Solutions – Practical 3

Michael P. Fourman

February 2, 2010

`Poly.ML`

Addition of polynomials is straightforward. This style of synchronised recursion, on two lists at once, is common.

Multiplication is more involved. The expression `(map (times a) bb)` multiplies each member of the list `bb` by `a` — the use of map hides an inner recursion. This inner recursion is analogous to one line of a long multiplication sum. In long multiplication, the remaining lines have to be shifted to multiply by 10; in polynomial multiplication, we shift (by consing 0.0) to give the effect of multiplication by x.

Evaluation directly implements Horner's rule.

Differentiation and integration are handled by introducing auxiliary functions, with an additional parameter to keep track of the power of x. In each case, we have to treat the constant term specially.

`SparsePoly.ML`

We keep the lists of terms in order of increasing powers of x. Our implementation of addition relies on this, and this property is preserved by the other operations.

Note the use of layered patterns in the definition of polynomial addition. This reduces the syntactic overhead of the record syntax. This style of recursion, keeping two ordered lists in step, is also common.

Multiplication is just as before; but this time we introduce an auxiliary function to handle a single line of the sum, and since powers are carried in the terms, we don't need an explicit shift.

Evaluation uses the explicit powers; Horner's rule is only good for dense polynomials.

Differentiation and integration can be done term-by term; again there is a special case for differentiation of a constant.

```
structure Poly : PolySig =
struct
type Poly = real list

fun x  ++ [] = x :Poly
  | [] ++ x  = x
  | (a :: aa) ++ (b :: bb) = (a + b) :: (aa ++ bb)

fun times a b : real = a * b

fun [] ** x  = [] :Poly
  | x  ** [] = []
  | (a :: aa) ** bb = map (times a) bb ++ (0.0 :: (aa ** bb))

fun eval []        v = 0.0
  | eval (a :: aa) v = a + v * (eval aa v)

fun mydiff n []       = [] :Poly
  | mydiff n (h :: t) = real n * h :: mydiff (n+1) t

fun diff [] = []
  | diff (h :: t) = mydiff 1 t

fun myint n [] = []
  | myint n (h :: t) = h / real n :: myint (n+1) t

fun int x = 0.0 :: myint 1 x ;
end;
```

```
structure SparsePoly : PolySig =
struct
type Poly = {coeff:real, power:int} list
(* keep lists in order of increasing powers *)

fun a ++ [] = a :Poly
  | [] ++ b = b
  | (poly1 as ((h1 as {coeff = c1, power = p1}) :: t1))
    ++
    (poly2 as ((h2 as {coeff = c2, power = p2}) :: t2))
    = if p1 = p2 then {coeff = c1 + c2, power = p1} :: (t1 ++ t2)
      else if p1 < p2 then h1 :: (t1 ++ poly2)
      else (* p2 < p1 *)   h2 :: (poly1 ++ t2)


fun times _ [] = [] :Poly
  | times (term as {coeff = c, power = p}) ({coeff = ch, power = ph} :: t)
        = {coeff = c * ch, power = p + ph} :: times term t

fun [] ** x  = [] :Poly
  | x  ** [] = []
  | (a :: aa) ** bb = (times a bb) ++ (aa ** bb)

infix 9 ^^;
fun a ^^ 0 = 1.0
  | a ^^ n = if n mod 2 = 0 then (a*a) ^^ (n div 2)
                            else (a*a) ^^ (n div 2) * a
fun eval []        v = 0.0
  | eval ({coeff, power} :: aa) v = (coeff * (v ^^ power)) + (eval aa v)

fun diff [] = []
  | diff ({coeff, power} :: t) =
         if power < 1 then diff t
         else {coeff = coeff * real power, power = power - 1} :: diff t

fun intterm {coeff, power} = {coeff = coeff/real(power + 1), power = power + 1}

fun int x = map intterm x ;
end;
```