# Automatic Differentiation in ACL2

Peter Reid[1] and Ruben Gamboa[2]

[1] University of Oklahoma, Norman, OK, USA,
peter.d.reid@gmail.com
http://www.cs.ou.edu
[2] University of Wyoming, Laramie, WY, USA
ruben@uwyo.edu
http://www.cs.uwyo.edu/~ruben

**Abstract.** In this paper, we describe recent improvements to the theory of differentiation that is formalized in ACL2(r). First, we show how the normal rules for the differentiation of composite functions can be introduced in ACL2(r). More important, we show how the application of these rules can be largely automated, so that ACL2(r) can automatically define the derivative of a function that is built from functions whose derivatives are already known. Second, we show a formalization in ACL2(r) of the derivatives of familiar functions from calculus, such as the exponential, logarithmic, power, and trigonometric functions. These results serve as the starting point for the automatic differentiation tool described above. Third, we describe how users can add new functions and their derivatives, to improve the capabilities of the automatic differentiator. In particular, we show how to introduce the derivative of the hyperbolic trigonometric functions. Finally, we give some brief highlights concerning the implementation details of the automatic differentiator.

**Key words:** ACL2, nonstandard analysis, automatic differentiation

## 1 Introduction

ACL2(r) is a variant of the theorem prover ACL2 that offers support for reasoning about the irrational real and complex numbers via nonstandard analysis [8]. Since its logic is strictly first order and the theorem prover has only limited support for quantifiers, ACL2 would not appear to be a good candidate for reasoning about real analysis. However, by introducing key concepts from nonstandard analysis, such as "classical," "standard part," and the transfer principle, ACL2(r) extends ACL2 just enough to take advantage of its strong support for induction, which serves a key role in arguments using nonstandard analysis.

As a result, many results from real analysis have been formalized in ACL2, including the fundamental theorem of calculus [10] and several results having to do with differentiability [6, 7]. However, much of this work is foundational in nature, while the intended use of ACL2(r) is to support reasoning about real-world software whose correctness relies on facts from basic engineering mathematics.

In this paper, we describe recent results that greatly expand the usefulness of ACL2(r) when reasoning about functions and their derivatives. In Sect. 3, we present a new ACL2(r) "event" that lets the user introduce a function that is the derivative of an old function. For example, the derivative of $\sqrt{1+x^2}$ can be introduced with the definition

```
(defderivative sqrt-1+x**2-derivative
  (acl2-sqrt (+ 1 (* x x))))
```

The event `derivative` symbolically differentiates the given expression and defines the corresponding function. It also proves the theorems that assert that the new function is indeed the derivative of the old one. The implementation of `defderivative` relies on metatheorems formalizing the familiar algebraic differentiation rules, such as $(f+g)'(x) = f'(x) + g'(x)$. This is similar to the approach used in [7], but with one key difference. The proof obligations required by the metatheorems in [7] are too unwieldy to be automated successfully. In Sect. 4, we present a different formalization that is much easier to automate when the derivative is known, as is the case when it is discovered using the algebraic differentiation rules. Of course, algebraic differentiation rules are of little use without a priori knowledge of *some* derivatives, i.e., a database of known derivatives. In Sect. 5, we show how the derivatives of many useful functions from calculus are formalized in ACL2(r). In particular, the exponential function had been defined in ACL2(r) since it was first developed, but its derivative was never determined. We report in this paper our recent formalization of this result in ACL2(r). Moreover, we use this result to find the derivatives of other functions, including the trigonometric functions. Finally, in Sect. 6, we show how a user can extend the database of known derivatives by proving a derivative fact, perhaps from first principles. In particular, we show how the user can introduce the hyperbolic trigonometric functions and their derivatives.

## 2 Related Work

Finding the derivative of functions is a task that has many applications, such as optimization and sensitivity analysis. Consequently, many researchers have tackled the problem of automatically finding the derivative of a function expressed as a computer program. In fact, automatic differentiation (AD) is an established research area [1, 4, 9, 5].

The approach used in AD is to compute the derivative of a program by examining the program statically. That is, the program's source code is transformed so that it can compute not only the original function, but also its derivative. This can be done either by using overloaded operators (in languages that support them), or by using preprocessing techniques to produce a new function. Naturally, this means that most solutions are program-specific, e.g., ADIC for C programs [3] and ADIFOR for FORTRAN programs [2], which use similar ideas but with different implementations.

