Implementing an Automatic Differentiator in ACL2

Peter Reid
University of Oklahoma
Norman, Oklahoma, USA

peter.d.reid@gmail.com

Ruben Gamboa
Department of Computer Science
University of Wyoming
Laramie, Wyoming, USA

ruben@cs.uwyo.edu

The foundational theory of differentiation was developed as part of the original release of ACL2(r). In work reported at the last ACL2 Workshop, we presented theorems justifying the usual differentiation rules, including the chain rule and the derivative of inverse functions. However, the process of applying these theorems to formalize the derivative of a particular function is completely manual. More recently, we developed a macro and supporting functions that can automate this process. This macro uses the ACL2 table facility to keep track of functions and their derivatives, and it also interacts with the macro that introduces inverse functions in ACL2(r), so that their derivatives can also be automated. In this paper, we present the implementation of this macro and related functions.

1 Introduction

This paper describes the implementation of an automatic differentiator (AD) [RG11] that can find and prove the derivative of algebraic expressions in ACL2(r). The tool is accessed through the macros `defderivative` and `derivative-hyps`. We will describe these macros more fully later, but for now, we introduce them with an example.

Suppose we have defined the function `square` that computes $x \cdot x$. The following event determines the derivative of the function `square` and proves the associated theorems:

```
(defderivative square-deriv-local (square x))
```

The key theorem that the macro `defderivative` introduces is as follows:

```
(defthm square-deriv-local
  (implies (and (acl2-numberp x)
                (acl2-numberp y)
                (standardp x)
                (i-close x y)
                (not (equal x y)))
           (i-close (/ (- (square x)
                          (square y))
                       (- x y))
                    (+ (* x 1) (* x 1))))
  :hints ...)
```

The conclusion of this theorem states that the derivative of square is $x \cdot 1 + x \cdot 1$, which of course simplifies to the expected value of $2x$. The macro does not automatically perform such simplifications, but

Hardin and Schmaltz (Eds): ACL2 2011
EPTCS 70, 2011, pp. 61–69, doi:10.4204/EPTCS.70.5

the user can easily provide the preferred form for the derivative function[1]. For example, the function `square-prime` can be defined as `(* 2 x)`, and the derivative theorems for it can be proved using the macro `derivative-hyps`:

```
(derivative-hyps square
 :close-hints
   (("Goal"
     :use ((:instance square-deriv-local))
     :in-theory (disable square-deriv-local))))
```

The macro `derivative-hyps` introduces all the theorems that establish that the function `square-prime` is in fact the derivative of `square`. The keyword arguments allow the user to provide hints for some of the necessary theorems. In this case, it is necessary to explicitly invoke the theorem `square-deriv-local`, which was previously introduced via `defderivative`.

The rest of this paper describes the implementation of these macros. Section 2 describes some differences in the theory of differentiation that proved useful in developing the AD macros. In particular, the proofs of the algebraic composition theorems differ from the ones described in [GC09] to take advantage of the fact the derivative is known in the current context. Section 3 describes the capabilities and limitations of our AD system. This is followed in Section 4 with a full description of their implementation. Finally, Section 5 describes future enhancements to these macros.

2 The Revised Story of Differentiation in ACL2(r)

The theory of differentiation that was developed in [Gam00] and [GC09] is strictly foundational. For one thing, the development is concerned more with differentiability than with derivatives. For example, the theorem that describes the derivative of sums is stated informally as follows: If f and g are differentiable functions, so is $f+g$. Notice that no mention is made of the derivatives of f, g, or $f+g$! Instead, the theorems deal directly with expressions corresponding to the differential of the functions, e.g., $\Delta f(x)/\Delta x = (f(x+\varepsilon)-f(x))/\varepsilon$.

