Mechanical Verification of a Square Root Algorithm using Taylor's Theorem

Jun Sawada[1] and Ruben Gamboa[2]

[1] IBM Austin Research Laboratory
Austin, TX 78759
sawada@austin.ibm.com

[2] Department of Computer Science
University of Wyoming
Laramie, WY 82071
ruben@cs.uwyo.edu

Abstract. The IBM Power4$^{\mathrm{TM}}$ processor uses series approximation to calculate square root. We formally verified the correctness of this algorithm using the ACL2(r) theorem prover. The proof requires the analysis of the approximation error on a Chebyshev series. This is done by proving Taylor's theorem, and then analyzing the Chebyshev series using Taylor series. Taylor's theorem is proved by way of non-standard analysis, as implemented in ACL2(r). Since Taylor series of a given order have less accuracy than Chebyshev series in general, we used hundreds of Taylor series generated by ACL2(r) to evaluate the error of a Chebyshev series.

1 Introduction

We discuss the formal verification of a floating-point square root algorithm used in the IBM Power4$^{\mathrm{TM}}$ processor. The same algorithm was first presented and proven, not formally, by Agarwal et al in [AGS99]. Obviously, the drawback of a hand-proof is that it does not provide an absolute assurance of correctness. Formal verification gives a higher-level of confidence by mechanically checking every detail of the algorithm.

The formal verification of square root algorithms used in industrial processors has been studied in the past. Russinoff used the ACL2 theorem prover [KM96] to verify the microcode of K5 Microprocessor [Rus99]. Later he also verified the square root algorithm in the K7 microprocessor and proved that the underlying RTL model actually implements such an algorithm [Rus98]. Aagaard et al. [AJK$^+$00] verified the square root algorithm used in an Intel processor with the Forte system [OZGS99] that combines symbolic trajectory evaluation and theorem proving.

The square root algorithms mentioned above use the Newton-Raphson algorithm or one of its variants [PH96]. This algorithm starts with an initial estimate and iteratively calculates a better estimate from the previous one. The formula to obtain the new estimate is relatively simple. It takes a few iterations to obtain an estimate that is accurate enough. This estimate is rounded to the final answer according to a specified rounding mode. In Newton-Raphson's algorithm, many instructions are dependent on earlier instructions. The algorithm may require more execution cycles on a processor with many pipeline stages and high latency.

The IBM Power4 processor and its predecessor Power3$^{\mathrm{TM}}$ processor use a different method of function iteration to calculate square root. From the initial approximation, it obtains a better approximation using a Chebyshev polynomial. Polynomial calculation needs more instructions than a single iteration of the Newton-Raphson algorithm. However, only a single iteration is sufficient to obtain the necessary precision. Since instructions in the polynomial calculation are less dependent on earlier instructions than those in the Newton-Raphson algorithm, more instructions can be executed in parallel with a pipelined floating-point unit.

The biggest challenge for the formal verification of this algorithm, and also the one that distinguishes our verification work from others, is the error size analysis on the polynomial approximating the square root function. We need to verify that a Chebyshev polynomial used in our algorithm has errors small enough to guarantee that the final estimate is rounded to the correct answer.

The verification was carried out with the ACL2(r) theorem prover [Gam99]. ACL2(r) is an extension of the ACL2 theorem prover that performs reasoning on real numbers using non-standard analysis [Rob59].

The verification of the square root algorithm took place in two steps:

1. Prove Taylor's theorem.
2. Prove the square root algorithm using Taylor's theorem.

Taylor's theorem represents a differentiable function as the sum of an approximating polynomial and a remainder term. Our approach is to use a Taylor polynomial in the measurement of the error size of a Chebyshev polynomial. One problem is that a Chebyshev polynomial in general gives a better approximation than a Taylor polynomial, thus it is not straightforward to verify the error size of a Chebyshev polynomial with a Taylor polynomial. In order to solve this problem, we analyzed the Chebyshev polynomial with hundreds of Taylor polynomials generated by the ACL2(r) prover.

This paper is organized as follows. In Section 2, we introduce the non-standard analysis features of ACL2(r) that form a basis for our proof. In Section 3, we describe the proof of Taylor's theorem in ACL2(r). In Section 4, we describe the square root algorithm used in the Power4 processor and its verification. This section assumes that certain proof obligations are met. These proof obligations are proved in Section 5, using Taylor's theorem. Finally, we conclude in Section 6. If the reader is not interested in the non-standard analysis, it is safe to skip Section 2 and the proof of Taylor's theorem in Section 3.

2 ACL2(r) : Real Analysis using Non-Standard Analysis

Non-standard analysis, introduced by Robinson in the 1960s using model-theoretic techniques and later given an axiomatization by Nelson [Rob59,Nel77], provides a rigorous foundation for the informal reasoning about infinitesimal quantities used by Leibniz when he co-invented the calculus and still used today by engineers and scientists when applying calculus. There are several good introductions to non-standard analysis, for example [Rob88,Nel]. In this section, we give the reader enough of the background to follow subsequent discussions.

Non-standard analysis changes our intuitive understanding of the real number line in a number of ways. Some real numbers, including all numbers that can be uniquely defined in first-order logic, such as 0, 1, e, and π, are called *standard*. There are real numbers that are larger in magnitude than all the *standard* reals; these numbers are called *i-large*. Numbers that are not *i-large* are called *i-limited*. Moreover, there are reals smaller in magnitude than any positive *standard* real; these numbers are called *i-small*. It follows that 0 is the only number that is both *standard* and *i-small*. Notice that if N is an *i-large* number, $1/N$ must be *i-small*. Two numbers are called *i-close* if their difference is *i-small*. It turns out that every *i-limited* number is *i-close* to a *standard* number. 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 called the *standard-part* of x.

The terms *i-large*, *i-small*, and *i-close* give mathematical precision to the informal ideas "infinitely large," "infinitely small," and "infinitely close." These informal notions are ubiquitous in analysis, where they are often replaced by formal statements about series or by $\epsilon - \delta$ arguments. A feature of non-standard analysis is that it restores the intuitive aspects of analytical proofs.

For example, the sequence $\{a_n\}$ is said to converge to the limit A if and only if a_N is *i-close* to A for all *i-large* N. This agrees with the intuitive notion of convergence: "a_n gets close to A when n is large enough." Similarly, consider the notion of derivatives: the function f has derivative $f'(x)$ at a *standard* point x if and only if $(f(x) - f(y))/(x - y)$ is *i-close* to $f'(x)$ whenever x is *i-close* to y. Again, the formal definition follows closely the intuitive idea of derivative as the slope of the chord with endpoints "close enough."

