Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 9

ML examples I: Symbolic Differentiation

Topics covered:

- Symbolic computation
- Data representation
- Prettyprinting
- Differentiation
- Simplification by rewriting

Symbolic computation

This covers applications where manipulation of mathematical *expressions*, in general containing variables, is emphasized at the expense of actual numerical calculation.

There are several successful 'computer algebra systems' such as Axiom, Maple and Mathematica, which can do certain symbolic operations that are useful in mathematics.

Examples include factorizing polynomials and differentiating and integrating expressions.

We will show how ML can be used for such applications. Our example will be symbolic differentiation.

This will illustrate all the typical components of symbolic computation systems: data representation, internal algorithms, parsing and prettyprinting.

Data representation

We will allow mathematical expressions to be built up from variables and constants by the application of n-ary operators.

Therefore we define a recursive type as follows:

```
#type term = Var of string
           | Const of string
           | Fn of string * (term list);;
Type term defined.
```

For example the expression $sin(x + y)/cos(x - exp(y)) - ln(1 + x)$ is represented by:

```
Fn("-",
   [Fn("/",[Fn("sin",[Fn("+",[Var "x";
                              Var "y"])]);
            Fn("cos",[Fn("-",[Var "x";
                              Fn("exp",
                                 [Var "y"])])])]);
     Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
```

The need for parsing and printing

Reading and writing expressions in their raw form is rather unpleasant. This is a general problem in all symbolic computation systems. Typically one wants:

- A *parser* to accept input in human-readable form and translate it into the internal representation
- A *prettyprinter* to translate output from the internal representation back to a human-readable form.

We will use parsing as another major example of functional programming, so we will defer that for now.

However we will now write a simple printer for our expressions.

Prettyprinting expressions

How do we want to print expressions?

- Variables and constants are just written as their names.
- Ordinary n-ary functions applied to arguments are written by juxtaposing the function and a bracketed list of arguments, e.g. $f(x_1, \ldots, x_n)$.
- Infix binary functions like $+$ are written in between their arguments.
- Brackets are used where necessary for disambiguation.
- Infix operators have a notion of precedence to reduce the need for brackets.

Storing precedences

We can have a list of binary operators together with their precedences, e.g.

```
#let infixes =
    ["+",10; "-",10; "*",20; "/",20];;
```

This sort of list, associating data with keys, is called an *association list*. To get the data associated with a key we use the following:

```
#let rec assoc a ((x,y)::rest) =
    if a = x then y else assoc a rest;;
```

In our case, we define:

```
#let get_precedence s = assoc s infixes;;
```

This procedure of linear search is inefficient, but it is simple and adequate for small examples. Techniques like hashing are better for heavyweight applications.

Making precedences modifiable

Because of static binding, changing the list of infixes does not change `get_precedence`.

However we can make `infixes` a reference instead and then it is modifiable:

```
  #let infixes =
     ref ["+",10; "-",10; "*",20; "/",20];;
...
  #let get_precedence s =
     assoc s (!infixes);;
  get_precedence : string -> int = <fun>
  #get_precedence "^";;
  Uncaught exception: Match_failure
  #infixes := ("^",30)::(!infixes);;
  - : unit = ()
  #get_precedence "^";;
  - : int = 30
```

This setup is not purely functional, but is perhaps more natural.

Finding if an operator is infix

We will treat an operator as infix just if it appears in the list of infixes with a precedence. We can do:

```
#let is_infix s =
   try get_precedence s; true
   with _ -> false;;
```

because `get_precedence` fails on non-infixes. An alternative coding is to use an auxiliary function `can` which finds out whether the application of its first argument to its seconds succeeds::

```
#let can f x =
   try f x; true
   with _ -> false;;
can : ('a -> 'b) -> 'a -> bool = <fun>
#let is_infix = can get_precedence;;
is_infix : string -> bool = <fun>
```

The printer: explanation

The printer consists of two mutually recursive functions.

The function `string_of_term` takes two arguments. The first is a 'currently active precedence', and the second is the term.

For example, in printing the right-hand argument of `x * (y + z)`, the currently active precedence is the precedence of `*`. If the function prints an application of an infix operator (here `+`), it puts brackets round it unless its own precedence is higher.

We have a second, mutually recursive, function, to print a list of terms separated by commas. This is for the argument lists of non-unary and non-infix functions of the form $f(x_1, \ldots, x_n)$.

The printer: code

```
#let rec string_of_term prec =
   fun (Var s) -> s
     | (Const c) -> c
     | (Fn(f,args)) ->
         if length args = 2 & is_infix f then
           let prec' = get_precedence f in
           let s1 = string_of_term prec'
                       (hd args)
           and s2 = string_of_term prec'
                       (hd(tl args)) in
           let ss = s1^" "^f^" "^s2 in
           if prec' <= prec then "("^ss^")" else ss
         else
           f^"("^(string_of_terms args)^")"

 and string_of_terms t =
   match t with
       [] -> ""
     | [t] -> string_of_term 0 t
     | (h::t) -> (string_of_term 0 h)^","^
                 (string_of_terms t);;
```

The printer: installing

CAML Light has special facilities for installing user-defined printers in the toplevel read-eval-print loop.

Once our printer is installed, anything of type `term` will be printed using it.