Using principles from non-standard analysis [Rob96, Rob88], these derivatives can be introduced implicitly by taking standard parts. That is, $f'(x) = {}^*(\Delta f(x)/\Delta x)$. However, this definition only works when x is standard. It can be generalized using `defthm-std`, but this process is unsatisfactory because the relationship between f', the expected derivative of the function f, and the standard part of $\Delta f/\Delta x$ is obscured. This may explain why previous results included many abstract theorems about differentiable functions, but few concrete derivatives. For instance, the derivatives of the trigonometric, exponential, and logarithmic functions had not been proven in ACL2(r) before this project.

A more significant challenge is the use of intervals in the formalization of differentiation. Intervals were used to define the domain (and sometimes range) of differentiable functions. This corresponds to typical mathematical statements. For example, one of the hypotheses of the mean-value theorem (MVT) is that f is continuous over an interval $[a,b]$ and differentiable over (a,b), and a user can select suitable values of a and b when the theorem is applied manually. But this is harder to do when the theorems are applied automatically, if for no other reason than the domain of some functions (such as $f(x) = 1/x$) is not a simple interval.

[1] As we write this, we are working on a version of the macro that lets the user provide this function when the macro is introduced. This will simplify the process described here.

Consequently, we developed new versions of the differentiability criterion that make explicit use of the differentiable function f and its derivative f'. We also redeveloped versions of the composition theorems, namely

- $(f+g)'(x) = f'(x)+g'(x)$.
- $(f \cdot g)'(x) = f(x)g'(x)+f'(x)g(x)$.
- $(f \circ g)'(x) = f'(g(x))g'(x)$.
- $(f^{-1})'(x) = 1/f'(f^{-1}(x))$, where f^{-1} is the (compositional) inverse of f.

These new versions represent the domain (and sometimes range) of f explicit via functions instead of intervals or some other data structure. The association between the function f, its derivative f' and its domain is kept using a naming convention; i.e., `f`, `f-prime`, and `f-domain-p`.

Using these new formalizations and the associated naming conventions was the key to automating the application of the algebraic composition rules first formalized in [GC09]. However, the new formalization does have some drawbacks. First, there is no guarantee that the domains used are, or even contain non-trivial intervals. This means, for example, that the derivative of $|x|$ could be vacuously formalized on the domain $x \in \{0\}$. More seriously, however, it prevents the application of the more foundational theorems established earlier, such as the MVT. We are investigating ways to bridge our current work with prior results to remedy this issue.

The final, significant challenge is that some of the results obtained previously were proven in contexts that turned out to be too restrictive. Specifically, important theorems, such as the chain rule, which is used repeatedly during automatic differentiation, was developed only for real-valued functions. However, the trigonometric functions in ACL2(r), such as sine and cosine, are defined in terms of the complex exponential function. For example, $\sin(x) \equiv (e^{ix} - e^{-ix})/2i$. So we developed a new formalization of the chain rule, which works for complex numbers.

3 The Automatic Differentiation Macros

Previously, we discussed how the macros `defderivative` and `derivative-hyps` to introduce the derivative of a function. In this section, we will explore these and other related macros more fully.

The macro `derivative-hyps` is used to generate automatically the theorems required to show that the function `f-prime` is the derivative of `f`. The theorems and the associated naming conventions are as follows:

- **f-number**

```
(implies (f-domain-p x)
         (acl2-numberp (f x)))
```

- **f-standard**

```
(implies (and (standardp x)
              (f-domain-p x))
         (standardp (f x)))
```

- **f-continuous**

```
(implies (and (f-domain-p x)
              (standardp x)
              (f-domain-p y)
              (i-close x y))
         (i-close (f x) (f y)))
```

- **f-prime-number**

```
(implies (f-domain-p x)
         (acl2-numberp (f-prime x)))
```

- **f-prime-standard**

```
(implies (and (standardp x)
              (f-domain-p x))
         (standardp (f-prime x)))
```

- **f-prime-continuous**

```
(implies (and (f-domain-p x)
              (standardp x)
              (f-domain-p y)
              (i-close x y))
         (i-close (f-prime x) (f-prime y)))
```

- **f-close**

```
(implies (and (f-domain-p x)
              (standardp x)
              (f-domain-p y)
              (i-close x y)
              (not (equal x y)))
         (i-close (/ (- (f x) (f y))
                     (- x y))
                  (f-prime x)))
```

These theorems are precisely the ones that will be used to establish the constraints of `encapsulates` that are used to encode the composition theorems. The names of the theorems are important, because the macros will generate hints with those names.

