.

After the constrained definition of `prop` is accepted, it is easy to define the function which generalizes `prop` over lists; that is, it returns true if and only if all elements of the argument list satisfy `prop`:

```
(defun prop-list-p (l)
  (if (endp l)
      t
    (and (prop (car l))
         (prop-list-p (cdr l)))))
```

The function `endp` succeeds when its argument is not a non-empty list. The pattern above is repeated by most functions which traverse over lists.

ACL2 is able to prove that this function is preserved by subsets:

```
(defthm subsetp-prop
  (implies (and (prop-list-p l)
                (subsetp l2 l))
           (prop-list-p l2)))
```

Readers who were discouraged by the proof of `2**n-even` may be encouraged to know that ACL2 is able to prove `subsetp-prop` without the use of any hints. ACL2 appears more prepared to reason about lists than about algebra.

Next, consider the definition of the function `even-list-p`:

```
(defun even-list-p (l)
  (if (endp l)
      t
    (and (evenp (car l))
              (even-list-p (cdr l)))))
```

This function would be true of the list `'(2 8 4)`, but false of `'(2 7 4)`. Clearly, this definition follows the pattern set by `prop-list-p`. Moreover, `evenp` is a boolean, so it satisfies the constraints of `prop`. This makes it possible to use the theorem `subset-prop` to prove the equivalent theorem about `even-list-p`:

```
(defthm subsetp-even
  (implies (and (even-list-p l)
                (subsetp l2 l))
           (even-list-p l2))
  :hints (("Goal"
           :by (:functional-instance subsetp-prop
                (prop        evenp)
                (prop-list-p even-list-p)))))
```

Notice how `encapsulate` had the result of introducing a derived inference rule. Once `(booleanp (evenp x))` is established, `subsetp-even` can be deduced.

A limitation of `encapsulate` is that ACL2 never considers using one of these derived rules automatically. The user must explicitly instantiate them via a `:hint` to the prover.

Bibliography

[1] W. E. Adams. Verifying adder circuits using powerlists. Technical Report CS-TR-94-02, University of Texas at Austin, 1994.

[2] A. M. Ballantyne. The Metatheorist: Automatic proofs of theorems in analysis using non-standard techniques, part ii. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 61–75. Kluwer Academic Publishers, 1991.

[3] A.M. Ballantyne and W. W. Bledsoe. Automatic proofs of theorems in analysis using non-standard techniques. *Journal of the Association for Computing Machinery (JACM)*, 24(3):353–371, 1977.

[4] K. Batcher. Sorting networks and their applications. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 32, 1968.

[5] M. J. Beeson. Mathpert: Computer support for learning algebra, trig, and calculus. *Logic Programming and Automated Reasoning (LPAR)*, 1992.

[6] M. J. Beeson. Using nonstandard analysis to justify the correctness of computations. *International Journal of Foundations of Computer Science*, 6(3), 1995.

[7] J. L. Bell and A. B. Slomson. *Models and Ultraproducts: An Introduction*. North-Holland, Amsterdam, 1969.

[8] W. W. Bledsoe. The UT natural deduction prover. Technical Report ATP-17B, University of Texas at Austin, 1983.

[9] W. W. Bledsoe. Some automatic proofs in analysis. *Contemporary Mathematics*, 29, 1984.

[10] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, Orlando, 1979.

[11] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, San Diego, 1988.

[12] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, November 1996.

[13] R. V. Churchill and J. W. Brown. *Complex Variables and Applications*. McGraw-Hill, fourth edition, 1984.

[14] E. M. Clarke and X. Zhao. Analytica — an experiment in combining theorem proving and symbolic computation. Technical Report CMU-CS-92-147, CMU, 1992.

[15] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 32. McGraw-Hill, New York, 1990.

[16] J. Cowles. Meeting a challenge of knuth. Available for ftp download at `ftp://ftp.cs.utexas.edu/pub/boyer/cli-notes/note-286.txt`, September 1993.

[17] J. Cowles. 91 function. Private communication, January 1999.

[18] J. Cowles. Square root of 2. Private communication, February 1999.

[19] F. Diener and M. Diener, editors. *Nonstandard Analysis in Practice*. Springer, 1995.

[20] K. Doets. *Basic Model Theory*. Studies in Logic, Language, and Information. CSLI Publications, 1996.

[21] B. Dutertre. Elements of mathematical analysis in PVS. In *Proceedings of the Ninth International Conference on Theorem Proving in Higher-Order Logics (TPHOL'96)*, 1996.

[22] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.

[23] J. D. Fleuriot and L. C. Paulson. A combination of nonstandard analysis and geometry theorem proving, with application to newton's principia. In C. Kirchner and H. Kirchner, editors, *Fifteenth International Conference on Automated Deduction (CADE-15)*, number 1421 in Lecture Notes in Artificial Intelligence. Springer, July 1998.

[24] W. Fulks. *Advanced Calculus: an introduction to analysis*. John Wiley & Sons, third edition, 1978.

[25] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[26] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[27] P. R. Halmos. *Naive Set Theory*. Springer-Verlag, 1960.

[28] J. Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, University of Cambridge, 1996.