The *non-standard definition principle* allows the definition of functions by specifying their behavior only at standard points. For example, consider the function $\sqrt{x}$. One way to define it is to provide an approximation scheme $f_n(x)$ so that $\{f_n(x)\}$ converges to the square root of x. For *standard* points x, the function $\sqrt{x}$ can be defined by $\sqrt{x} = (f_N(x))^*$, where N is an *i-large* integer. Using the non-standard definitional principle, this function defined over *standard* numbers is extended to the function $\sqrt{x}$ defined over the entire real number line.

The *transfer principle* allows us to prove a first-order statement $P(x)$ about the reals by proving it only when x is *standard*. This principle can be applied only when the statement $P(x)$ is a first-order statement in the language of real analysis, *without* using the new functions of non-standard analysis, such as *standard*, *i-large*, *i-small*, *i-close*, or *standard-part*. Consider the example given above for $\sqrt{x}$. The function $f_N(x)$ is an approximation to the square root of x, so it is reasonable that $f_N(x) \cdot f_N(x)$ is *i-close* to x when x is *i-limited* and N is *i-large*. In fact, such a theorem can proved in ACL2(r) using induction on N. What this means is that for *standard* x, $\sqrt{x} \cdot \sqrt{x} = (f_N(x))^* \cdot (f_N(x))^* = (f_N(x) \cdot f_N(x))^* = x$. The transfer principle then establishes $\sqrt{x} \cdot \sqrt{x} = x$ for *all* x.

Using the non-standard definition and transfer principles in tandem is a powerful and ubiquitous technique in ACL2(r). To illustrate it, we present a proof of the maximum theorem in ACL2(r). The theorem states that if f is a continuous function on the closed interval $[a, b]$, there is a point $x \in [a, b]$ so that $f(x) \geq f(y)$ for all $y \in [a, b]$. This theorem is used in the proof of Rolle's Lemma, which in turn is the key to proving Taylor's Theorem.

We begin by introducing an arbitrary continuous function f in a domain. This can be done in ACL2 using the `encapsulate` event:

```
(encapsulate
 ((f (x) t)
  (domain-p (x) t))

 (local (defun f (x) x))
 (local (defun domain-p (x) (realp x)))

 (defthm domain-real
   (implies (domain-p x)
            (realp x)))

 (defthm domain-is-interval
   (implies (and (domain-p l) (domain-p h)
                 (realp x) (<= l x) (<= x h))
            (domain-p x)))

 (defthm f-standard
   (implies (and (domain-p x)
                 (standard-numberp x))
            (standard-numberp (f x))))

 (defthm f-real
   (implies (domain-p x)
            (realp (f x))))

 (defthm f-continuous
   (implies (and (domain-p x) (standard-numberp x)
                 (domain-p y) (i-close x y))
            (i-close (f x) (f y))))
 )
```

ACL2's `encapsulate` mechanism allows the introduction of constrained functions. This event introduces the functions `f` and `domain-p`. The first argument of the `encapsulate` establishes that they are unary functions. The definitions of `f` and `domain-p` are marked `local`, and they are not available outside of the encapsulate. Their only purpose is to demonstrate that there are *some* functions which satisfy the given constraints. The constraints are specified by the `defthm` events inside of the `encapsulate`. These constraints serve to make `domain-p` an arbitrary function that accepts intervals of real numbers, and `f` an arbitrary *standard*, real, and continuous function.

To show that `f` achieves its maximum on a closed interval, we split the interval $[a, b]$ into n subintervals of size $\epsilon = \frac{b-a}{n}$. It is easy to define a function that finds the point $a + k \cdot \epsilon$ where `f` achieves the maximum of the points in the ϵ-grid of points $a + i \cdot \epsilon$. That much of the reasoning uses only the traditional concepts in ACL2, notably recursion and induction. Non-standard analysis takes center stage when we consider what happens when n is *i-large*, hence when ϵ is *i-small.* Consider the point $x_{max} = (a + k \cdot \epsilon)^*$. This is a *standard* point, since it is the *standard-part* of an *i-limited* point. Let y be a *standard* point in $[a, b]$. Since $y \in [a, b]$, there must be an i so that $y \in [a+(i-1)\cdot\epsilon, a+i\cdot\epsilon]$. Since ϵ is *i-small*, it follows that y is *i-close* to $a+i\cdot\epsilon$, so $f(y) = (f(a+i\cdot\epsilon))^*$ from the continuity of f. But from the definition of x_{max} it follows that $f(y) = (f(a + i \cdot \epsilon))^* \leq (f(a + k \cdot \epsilon))^* = f(x_{max})$. This suffices to show that `f` achieves its maximum over *standard* points $y \in [a, b]$ at x_{max}. Using the transfer principle, we have that `f` achieves its maximum over $[a, b]$ at x_{max}.

In ACL2(r), we prove the result by defining the function `find-max-f-x-n` which finds the point $a + k \cdot \epsilon$ where `f` achieves its maximum over the ϵ-grid:

```
(defun find-max-f-x-n (a max-x i n eps)
  (if (and (integerp i) (integerp n) (<= i n)
           (realp a) (realp eps) (< 0 eps))
      (if (> (f (+ a (* i eps))) (f max-x))
          (find-max-f-x-n a (+ a (* i eps)) (1+ i) n eps)
        (find-max-f-x-n a max-x (1+ i) n eps))
    max-x))
```

It is a simple matter to prove that this function really does find the maximum in the grid. Moreover, under natural conditions, the point it finds is in the range $[a, b]$. The next step is to use the non-standard definitional principle to introduce the function that picks the actual maximum over $[a, b]$. In ACL2(r) this is done by using the event `defun-std` in place of `defun` to define the function:

```
(defun-std find-max-f-x (a b)
  (if (and (realp a) (realp b) (< a b))
      (standard-part
       (find-max-f-x-n a a 0 (i-large-integer)
                       (/ (- b a) (i-large-integer))))
    0))
```

The function `i-large-integer` in ACL2(r) is used to denote a positive *i-large* integer. Since `find-max-f-x-n` is in the range $[a, b]$, we can use the transfer principle to show that `find-max-f-x` is also in $[a, b]$.