Our interest is in finding the derivatives of functions expressed as programs in Common LISP. To that extent, our work is related to that in [11]. However, our primary interest is in automatically finding the *proof* that the derivative is correct, not just in finding the derivative. The techniques described in [11] go far beyond the work described in this paper as far as automatic differentiation, e.g., handing general derivatives of multivariate functions $f : R^n \to R^m$. But the emphasis there is in programming, not proving formal correctness using an automated theorem prover.

In spirit, our work has more in common with [12]. There, the concept of proof-carrying codes is applied to the AD transformations. The result is that the AD tool can produce a certificate that can be verified by a formal tool, thus establishing that the transformed function correctly computes the derivative of the input function. Our approach is quite different from that in [12] in that we are working with functional programs written in Common LISP, instead of abstract programs in a Hoare-style WHILE language.

## 3 Introducing the Defderivative Event

We begin our presentation by showing how `defderivative` looks to the end user. Consider the expression $\sqrt{1+x^2}$, and suppose that the user wants to introduce its derivative in ACL2(r). This is trivial to do with `defderivative`:

```
(defderivative sqrt-1+x**2-derivative
  (acl2-sqrt (+ 1 (* x x))))
```

`Defderivative` introduces the function `sqrt-1+x**2-derivative`, with a definition that is, of course, equivalent to $x/\sqrt{1+x^2}$. This definition is computed automatically using symbolic differentiation. `Defderivative` also introduces the theorem `sqrt-1+x**2-derivative-proof-obligation` that shows that this function is, in fact, the derivative of $\sqrt{1+x^2}$. This theorem is equivalent to the following ACL2(r) statement:

```
(defthm sqrt-1+x**2-derivative-proof-obligation
  (implies (and (acl2-numberp x)
                (realp (+ 1 (* x x)))
                (< 0 (+ 1 (* x x)))
                (acl2-numberp y)
                (realp (+ 1 (* y y)))
                (< 0 (+ 1 (* y y)))
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sqrt (+ 1 (* x x)))
                          (acl2-sqrt (+ 1 (* y y))))
                       (- x y))
                    (* (/ 1/2 (acl2-sqrt (+ 1 (* x x))))
                       (+ 0 (+ (* x 1) (* x 1)))))))
```

The hypotheses in the theorem are formed by combining the hypotheses required by each of the various composition rules applied during symbolic differentiation. It is evident that this combination is "blind," as many of the hypotheses are trivially true. The expression that defines the derivative is also raw. I.e., it is formed by blindly following of the composition rules. This is why we used the phrase "equivalent to" above, when referring to the definition of `sqrt-1+x**2-derivative`. We have experimented with using ACL2's rewriter to simplify the body of the derivative, but we have found that the simplified (according to ACL2) form rarely corresponds to the user's expectation. For example, ACL2(r) simplifies the term above to the following

```
(+ (* 1/2 x (/ (acl2-sqrt (+ 1 (* x x)))))
   (* 1/2 x (/ (acl2-sqrt (+ 1 (* x x))))))
```

So we have found it better in practice to leave the formula discovered by automatic differentiation as is, and let the user provide a simpler definition, if she wishes. Typically, ACL2(r) can prove that these definitions are equivalent, so the function defined by the user is also shown to be the derivative of the original function. In this way, the user can choose the definition used, but avoid the tedious steps required to prove that it is the actual derivative. It is important to note that it is usually much easier to prove that these two functions are equal than to show that they are the derivative of the original function! For example, ACL2(r) can prove the following with completely automatically:

```
(equal (* (/ 1/2 (acl2-sqrt (+ 1 (* x x))))
          (+ 0 (+ (* x 1) (* x 1))))
       (/ x (acl2-sqrt (+ 1 (* x x)))))
```

In turn, that makes it trivial to simplify the derivative of $\sqrt{1+x^2}$, so that it matches the user's expectations:

```
(defthm sqrt-1+x**2-derivative-clean
  (implies (and (realp x)
                (realp y)
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sqrt (+ 1 (* x x)))
                          (acl2-sqrt (+ 1 (* y y))))
                       (- x y))
                    (/ x (acl2-sqrt (+ 1 (* x x))))))
  :hints (("Goal" :use (:instance
                        sqrt-1+x**2-derivative-proof-obligation))))
```

Notice that we have simplified not only the formula for the derivative, but also the hypotheses.

