Jacques Garrigue, 2007 年 11 月 28 日

再帰関数と数式処理

11　数式の微分と正規化

形式的微分

9 章では既に簡単な数式の定義の仕方を見た．ここでは微分を目的としているので，表現力を少し増やしたい．

```
type expr =
    X                                          (* 変数は X だけ *)
  | Cst of float                               (* 定数には実数を使う *)
  | Add of expr * expr
  | Mul of expr * expr
  | CMul of float * expr                       (* 定数の掛け算 *)
  | Pow of expr * float                        (* 定数でのべき *)
  | Sin of expr
  | Cos of expr
  | Exp of expr

let rec eval e x =                             (* 評価は通常どおり *)
  match e with
    X     -> x
  | Cst a -> a
  | Add (e1, e2) -> eval e1 x +. eval e2 x
  | Mul (e1, e2) -> eval e1 x *. eval e2 x
  | CMul (a, e1) -> a *. eval e1 x
  | Pow (e1, a)  -> eval e1 x ** a
  | Sin e1 -> sin (eval e1 x)
  | Cos e1 -> cos (eval e1 x)
  | Exp e1 -> exp (eval e1 x)

# eval (Pow (Sin X, 3.)) 1.;;
- : float = 0.595823236590955596
```

こんな式に対して，微分が簡単に定義できる．

```
let derive_op1 = function                      (* 引数に対する微分 *)
    CMul(a, x) -> Cst a
  | Pow (x, b) -> CMul (b, Pow (x, b-.1.))
  | Sin x -> Cos x
  | Cos x -> CMul(-1., Sin x)
  | Exp x -> Exp x
  | _ -> failwith "derive_op1"
val derive_op1 : expr -> expr

let get_arg = function                         (* 1 引数の構文の引数を返す *)
    CMul (_, e1) | Pow (e1, _)
  | Sin e1 | Cos e1 | Exp e1 -> e1
```

```
  | _ -> failwith "get_arg"                          (* それ以外の構文でエラー *)

let rec derive = function
    X -> Cst 1.
  | Cst _ -> Cst 0.
  | Add (e1, e2) -> Add(derive e1, derive e2)
  | Mul (e1, e2) -> Add (Mul(derive e1, e2), Mul(e1, derive e2))
  | e -> Mul (derive (get_arg e), derive_op1 e)

# derive (Mul (Sin X, Cos X));;
- : expr =
Add (Mul (Mul (Cst 1., Cos X), Cos X),
 Mul (Sin X, Mul (Cst 1., CMul (-1., Sin X))))
```

ここで **function** というキーワードを使っているが，以下の構文の略になる．

```
fun e -> match e in
```

短かい上に引数の名前を考えなくていいから便利だ．

練習問題 11.1　*1.* 様々な式を微分し，正しさを確かめよ．問題 *4.1.2* の derive とも比較せよ．

2. 新しい基本関数を追加せよ．

式の単純化

微分を正しく定義できたものの，式をそのまま出すと冗長な部分が多い．やはり単純しなければならない．

```
# derive (Sin (Add(X, Cst 1.)));;
- : expr = Mul (Add (Cst 1., Cst 0.), Cos (Add (X, Cst 1.)))
```

まず考えられるのは，変数を含まない部分を完全に計算してしまうことだ．

```
let map f e =                                       (* map も要る *)
  match e with
    X | Cst _ -> e
  | Add (e1, e2) -> Add (f e1, f e2)
  | Mul (e1, e2) -> Mul (f e1, f e2)
  | CMul (a, e1) -> CMul (a, f e1)
  | Pow (e1, a)  -> Pow (f e1, a)
  | Sin e1 -> Sin (f e1)
  | Cos e1 -> Cos (f e1)
  | Exp e1 -> Exp (f e1)

let rec const = function                            (* X を含むかどうか *)
    X -> false
  | Cst _ -> true
  | Add (e1, e2) | Mul (e1, e2) -> const e1 && const e2
  | e -> const (get_arg e)

let rec simpl e =
  if closed e then Cst (eval e 0.) else map simpl e

# simpl (derive (Sin (Add(X, Cst 1.))));;
- : expr = Mul (Cst 1., Cos (Add (X, Cst 1.)))
```