A simple inductive argument can establish that `find-max-f-x-n` finds the point in the ϵ-grid where `f` achieves its maximum. Taking the *standard-part* of both sides of this inequality shows that the point selected by `find-max-f-x` is also a maximum over the points in the grid:

```
(defthm find-max-f-is-maximum-of-grid
  (implies (and (realp a) (standard-numberp a)
                (realp b) (standard-numberp b)
                (< a b) (domain-p a) (domain-p b)
                (integerp i) (<= 0 i) (<= i (i-large-integer)))
           (<= (standard-part (f (+ a (* i (/ (- b a)
                                              (i-large-integer))))))
               (f (find-max-f-x a b)))))
```

Once we have this result, we can see that `find-max-f-x` finds the point where `f` achieves its maximum on the *standard* points of $[a, b]$. It is simply necessary to observe that any *standard* point $x \in [a, b]$ must be *i-close* to some point $a + i \cdot \epsilon$ in the ϵ-grid of $[a, b]$.

```
(defthm find-max-f-is-maximum-of-standard
  (implies (and (realp a) (standard-numberp a)
                (realp b) (standard-numberp b)
                (realp x) (standard-numberp x)
                (domain-p a) (domain-p b)
                (<= a x) (<= x b) (< a b))
           (<= (f x) (f (find-max-f-x a b)))))
```

To complete the proof, it is only necessary to invoke the transfer principle on the theorem above. In ACL2(r) this is done by using the event `defthm-std` when the theorem is proved:

```
(defthm-std find-max-f-is-maximum
  (implies (and (realp a) (domain-p a)
                (realp b) (domain-p b)
                (realp x) (<= a x) (<= x b) (< a b))
           (<= (f x) (f (find-max-f-x a b)))))
```

The techniques described above, combining `defun-std` and `defthm-std`, have been used extensively in ACL2(r), resulting in proofs as varied as the correctness of the Fast Fourier Transform and the fundamental theorem of calculus [Gam02,Kau00]. A more complete account of ACL2(r) can be found in [Gam99,GK01].

3 Proof of Taylor's Theorem

Given a function f with n continuous derivatives on an interval $[a, b]$ Taylor's formula with remainder provides a means for estimating $f(x)$ for an arbitrary $x \in [a, b]$ from the values of f and its derivatives at a. Formally, Taylor's Theorem can be stated as follows:

Taylor's Theorem *If $f^{(n)}(x)$ is continuous in $[x_0, x_0+\delta]$ and $f^{(n+1)}(x)$ exists in $[x_0, x_0+\delta]$, then there exists $\xi \in (x_0, x_0+\delta)$ such that*

$$\begin{aligned} f(x_0+\delta) &= f(x_0) + f'(x_0)\delta + \frac{f''(x_0)}{2!}\delta^2 + \cdots + \frac{f^{(n-1)}(x_0)}{(n-1)!}\delta^{(n-1)} + \frac{f^{(n)}(\xi)}{n!}\delta^n \\ &= \sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}\delta^i + \frac{f^{(n)}(\xi)}{n!}\delta^n \end{aligned}$$

The term $\sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}\delta^i$ is called the Taylor polynomial of f of degree $n-1$, and $\frac{f^{(n)}(\xi)}{n!}\delta^n$ is called the corresponding Taylor remainder. The Taylor polynomial is often used to approximate $f(x)$; the approximation error can be estimated using the Taylor remainder.

The proof of this theorem, presented in [Ful78] among others, is similar to the proof of the mean value theorem. First, a special function F is constructed, and then Rolle's Lemma is applied to F to find a ξ for which $F'(\xi) = 0$. Taylor's formula follows from solving $F'(\xi) = 0$ for $f(x_0+\delta)$. From an automated theorem perspective, the main challenge is to find F', which can be done by repeated application of the theorems concerning the derivatives of composition of functions, and then to solve the equation accordingly.

To formalize Taylor's Theorem in ACL2(r), we use `encapsulate` to introduce the constrained function `f`. The derivatives of `f` are constrained in the function `f-deriv`, which takes arguments corresponding to i and x in $f^{(i)}(x)$. The constraints on `f-deriv` follow the non-standard definition of derivatives as discussed in section 2. Also constrained is the bound `tay-n` on the order of the Taylor approximation. Note in particular that for i greater than `tay-n`, we do not assume that `f-deriv` continues to find derivatives, since these higher-order derivatives are not even assumed to exist as part of Taylor's theorem. In the interest of brevity, we present only the constraints of `f-deriv`:

```
(encapsulate
 ((f (x) t)
  (f-deriv (i x) t)
  (domain-p (x) t)
  (tay-n () t))
 ...
 (defthm f-deriv-0
   (implies (domain-p x)
            (equal (f-deriv 0 x)
                   (f x))))
 (defthm f-deriv-chain
   (implies (and (standard-numberp x)
                 (domain-p x) (domain-p y)
                 (i-close x y) (not (= x y))
                 (integerp i) (<= 0 i) (<= i (tay-n)))
            (i-close (/ (- (f-deriv i x) (f-deriv i y))
                        (- x y))
                     (f-deriv (1+ i) x))))
 )
```

The Taylor series can be defined in terms of the constrained functions `f` and `f-deriv`:

```
(defun tay-term (i x0 delta)
  (* (fn-deriv i x0)
     (expt delta i)
     (/ (factorial i))))

(defun tay-sum (i n x0 delta)
  (if (and (integerp i) (integerp n) (<= i n))
      (+ (tay-term i n x0 delta)
         (tay-sum (1+ i) n x0 delta))
    0))
```

We found it useful to split this into two functions, because it makes the statement of Taylor's theorem more succinct. In particular, the error term uses `tay-term`. The formal statement of Taylor's theorem in ACL2(r) is as follows:

```
(defthm taylor-series
  (implies (and (domain-p x0) (domain-p (+ x0 delta))
                (integerp n) (< 1 n) (<= n (tay-n)))
           (and (domain-p (tay-xi n x0 (+ x0 delta)))
                (<= (min x0 (+ x0 delta)) (tay-xi n a (+ x0 delta)))
                (<= (tay-xi n a (+ x0 delta)) (max x0 (+ x0 delta)))
                (equal (f (+ x0 delta))
                       (+ (tay-sum 0 (1- n) x delta)
                          (tay-term n (tay-xi n a x) delta))))))
```

where the function `tay-xi` finds the point ξ used in the error term in Taylor's theorem. It is defined in terms of `find-max-f-x`, and its properties follow from Rolle's Lemma. Details of this proof can be found in [GM02].