# 4 The Implementation of Defderivative

## 4.1 Finding the Derivative

`Defderivative` can differentiate a function that is defined according to the following forms, where the derivative is taken with respect to the variable $x$:

- The identity function, i.e., $x$.
- A constant. This can take be a literal number, a variable other than $x$, or a function of zero arguments.
- Addition, i.e., $f(x) + g(x)$.
- Multiplication, i.e., $f(x) \times g(x)$.
- Composition, i.e., $f(g(x))$.
- Functional inverse, i.e., $f^{-1}(x)$.

In these forms, $f$ and $g$ are either functions whose derivatives have been previously defined, or formulas that `defderivative` can derive recursively.

The list of forms does not include subtraction or division. This is because ACL2 defines these operations by using the corresponding inverses. So $f - g$ is really handled as $f + (-g)$, and we treat $(-g)$ as the composition of the functions unary minus and $g$. Once the derivatives of unary minus and unary division (i.e., the multiplicative inverse) are known, `defderivative` handles subtraction and division through the functional composition rule.

As `defderivative` computes the derivative of functions defined using any of the given forms, is also proves the theorems that establish that the new expression is the derivative of the given function. We refer to these theorems as the *derivative theorems*. The most important of these theorems relates the function's differential between two i-close points to its derivative, which captures the non-standard notion of derivative. Letting `F` be the function, `F-PRIME` its derivative as found using symbolic differentiation, and `DOMAIN-P` the domain over which `F` is defined and is differentiable, this theorem takes the following form:

```
(implies (and (DOMAIN-P x)
              (DOMAIN-P y)
              (standardp x)
              (i-close x y)
              (not (equal x y)))
         (i-close (/ (- (F x) (F y))
                     (- x y))
                  (F-PRIME x)))
```

The remaining six derivative theorems play supporting roles.

Readers familiar with ACL2(r) may notice that the formal statement of differentiability given above differs from the one used in prior formalizations. There are two important differences:

- In earlier work, we separated the notions of derivative and differentiability. So the definition of differentiability was stated entirely in terms of `F` and not

`F-PRIME`. The notions are equivalent, of course, but in the context of this work, the formal statement above is much more convenient, since we already have `F-PRIME`.
- The predicate `DOMAIN-P` describes the domain over which `F` is differentiable. In earlier work, we used intervals to define this domain. This has the advantage that we can quantify over intervals in a first-order logic, like ACL2's. However, treating this domain as a function makes it easier to automate the process of defining the appropriate domain over which the algebraic differentiation rules are applicable, e.g., such as $f(x) \neq 0$.

For functions defined according to the forms described above, a calculus text would prescribe applying differentiation rules such as the sum rule, the product rule, and the chain rule. Each of these rules expresses the derivative of the whole ($f$ and $g$ composed) in terms of the derivative of its parts ($f$ and $g$ individually). I.e, these correspond to theorems involving general functions—higher order logic. A first-order logic, ACL2(r) does not deal in higher-order logic directly, but theorems such as these can be proved using ACL2(r)'s `encapsulate` feature. An encapsulate invocation lists function signatures followed by assumptions that describe how those functions behave. These assumptions are referred to as constraints. Proofs about the encapsulated functions can be constructed using the encapsulated assumptions. Finally, concrete functions can be substituted into those encapsulated function signatures in whatever proofs were constructed, as long as the encapsulated assumptions can be shown to hold given the same assumptions, which become proof obligations from the perspective of the concrete function.

The algebraic differentiation rules are encapsulated as follows. The encapsulated functions are

- $f$, its derivative, and its domain;
- $g$, its derivative, and its domain; and
- the composed function (e.g., $f + g$), its derivative, and its domain.

The constraints in the encapsulate are as follows:

- $f$ satisfies the derivative theorems.
- $g$ satisfies the derivative theorems.
- The composed function is related to $f$ and $g$ in some way. For example, the sum rule is encapsulated using the constraint

```
(equal (f+g x)
       (+ (f x) (g x))
```

- The derivative of the composed function is related to $f$, $g$, and their derivatives in some way. For example, in the sum rule, the constraint has the form

```
(equal (f+g-prime x)
       (+ (f-prime x) (g-prime x)))
```

– The composed function is "type-safe." I.e., when the composed function (e.g., $f+g$) is evaluated on a number in its domain, its value depends on the value of the functions $f$ and $g$ applied to numbers on their respective domains.

Using these assumptions, the composition books proceed to prove the derivative theorems about the composed function. When the derivative of a specific composition needs to be proved, these theorems can be instantiated with the specific functions $f$ and $g$.

