Taylor's Formula with Remainder

Ruben A. Gamboa
Computer Science Department
University of Wyoming
PO Box 3295
Laramie, WY 82071-3295
ruben@cs.uwyo.edu

Brittany Middleton
Computer Sciences Department
Taylor Hall 2.124
The University of Texas at Austin
Austin, TX 78712-1188
brittany@cs.utexas.edu

February 28, 2002

Abstract

In this paper, we present a proof in ACL2(r) of Taylor's formula with remainder. This important theorem allows a function f with $n+1$ derivatives on the interval $[a, b]$ to be approximated with a Taylor series of n terms centered at a. Moreover, the formula allows the error in the approximation to be bounded by a term involving the $(n+1)$st derivative of f on (a, b).

The results in this paper were motivated in part by Jun Sawada's work with ACL2(r) verifying that the approximation used in the square root calculation of the IBM Power4 processor has the accuracy required. Sawada's proof effort used a Taylor approximation to the square root function. However, the support for such development in ACL2(r) is lacking [17]. This paper shows how such results can be proved in ACL2(r). It also shines a spotlight on some limitations of ACL2(r) that complicate the proof. Future work will address these limitations.

1 Introduction

ACL2(r) is a modified version of ACL2 with support for irrational real and complex numbers [5]. The logical foundation for ACL2(r) is provided by non-standard analysis, initially developed by Robinson and later axiomatized by Nelson [13, 15]. In essence, non-standard analysis formalizes the intuitive arguments in calculus that appeal to infinitesimal quantities, giving a rigorous foundation to familiar calculus notions such as "infinitely small," "infinitely close," and "infinitely large." There are several good introductions to non-standard analysis, for example [1, 10, 11, 12, 14]. This paper assumes that the reader has some familiarity with non-standard analysis, though it will introduce important notions from non-standard analysis as they appear.

As a tool, ACL2(r) is sufficient to reason about foundational questions regarding the irrationals. For example, it has been used to define the basic trigonometric functions, and to prove various trigonometric identities. On a more foundational level, it has been used to prove key theorems from elementary analysis, such as the intermediate value theorem, mean value theorem, and the fundamental theorem of calculus [3, 4, 6].

However, the majorify of these proofs are at the foundational level. Lacking in ACL2(r) are simple mechanisms for defining new irrational functions. In this paper, we formalize one approach to these definitions, Taylor's formula. In its incantation as a power series, this formula was used explicitly to define the exponential function in ACL2(r) from scratch. Here we use it to approximate a function whose first n derivatives are known at a specific point. This was motivated in part by Jun Sawada's efforts at approximating the square root function in ACL2(r) [17].

The remainder of the paper is organized as follows. Section 2 presents a hand proof of Taylor's formula with remainder. This is followed in section 3 by a discussion of the lemmas required by the proof. The focus will be on identifying which results can be reused from previous ACL2(r) proofs and which results require new proof development. Section 4 describes an unexpected obstacle, an unfortunate combination of the limitations of functional instantiation with pseudo-lambda expressions and with non-classical functions. The main ACL2(r) proof of Taylor's formula with remainder is given in section 5. Finally, some conclusions and directions for future work are listed in section 6.

2 The Hand Proof

Given a function f with n continuous derivatives on the interval $[a, b]$ and its $(n+1)$st derivative defined on (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. Specifically,

$$f(x) = f(a) + \sum_{i=1}^{n} \frac{f^{(i)}(a) \cdot (x-a)^i}{i!} + \frac{f^{(n+1)}(\beta)}{(n+1)!}(x-a)^{n+1},$$

where β is some point in the interval (a, b).

The proof, presented in [2] among others, follows 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 β for which $F'(\beta) = 0$. Taylor's formula follows from solving $F'(\beta) = 0$ for $f(x)$.