4 Verification of a Square Root Algorithm

4.1 Description of the Algorithm

Following is the description of the square root algorithm used in the Power4 processor. First we introduce a few functions. We define $\text{expo}(x)$ as the function that returns the exponent of x. Function $\text{ulp}(x, n)$ returns the unit of least position; that is defined as:

$$\mathrm{ulp}(x,n) = 2^{\mathrm{expo}(x)-n+1}.$$

This corresponds to the magnitude of the least significant bit of an n-bit precision floating point number x. Function $\mathrm{near}(x,n)$ rounds a rational number x to a floating-point number with n-bit precision using IEEE nearest rounding mode. Function $\mathrm{rnd}(x,m,n)$ rounds x to an n-bit precision floating-point number using rounding mode m, where m can be *near*, *trunc*, *inf* or *minf* corresponding to the four IEEE rounding modes [Ins]. The predicate $\mathrm{exactp}(x,n)$ is true if x can be exactly represented as an n-bit IEEE floating-point number. Thus $\mathrm{exactp}(\mathrm{near}(x,n),n)$ and $\mathrm{exactp}(\mathrm{rnd}(x,m,n),n)$ are true. These functions, except $\mathrm{ulp}(x,n)$, are defined in the library distributed with the ACL2 system.

We assume that b is an IEEE double precision floating-point number satisfying $1/2 \leq b < 2$ and discuss the algorithm to calculate the square root of b. This restricted algorithm can be easily extended to the full range of IEEE floating-point numbers, because the exponent of the square root is calculated by dividing the exponent of b by 2.

In this algorithm, an on-chip table provides the initial 12-bit estimate of the square root of b, which is named q_0. Another on-chip table gives the 53-bit rounded value of $1/q_0^2$, which is denoted y_0. In order to reduce the size of the on-chip tables, the tables entries exist only for $1/2 \leq b < 1$. When $1 \leq b < 2$, the algorithm looks up the table entries for $b/2$ and adjusts them by dividing y_0 by 2 and multiplying q_0 by `*root2*`, which is a precomputed 53-bit representation of $\sqrt{2}$. The adjusted values are called y_{0s} and q_{0s}, respectively.

Let $e = 1 - y_{0s}b$. The difference between the squares of q_{0s} and $\sqrt{b}$ can be calculated as:

$$q_{0s}^2 - b \simeq q_{0s}^2(1 - by_{0s}) = q_{0s}^2 e.$$

By solving this equation with respect to $\sqrt{b}$, we get

$$\sqrt{b} \simeq q_{0s}\sqrt{1-e} \simeq q_{0s}(1 + c_0 e + c_1 e^2 + c_2 e^3 + c_3 e^4 + c_4 e^5 + c_5 e^6)$$

where $1 + c_0 e + \cdots + c_5 e^6$ is a Chebyshev polynomial approximating $\sqrt{1-e}$. Further manipulation of the right-hand side leads to the following:

$$\begin{aligned}
\sqrt{b} &\simeq q_{0s} + q_{0s}e(c_0 + \cdots c_5 e^5) \\
&= q_{0s} + q_{0s}(1 - y_{0s}b)(c_0 + \cdots c_5 e^5) \\
&\simeq q_{0s} + q_{0s}(q_{0s}^2 y_{0s} - y_{0s}b)(c_0 + \cdots c_5 e^5) \\
&= q_{0s} + q_{0s}y_{0s}(q_{0s}^2 - b)(c_0 + \cdots c_5 e^5).
\end{aligned}$$

The algorithm in Table 1 uses this equation to calculate a better approximation q_1 of $\sqrt{b}$. The procedure to obtain y_{0s} from y_0 is not explicit in Table 1 as it is simply an exponent adjustment. Chebyshev coefficients c_0 through c_5 are 53-bit precision floating-point numbers obtained from an on-chip table. In fact, we use two sets of Chebyshev coefficients, one of which is intended to be used for

Table 1. Double-precision floating-point square root algorithms used in Power4.

Algorithm to calculate $\sqrt{b}$
Look up y_{0s}
Look up q_0
$e := \text{near}(1 - y_{0s} \times b, 53)$
$q_{0s} := \text{near}(\texttt{*root2*} \times q_0, 53)$ if $1 \le b < 2$
$\quad := q_0$ if $1/2 \le b < 1$
$t_3 := \text{near}(c_4 + c_5 \times e, 53)$
$t_4 := \text{near}(c_2 + c_3 \times e, 53)$
$esq := \text{near}(e \times e, 53)$
$t_5 := \text{near}(c_0 + c_1 \times e, 53)$
$e_1 := \text{near}(q_{0s} \times q_{0s} - b, 53)$
$t_1 := \text{near}(y_{0s} \times q_{0s}, 53)$
$t_6 := \text{near}(t_4 + esq \times t_3, 53)$
$q_{0e} := \text{near}(t_1 \times e_1, 53)$
$t_7 := \text{near}(t_5 + esq \times t_6, 53)$
$q_1 := q_{0s} + q_{0e} \times t_7$
sqrt-round(q_1,b,mode)

$0 \le e \le 2^{-6}$ and the other for $-2^{-6} \le e < 0$. Let c_{0p}, c_{1p}, c_{2p}, c_{3p}, c_{4p} and c_{5p} be the set of coefficients intended for the positive case, and c_{0n}, c_{1n}, c_{2n}, c_{3n}, c_{4n} and c_{5n} for the negative case. In our algorithm, the 6th fraction bit of b, instead of the polarity of e, determines which set of coefficients will be used. This can be justified by the fact that e tends to be positive when the 6th fraction bit of b is 0, and negative otherwise. However, this relation between the 6th fraction bit of b and the polarity of e is not always true, and we must verify that this heuristic in selecting Chebyshev coefficients does not cause too much error. We will come back to this in Section 5.

The function sqrt-round(q_1, b, m) at the end of the algorithm represents the hardwired rounding mechanism for the square root algorithm. It rounds the final estimate q_1 to the correct answer $\text{rnd}(\sqrt{b}, m, 53)$, if the error of the final estimate q_1 is less than a quarter of the ulp, i.e.,

$$|q_1 - \sqrt{b}| < \text{ulp}(q_1, 53)/4.$$

Our verification objective is to prove that the final estimate q_1 falls into this required error margin.

4.2 Verification of the Algorithm

The proof of the square root algorithm has been mechanically checked by the ACL2(r) prover. The proof outline is basically the same as that provided by Agarwal et al[AGS99]. We describe the proof from the perspective of mechanization, and explain what must be proven with Taylor's theorem.

