Autodifferentiation in Racket and Lua

Jaime Guerrero

May 6, 2015

1 Introduction

Differentiation of functions plays a foundational role in countless domains, including mathematics, engineering, economics, and various social sciences. The most familiar differentiation techniques are *symbolic differentiation* and *numerical approximation*. Symbolic differentiation takes a function and calculates a derivative by way of algebraic manipulations, while numerical approximation produces a value by using the limit definition of the derivative. Both of these techniques are popular and well-understood but have their drawbacks. For instance, symbolic differentiation often generates complicated expressions that are hard to understand, and numerical approximation with infinitesimals must deal with roundoff errors. Automatic differentiation (autodiff) is a technique built upon the chain rule which addresses these issues.

In this paper we first give a high-level description of autodiff. We then outline a Haskell implementation that serves as the basis of implementations in Racket and Lua. After describing the implementations, we finish with some thoughts on the role of metaprogramming in autodiff.

2 Autodifferentiation Overview

The two primary forms of automatic differentiation, reverse mode and forward mode, both exploit the compositional nature of the chain rule. The forms differ in the way they approach the multiplications within the chain rule; forward mode goes from right to left, while reverse mode goes from left to right. Forward mode is the more straightforward of the two, and it has been implemented here. This implementation does not allow for functions with multiple unknowns, but can calculate partial derivatives with respect to one unknown. Reverse mode is used in determining more sophisticated derivatives.

3 Karczmarczuk Implementation

The bulk of this code is inspired from Jerzy Karczmarczuk's Haskell implementation[1]. In his paper, Karczmarczuk exploits Haskell's lazy evaluation to create an intricate system. However, the first part of the paper works perfectly well in a strict setting. His implementation utilizes the most common technique for forward mode differentiation: an algebra of *dual numbers*. A dual number is a pair of numbers, wherein the first component is a function's value at a point, and the second component is the function's derivative at that same point. In order to instantiate a dual number, a higher-order function dlift function is given some function and its derivative. When provided with a point, dlift 's function creates a dual and calculates its derivative.

The use of dual numbers is a concise way to describe a function's value and derivative. But the above description of differentiation amounts to nothing more than a large table that maps functions to their derivatives. In order for more complex calculations, standard arithmetic operators must be overloaded to work upon the dual numbers that dlift generates. And since duals already carry derivatives along with them, the operators are where the standard derivative rules of calculus can be encoded. For example, $*$ takes two dual numbers and returns a third, where the returned value consists of multiplied function values and an encoding of the product rule as its second component. It is in this overloading that we can find the derivatives of all functions consisting of elementary operators upon algebraic and trigonometric functions.

4 Racket and Lua Implementations

We now explore the two provided implementations of autodifferentiation. Both are similar in their approaches, but differ in the particulars. We achieve an algebra on duals by way of operator overloading in both cases.

4.1 Explicitly Renaming Operators

Within Racket, operator overloading is performed by explicitly renaming the primitive operator, and changing the "old" operator to act upon dual numbers. With help from the rename−in function, the builtin Racket operators have been prepended with the letter "r".

Consider a dual structure along with dual multiplication:

```
(define-struct dual [x dx])

(define const
  (lambda (n)
    (make-dual n 0.0)))

(define var
  (lambda (n)
    (make-dual n 1.0)))

(define *
  (lambda (d1 d2)
    (when (number? d1) (set! d1 (const d1)))
    (when (number? d2) (set! d2 (const d2)))
    (let ([x (dual-x  d1)]
          [a (dual-dx d1)]
          [y (dual-x  d2)]
          [b (dual-dx d2)])
      (make-dual (r* x y)
                 (r+ (r* x b)
                     (r* a y))))))
```

Listing 1: Dual Contstructors and Dual Multiplication in Racket

Of interest is the calculations of each component that occur within the returned dual. Because the components of a dual are numbers, creating a dual forces us to "drop down" to the level of the renamed operators to perform normal arithmetic.

It is important to also notice the two when clauses above. Both of these coerce a normal number into a dual with derivative zero. This is necessary for two reasons. Firstly, when supplying a point to a function, the user must annotate whether they are interested in a both a function value and derivative calculation, or just a function value. This is done through use of the const and var keywords, which construct duals with initial derivative values of 0 and 1, respectively. In the cases when a user is interested in only a function value, we can save them a few keystrokes. So (dual−x (cos (const 1))) is the same as (dual−x (cos 1)). However, if the user does not want to deal with dual numbers, they can use any of the renamed base operators.

The second (and probably more important reason) is that more complex functions, especially those involving scalar multiples, need to have each piece of the function in dual form. Given the call (∗ 3 (cos (var 1))) and an overloaded definition of ∗, the 3 must be changed in order to satisfy the contract of ∗.

4.2 Operator Overloading via Metatables

The Lua implementation is very similar to the Racket implementation. However, rather than renaming primitive operations so they work upon dual values, we extend their behavior through manipulation of *metatables*. A metatable, in this case, is a collection of methods that are assigned to the representation of duals we have chosen. In this case a dual is a table with two components. In the following code, duals have their metatables set so that addition works component-wise on the dual's elements.

```
-- The metatable
mt = {
    __add = function (d1, d2)
        if type(d1) == "number" then d1 = const(d1) end
        if type(d2) == "number" then d2 = const(d1) end
        return {x = d1.x + d2.x, dx = d1.dx + d2.dx}
    end
}

function const (n)
    return setmetatable({x = n, dx = 0}, mt)
end

function var (n)
    return setmetatable({x = n, dx = 1.0}, mt)
end
```

Listing 2: Dual Contstructors and Dual Addition in Lua

One disadvantage to the Racket approach was that users needed to understand more carefully the types of the overloaded operators. With metatable overloading, coercion of a constant into a dual only occurs when the const or var keywords are found within the expression. Put another way, a call to 2 + 2 returns 4, and not a table with the components 4 and 0.

5 What is Metaprogramming?

In the broadest strokes, metaprogramming is understood to be the transformation of one program into another. Autodiff does not exactly fall into this space, as the numerical values returned are not (in a strict sense) programs.

At the very highest level, operator overloading can be seen as a form of metaprogramming, wherein the new operator definitions are programming the host language itself. This is reflected most clearly in the work of Hudak[2].

Following a more traditional definition of metaprogramming, we note that program generation is found within derivative calculations that occur at each step of the computation. Though the representation of the derivative is lost once its value is found, a natural extension to this program would be to provide those lost representations to the user. Doing so moves these programs more in the direction of symbolic differentiation systems, and would bring in those problems that autodiff was supposed to remedy. This gives rise to another extension: once there is a representation of the derivative, it can be passed through an algebraic optimizer. This would add a taste of staging while giving the user a sense that they are "doing metaprogramming".

6 Acknowledgements

Thanks to Rob Zinkov and Tim Zakian for their insights and clarifications.

References

[1] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Higher-Order and Symbolic Computation, Volume 14, Issue 1, pp 35-57, 2001.

[2] Paul Hudak, *Building Domain-Specific Embedded Languages*, ACM Computing Surveys, Volume 28, 1996.

[3] The Racket v.6.1.1 Documentation, PLT.

[4] The Lua 5.3 Reference Manual, Lua.org, 2015.