The function F is defined differently for each point x in $[a, b]$. In the following discussion, let x be a specific, fixed point in $[a, b]$. Define F over the interval $[a, x]$ as follows:

$$F(t) = f(x) - f(t) - \sum_{i=1}^{n} \frac{f^{(i)}(t) \cdot (x-t)^i}{i!} + \frac{(x-t)^{n+1}}{(n+1)!}A,$$

where A is a constant that does not depend on t. The specific value of A is chosen to satisfy the criteria for Rolle's lemma. Specifically, before Rolle's lemma can be applied to F on $[a, x]$, we must show that F is differentiable, and that $F(a) = F(x) = 0$. Clearly $F(x) = 0$, since all the $(x-t)^i$ terms vanish, leaving only $f(x) - f(t)$ with $t = x$. To ensure $F(a) = 0$, it is only necessary to set $F(a) = 0$ in the expression above and solve for A. This gives

$$A = -\left\{ f(x) - f(a) - \sum_{i=1}^{n} \frac{f^{(i)}(a) \cdot (x-a)^i}{i!} \right\} \frac{(n+1)!}{(x-a)^{n+1}}.$$

Moreover, F is clearly differentiable since it is the sum of differentiable terms. Using Rolle's lemma, we can conclude that there is some $\beta \in (a, x)$ such that $F'(\beta) = 0$. But observe that the terms in $F'(t)$ neatly cancel out. That is, the derivative of $\sum_{i=1}^{n} \frac{f^{(i)}(t) \cdot (x-t)^i}{i!}$ simplifies to $-f'(t) + \frac{f^{(n+1)}(t)(x-t)^n}{n!}$. This means that

$$F'(\beta) = 0 = -\frac{f^{(n+1)}(\beta)(x-\beta)^n}{n!} - \frac{(x-\beta)^n}{n!}A.$$

Since $x \neq \beta$, the terms $\frac{(x-\beta)^n}{n!}$ can be factored and eliminated, leaving

$$\begin{aligned} 0 &= -f^{(n+1)}(\beta) - A \\ &= -f^{(n+1)}(\beta) + \left\{ f(x) - f(a) - \sum_{i=1}^{n} \frac{f^{(i)}(a) \cdot (x-a)^i}{i!} \right\} \frac{(n+1)!}{(x-a)^{n+1}}. \end{aligned}$$

Solving for $f(x)$ in the formula above results in Taylor's formula with remainder:

$$f(x) = f(a) + \sum_{i=1}^{n} \frac{f^{(i)}(a) \cdot (x-a)^i}{i!} + \frac{f^{(n+1)}(\beta)}{(n+1)!}(x-a)^{n+1}.$$

3 The Foundational Lemmas

The proof outlined in section 2 uses several basic lemmas from analysis. In this section, we try to identify these lemmas and make sure we have proved them before proceeding to the main argument.

Clearly, the proof depends on Rolle's lemma, which states that if a function f is differentiable on the interval $[a, b]$, and if $f(a) = f(b)$, then there is some $\beta \in [a, b]$ such that $f'(\beta) = 0$. Rolle's lemma was proved in ACL2(r) in [4], and we will use that proof here.

The hand proof applies Rolle's lemma to the following function F on the range $[a, x]$:

$$F(t) = f(x) - f(t) - \sum_{i=1}^{n} \frac{f^{(i)}(t) \cdot (x-t)^i}{i!} + \frac{(x-t)^{n+1}}{(n+1)!} A.$$

So we need to verify that F satisfies the hypothesis of Rolle's lemma. In particular, we need to show that F is differentiable on $[a, x]$ and that $F(a) = F(x)$. The latter test is easily verified, since we can choose the constant (with respect to t) A to force $F(a) = F(x) = 0$. That F is differentiable (with respect to t) follows from the fact that the first $n + 1$ derivatives of f are defined on $[a, b]$ and the rules governing derivatives of sums and products. Also needed is the fact that $(x - t)^n$ is differentiable (with respect to t), which can be established from the chain rule and the differentiability of x^n (with respect to x). This suggests that we need to prove basic lemmas involving the derivatives of compositions of functions.

The basic lemmas include $(f \cdot g)' = fg' + gf'$, $(k \cdot f)' = k \cdot f'$, $(f \circ g)' = f' \circ g \cdot g'$, and $(x^n)' = n \cdot x^{n-1}$, the first of which is shown below. As in [4], we proceed by using `encapsulate` to introduce generic differentiable functions, the f and g of the claim above:

```
(encapsulate
 ((dc-fn1 (x) t)
  (dc-fn2 (x) t)
  (dc-fn1-deriv (x) t)
  (dc-fn2-deriv (x) t)
  (dc-fn-domain-p (x) t))

 ;; The function dc-fn-domain-p recognizes a standard
 ;; interval of reals

 (local (defun dc-fn-domain-p (x) (realp x)))

 (defthm dc-fn-domain-standard
   (implies (dc-fn-domain-p x)
            (dc-fn-domain-p (standard-part x))))

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

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