これで余計な部分が少し減ったが，まだまだ残っている．
次の考え方は，単純化に使える方程式を利用する．例えば，上の例では，

$$\texttt{Mul}(\texttt{Cst 1.}, \texttt{E}) = \texttt{E}$$

は知られている．

```
exception Nomatch

let simpl1 = function
  | Add(Cst 0., e1) | Add(e1, Cst 0.)           -> e1
  | Mul(Cst a, e1) | Mul(e1, Cst a)             -> CMul(a, e1)
  | Mul(CMul(a, e1), e2) | Mul(e1, CMul(a, e2)) -> CMul(a, Mul(e1, e2))
  | CMul(1., e1) -> e1
  | CMul(0., e1) -> Cst 0.
  | CMul(a, CMul(b, e1)) -> CMul(a*.b, e1)
  | Pow(e1, 1.) -> e1
  | Pow(e1, 0.) -> Cst 1.
  | Pow(Pow(e1, a), b) -> Pow(e1, a*.b)
  | _ -> raise Nomatch

let rec simpl e =
  if closed e then eval e 0. else
  let e = map simpl e in
  try simpl (simpl1 e) with Nomatch -> e

# simpl (derive (Sin (Add(X, Cst 1.))));;
- : expr = Cos (Add (X, Cst 1.))
```

多項式と正規化

こんな単純化方法では限界がある．例えば，

```
# simpl (derive (Mul (Sin X, Cos X)));;
- : expr = Add (Mul (Cos X, Cos X), CMul (-1., Mul (Sin X, Sin X)))
```

では，結果を

```
- : expr = Add (Pow (Cos X, 2.), CMul (-1., Pow (Sin X, 2.)))
```

として表示したい．
この例では，simpl1 に規則を追加するだけでできるが，一般的には同じ掛け算では関係のないものが入っているかも知れないので，局所的な規則だけではできない．

練習問題 11.2 simpl1 の規則を増やして，Mul(Sin X, Mul (Sin X, Sin X)) が Pow(Sin X, 3.) に正規化されるようにせよ．パターンマッチングの **when** 構文を使う必要がある．

単純化を越えて，式を正規化するために，Add と Mul を多項式に変えるといい．
話をはっきりさせるために，もっと制限された式に戻す．

```
type expr0 =
    PowX of float                                    (* X の a 乗 *)
  | Cst of float
  | Add of expr0 * expr0
  | Mul of expr0 * expr0
```

これだったら，変数 X の多項式を次数と係数のリストと見做せばよい．

```
type poly = (float * float) list
let eval_poly p x =
  List.fold_left (fun r (expn, coeff) -> r +. coeff *. (x ** expn)) 0. p
```

もっとも重要な操作は多項式の和だが，次数の順序で並べられていると仮定すると効率よくできる．

```
let rec add_poly p1 p2 =
  match p1, p2 with
    [], _ -> p2
  | _, [] -> p1
  | (x,a)::p1', (y,b)::p2' ->
      if x = y then
        if a +. b = 0. then add_poly p1' p2'
        else (x,a+.b)::add_poly p1' p2'
      else if x < y then (x,a)::add_poly p1' p2
      else (y,b)::add_poly p1 p2'
```

練習問題 11.3 *1.* この add_poly を使って，多項式の掛け算 mul_poly を定義せよ．

2. add_poly と mul_poly を使って，expr0 を poly に変換する関数 poly_of_expr0 : expr0 -> poly を定義せよ．

変数の数を増やすと，上の定義では足りない．

```
type expr1 =
    Pow of string * float
  | ...
type poly = ((string * float) list * float) list
```

数式で書くと

$$P = \Sigma_i(\Pi_j X_{ij}^{a_{ij}}) \cdot b_i$$

このとき，add_poly の定義は変えなくていい．mul_poly では (string * float) list の掛け算を行わなければならないが，実は add_poly がそこでも使える．

練習問題 11.4 多変数多項式における mul_poly および poly_of_expr1 を定義せよ．