We use some special functions from the CAML Light **`format`** library to make sure that our printer interacts with the toplevel loop.

```
##open "format";;
#let print_term s =
    open_hvbox 0;
    print_string
     ("‘"^(string_of_term 0 s)^"‘");
    close_box();;
print_term : term -> unit = <fun>
#install_printer "print_term";;
- : unit = ()
```

Before and after

```
#t;;
t : term =
 Fn
  ("-",
   [Fn
     ("/",
      [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
       Fn ("cos", [Fn ("-", [Var "x"; Fn ("exp",
                                          [Var "y"])])])]);
    Fn ("ln", [Fn ("+", [Const "1"; Var "x"])])])
#install_printer "print_term";;
- : unit = ()
#t;;
t : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#let x = t;;
x : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#(x,t);;
- : term * term =
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`,
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#[x; t; x];;
- : term list =
 [`sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`]
```

Differentiation: algorithm

There is a well-known method (taught in schools) to differentiate complicated expressions.

- If the expression is one of the standard functions applied to an argument, e.g. $sin(x)$, return the known derivative.
- If the expression is of the form $f(x) + g(x)$ then apply the rule for sums, returning $f'(x) + g'(x)$. Likewise for subtraction etc.
- If the expression is of the form $f(x) * g(x)$ then apply the product rule, i.e. return $f'(x) * g(x) + f(x) * g'(x)$.
- If the expression is one of the standard functions applied to a composite argument, say $f(g(x))$ then apply the Chain Rule and so give $g'(x) * f'(g(x))$.

This translates very easily into a recursive computer algorithm.

Differentiation: code

```
let rec differentiate x tm = match tm with
    Var y -> if y = x then Const "1" else Const "0"
  | Const c -> Const "0"
  | Fn("-",[t]) -> Fn("-",[differentiate x t])
  | Fn("+",[t1;t2]) -> Fn("+",[differentiate x t1;
                               differentiate x t2])
  | Fn("-",[t1;t2]) -> Fn("-",[differentiate x t1;
                               differentiate x t2])
  | Fn("*",[t1;t2]) ->
     Fn("+",[Fn("*",[differentiate x t1; t2]);
             Fn("*",[t1; differentiate x t2])])
  | Fn("inv",[t]) -> chain x t
     (Fn("-",[Fn("inv",[Fn("^",[t;Const "2"])])]))
  | Fn("^",[t;n]) -> chain x t
    (Fn("*",[n; Fn("^",[t; Fn("-",[n; Const "1"])])]))
  | Fn("exp",[t]) -> chain x t tm
  | Fn("ln",[t]) -> chain x t (Fn("inv",[t]))
  | Fn("sin",[t]) -> chain x t (Fn("cos",[t]))
  | Fn("cos",[t]) -> chain x t
     (Fn("-",[Fn("sin",[t])]))
  | Fn("/",[t1;t2]) -> differentiate x
     (Fn("*",[t1; Fn("inv",[t2])]))
  | Fn("tan",[t]) -> differentiate x
     (Fn("/",[Fn("sin",[t]); Fn("cos",[t])]))
and chain x t u =  Fn("*",[differentiate x t; u]);;
```

Differentiation examples

Let's try a few examples:

```
#let t1 = Fn("sin",[Fn("*",[Const "2";
                            Var "x"])]);;
t1 : term = `sin(2 * x)`
#differentiate "x" t1;;
- : term = `(0 * x + 2 * 1) * cos(2 * x)`
#let t2 = Fn("tan",[Var "x"]);;
t2 : term = `tan(x)`
#differentiate "x" t2;;
- : term =
 `(1 * cos(x)) * inv(cos(x)) +
  sin(x) * ((1 * -(sin(x))) *
  -(inv(cos(x) ^ 2)))`
#differentiate "y" t2;;
- : term =
 `(0 * cos(x)) * inv(cos(x)) +
  sin(x) * ((0 * -(sin(x))) *
  -(inv(cos(x) ^ 2)))`
```

Simplification

It seems to work OK, but it isn't making certain obvious simplifications, like $0 * x = 0$. These arise partly because we apply the recursive rules like the chain rule even in trivial cases. We'll write a simplifier:

```
#let simp = fun
    (Fn("+",[Const "0"; t])) -> t
  | (Fn("+",[t; Const "0"])) -> t
  | (Fn("-",[t; Const "0"])) -> t
  | (Fn("-",[Const "0"; t])) -> Fn("-",[t])
  | (Fn("+",[t1; Fn("-",[t2])])) -> Fn("-",[t1; t2])
  | (Fn("*",[Const "0"; t])) -> Const "0"
  | (Fn("*",[t; Const "0"])) -> Const "0"
  | (Fn("*",[Const "1"; t])) -> t
  | (Fn("*",[t; Const "1"])) -> t
  | (Fn("*",[Fn("-",[t1]); Fn("-",[t2])])) ->
       Fn("*",[t1; t2])
  | (Fn("*",[Fn("-",[t1]); t2])) ->
       Fn("-",[Fn("*",[t1; t2])])
  | (Fn("*",[t1; Fn("-",[t2])])) ->
       Fn("-",[Fn("*",[t1; t2])])
  | (Fn("-",[Fn("-",[t])])) -> t
  | t -> t;;
```

Simplification examples

We need to apply `simp` recursively, bottom-up:

```
#let rec dsimp = fun
    (Fn(fn,args)) ->
         simp(Fn(fn,map dsimp args))
  | t -> simp t;;
```

Now we get better results:

```
#dsimp(differentiate "x" t1);;
- : term = `2 * cos(2 * x)`
#dsimp(differentiate "x" t2);;
- : term = `cos(x) * inv(cos(x)) +
            sin(x) *
            (sin(x) * inv(cos(x) ^ 2))`
#dsimp(differentiate "y" t);;
- : term = `0`
```

There are still other simplifications to be made, e.g. $cos(x) * inv(cos(x)) = 1$. Good algebraic simplification is a difficult problem!