Part II: Symbolic Evaluation of Multiple Integrals

1 Introduction

So I decided to finally take the plunge, and try my hand with functional programming - with the intent of being at least familiar with the basics. I like the idea of a language that's more mathematical in nature, although I'm hesitant about too many layers of abstraction above the machine (well I mean this *is* coming from someone who's spent the past two years working in C and assembly). Anyway, I have no idea what I'm doing, but I'll try to tackle a small math problem and see how it goes. An interesting problem might be multiple integration, something that is very inefficient to do numerically, but fairly straightforward to do analytically. It also seems a good fit for a functional language - it's a recursive problem and one that really needs a conceptual base to work from. It's not difficult to find the indefinite integral of some rational polynomial function w.r.t. more than one variable, but you need to think through it in order to understand what's going on. I don't really have any idea how to do this in an imperative language, let alone in something like Scheme, but I want to give it a try. (Side note: not all mathematical problems are inherently conducive to functional programming. Consider the gradient descent algorithm or, more on topic, computing a Riemann Sum $\sum_{i=0}^{\infty} f(x_i^*)(x_i - x_{i-1})$)

The first problem we'd want to tackle is just finding the antiderivative (or indefinite integral) of a polynomial with respect to a single variable, which is just high school calculus. You take the exponent of the term, add one to it, then multiply the coefficient by that value. We can use tail recursion to go through each term, joining up the result of each to form a sum in the same format as the input. You might notice the omission of the constant c of integration, and that's for the very good reason that I'm feeling particularly lazy.

2 Single-variable Integration

So we need to figure out how to integrate a function $p : \mathbb{R} \to \mathbb{R}$, $p(x) = \sum_{i=0}^{n} a_i x^i$, where n is the order of the polynomial (assuming $a_n \neq 0$). We'll just explore indefinite integrals, which are easier to do, since we don't have to consider computing the bounds of the integral at each successive projection (although that should not be too difficult for simple domains).

I just wanted to write something very simple to get the basic concept done; the program accepts a symbol of a list of terms, each with the format (* a (** x k)), where a is the coefficient and k the exponent (* denotes multiplication, ** exponentiation, with prefix notation). A call to the function integrate might look something like:

```
(integrate '((* 3 (** x 4)) (* 9 (** x 6)) (* 2 (** x 7)) (* 4 (** x 9)) (*
    7 (** x 11))))
```

Which would find the antiderivative of the function $3x^4 + 9x^6 + 2x^7 + 4x^9 + 7x^{11}$. Of course, we would expect the result to be: $\frac{3}{5}x^5 + \frac{9}{7}x^7 + \frac{1}{4}x^8 + \frac{2}{5}x^{10} + \frac{7}{12}x^{12} + c$. And indeed, we get the output (correct, bar the integrating factor which I have ommitted):

```
((* 3/5 (** x 5)) (* 1 2/7 (** x 7)) (* 1/4 (** x 8)) (* 2/5 (** x 10)) (*
    7/12 (** x 12)))
```

The code is as follows, less the trivial make-product and make-exponent functions:

```
#lang scheme

(define (make-sum a1 a2)
  (list '+ a1 a2))
```

```
(define (make-product m1 m2)
  (list '* m1 m2))
(define (make-exponent e1 e2)
  (list '** e1 e2))

(define (term-integrate expr)
  (define (order term) (caddr (caddr term)))
  (define (var term) (cadr (caddr term)))
  (define (coeff term) (cadr term))
  (cond ((eq? (length expr) 0))
        (else (make-product
               (/ (coeff expr) (+ (order expr) 1))
               (make-exponent (var expr) (+ (order expr) 1))))))

(define (integrate expr)
  (cond ((empty? expr) empty)
        (else (cons
               (term-integrate (car expr))
               (integrate (cdr expr))))))
```

This becomes more interesting if we want to take the integral of the function with respect to more than one variable. Many more conditions are implied, and we have to handle several cases instead of just one. The integration has to be recursive as well, as the operation "cascades" from the inner expression to the outer one, which is evaluated after all the rest. In other words, the variable of integration that comes last (the rightmost) is the last one we integrate with respect to. More formally, we want to integrate a function:

$$p(x_1, x_2, ..., x_b) = \sum_{j=1}^{n} a_j \prod_{k=1}^{m_j} x_k^{c_{kj}} \tag{1}$$

Where b is the number of variables, n is the number of sum terms, m_j is the number of product terms in sum term j, and c_{kj} is the order of each x_i in the sum term j. This may seem like a lot to take in, but it's really just a formal way to say that we need to integrate a polynomial function of several variables, something like $3xy^2z + 5x^2y^3z^2$, where we can let $x = x_1, y = x_2, z = x_3$.