This work is similar to the composition rules presented in [7]. In fact, originally we tried to use the compositions theorems from [7], but we discovered that these were not amenable to automation. One problem was that the existing theorems allowed differentiation over a single interval. But that made it impossible to reason automatically about the derivative of tangent, for example. Another problem was in ease of application. The composition theorems in [7] never state the derivative except as the standard part of a small differential. This introduces complexity, since that small differential needs to be shown to behave as a derivative should. The new composition theorems take an expression for the derivative explicitly, which greatly simplifies their proofs. This comes at virtually no cost to `defderivative`, since it already computes expressions for the derivative.

### 4.2 Proof Structure

Composition rules are useful, but putting them together to verify the derivative of a complicated function can be prohibitively tedious. Each function application in the expression being differentiated requires several dozen lines of carefully written theorems to instantiate the appropriate compositions, adding up to hundreds of lines of proof for a typical expression. The root cause of this fact is that ACL2 has little support for higher-order functions and requires that virtually every step of a higher-order proof be explicitly pointed out to it. Fortunately, macros provide a way out. `Defderivative` composes the theorem code that the user otherwise would have had to and submits it to ACL2, making differentiation take a few lines rather than a few hundred. At the heart of the system is a function, named `differentiate-fn`, which we have added to the theorem prover. Its signature is (roughly) as follows: Inputs:

1. Function expression. For example, this could be `(acl2-sqrt (+ 1 (* x x)))`.
2. Derivative name. This is the name of derivative function, and is also serves as a prefix for the derivative theorems.

Outputs:

1. A proof of the derivative theorems.
2. The function's derivative.
3. The function's domain.

Recall that there are several forms that a differentiable expression can take. There is a branch for each case in `differentiate-fn`. The first two cases, where

the expression to differentiate is $x$ or a fixed number, are relatively trivial to implement. `Differentiate-fn` simply returns a canned proof of the appropriate theorems, renamed according to the prefix requested, along with a canned derivative (1 or 0, respectively) and domain. The other cases are more interesting. The proofs they return take the following form.

1. Prove (recursively) the derivative theorems about $f$.
2. Prove (recursively) the derivative theorems about $g$.
3. Disable all theorems, except the derivative theorems of $f$ and $g$. This allows us to limit ACL2's proof search, so that we can automate the rest of the proof.
4. Instantiate the appropriate algebraic differentiation theorems to prove the derivative theorems of the composite function.
5. These derivative theorems are the only ones introduced by `differentiate-fn`.

In short, these proofs recursively verify the derivative of the two functions being composed and then combine those proofs by instantiating some of the composition proofs discussed in section 4.

The derivative of a function is automatically recognized if the function has been registered with `defderivative`. For a function to be registered, `defderivative` must be informed of that function's derivative and its domain and provided with the seven proofs showing its correctness. Using `defderivative` to differentiate a function automatically registers it. In the following sections, we will describe how the original set of functions are registered, and how the user can register new functions.

We conclude this section with a simple example that shows `defderivative` in action. Imagine that `acl2-sqrt` has been registered and `defderivative` is then asked to differentiate `(acl2-sqrt (+ x 3))`. First, `defderivative` will use the proofs, provided on registration, concerning `acl2-sqrt`, its derivative, and its domain to fill in the first recursive section of the proof structure; this is simply a matter of renaming those proofs. Second, `defderivative` will recursively differentiate `(+ x 3)`. This will use the differentiation rules for sum, with $f(x) = x$ and $g(x) = 3$, and these functions will be differentiated recursively. Of course, their derivatives are trivial to compute, so `defderivative` combines them to find the derivative of `(+ x 3)`. Finally `defderivative` will use the theorems about the derivative of $f \circ g(x)$, using $f = \sqrt{x}$ and $g = x + 3$.

## 5 The Path to Elementary Functions

In this section, we will describe how we have seeded `defderivative` with the derivatives of several functions from elementary calculus. This list includes $x^n$, $e^x$, the natural logarithm, $\sqrt{x}$, sine, cosine, arcsine, arccosine, and arctangent. Other functions, such as tangent, can be derived automatically with `defderivative`, since they are defined using elementary operations over the built-in functions, e.g., $\tan(x) = \sin(x)/\cos(x)$. The proof effort required to establish the derivatives of these functions was significant. Fig. 1 shows how the proofs are based on one another.