First we define the intermediate values $q_{0s}, y_{0s}, e, t_3, t_4, esq, t_5, e_1, t_1, t_6, q_{0e}, t_7$ and q_1 that appear in Table 1 as ACL2(r) functions. These are, in fact, functions of b, but we omit the argument b for simplicity in the paper. The same is true for the Chebyshev coefficients $c0$, $c1$, $c2$, $c3$, $c4$ and $c5$, which are selected from two sets of coefficients depending on b. For each of the intermediate values, we define $\tilde{e}$, $\tilde{t_3}$, $\tilde{t_4}$, $\tilde{esq}$, $\tilde{t_5}$, $\tilde{e_1}$, $\tilde{t_1}$, $\tilde{t_6}$, $\tilde{q_{0e}}$ and $\tilde{t_7}$ as the infinitely precise value before rounding. We define r_e, r_{t_3}, r_{t_4}, r_{esq}, r_{t_5}, r_{e_1}, r_{t_1}, r_{t_6}, $r_{q_{0e}}$ and r_{t_7} as the values added to the infinitely precise values by rounding. Formally speaking:

$$\begin{array}{llllll}
\tilde{e} & = 1 - y_{0s} \times b & e & = \text{near}(\tilde{e}, 53) & r_e & = e - \tilde{e} \\
\tilde{t_3} & = c_4 + c_5 \times e & t_3 & = \text{near}(\tilde{t_3}, 53) & r_{t_3} & = t_3 - \tilde{t_3} \\
\tilde{t_4} & = c_2 + c_3 \times e & t_4 & = \text{near}(\tilde{t_4}, 53) & r_{t_4} & = t_4 - \tilde{t_4} \\
\tilde{esq} & = e \times e & esq & = \text{near}(\tilde{esq}, 53) & r_{esq} & = esq - \tilde{esq} \\
\tilde{t_5} & = c_0 + c_1 \times e & t_5 & = \text{near}(\tilde{t_5}, 53) & r_{t_5} & = t_5 - \tilde{t_5} \\
\tilde{e_1} & = q_{0s} \times q_{0s} - b & e_1 & = \text{near}(\tilde{e_1}, 53) & r_{e_1} & = e_1 - \tilde{e_1} \\
\tilde{t_1} & = y_{0s} \times q_{0s} & t_1 & = \text{near}(\tilde{t_1}, 53) & r_{t_1} & = t_1 - \tilde{t_1} \\
\tilde{t_6} & = t_4 + esq \times t_3 & t_6 & = \text{near}(\tilde{t_6}, 53) & r_{t_6} & = t_6 - \tilde{t_6} \\
\tilde{q_{0e}} & = t_1 \times e_1 & q_{0e} & = \text{near}(\tilde{q_{0e}}, 53) & r_{q_{0e}} & = q_{0e} - \tilde{q_{0e}} \\
\tilde{t_7} & = t_5 + esq \times t_6 & t_7 & = \text{near}(\tilde{t_7}, 53) & r_{t_7} & = t_7 - \tilde{t_7}
\end{array}$$

We also define μ as $\mu = y_{0s} q_{0s}^2 - 1$.

From an automatic case-analysis of the look-up table, ACL2(r) can show that $|\tilde{e}| < 2^{-6}$, $|\mu| \le 397/128 \times 2^{-53}$, $50/71 \le q_{0s} < 71/50$ and $1/2 \le y_{0s} < 2$. The amount rounded off by the nearest-mode rounding is at most half of the ulp as stated in the following lemma.

Lemma 1. *For rational number x and a positive integer n,*

$$|near(x, n) - x| \le ulp(x, n)/2$$

By applying this lemma, we can show that $|r_e| \le 2^{-60}$.

Furthermore, from the definition given above and Lemma 1, ACL2(r) proves that other intermediate values satisfy the following conditions.

$$\begin{array}{llllll}
|\tilde{t_3}| & \le 2^{-5} + 2^{-11} & |t_3| & \le 2^{-5} + 2^{-11} & |r_{t_3}| & \le 2^{-58} \\
|\tilde{t_4}| & \le 2^{-4} + 2^{-10} & |t_4| & \le 2^{-4} + 2^{-10} & |r_{t_4}| & \le 2^{-57} \\
|\tilde{esq}| & \le 2^{-12} & |esq| & \le 2^{-12} & |r_{esq}| & \le 2^{-66} \\
|\tilde{t_5}| & \le 2^{-1} + 2^{-7} & |t_5| & \le 2^{-1} + 2^{-7} & |r_{t_5}| & \le 2^{-54} \\
|\tilde{e_1}| & \le 2^{-5} + 2^{-50} & |e_1| & \le 2^{-5} + 2^{-50} & |r_{e_1}| & \le 2^{-58} \\
|\tilde{t_1}| & < \frac{182}{128} & |t_1| & < \frac{182}{128} & |r_{t_1}| & \le 2^{-53} \\
|\tilde{t_6}| & \le 2^{-4} + 2^{-9} & |t_6| & \le 2^{-4} + 2^{-9} & |r_{t_6}| & \le 2^{-57} \\
|\tilde{q_{0e}}| & \le \frac{182}{128} \times 2^{-5} & |q_{0e}| & \le \frac{182}{128} \times 2^{-5} & |r_{q_{0e}}| & \le 2^{-58} \\
|\tilde{t_7}| & \le 2^{-1} + 2^{-6} & |t_7| & \le 2^{-1} + 2^{-6} & |r_{t_7}| & \le 2^{-54}
\end{array}$$

Next we represent each intermediate value as the sum of the formula the intermediate value is intended to represent and an error term. For example, esq is the sum of $\tilde{e} \times \tilde{e}$ and the error term $E_{esq} = 2\tilde{e}r_e + r_e^2 + r_{esq}$.

$$\begin{aligned}
esq &= (\tilde{e} + r_e) \times (\tilde{e} + r_e) + r_{esq} \\
&= \tilde{e} \times \tilde{e} + 2\tilde{e}r_e + r_e^2 + r_{esq} \\
&= \tilde{e} \times \tilde{e} + E_{esq},
\end{aligned}$$

From the magnitude of the intermediate values, the size of the error term E_{esq} can be calculated as:

$$|E_{esq}| < 2|\tilde{e}||r_e| + |r_e|^2 + |r_{esq}| < 2^{-64}$$

Similarly, with appropriate error terms $E_{q_{0e}}$, E_{t_3}, E_{t_4}, E_{t_5}, E_{t_6} and E_{t_7}, we can represent $q_{e0}, t_3, t_4, t_5, t_6$ and t_7 in the following way.

$$\begin{array}{ll}
q_{0e} = q_{0s}(\tilde{e} + \mu) + E_{q0e} & |E_{q_{0e}}| \le 2^{-56} \\
t_3 = c_4 + c_5\tilde{e} + E_{t_3} & |E_{t_3}| \le 2^{-58} + 2^{-65} \\
t_4 = c_2 + c_3\tilde{e} + E_{t_4} & |E_{t_4}| \le 2^{-57} + 2^{-64} \\
t_5 = c_0 + c_1\tilde{e} + E_{t_5} & |E_{t_5}| \le 2^{-54} + 2^{-61} \\
t_6 = c_2 + c_3\tilde{e} + c_4\tilde{e}^2 + c_5\tilde{e}^3 + E_{t_6} & |E_{t_6}| \le 2^{-56} + 2^{-63} \\
t_7 = c_0 + c_1\tilde{e} + c_2\tilde{e}^2 + c_3\tilde{e}^3 + c_4\tilde{e}^4 + c_5\tilde{e}^5 + E_{t_7} & |E_{t_7}| \le 2^{-53} + 2^{-60}
\end{array}$$

Let $P(x)$ denote the polynomial[3] $c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4 + c_5x^5$. Then we can represent the final estimate q_1 as:

$$q_1 = q_{0s} + q_{0s}(\tilde{e} + \mu)P(\tilde{e}) + E_{q1} \qquad |E_{q1}| \le 2^{-56}$$

with an appropriate error term E_{q1}.