```
;; fn1 and fn1-deriv are standard real-valued functions,
;; and fn1-deriv is the derivative of fn1

(local (defun dc-fn1 (x) x))
(local (defun dc-fn1-deriv (x) (declare (ignore x)) 1))

(defthm dc-fn1-standard
  (implies (and (dc-fn-domain-p x)
                (standard-numberp x))
           (standard-numberp (dc-fn1 x))))

(defthm dc-fn1-deriv-standard
  (implies (and (dc-fn-domain-p x)
                (standard-numberp x))
           (standard-numberp (dc-fn1-deriv x))))

(defthm dc-fn1-real
  (implies (dc-fn-domain-p x)
           (realp (dc-fn1 x))))

(defthm dc-fn1-deriv-real
  (implies (dc-fn-domain-p x)
           (realp (dc-fn1-deriv x))))

(defthm dc-fn1-derivative
  (implies (and (standard-numberp x)
                (dc-fn-domain-p x)
                (dc-fn-domain-p y)
                (i-close x y) (not (= x y)))
           (i-close (/ (- (dc-fn1 x) (dc-fn1 y)) (- x y))
                    (dc-fn1-deriv x))))

;; similar definitions and constraints for fn2...

)
```

To prove that the derivative of the sums is the sum of the derivatives, we introduce functions for the sums and their derivative:

```
(defun dc-fn1+fn2 (x)
  (+ (dc-fn1 x) (dc-fn2 x)))

(defun dc-fn1+fn2-deriv (x)
  (+ (dc-fn1-deriv x) (dc-fn2-deriv x)))
```

Suppose x is *standard* and y is *i-close* to x. From `dc-fn1-derivative` it follows that $\frac{(f_1(x)-f_1(y))}{(x-y)}$ is *i-close* to ${f_1}'(x)$. Similarly, $\frac{(f_2(x)-f_2(y))}{(x-y)}$ is *i-close* to ${f_2}'(x)$. Adding these two and simplifying yields the desired result. The key lemma is that when x_1 is *i-close* to x_2 and y_1 is *i-close* to y_2, x_1+y_1 is *i-close* to x_2+y_2. ACL2(r) proves this lemma quickly:

```
(defthm close-plus
  (implies (and (i-close x1 x2)
                (i-close y1 y2))
           (i-close (+ x1 y1) (+ x2 y2)))
  :hints (("Goal" :in-theory (enable i-close))))
```

With this lemma in the ACL2(r) database, ACL2(r) can prove the main result:

```
(defthm dc-fn1+fn2-derivative
  (implies (and (standard-numberp x)
                (dc-fn-domain-p x)
                (dc-fn-domain-p y)
                (i-close x y) (not (= x y)))
           (i-close (/ (- (dc-fn1+fn2 x)
                          (dc-fn1+fn2 y))
                       (- x y))
                    (dc-fn1+fn2-deriv x)))
  :hints (("Goal"  :
           in-theory (disable close-plus)
           :use (dc-fn1-derivative
                 dc-fn2-derivative
                 (:instance close-plus
                    (x1 (/ (- (dc-fn1 x)
                              (dc-fn1 y))
                           (- x y)))
                    (x2 (dc-fn1-deriv x))
                    (y1 (/ (- (dc-fn2 x) (dc-fn2 y))
                           (- x y)))
                    (y2 (dc-fn2-deriv x)))))))
```

The hints ensure that ACL2(r) has all the appropriate lemma instances to prove the result. In the remainder of this paper, we will omit the specific hints, leaving only a reminder that they are required.

Similar theorems take care of the rules $(f \cdot g)' = fg' + gf'$, $(k \cdot f)' = k \cdot f'$, $(f \circ g)' = f' \circ g \cdot g'$, and $(x^n)' = n \cdot x^{n-1}$. Most of these theorems require substantially more work to prove than the $f + g$ case, but their proofs follow the established non-standard analysis arguments.

4 An Unexpected Obstacle

A naive application of the lemmas described in section 3 runs into an unfortunate limitation of ACL2(r). Consider the functions $G_1(x) = x$ and $G_2(x) = x + a$ where a is some constant, or more specifically a value held fixed for a portion of the proof. We would like to apply the lemma `dc-fn1+fn2-derivative` to show that the derivative of $(G_1 + G_2)(x)$ is 2:

```
(defun G1+G2 (x a)
  (+ x x a))

(defun G1+G2-deriv (x)
  (declare (ignore x))
  2)

(defthm G1+G2-derivative
  (implies (and (standard-numberp x)
                (standard-numberp a)
                (realp x)
                (realp y)
                (realp a)
                (i-close x y) (not (= x y)))
```

```
          (i-close (/ (- (G1+G2 x a) (G1+G2 y a))
                      (- x y))
                   (G1+G2-deriv x)))
  :hints (("Goal"
          :use ((:functional-instance
                 dc-fn1+fn2-derivative
                 (dc-fn-domain-p realp)
                 (dc-fn1 (lambda (x) x))
                 (dc-fn2 (lambda (x) (+ x a)))
                 (dc-fn1+fn2 (lambda (x) (G1+G2 x a)))
                 (dc-fn1-deriv (lambda (x) 1))
                 (dc-fn2-deriv (lambda (x) 1))
                 (dc-fn1+fn2-deriv G1+G2-deriv))))))
```

This fails because ACL2(r) has to establish that the functional instance satisfies the constraints on `dc-fn1` and `dc-fn2`. In particular, one of the constraints is the following:

```
(defthm dc-fn2-real
  (implies (dc-fn-domain-p x)
           (realp (dc-fn2 x))))
```

When applied to the expression `(+ x a)`, this yields the goal

```
(implies (realp x) (realp (+ x a)))
```

which is false by itself. It is, however, true in the context of the hypotheses of the theorem, which ensure `a` is a real number. There is an established way to deal with this hurdle in ACL2, namely to use a term that preserves the constraint while simplifying to `(+ x a)` when the hypothesis of the theorem are considered. This ACL2 trick is briefly mentioned in [16]. In this case, an obvious candidate is the term `(+ x (realfix a))` which is equal to `(+ x a)` when `a` is real and which is real whenever `x` is real, regardless of whether `a` is also real or not.

However, this only delays the problem. Another constraint is the following:

```
(defthm dc-fn2-standard
  (implies (and (dc-fn-domain-p x)
                (standard-numberp x))
           (standard-numberp (dc-fn2 x))))
```

This results in the following proof obligation:

```
(implies (and (realp x) (standard-numberp x))
         (standard-numberp (+ x (realfix a))))
```

Again, this constraint is false. Moreover, attempting to fix it using the same trick as above will fail. For example, suppose we instantiate dc-fn2 with `(if (standard-numberp a) (+ x (realfix a)) x)`. ACL2(r) will not allow this instantiation, because the term uses tht non-classical function `standard-numberp`, and ACL2(r) requires that functions used in a functional instantiation be classical.

What we need is a way to add `(standard-numberp a)` to the hypothesis of the instantiated version of `dc-fn2-standard`. Smuggling the extra hypothesis into the substitution of `dc-fn2` does not work, so we have little choice but to add it to the constraint `dc-fn2-standard` itself. To do this, we define `dc-fn2` as a constrained function of two arguments, both `x` and `a`. Throughout the definition, the extra argument `a` is ignored.

An obvious drawback from this approach is that you have to know how many extra variables to add a priori — in fact, we needed four extra variables for the proof of Taylor's formula. What this means, in practice, is that the book containing the `encapsulate`

must be specialized: We had to create special versions of the books `continuity.lisp` and `derivatives.lisp`.

In general, the restrictions on functionally instantiating constrained functions with non-classical terms are required to preserve soundness. However, it is possible that a less draconian set of requirements will suffice, and we are currently investigating whether some relaxation of the requirement can be permitted.

5 The Main Proof

We are now ready to present the proof in ACL2(r) of Taylor's formula with remainder. The first step is to constrain the function f using `encapsulate` as follows:

```
(encapsulate
 ((tay-fn (i n a x) t)
  (tay-fn-deriv (i n a x) t)
  (tay-domain-p (x) t)
  (tay-n () t))

 ;; tay-domain-p recognizes a standard real interval

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

 (defthm tay-domain-standard
   (implies (tay-domain-p x)
            (tay-domain-p (standard-part x))))