In tackling these proofs, we used three different approaches. The first was proving the derivative from first principles, i.e., algebraic manipulation of the differential into an expression that approaches the derivative as the difference becomes small. The second approach was using earlier, simpler proofs to bootstrap later ones. For example, because ACL2(r) defines sine and cosine in terms of exponentials, one can use `defderivative` to differentiate sine's definition and then show that the derivative is cosine. Proofs with these approach tended to be trivial. The third approach was to use `defderivative` to differentiate functions that are defined as the inverse of a differentiable function, e.g., $\ln(x)$.

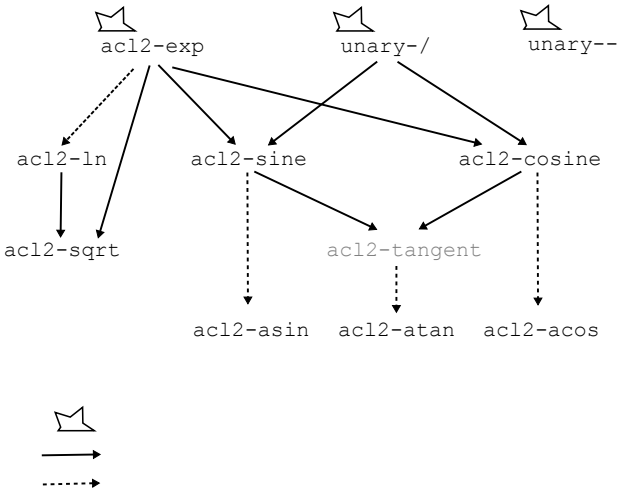


**Fig. 1.** Dependency graph of the functions built into `defderivative`. Symbols leading into a function represent how its derivative theorems were proved.

As the figure suggests, the most difficult proof was the derivative of $e^x$. In some settings, this is a trivial result. For example, some calculus books show that the derivative of $a^x$ is proportional to $a^x$, then define $e$ as the unique real number such that the proportion is equal to 1. Others start by defining $e^x$ using its Taylor expansion, then observe that this infinite polynomial is its own derivative. But neither of this options were open to us. The function $e^x$ is defined in ACL2(r) indirectly, using partial Taylor sums and the nonstandard transfer principle. Moreover, $a^x$ is defined in terms of $e^x$. So we had to follow a more direct approach.

To find the proof, we examined the value of the differential, $\frac{e^{x+\Delta x}-e^x}{\Delta x}$. Using the law of exponents, this reduces to $e^x \frac{e^{\Delta x}-1}{\Delta x}$. When $x$ is standard, so is $e^x$, so it is sufficient to show that $\frac{e^{\Delta x}-1}{\Delta x} \approx 1$, i.e., is close to 1.

Proving that lemma was the biggest challenge. First, we defined $f(x) = \sum_{k=0}^{N} \frac{x^k}{(k+2)!}$. Then we showed that this series converged. That is, we shoed that the partial sums are limited whenever $x$ is limited, by comparing the partial sums with the Taylor expansion of $e^x$, which we had already shown converges. Since $f$ converges, we can use the transfer principle to define the function $g(x) = {}^*f(x)$, the unique standard function that $f$ converges to pointwise. It follows from the transfer principle and the definitions of $g$ and $e^x$ that $\frac{e^{\Delta x}-1}{\Delta x} = 1 + \Delta x \cdot g(\Delta x)$. Finally, we show that whenever $\Delta x \leq 1$, $||g(\Delta x)|| \leq ||g(1)||$ and therefore limited. So when $\Delta x$ is infinitesimal, so is $\Delta x \cdot g(\Delta x)$, and it follows that $\frac{e^{\Delta x}-1}{\Delta x} = 1 + \Delta x \cdot g(\Delta x) \approx 1$.

## 6 Adding New Derivative Facts: Hyperbolic Trigonometric Functions

In this section, we show how a user can differentiate expressions involving a function that is not among those already registered with `defderivative`. To register a new function with `defderivative`, there are essentially two steps. First, the derivative theorems must be proved about that function. Second, `defderivative` must be informed of the new function, its derivative, its domain, and the associated proofs through a call to `def-elem-derivative`. This section provides an example of going through that process.

This example, in which we make `defderivative` able to differentiate hyperbolic sine and cosine, uses a bootstrapping approach. The hyperbolic functions are defined in terms of exponential functions, which `defderivative` already knows how to differentiate. The strategy will be to use `defderivative`'s existing capability to differentiate hyperbolic sine's definition and then to associate hyperbolic sine itself with the resulting derivative.