We are going to rewrite the last equation using a number of series approximation. Let us define

$$\begin{aligned}
E_{su} &= \sqrt{1+\mu} - (1 + \mu/2) \\
E_{se} &= \sqrt{1-\tilde{e}} - (1 - \tilde{e}/2) \\
E_{cheb} &= \sqrt{1-\tilde{e}} - (1 + \tilde{e}P(\tilde{e}))
\end{aligned}$$

Further we define the following error terms:

$$\begin{aligned}
E_{pet2} &= P(\tilde{e}) - (-1/2 - \tilde{e}/8) \\
E_{sb} &= q_{0s} \times E_{se} - \sqrt{b} \times (E_{su} + \mu/2) \\
E_{final} &= -3/8 \times q_{0s}\mu\tilde{e} - q_{0s}E_{cheb} + \mu E_{sb}/2 + \sqrt{b}E_{su} + E_{q1} + q_{0s}\mu E_{pet2}
\end{aligned}$$

Then we can prove that

$$q_1 = \sqrt{b} + E_{final}.$$

This proof of this equation is long and tedious, but ACL2(r) can prove it using only five `defthm` proof commands. The details of the proof are provided in [Saw02].

Let us assume that the following inequalities hold:

$$|E_{su}| \le 2^{-105} \tag{1}$$

$$|E_{se}| \le 2^{-15} + 2^{-19} \tag{2}$$

$$|E_{cheb}| \le 3/2 \times 2^{-58} \tag{3}$$

[3] Since coefficients c_0 through c_5 depend on b, $P(x)$ depends on b as well. In ACL2(r), we define it as a function that takes b and x as its arguments.

Then, we can prove from the definition of E_{final} that

$$|E_{final}| < 2^{-55} \leq \text{ulp}(b, 53)/4.$$

This implies that the final estimate q_1 is less than one quarter of the ulp away from $\sqrt{b}$. The inequalities (1)-(3) are proof obligations that we must resolve using Taylor's theorem.

5 Use of Taylor's Theorem in Error Calculation

In this section, we prove the inequalities (1)-(3) from the previous section that give the upper bounds of $|E_{su}|, |E_{se}|$ and $|E_{cheb}|$.

Since the square root function is infinitely differentiable and its derivatives are continuous on the positive domain, we can apply Taylor's theorem on its entire positive domain. Let us define $a(n, x)$ as:

$$a(n, x) = \frac{1}{n!} \prod_{i=0}^{n-1} (\frac{1}{2} - i) x^{\frac{1}{2} - n}.$$

Function $a(n, x_0)$ gives the n'th Taylor coefficient for $\sqrt{x}$ at x_0. Thus Taylor's equation presented in Section 3 can be written for square root as:

$$\sqrt{x_0 + \delta} = \sum_{i=0}^{n-1} a(i, x_0)\delta^i + a(n, \xi)\delta^n.$$

Given that (`nth-tseries-sqrt` i x_0 δ) represents i'th term in the Taylor series $a(i, x_0)\delta^i$, we define the n'th degree Taylor polynomial in the ACL2(r) logic as follows:

```
(defun tseries-sqrt (n x delta)
  (if (zp n)
      0
    (+ (nth-tseries-sqrt (- n 1) x delta)
       (tseries-sqrt (- n 1) x delta))))
```

We also define the term ξ in Taylor's theorem as the function (`taylor-sqrt-xi` n x δ) using the same principle as `tay-xi`. Then Taylor's theorem as proved in Section 3 can be instantiated to the following theorem:

```
(defthm taylor-theorem-on-sqrt
  (implies (and (integerp n) (< 1 n) (<= n (tay-degree-ubound))
                (realp x) (realp delta) (< 0 x) (< 0 (+ x delta)))
   (equal (acl2-sqrt (+ x delta))
          (+ (tseries-sqrt n x delta)
             (nth-tseries-sqrt n (taylor-sqrt-xi n x delta) delta)))))
```

We can also prove that `(taylor-sroot-xi` n x δ`)` returns a real number that is in the open segment $(x, x+\delta)$. The condition `(<= n (tay-degree-ubound))` guarantees that `n` is *i-limited* in the sense discussed in Section 2.

Using this theorem, we will prove the upper bound of the error terms. An upper bound of $|E_{su}|$ can be directly calculated by applying Taylor's theorem. Since E_{su} is equal to the second degree Taylor remainder for the function $\sqrt{1+\mu}$ at $\mu = 0$:

$$E_{su} = \sqrt{1+\mu} - (1+\mu/2) = -\frac{1}{8}(1+\xi)^{-\frac{3}{2}}\mu^2.$$

Since $|\xi| < |\mu| \leq \frac{397}{128} \times 2^{-53}$, an upper bound of $|E_{su}|$ is given as

$$|E_{su}| < \frac{1}{8}(1 - \frac{397}{128} \times 2^{-53})^{-\frac{3}{2}} \times (\frac{397}{128} \times 2^{-53})^2 < 2^{-105}.$$

Similarly, the upper bound for $|E_{se}|$ can be calculated as

$$|E_{se}| < \frac{1}{8} \times (1 - 2^{-6})^{-\frac{3}{2}} \times (2^{-6})^2 < 2^{-15} + 2^{-19}.$$

We also used the Taylor series in the calculation of an upper bound of $|E_{cheb}| = |\sqrt{1-\tilde{e}} - (1 + \tilde{e}P(\tilde{e}))|$. Since the Chebyshev polynomial $1 + \tilde{e}P(\tilde{e})$ is a better approximation of $\sqrt{1-\tilde{e}}$ than the Taylor polynomial of the same degree, using a Taylor polynomial in the measurement of this approximation error is not straightforward.