Similarly, the macro `inverse-hyps` generates the theorems that establish that the function `f-inverse` is the compositional inverse of `f`.

- **f-inverse-in-range**

```
(implies (f-inverse-domain-p x)
         (f-domain-p (f-inverse x)))
```

- **f-domain-is-number**

```
(implies (f-domain-p x)
         (acl2-numberp x))
```

- **f-inverse-relation**

```
(implies (f-inverse-domain-p x)
         (equal (f (f-inverse x)) x))
```

- **f-d/dx-f-relation**

```
(implies (f-inverse-domain-p x)
         (equal (f-inverse-prime x)
                (/ (f-prime (f-inverse x)))))
```

- **f-prime-not-zero**

```
(implies (f-domain-p x)
         (not (equal (f-prime x) 0)))
```

- **f-preserves-not-close**

```
(implies (and (f-domain-p x)
              (f-domain-p y)
              (i-limited x)
              (not (i-close x y)))
         (not (i-close (f x) (f y))))
```

The theorems generated by this macro are precisely the constraints needed to establish that the differentiable function f has an inverse. Note, for example, the last two theorems above. The theorem **f-prime-not-zero** ensures that $f'(x) \neq 0$ in the domain of f. This is, in fact, one of the hypotheses of the theorem that states that $(f^{-1})'(x) = 1/f'(f^{-1}(x))$, since the expression contains $1/f'(\ldots)$.

Both `defderivative` and `derivative-hyps` can be used in two different contexts. First, our own functions and macros use them the generate constraints in various `encapsulate`s. Second, the user can use these macros to generate the theorems that correspond to these constraints, thus making sure that (a) the constraints will be satisfied, and (b) the theorems satisfy the naming conventions assumed by the macros.

One final point is related to the theorems generated by these macros: These have to be proved by ACL2(r). Many times, the proofs succeed automatically, because the macros are careful to use the minimal theory required for the proofs to go through, but sometimes ACL2(r) needs a little help. The macros use keyword arguments to accept hints that will be passed on to the appropriate generated theorems. For example, the keyword argument `not-close-hints` is used to provide a hint to the theorem `f-preserves-not-close`.

The macro `defderivative` is the main entry point into the automatic differentiator. It takes two arguments, a prefix used to scope the generated theorem names, and the arithmetic expression that is to be derived. By "arithmetic expression", we mean an ACL2 term that is composed only of numbers, variables, arithmetic operators, and the application of functions with known derivatives or functions that are defined in terms of arithmetic expressions or as the inverse of functions that have known derivatives or are defined in terms of arithmetic expressions. In particular, recursive functions are not allowed, nor are functions that use `if` or `cond`.

What `defderivative` does is first to compute symbolically the derivative of the expression, and then to generate the lemmas necessary to demonstrate that the derivative computed symbolically is correct. Note that the lemmas follow the pattern and naming convention defined by `derivative-hyps`, so the derivatives of deeply nested expressions can be done automatically.

Finally, the macros `def-elem-derivative` and `def-elem-inverse` are used to register functions with known derivatives or inverses, respectively. For example, the AD system automatically defines the derivative of $f(x) = 1/x$ with the following event:

```
(def-elem-derivative
  unary-/
  elem-unary-/
  (and (acl2-numberp x)
       (not (equal x 0)))
  (- (/ (* x x))))
```

The arguments to this macro are (1) the name of the function that has a known derivative (in this case, `unary-/`), (2) a prefix used to name all the theorems generated by `derivative-hyps`, (3) an ACL2 expression for the domain of the function, and (4) an ACL2 expression for the derivative. Note that `def-elem-derivative` does not prove any theorems. Rather, it registers in a global database that the given function has the specified derivative. It is expected that the proofs have been previously generated, e.g., using `derivative-hyps`.