Hyperbolic sine and cosine and their derivatives are defined as follows:

$$sinh(x) = \frac{e^x - e^{-x}}{2} \qquad \frac{d}{dx} sinh(x) = cosh(x)$$
$$cosh(x) = \frac{e^x + e^{-x}}{2} \qquad \frac{d}{dx} cosh(x) = sinh(x)$$

These definitions are trivial to enter in ACL2(r).

```
(defun acl2-sinh (x)
  (/ (- (acl2-exp x) (acl2-exp (- x)))
     2))

(defun acl2-cosh (x)
  (/ (+ (acl2-exp x) (acl2-exp (- x)))
     2))
```

Next, we use `defderivative` to find the derivative of the body of `acl2-sinh`. Normally, we would differentiate `acl2-sinh` directly (instead of differentiating

its body), but we do not do so here, since `defderivative` would automatically register `acl2-sinh`!

```
(defderivative acl2-sinh-lemma
  (/ (- (acl2-exp x) (acl2-exp (- x)))
     2))
```

As expected, this results in an unsimplified domain and derivative. However, we can simplify it and introduce the derivate of `acl2-sinh` with the following theorem:

```
(defthm acl2-sinh-derivative
  (implies (and (acl2-numberp x)
                (acl2-numberp y)
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (acl2-sinh x)
                          (acl2-sinh y))
                       (- X Y))
                    (acl2-cosh x)))
  :hints (("Goal" :use (:instance acl2-sinh-lemma))))
```

That is the most difficult of the proof obligations that the user must prove before she can register the derivative of hyperbolic sine. The other obligations concern the remaining derivative theorems, and those are far simpler, such as showing that values in the domain of hyperbolic sine do not requires call outside the domain of $e^x$. Once these obligations are established, the user can register the derivative with the following event:

```
(def-elem-derivative
  acl2-sinh              # function to differentiate
  elem-acl2-sinh         # prefix of theorems' name
  (acl2-numberp x)       # domain
  (acl2-cosh x))         # derivative
```

## 7 Conclusions

In this paper, we described the macro `defderivative` and its implementation. This macro symbolically differentiates ACL2(r) expressions involving functions whose derivatives have been established previously, e.g., built-in functions, functions derived using `defderivative`, and functions registered by the user. The macro also computes the appropriate domain for the function and proves the required derivative theorems.

In the process of implementing `defderivative`, we identified some impediments to automation in our previous treatment of algebraic differentiation rules,

and we addressed those shortcomings as part of this project. The resulting framework is much easier to use, hence more widely applicable. For example, the previous work was foundational, allowing one to prove (often tediously) when one function was the derivative of another. There were very few practical results. While the derivative of $x^n$ was formalized in ACL2(r), that of $e^x$ was not, nor were those of the trigonometric functions. We proved the derivative of $e^x$ using techniques similar to the ones used previously with ACL2(r), but the remaining derivatives were derived automatically.

The macro `defderivative` can be readily extended to compute partial derivatives. However, the treatment of differentiation in ACL2(r) is derived from nonstandard analysis, and this imposes technical restrictions on the treatment of free variables. The result is that we must anticipate the number of variables that will be required. Thus far, we have implemented partial derivatives for functions of two variables, such as `expt`, which can represent either $a^x$ or $x^n$, depending on which variable is fixed. To generalize this to functions of three or more variables, it will be more convenient to use a classical notion of derivative, i.e., one based on limits instead of standard part. We are currently working on a proof of the equivalence of these notions in ACL2(r). However, the proof is quite challenging, because it involves quantifiers and infinite sets, neither of which is supported well by ACL2(r).

## References

1. Community portal for automatic differentiation. `http://www.autodiff.org`.
2. Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from fortran programs. *Scientific Programming*, (1), 1991.
3. Christian Bischof, Lucas Roh, and Andrew Mauer-oats. ADIC: An extensible automatic differentiation tool for ANSI-C. 27:1427–1456, 1997.
4. Christian H. Bischof, Paul D. Hovland, and Boyana Norris. On the implementation of automatic differentiation tools. *Higher Order Symbol. Comput.*, 21:311–331, September 2008.
5. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Computer and Information Science. Springer, 2001. Selected papers from the AD2000 conference, Nice, France, June 2000.
6. 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.
7. R. Gamboa and J. Cowles. The chain rule and friends in ACL2(r). In *Proceedings of the Eighth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2009)*, 2009.
8. R. Gamboa and M. Kaufmann. Nonstandard analysis in ACL2. *Journal of Automated Reasoning*, 27(4):323–351, November 2001.
9. A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Other Titles in Applied Mathematics. SIAM, 2008.