Our approach is to divide the range of $\tilde{e}$ into small segments, generate a Taylor polynomial for each segment, and use it to calculate the error of the Chebyshev polynomial for every segment one at a time. Each segment should be small enough so that the generated Taylor polynomial is far more accurate than the Chebyshev polynomial. The range of $\tilde{e}$ is $[-2^{-6}, 2^{-6}]$. We divided it into 128 segments of size 2^{-12} and performed error analysis on each segment.

In order to carry out the proof on many segments efficiently, it is critical to automate the error analysis at each segment. One of the major obstacles to automatic analysis is that ACL2(r) cannot directly compute the square root, because `acl2-sqrt` is defined using non-standard analysis, and it might return irrational numbers which cannot be computed by ACL2(r).

Consequently, we used a function approximating the square root function. The ACL2(r) function `(iter-sqrt` x ϵ`)` returns a rational number close to $\sqrt{x}$. In this paper, we write $\sqrt{x}^{\star}_{\epsilon}$ to denote this function. This function satisfies $\sqrt{x}^{\star}_{\epsilon} \times \sqrt{x}^{\star}_{\epsilon} \leq x$ and $x - \sqrt{x}^{\star}_{\epsilon} \times \sqrt{x}^{\star}_{\epsilon} < \epsilon$ for a positive rational number ϵ. From this, we can easily prove that $\sqrt{x} - \sqrt{x}^{\star}_{\epsilon} \leq max(\epsilon, \epsilon/x)$.

Using this function, we define an ACL2(r) function $a^{\star}(x, i, \eta)$ that calculates the approximation of $a(i, x_0)$. More precisely,

$$a^{\star}(x, i, \eta) = \frac{1}{n!}\prod_{i=0}^{n-1}(1/2 - i)\sqrt{x}^{\star}_{\eta}x^{-n}.$$

Then we can show

$$|a(x,n) - a^\star(x,n,\eta)| \le \frac{1}{n!}\prod_{i=0}^{n-1}(1/2 - i) \times max(\eta, \eta/x)x^{-n}.$$

As discussed in Section 4, our algorithm selects Chebyshev coefficients from two sets of constants depending on the 6th fraction bit of b. Let

$$Cheb_p(e) = 1 + c_{0p}e + c_{1p}e^2 + c_{2p}e^3 + c_{3p}e^4 + c_{4p}e^5 + c_{5p}e^6$$
$$Cheb_n(e) = 1 + c_{0n}e + c_{1n}e^2 + c_{2n}e^3 + c_{3n}e^4 + c_{4n}e^5 + c_{5n}e^6$$

Then $E_{cheb} = \sqrt{1-\tilde{e}} - Cheb_p(\tilde{e})$ when the 6th fraction bit of b is 0, and $E_{cheb} = \sqrt{1-\tilde{e}} - Cheb_n(\tilde{e})$ when it is 1.

Let us calculate the size of E_{cheb} for the case where the 6th fraction bit of b is 1. Even though the heuristics discussed in Section 4 suggests that $\tilde{e}$ tends to be negative in this case, a simple analysis shows that $-2^{-6} \le \tilde{e} \le 3/2 \times 2^{-12}$; e could take some positive numbers. We analyze the entire domain of $\tilde{e}$ by dividing it into 66 small segments. We substitute $e_0 - e_\delta$ for $\tilde{e}$ in $\sqrt{1-\tilde{e}} - Cheb_n(\tilde{e})$, where e_0 is one of the 66 constants -63×2^{-12}, -62×2^{-12}, …, 2×2^{-12}, while e_δ is a new variable that satisfies $0 \le e_\delta \le 2^{-12}$. The upper bound for the entire domain of $\tilde{e}$ is simply the maximum value of all the upper bounds for the 66 segments.

The upper bound for $|E_{cheb}|$ can be represented as the summation of three terms.

$$\begin{aligned}|\sqrt{1-e_0+e_\delta} - Cheb_n(e_0 - e_\delta)| \le & \left|\sqrt{1-e_0+e_\delta} - \sum_{i=0}^{5} a(1-e_0, i)e_\delta^i\right| + \\ & \left|\sum_{i=0}^{5} a(1-e_0, i)e_\delta^i - \sum_{i=0}^{5} a^\star(1-e_0, i, \eta)e_\delta^i\right| + \\ & \left|\sum_{i=0}^{5} a^\star(1-e_0, i, \eta)e_\delta^i - Cheb_n(e_0 - e_\delta)\right|.\end{aligned}$$

An upper bound for the first term can be given by applying Taylor's theorem.

$$\begin{aligned}|\sqrt{1-e_0+e_\delta} - \sum_{i=0}^{5} a(1-e_0,i)e_\delta^i| \le |a(\xi,6)e_\delta^6| \le \frac{1}{6!}\prod_{i=0}^{5}|\frac{1}{2} - i||\xi^{-\frac{11}{2}}||e_\delta^6| \\ < \frac{1}{6!}\prod_{i=0}^{5}(\frac{1}{2} - i) \times max((1-e_0)^{-6}, (1-e_0)^{-5}) \times 2^{-78}.\end{aligned}$$

Here ξ is the constant satisfying Taylor's theorem such that $1 - e_0 < \xi < 1 - e_0 + e_\delta$. Note that this upper bound can be calculated by ACL2(r) as it does not contain square root nor variables.

The upper bound for the second term can be calculated as follows:

$$\left|\sum_{i=0}^{n-1} a(1-e_0,i)e_\delta^i - \sum_{i=0}^{n-1} a^\star(1-e_0,i,\eta)e_\delta^i\right| \le \sum_{i=0}^{n-1} |a(1-e_0,i) - a^\star(1-e_0,i,\eta)|\, e_\delta^i$$

$$\le \sum_{i=0}^{n-1}\left\{\frac{2^{-13i}}{i!}\prod_{j=0}^{i-1}(1/2-j)\times max(\eta,\eta/(1-e_0))\times(1-e_0)^{-i}\right\}.$$

We chose η to be 2^{-60} to make this term small enough. Again the upper bound has no variables involved and can be calculated by ACL2(r).

