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.

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. What is special about reverse mode? This is a numerical technique.

3 A Haskell Implementation

The bulk of this code is inspired from Jerzy Karczmarczuk's Haskell implementation SOURCE. In his paper, Karczmarczuk exploits Haskell's lazy evaluation to create a sophisticated 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 sophisticated 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 Scheme 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))))))
```

Of interest is the calculations of each component that ocurr wtihin 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 dual with a derivative of 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)).

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

5 Takeaways

5.1 Operator overloading is DSL construction

Hudak's paper? At the highest level, we can understand operator overloading as a way to implement a DSL. This is particularly useful in our case, where both implementations (more or less) retain their mathematical flavor.

5.2 Derivative Construction as Metaprogramming

5.3 Optimizations