Similarly, `def-elem-inverse` is used to register an inverse function. For example, the following expression registers the inverse of the function `square`:

```
(def-elem-inverse
  square-inverse
  square-inverse
  (square-domain-p x)
  (square-inverse-domain-p x)
  square)
```

The arguments are similar to `def-elem-derivative`, except both the domain and range (or inverse-domain) need to be specified.

As we mentioned previously, the AD automatically uses `def-elem-derivative` to register the derivatives of $f(x) = 1/x$ and $f(x) = -x$. This explains how subtraction and division are handled, namely through the chain rule and those two derivative facts. In addition, we have developed several ACL2 books that establish the derivative of more complex functions, such as e^x, $\ln x$, $\sin(x)$, $\sin^{-1}(x)$, etc. The derivative of e^x was done using first principles, but the others were done using the macros described in this section. The relevant books also register the derivatives of these functions using `def-elem-derivative`. The derivative facts we have established so far are summarized in Figure 1. We note in passing that the macros support not just unary functions but binary functions *where one argument is held fixed*. This allows us to differentiate the `raise` function to find the derivatives of x^n and a^x. The trick of holding an argument fixed is essentially the same one used in [SG02].

4 Implementing the Macros

The macros `derivative-hyps` and `inverse-hyps` simply generate a `progn` containing several theorems. There is not much to their implementation.

Similarly, `def-elem-derivative` and `def-elem-inverse` are nothing more than syntactic sugar for ACL2's built-in `table` facility.

That leaves the definition of `defderivative`. In a nutshell, this macro works by repeatedly applying the chain rule to an expression, until each of its subexpressions is either a constant, a variable, a function with a known derivative, or the inverse of a function. Before describing this macro, we want to make a minor point. Obviously, `defderivative` needs access to ACL2's definition database, so that it can expand function applications to compute derivatives. However, access to these definitions depends on

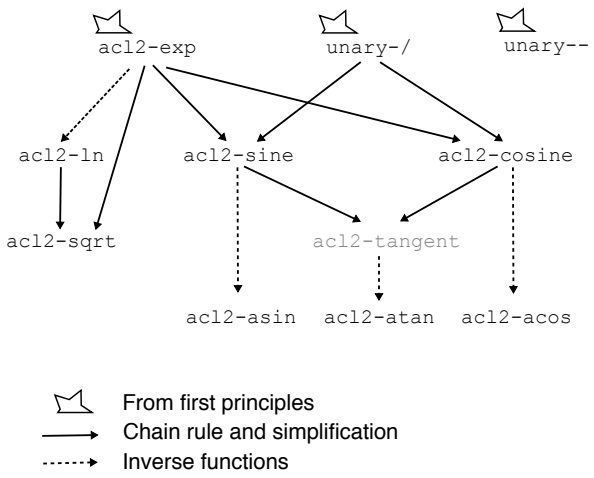

Figure 1: Dependency graph of the functions built into `defderivative`. Symbols leading into a function represent how its derivative theorems were proved.

access to ACL2 `state`, and ACL2 macros have traditionally not allowed access to `state`. However, such access is now permitted via `make-event`[2] [KM].

The first thing that `defderivative` does is translate the term to the derived, so that it does not contain any macros. Among other things, this replaces terms using `+` with terms using `binary-+`. Then, the resulting expression is differentiated symbolically. During this process, the necessary proofs are collected and laid out using `encapsulate`. Many of the proofs are done automatically by instantiating the relevant composition theorems. The macros know the name of the composition theorems and their constrained functions, so they can generate the appropriate hints. Note that this process also involves the symbolic computation of the domain of intermediate functions.