The third term is the difference between the Chebyshev series approximation and the Taylor series approximation. Since e_0 and η are constant in the third term, we can simplify the term $\sum_{i=0}^{5} a^\star(1-e_0,i,\eta)e_\delta^i - Cheb_n(e_0-e_\delta)$ into a polynomial of e_δ of degree 6. Here having the computational function $a^\star(1-e_0,i,\eta)$ rather than the real Taylor coefficient allows ACL2(r) to automatically simplify the formula. We denote the resulting polynomial as $\sum_{i=0}^{6} b_i e_\delta^i$, where coefficient b_i is a constant automatically calculated by ACL2(r) during the simplification. Then the upper bound can be given as

$$\left|\sum_{i=0}^{5} a^\star(1-e_0,i,\eta)e_\delta^i - Cheb_n(e_0-e_\delta)\right| = |\sum_{i=0}^{6} b_i e_\delta^i| \le \sum_{i=0}^{6} |b_i|\times 2^{-13i}.$$

By adding the three upper bounds, we can prove that

$$|\sqrt{1-e_0+e_\delta} - Cheb_n(e_0-e_\delta)| < 3/2\times 2^{-58}.$$

for all 66 values for e_0. This is the upper bound of $|E_{cheb}|$ when the 6th fraction bit of b is 1. Similarly, we can prove that $|\sqrt{1-e_0+e_\delta} - Cheb_p(e_0-e_\delta)| < 3/2\times 2^{-58}$ for the case where the 6th fraction bit is 0. In this case, $-6/5\times 2^{-12} \le \tilde{e} \le 2^{-6}$. Since the ranges of $\tilde{e}$ are overlapping for the two cases, we repeat the upper bound analysis on some segments. Summarizing the two cases, $|E_{cheb}|$ has the upper bound $3/2\times 2^{-58}$. This will complete the proof of the previous section.

Each step of this proof has been mechanically checked by ACL2(r). By making the upper bounds of the error terms computational by ACL2(r), this unbound calculation for the hundreds of segments was performed automatically.

6 Discussion

We mechanically proved Taylor's theorem using non-standard analysis implemented in ACL2(r), and then we used it to formally verify that the Power4 square root algorithm satisfies an error size requirement. One major challenge for its verification was evaluating the approximation error for the Chebyshev polynomial. We have performed error size calculation of the Chebyshev polynomial in hundreds of small segments. For each segment, a Taylor series is generated to evaluate the approximation error of the Chebyshev series. This type of proof can be carried out only with a mechanical theorem prover or other type of computer program, because the simplification of hundreds of formulae is too tedious for humans to carry out correctly.

One might wonder why we did not prove theorems about Chebyshev series and use them in the error size analysis. One answer is that the mathematics behind Chebyshev series is much more complex than Taylor series. We believe our approach is a good mix of relatively simple mathematics and the power of mechanical theorem proving.

The upper bound proof of the Chebyshev series approximation was carried out automatically after providing the following:

1. ACL2(r) macros that automates the proof for small segments.
2. Computed hints that guides the proof by case analysis.
3. A set of rewrite rules that simplify a polynomial of rational coefficients.

Since this proof is automatic, we could change a number of parameters to try different configurations. For example, we changed the segment size and η used to calculate $\sqrt{x}^*_\eta$. In fact, Chebyshev series approximation error was obtained by trial-and-error. At first, we set a relatively large number to an ACL2(r) constant `*apx_error*` and ran the prover to verify $|E_{cheb}| <$`*apx_error*`. If it is successful, we lowered the value of `*apx_error*`, iterated the process until the proof failed.

The approximation error analysis using Taylor series requires less computational power than brute-force point-wise analysis. When $|\tilde{e}| \leq 2^{-6}$, the value $\sqrt{1-\tilde{e}} \simeq 1-\tilde{e}/2$ ranges approximately from $1-2^{-7}$ to $1+2^{-7}$. In order to prove that the error of its Chebyshev series approximation is less than 1.5×2^{-58}, our estimate suggests that we need to check over 2^{50} points, assuming the monotonicity of the square root function. On the other hand, the entire verification of the square root algorithm using Taylor's polynomials took 673 seconds on a Pentium III 400MHz system. It is not sheer luck that we could finish the error calculation by analyzing only hundreds of segments. Because the n'th degree Taylor remainder for the square root function is $O(d^n)$ for the segment size d, the approximation error by a Taylor series quickly converges to 0 by making the segment smaller when n is, say, 6. We believe that we can apply our technique to other algorithms involving series calculations.

Acknowledgment We thank Brittany Middleton for proving basic theorems on continuous functions and their derivatives. We also acknowledge the usefulness of the floating-point library distributed with ACL2, which was developed by David Russinoff.

References

[AGS99] Ramesh C. Agarwal, Fred G. Gustavson, and Martin S. Schmookler. Series approximation methods for divide and square root in the power3 processor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 116–123, 1999.

[AJK$^+$00] Mark D. Aagaard, Robert B. Jones, Roope Kaivola, Katherine R. Kohatsu, and Carl-Johan H. Seger. Formal verification of iterative algorithms in microprocessors. *Proceedings Design Automation Conference (DAC 2000)*, pages 201 – 206, 2000.

[Ful78] W. Fulks. *Advanced Calculus: an introduction to analysis.* John Wiley & Sons, third edition, 1978.

[Gam99] Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2.* PhD thesis, University of Texas at Austin, 1999.

[Gam02] R. Gamboa. The correctness of the Fast Fourier Trasnform: A structured proof in ACL2. *Formal Methods in System Design*, 20:91–106, January 2002.

[GK01] R. Gamboa and M. Kaufmann. Nonstandard analysis in ACL2. *Journal of Automated Reasoning*, 27(4):323–351, November 2001.

[GM02] R. Gamboa and B. Middleton. Taylor's formula with remainder. In *Proceedings of the Third International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2002)*, 2002.

[Ins] Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985.

[Kau00] M. Kaufmann. Modular proof: The fundamental theorem of calculus. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 6. Kluwer Academic Press, 2000.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34. IEEE Computer Society Press, June 1996.

[Nel] E. Nelson. On-line books: Internal set theory. Available on the world-wide web at `http://www.math.princeton.edu/ñelson/books.html`.

[Nel77] E. Nelson. Internal set theory. *Bulletin of the American Mathematical Society*, 83:1165–1198, 1977.

[OZGS99] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, February 1999.

[PH96] David A. Patterson and John L. Hennessey. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[Rob59] A. Robinson. Model theory and non-standard arithmetic, infinitistic methods. In *Symposium on Foundations of Mathematics*, 1959.

[Rob88] A. Robert. *Non-Standard Analysis.* John Wiley, 1988.

[Rus98] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the Floating Point Multiplication, Division, and Square Root Algorithm of the AMDK7 Processor. *J. Comput. Math. (UK)*, 1, 1998.

[Rus99] D. Russinoff. A Mechanically Checked Proof of Correctness of the AMDK5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1), 1999.

[Saw02] J. Sawada. Formal verification of divide and square root algorithms using series calculation. In *Proceedings of the Third International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2002)*, 2002.