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

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

 ;; tay-fn is a standard real function, depending
 ;; only on x

 (local
  (defun tay-fn (i n a x)
    (declare (ignore i n a x))
    0))

 (defthm tay-fn-ignores-extra-args
   (equal (tay-fn i n a x)
          (tay-fn i2 n2 a2 x))
   :rule-classes nil)
```

```
(defthm tay-fn-standard
  (implies (and (tay-domain-p x)
                (standard-numberp i)
                (standard-numberp n)
                (standard-numberp a)
                (standard-numberp x))
           (standard-numberp (tay-fn i n a x))))

(defthm tay-fn-real
  (implies (tay-domain-p x)
           (realp (tay-fn i n a x))))

;; similarly, tay-fn-deriv is a standard real function,
;; depending only on i and x

...

;; tay-n is a standard natural number

(local
 (defun tay-n ()
   1))

(defthm natural-tay-n
  (and (integerp (tay-n))
       (<= 0 (tay-n)))
  :rule-classes (:rewrite :type-prescription))

(defthm limited-tay-n
  (i-limited (tay-n)))

;; tay-fn-deriv(i,x) is the ith derivative of tay-fn
;; at x, for 0<=i<=tay-n

(defthm tay-fn-deriv-0
  (implies (tay-domain-p x)
           (equal (tay-fn-deriv 0 n a x)
                  (tay-fn 0 n a x))))

(defthm tay-fn-deriv-chain
  (implies (and (standard-numberp x)
                (tay-domain-p x)
                (tay-domain-p y)
                (integerp i)
                (<= 0 i)
                (<= i (tay-n))
                (i-close x y) (not (= x y)))
           (i-close (/ (- (tay-fn-deriv i n a x)
                          (tay-fn-deriv i n a y))
                       (- x y))
                    (tay-fn-deriv (1+ i) n a x))))
)
```

Notice the definition of `tay-fn` includes four parameters, though only one parameter contributes to its value. This is in accordance with the discussion on section 4. The parameters in question are `x`, which is the point where the function is evaluated, and `i`, `n`, and `a`, which are standins for the current term in the approximation, the number of terms in the approximation, and the point over which the Taylor series is expanded, respectively. The constraint `tay-fn-ignores-extra-args` guarantees that `tay-fn` uses only `x` to determine its value. However, this does not mean that other functions using `tay-fn`, such as the partial sums of the Taylor series, are bound by the same restriction. In effect, what `tay-fn-ignores-extra-args` guarantees is that if the value of `tay-fn` of `x` is the same in the first term of Taylor's formula as in the last — which of course it should be, since `tay-fn` is really a function of only one variable.

Notice also that the constraints on `tay-fn` and `tay-fn-deriv` state explicitly that the derivative of `tay-fn` is `tay-fn-deriv`. This is a stronger claim than the claim that `tay-fn` is differentiable, as used in [4]. Recall that the books `continuity.lisp` and `derivatives.lisp`, where Rolle's lemma is proved, had to be customized to include the four extra variables (three of which correspond to `i`, `n`, and `a` in `tay-fn`). For convenience, we also changed the differentiability constraint in `derivatives.lisp` to match the one given above; i.e., the modified books require that the derivative of f be known.

From the definition of `tay-fn` and `tay-fn-deriv`, we can define Taylor's formula as follows:

```
(defun tay-term (i n a x)
  (* (tay-fn-deriv i n a a)
     (expt (- x a) i)
     (/ (factorial i))))

(defun tay-sum (i n a x)
  (declare (xargs :measure (nfix (1+ (- n i)))))
  (if (and (integerp i)
           (integerp n)
           (<= i n))
      (+ (tay-term i n a x)
         (tay-sum (1+ i) n a x))
    0))
```

The function `tay-term` returns the ith term in the Taylor expansion of `tay-fn` around `a`, and `tay-sum` adds the terms from `i` to `n`, inclusive.

Following the hand proof presented in 2, we now define the function `tay-rolle-fn`, which is the intermediate function on which Rolle's lemma is invoked:

```
(defun tay-extra (n a x)
  (* (- (tay-fn 0 n a x) (tay-sum 0 n a x))
     (/ (factorial (1+ n))
        (expt (- x a) (1+ n)))))

