Topic 10
Example: Symbolic Differentiation

Section 2.3.2

October 2008

Symbolic differentiation

- Polynomial a + bx + cx^2 can be represented by the list (+ a (+ (* b x) (* c (* x x))))
- Derivative with respect to x is b + 2cx, which can be represented by (+ b (* 2 (* c x)))
- How can the derivative be computed?

Basic calculus (2 arguments only)

dy/dx = 0 if y is a constant or a variable other than x

dx/dx = 1

d(u+v)/dx = du/dx + dv/dx
(NOTE Recursive!)

d(u*v)/dx = u * dv/dx + v * du/dx
(NOTE Recursive!)

Want to build a procedure to do differentiation

- Think of the first rules as base conditions, then other rules can be used to decompose a problem into something easier.
- What do we need to do to tell which rule is applicable?
  - Differentiate between a constant, variable (and what it is), product, and sum
  - Extract parts of an expression

Data abstraction to the rescue!

Some constructors, selectors and predicates:

```
(variable? x)     (same-variable? x y)
(sum? x)          (product? x)
(make-sum x y)    (make-product x y)
(sum-arg1 x)      (product-arg1 x)
(sum-arg2 x)      (product-arg2 x)
```


Now we can compute derivatives

```
; takes an expression and a variable and
; returns the derivitive of expr wrt var
(define (deriv expr var)
  (cond ((number? expr) 0)
        ((variable? expr)
         (if (same-variable? expr var)
             1 0))
        ((sum? expr)
         (make-sum
           (deriv (sum-arg1 expr) var)
           (deriv (sum-arg2 expr) var)))
```

(`deriv` continued)

```
((product? expr)
   (make-sum
    (make-product
     (product-arg1 expr)
     (deriv (product-arg2 expr)
            var))
    (make-product
     (product-arg2 expr)
     (deriv (product-arg1 expr)
            var))))
  (else (error "Unknown type"
         expr))))
```


Implementation of lower layer

```
; takes an expression and returns #t
; if it is a variable
(define (variable? x) (symbol? x))

; takes two expressions and is #t if
; they are both the same variable
(define (same-variable? x y)
 (and (variable? x)
      (variable? y)
      (eq? x y)))
```


Sums

```
; takes two expressions and creates a sum
; with them as the arguments -- sums are
; simply represented as lists
(define (make-sum x y)
 (list '+ x y))

; returns #t if the argument is a sum
; a sum is a list whose first element is
; the symbol +
(define (sum? x)
 (and (pair? x)
      (eq? (car x) '+)))
```


Sum Selectors

```
; selectors for a sum retrieve
; the first and second arguments
; to be added
(define (sum-arg1 x) (cadr x))
(define (sum-arg2 x) (caddr x))
```


Products

```
; takes two expressions and creates a product
; with them as the arguments -- products are
; simply represented as lists
(define (make-product x y)
 (list '* x y))

; returns #t if the argument is a product
; a product is a list whose first element is
; the symbol *
(define (product? x)
 (and (pair? x)
      (eq? (car x) '*)))
```


Product Selectors

```
; selectors for a product retrieve
; the first and second arguments
; to be multiplied
(define (product-arg1 x) (cadr x))
(define (product-arg2 x) (caddr x))
```

It works! (sort of)

```
(define expr (make-product 'x 'y))
expr --> (* x y)

(deriv expr 'x) -->
  (+ (* x 0) (* y 1))
Should be y

Need to make simple reductions – use same
  trick as was used to reduce rational
  numbers – in the constructor
```


A better `make-sum`

```
; a new constructor that simplifies the sum a bit
(define (make-sum x y)
 (cond ((and (number? x) (= x 0)) y)
       ((and (number? y) (= y 0)) x)
       ((and (number? x) (number? y))
        (+ x y))
       (else (list '+ x y))))
```


A better `make-product`

```
; a new constructor that simplifies the product a bit
(define (make-product x y)
  (cond ((or (and (number? x) (= x 0))
             (and (number? y) (= y 0)))
          0)
         ((and (number? x) (= x 1)) y)
         ((and (number? y) (= y 1)) x)
         ((and (number? x) (number? y))
          (* x y))
         (else (list '* x y))))
```


Still room for improvement

```
(define expr (make-product 'x 'y))

expr --> (* x y)
(deriv expr 'x) --> y
but

(define expr (make-product 'x 'x))

expr --> (* x x)
(deriv expr 'x) --> (+ x x)
(* 2 x) would be better
```


Always room for improvement

More sophisticated simplifications are done in Reduce, Macsyma, Mathematica

Theoretically, no matter how many simplifications we build into the software, there are always more simplifications that can be made