The proofs of these theorems need to be fully automated, since there is no way for the user to give hints, as the theorems may be associated with arbitrarily deep subterms of the original formula. We try to guarantee this automation by using only minimal theories, usually only the names of the theorems generated by `derivative-hyps` and `inverse-hyps`. This is one reason why users must conform to these naming conventions, even when they prove a derivative fact from first principles, as we did for e^x.

The last thing that `defderivative` does is to clean up the expressions generated for the derivative and domain of the function. This is, perhaps, the biggest weakness of our current implementation. The clean-up process is simplistic, consisting mainly of converting `binary-+` back to `+`. We have considered invoking ACL2's rewriter at this point, if only to perform arithmetic simplification, e.g., to convert `(+ (* x 1) (* x 1))` to `(+ x x)`. But we have not found a satisfactory set of rewrite rules, so we are leaving the sophisticated rewrites to the user.

[2] What *did* we do before `make-event`?

5 Conclusions

This paper described the implementation of an automatic differentiation (AD) system for ACL2(r). The implementation brought up several points of interest to the ACL2 community.

First, the idea of using macros to generate theorems according to some pattern is as old as ACL2 (at least). However, the current work shows how different macros can cooperate by keeping information in the ACL2 `state`. For example, the macro `definv` can register information about inverse functions, which is subsequently used by the macros `defderivative`.

Second, since ACL2 `state` is now available to macros, the macros can generate code that depends on the ACL2 database. For instance, the macros can use the definitions of ACL2 functions to generate theorems according to some pattern.

Third, our approach demonstrates the care that must be taken when designing libraries intended for automation. The history of ACL2 and the Boyer-Moore theorem prover includes several examples of libraries that are carefully designed so that new theorems can be proved almost automatically. The lemmas in these libraries are chosen carefully so that they work well with ACL2's heuristics (and vice versa). But when a macro develops a complex theory from arbitrary ACL2 expressions, it becomes increasingly likely that some rewrite rules triggered by the ACL2 expression interfere with the proof plan of the macro. So the macro has to take careful control of the proof execution, especially if hints are involved to instantiate constrained functions. In our experience, we have improved our chances by explicitly controlling the active ACL2 theory, and making sure it only has the rewrite rules that we think are absolutely necessary for the theorems to prove. We tried to use `:by` hints in these instantiations, so that the proof plan was completely controlled by the hints. Unfortunately, we ran into too many cases where `lambda` expressions created for derivatives did not exactly match the functional instantiation generated with a `:by` hint, so we had to switch to regular `:use` hints instead. So far, the proof plans are succeeding, but we would prefer to have a more robust mechanism.

Fourth, our solution illustrates how naming conventions can be used to associate facts with functions. For example, the fact that `square` is 1-to-1 in its domain can be stored in the theorem `square-is-1-to-1` and the domain itself in the function `square-domain-p`. Such conventions enable macros to generate appropriate hints. Moreover, maintaining those conventions is easy if the macros themselves generate the required theorems. Things are more complicated when some of the theorems need to be generated by hand, e.g., to show the derivative of e^x.

Finally, the techniques described in this paper have enabled us to vastly extend the derivative facts that have been certified with ACL2(r). As part of this project, we demonstrated from first principles that $de^x/dx = e^x$. Then we used the macros that we developed to formalize the derivatives of the trigonometric functions, and the inverse trigonometric and logarithmic functions.

References

[Gam00] 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.

[GC09] 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.

[KM] M. Kaufmann and J S. Moore. ACL2 documentation for `make-event`. `http://www.cs.utexas.edu/users/moore/acl2/v4-2/MAKE-EVENT.html`.

[RG11] P. Reid and R. Gamboa. Automatic differentiation in ACL2. In *Proc of the Second Conference on Interactive Theorem Proving (ITP-2011)*, 2011. doi:10.1007/978-3-642-22863-6_23.

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

[Rob96] A. Robinson. *Non-Standard Analysis*. Princeton University Press, 1996.

[SG02] J. Sawada and R. Gamboa. Mechanical verification of a square root algorithm using Taylor's theorem. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, 2002. doi:10.1007/3-540-36126-X_17