[29] E. R. Heineman and J. D. Tarwater. *Plane Trigonometry*. McGraw-Hill, Inc., seventh edition, 1993.

[30] L. W. Hines and W. W. Bledsoe. The STRIVE prover. Technical Report ATP-100, University of Texas at Austin, 1989.

[31] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[32] D. Kapur. Constructors can be partial too. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press, 1997.

[33] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. Technical Report TR-95-14, State University of New York at Albany, 1995.

[34] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using powerlists. Technical report, State University of New York at Albany, 1995.

[35] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *Computers in Mathematics with Applications*, 1994.

[36] M. Kaufmann. The sine of $1/2$. Private communication, 1996.

[37] M. Kaufmann and J S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual*.

[38] M. Kaufmann and J S. Moore. A precise description of the ACL2 logic. Available on the world-wide web at http://www.cs.utexas.edu/users/moore/publications/km97a.ps.Z.

[39] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. in preparation.

[40] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[41] H. J. Keisler. *Elementary Calculus*. Prindle, Weber and Schmidt, Boston, 1976.

[42] H. J. Keisler. *Foundations of Infinitesimal Calculus*. Prindle, Weber and Schmidt, Boston, 1976.

[43] H. J. Keisler. Fundamentals of model theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter A.2. North Holland, 1977.

[44] J. Kornerup. Parlists: A generalization of powerlists. In *Proceedings of Euro-Par'97*, 1997.

[45] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery (JACM)*, 27:831–838, 1980.

[46] J. Misra. Powerlists: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1737–1767, November 1994.

[47] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5$_K$86 floating-point division program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998.

[48] E. Nelson. On-line books: Internal set theory. Available on the world-wide web at `http://www.math.princeton.edu/~nelson/books.html`.

[49] E. Nelson. Internal set theory. *Bulletin of the American Mathematical Society*, 83:1165–1198, 1977.

[50] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[51] A. Robert. *Non-Standard Analysis*. John Wiley, 1988.

[52] A. Robinson. *Non-Standard Analysis*. Princeton University Press, 1996.

[53] P. Rudnicki. An overview of the MIZAR project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[54] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating-point square root microcode. *Formal Methods in System Design*, To appear.

[55] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7(tm) processor. *London Math. Soc. J. of Comp. Math*, To appear.

[56] J. R. Shoenfield. Axioms of set theory. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter B.1. North Holland, 1977.

[57] A. Trybulec. The Mizar-QC/6000 logic information language. *Bulletin of the Association for Literary and Linguistic Computing (LLAC)*, 6(2), 1978.

Index

Vita

Ruben Antonio Gamboa, son of Andres and Beatriz Gamboa, was born in Bucaramanga, Colombia on October 15, 1967. In 1981 he graduated from London Central High School, an American school for dependants of military personnel stationed in England. He began college in the extension universities available to military personnel and dependants at R.A.F. Chicksands, namely Troy State University and City Colleges of Chicago, which granted him an Associate of Applied Sciences in Data Processing in 1982. The following year he entered Angelo State University (ASU), receiving a Bachelor of Science in Computer Science in 1984. Then he enrolled in Texas A&M University, receiving a Master of Computer Science degree in 1986. He was the recipient of the Forsythe Graduate Fellowship in the 1985-86 academic year. In 1988 he joined the Microelectronics and Computer Technology Corporation (MCC), where he worked in the deductive database group, helping to implement $\mathcal{LDL}$, a logical query language. In 1990 he joined Logical Information Machines, Inc. (LIM), a financial database company he co-founded. In September 1992, he entered the Ph.D. program of the Computer Sciences department of the University of Texas at Austin. Ruben Gamboa has authored various publications, including the following:

- “Mechanically Verifying the Correctness of the Fast Fourier Transform in ACL2,” Proceedings of the IPPS/SPDP’98 Workshops, 1998.
- “Defthms About Zip and Tie: Reasoning About Powerlists in ACL2,” University of Texas Computer Sciences Technical Report No. TR97-02.

- “Square Roots in ACL2: A Study in Sonata Form.” University of Texas Computer Sciences Technical Report No. TR96-34.

- “Towards an Open Architecture for LDL,” with Danette Chimenti and Ravi Krishnamurthy, Proceedings of the 15th Conference on Very Large Databases (VLDB), 1989.

- “Abstract Machine for LDL,” with Danette Chimenti and Ravi Krishnamurthy, Proceedings of the 2nd Conference on Extending Database Technology (EDBT), 1990.

- “The LDL System Prototype,” with D. Chimenti et al., IEEE Transactions on Data and Knowledge Engineering, March, 1990.

- “Using Modules and Externals in LDL,” with D. Chimenti and R. Krishnamurthy, MCC Technical Report No. ACA-ST-036-89.

- “The SALAD Cookbook: A User’s/Programmer’s Guide,” with D. Chimenti, MCC Technical Report No. ACA-ST-346-89.

- “Difference Bounds on the Approximation of NP Optimization Problems,” with Donald Friesen, Texas A&M Computer Science Technical Report, 1986.

Permanent Address: 108 Stonehedge Blvd.
Georgetown, Texas 78626

This dissertation was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1]LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.