(defun tay-rolle-fn (n a x b)
  (+ (tay-fn 0 n x x)
     (- (tay-sum 0 n b x))
     (- (* (expt (- x b) (1+ n))
           (/ (factorial (1+ n)))
           (tay-extra n a x)))))
```

To satisfy that the requirement that $F(a) = F(b)$ in order to invoke Rolle's lemma, it is necessary to show that `(tay-rolle-fn n a x a)` is equal to `(tay-rolle-fn n a x x)`. This is mostly straight-forward, typical of ACL2 efforts.

More interesting is the claim that `(tay-rolle-fn n a x b)` is differentiable with respect to `b`. More precisely, we need to show that `(tay-rolle-fn n a x b)` has a specific derivative. The proof proceeds by using the lemmas about the derivatives of function compositions, e.g., $f + g$, $f \cdot g$, and $f \circ g$.

Consider the term `(expt (- x b) n)`. Its derivative with respect to `b` can be found by applying the chain rule to the functions $f(b) = b^n$ and $g(b) = x - b$:

```
(defun expt-x-b (i b x)
  (expt (- x b) i))

(defun expt-x-b-deriv (i b x)
  (- (* i (expt (- x b) (1- i)))))

(defthm expt-x-b-derivative
  (implies (and (standard-numberp b)
                (realp b)
                (realp b0)
                (i-close b b0)
                (not (= b b0))
                (realp xx)
                (standard-numberp xx)
                (integerp ii)
                (<= 0 ii)
                (<= ii (tay-n)))
           (i-close (/ (- (expt-x-b ii b xx)
                          (expt-x-b ii b0 xx))
                       (- b b0))
                    (expt-x-b-deriv ii b xx)))
  :hints (("Goal"
           :use ((:instance
                  (:functional-instance
                   dc-fn1-o-fn2-derivative ...)
                  (x b)
                  (y b0))))
          ...))
```

Similar uses of the product and sum rules for derivatives yield the derivative of `tay-sum`:

```
(defun tay-term-deriv (i n a x)
  (if (= i 0)
      (tay-fn-deriv (1+ i) n a a)
    (+ (* (tay-fn-deriv (1+ i) n a a)
          (expt (- x a) i)
          (/ (factorial i)))
       (- (* (tay-fn-deriv i n a a)
             (expt (- x a) (1- i))
             (/ (factorial (1- i))))))))

(defun tay-sum-deriv (i n a x)
  (declare (xargs :measure (nfix (1+ (- n i)))))
  (if (and (integerp i)
           (integerp n)
           (<= i n))
      (+ (tay-term-deriv i n a x)
         (tay-sum-deriv (1+ i) n a x))
```

```
    0))

  (defthm tay-sum-derivative
    (implies (and (standard-numberp a)
                  (standard-numberp xx)
                  (tay-domain-p a)
                  (tay-domain-p a0)
                  (i-close a a0)
                  (not (= a a0))
                  (tay-domain-p xx)
                  (integerp ii)
                  (integerp nn)
                  (<= 0 ii)
                  (<= ii nn)
                  (<= nn (tay-n)))
             (i-close (/ (- (tay-sum ii nn a xx)
                            (tay-sum ii nn a0 xx))
                         (- a a0))
                      (tay-sum-deriv ii nn a xx)))
    :hints ...)
```

Notice that the terms in `tay-term-deriv` form a nearly telescopic series, with the case $i = 0$ being the only exception. Hence, the terms in the sum cancel each other out, leaving only the last term:

```
  (defthm tay-sum-deriv->sum-deriv-simplified
     (implies (and (integerp n)
                   (< 0 n))
              (equal (tay-sum-deriv 0 n a x)
                     (* (tay-fn-deriv (1+ n) n a a)
                        (expt (- x a) n)
                        (/ (factorial n)))))
     :instructions ...)
```

The remaining portion of the derivative of `tay-rolle-fn` can be computed as follows:

```
  (defun expt/factorial*extra (n a x b)
    (* (expt (- x b) (1+ n))
       (/ (factorial (1+ n)))
       (tay-extra n a x)))

  (defun expt/factorial*extra-deriv (n a x b)
    (if (= n -1)
        0
      (- (* (expt (- x b) n)
            (/ (factorial n))
            (tay-extra n a x)))))

  (defun tay-rolle-fn-deriv (n a x b)
    (+ (- (tay-sum-deriv 0 n b x))
       (- (expt/factorial*extra-deriv n a x b))))