3 Multiple Integration

So now our single-variable integration function has to deal with handling terms that are not dependent on the variable of integration. I wrote this really inelegantly, and you can see it makes three passes through each sum term; one to check the entire term for dependence on var, one to find the new coefficient, and one to evaluate the result of integrating each product term w.r.t. var. I'm 99 percent sure there's a better way to do this, but the term integration isn't something that's important - conceptually, the order and process is far more integral (haha).

```
(define (make-sum a1 a2)
  (list '+ a1 a2))
(define (make-product m1 m2)
  (list '* m1 m2))
(define (make-exponent e1 e2)
  (list '** e1 e2))
```

```
(define (term-var-integrate expr var)
  (define (term-var term) (cadr term))
  (define (term-exp term) (caddr term))
  (define (dependent? expr) ; test for dependence on integrating variable
    (cond ((empty? expr) #f)
          ((eq? (term-var (car expr)) var) #t)
          (else
           (dependent? (list-tail expr 1)))))
  (define (coeff expr) ; determine the divisor of the coefficient
    (cond ((empty? expr) 0)
          ((eq? (term-var (car expr)) var)
           (+ (+ (term-exp (car expr)) 1) (coeff (cdr expr))))
          (else
           (coeff (list-tail expr 1)))))
  (define (product-terms expr) ; determine each product term after
     integrationq
    (cond ((empty? expr) empty)
          ((eq? (term-var (car expr)) var)
           (cons (make-exponent var (+ (term-exp (car expr)) 1)) (product-
              terms (cdr expr))))
          (else
           (cons (car expr) (product-terms (list-tail expr 1))))))
  (cond ((empty? expr) empty)
        ((dependent? (list-tail expr 2))
         (cons '* (cons (/ (cadr expr) (coeff (list-tail expr 2))) (product
            -terms (list-tail expr 2)))))
        (else empty)))

(define (integrate expr var-list)
  (define (var-integrate expr var)
    (cond ((empty? expr) empty)
        (else (cons
               (term-var-integrate (car expr) var)
               (var-integrate (cdr expr) var)))))
  (cond ((empty? var-list) expr)
        (else
         (integrate (var-integrate expr (car var-list)) (cdr var-list)))))
```

There's nothing particularly complex about the integrate procedure (/function/whatever you call it here I don't know lambda calculus), but it defines the problem very well. Multiple integration really means that you integrate [the result of integrating the expression with respect to the first variable] with respect to [the rest of the variables]. We also have to define the integral of an expression with respect to no variable to be the expression itself (I mean, this isn't really so much due to mathematical rigor as simply the need to establish a base case). What I love about this is that this is completely generalized and pure, independent of the way you actually do the term integration. I mean, if you had a lookup table of all the possible integrands, or used something crazy like complex analysis to evaluate the terms, the general principle still works, and that one line would still do what it's supposed to do. It's a nice function composition.

If you want a more rigorous perspective on this, just think how you would solve an iterated integral by hand; say if you're given $\iiint f(x,y,z)\mathrm{d}x\mathrm{d}y\mathrm{d}z$. The first thing you do is find the antiderivative $g(y,z) = \int f(x,y,z)\mathrm{d}x$, and then evaluate $\iint g(y,z)\mathrm{d}y\mathrm{d}z$. And that's exactly what happens here; we ensure that the first thing we do is integrate the entire term with respect to the first variable (in our example with f above, x), and the last thing we do is evaluate the integral of the result of all the previous integrations, with respect to the final variable (in our example, z). Isn't that cool? We're saying (integrate (var-integrate expr (car var-list)) (cdr var-list)), which corresponds *exactly* to what happens conceptually and mathematically.

I think the next step from here would probably be to evaluate definite, bounded integrals, but I'm getting a bit of a headache thinking about how the substitution and bounds evaluation would work; so I'll just leave that to Wolfram Alpha...