  (defthm tay-rolle-fn-derivative
     (implies (and (standard-numberp b)
                   (tay-domain-p b)
```

```
                (tay-domain-p b0)
                (i-close b b0)
                (not (= b b0))
                (tay-domain-p aa)
                (standard-numberp aa)
                (tay-domain-p xx)
                (standard-numberp xx)
                (< aa xx)
                (integerp nn)
                (<= 0 nn)
                (<= (1+ nn) (tay-n)))
           (i-close (/ (- (tay-rolle-fn nn aa xx b)
                          (tay-rolle-fn nn aa xx b0))
                       (- b b0))
                    (tay-rolle-fn-deriv nn aa xx b)))
    :hints ...)
```

We are now ready to apply Rolle's lemma to `tay-rolle-fn`. To do so, we need to create the function `tay-rolle-fn-critical-point` which selects the appropriate critical point — i.e., a local maximum or minimum — for `tay-rolle-fn` on the range $[a, x]$. This requires mimicking the appropriate definitions in `derivatives.lisp` and `continuity.lisp`. The first function finds a point in $[a, x]$ where `tay-rolle-fn` achieves its maximum:

```
(defun find-max-tay-rolle-fn-n (nn aa xx a max-x i n eps)
  (declare (xargs :measure (nfix (1+ (- n i)))))
  (if (and (integerp i)
           (integerp n)
           (<= i n)
           (realp a)
           (realp eps)
           (< 0 eps))
      (if (> (tay-rolle-fn nn aa xx (+ a (* i eps)))
             (tay-rolle-fn nn aa xx max-x))
          (find-max-tay-rolle-fn-n nn aa xx a
                                   (+ a (* i eps))
                                   (1+ i) n eps)
        (find-max-tay-rolle-fn-n nn aa xx a max-x
                                 (1+ i) n eps))
    max-x))

  ...

(defun-std find-max-tay-rolle-fn (nn aa xx a b)
  (if (and (realp a)
           (realp b)
           (< a b))
      (standard-part
       (find-max-tay-rolle-fn-n nn aa xx a
                                a
                                0
                                (i-large-integer)
                                (/ (- b a)
                                   (i-large-integer))))
    0))
```

This illustrates the classic way to define an irrational function in ACL2(r): First, an approximation function is defined recursively, and its properties proved with induction, and second the non-standard definitional principle is used to define the irrational function implicitly by giving its values only on *standard* arguments — essentially, as the function to which the approximation functions converge.

The definition of `find-min-tay-rolle-fn` which finds the point where the function `tay-rolle-fn` achieves its minimum is similar. With these two functions, it is possible to find a critical point — i.e., a local minimum or maximum — *inside* the range (a, x) as follows:

```
(defun tay-rolle-fn-critical-point (nn aa xx a b)
  (if (equal (tay-rolle-fn nn aa xx
                           (find-min-tay-rolle-fn
                            nn aa xx a b))
             (tay-rolle-fn nn aa xx
                           (find-max-tay-rolle-fn
                            nn aa xx a b)))
      (/ (+ a b) 2)
    (if (equal (tay-rolle-fn nn aa xx
                             (find-min-tay-rolle-fn
                              nn aa xx a b))
               (tay-rolle-fn nn aa xx a))
        (find-max-tay-rolle-fn nn aa xx a b)
      (find-min-tay-rolle-fn nn aa xx a b))))
```

And it is now possible to apply Rolle's lemma to `tay-rolle-fn`:

```
(defthm tay-rolle-fn-rolles-theorem
  (implies (and (tay-domain-p a)
                (tay-domain-p b)
                (realp nn)
                (realp aa)
                (realp xx)
                (= (tay-rolle-fn nn aa xx a)
                   (tay-rolle-fn nn aa xx b))
                (< a b)
                (tay-domain-p aa)
                (tay-domain-p xx)
                (< aa xx)
                (integerp nn)
                (<= 0 nn)
                (<= (1+ nn) (tay-n)))
           (and (tay-domain-p (tay-rolle-fn-critical-point
                               nn aa xx a b))
                (< a (tay-rolle-fn-critical-point
                      nn aa xx a b))
                (< (tay-rolle-fn-critical-point
                    nn aa xx a b)
                   b)
                (equal (tay-rolle-fn-deriv
                        nn aa xx
                        (tay-rolle-fn-critical-point
                         nn aa xx a b))
                       0)))
  :hints ...)
```

This expression can be simplified considerably by removing the "extra" variables nn, aa, and xx, which have served their purpose. The result is a more immediate translation of Rolle's lemma:

```
(defun tay-error-point (n a x)
  (tay-rolle-fn-critical-point n a x a x))

(defthm tay-rolle-fn-rolles-theorem-corollary
  (implies (and (tay-domain-p a)
                (tay-domain-p x)
                (< a x)
                (integerp n)
                (<= 0 n)
                (<= (1+ n) (tay-n)))
           (and (tay-domain-p (tay-error-point n a x))
                (< a (tay-error-point n a x))
                (< (tay-error-point n a x) x)
                (equal (tay-rolle-fn-deriv
                        n a x (tay-error-point n a x))
                       0)))
  :hints ...)
```

What remains is simply to solve the equation

```
(equal (tay-rolle-fn-deriv n a x (tay-error-point n a x))
       0)
```

for `(tay-fn x)`. This involves only algebraic manipulations:

```
(defthm taylor-series-with-remainder
  (implies (and (tay-domain-p a)
                (tay-domain-p x)
                (< a x)
                (integerp n)
                (< 0 n)
                (<= (1+ n) (tay-n)))
           (and (tay-domain-p (tay-error-point n a x))
                (< a (tay-error-point n a x))
                (< (tay-error-point n a x) x)
                (equal (tay-fn 0 n a x)
                       (+ (tay-sum 0 n a x)
                          (* (tay-fn-deriv
                              (1+ n) n
                              (tay-error-point n a x)
                              (tay-error-point n a x))
                             (expt (- x a) (1+ n))
                             (/ (factorial (1+ n)))))))))
  :hints ...)
```

That is, we have shown that

$$f(x) = f(a) + \sum_{i=1}^{n} \frac{f^{(i)}(a) \cdot (x-a)^i}{i!} + \frac{f^{(n+1)}(\beta)}{(n+1)!}(x-a)^{n+1},$$

for some $\beta \in (a, b)$.

6 Conclusion

Taylor's formula with remainder, as proved in 5, is sufficient to find approximations to analytic functions whose derivatives at a point are known. Such was the case in Sawada's motivating application, where he provided an elegant abstraction of the derivatives of $\sqrt{x}$ [17]. We hope that our work here will assist that verification effort.

The result can be extended in two obvious directions. First of all, as it stands the approximation can only be used to find the values of the function f for x greater than a. This is typical of the way the proof is presented in analysis textbooks, with an appeal to symmetry for the remainder of the proof. With a little bit of work, this restriction can be removed from the theorem presented. Another direction will be to consider the infinite Taylor series. We expect to have these results soon.

However, the methods used in the proof show some unfortunate limitations of ACL2(r). Having to add "extra" variables to encapsulated functions is particularly distasteful and cumbersome. This suggests that the rules for dealing with non-classical constraints be relaxed somewhat. Whether that means allowing non-classical functions as functional instances, considering only *standard* instances of variables in pseudo-lambda expressions, or an entirely different approach is currently being investigated.

References

[1] F. Diener and M. Diener, editors. *Nonstandard Analysis in Practice*. Springer, 1995.

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

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

[4] R. Gamboa. Continuity and differentiability in ACL2. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 18. Kluwer Academic Press, 2000.

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

[6] 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.

[7] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[8] M. Kaufmann and J S. Moore. The ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[9] M. Kaufmann and J S. Moore. Design goals for ACL2. Technical Report 101, Computational Logic, Inc., 1994. See URL `http://www.cs.utexas.edu/users/-moore/publications/acl2-paper s.html#Overviews`.

[10] H. J. Keisler. *Elementary Calculus*. Prindle, Weber and Schmidt, Boston, 1976.

[11] H. J. Keisler. *Foundations of Infinitesimal Calculus*. Prindle, Weber and Schmidt, Boston, 1976.

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

[13] E. Nelson. Internal set theory: A new approach to nonstandard analysis. *Bulletin of the American Mathematical Society*, 83:1165–1198, 1977.

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

[15] A. Robinson. *Non-Standard Analysis*. Princeton University Press, 1996.

[16] D. Russinoff and A. Flatau. RTL verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 13. Kluwer Academic Press, 2000.

[17] 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.