Programming ~~Paradigms~~ Languages F28PL, Lecture 4
Polymorphic and higher-order list operations

Jamie Gabbay

November 23, 2015

Polymorphic list length function

Let's design `length:'a list -> 'a list` to calculate the length of a list. Note the polymorphic type. Recall our slogan: A list is either empty or a head and a tail.

- Base case: `[] ==> 0`.
- Recursion case: `(h::t) ==> 1 + length of t`.

This design is almost precisely the ML code itself:

```
- fun length [] = 0
    | length (_::t) = 1+length t;
> val length = fn : 'a list -> int
- length ["a","b","c"];
> 3 : int
length ["a","b","c"] ==> 1+length ["b","c"] ==>
1+1+length ["c"] ==> 1+1+1+length [] ==> 1+1+1+0 ==> 3
```

Polymorphic list append

First, choose the type:
`append:'a list -> 'a list -> 'a list`.

Next, recursively design the function. Classic n00b error: pattern-match on everything in sight. Don't; it's not necessary here. Just pattern-match on the first argument.

- Appending `[]` to `l` gives `l`.
- Appending `hd::tl` to `l` gives `hd` appended to the result of appending `tl` to `l`.

Obvious, isn't it? Corresponding ML code:

```
- fun append [] l2 = l2
    | append (h1::t1) l2 = h1::append t1 l2;
> val append =
   fn : 'a list -> 'a list -> 'a list
- append ["a","b","c"] ["d","e","f"];
> ["a","b","c","d","e","f"] : string list
```

Polymorphic list append

```
append ["a","b","c"] ["d","e","f"]
==> "a"::append ["b","c"] ["d","e","f"]
==> "a"::"b"::append ["c"] ["d","e","f"]
==> "a"::"b"::"c"::append [] ["d","e","f"]
==> "a"::"b"::"c"::["d","e","f"]
==> ["a","b","c","d","e","f"]
```

In fact, append is primitive in ML.

```
> op @; (* infix operator so write "op" to fetch function *)
val it = fn : 'a list * 'a list -> 'a list
- [1,2,3]@[4,5,6];
> [1,2,3,4,5,6] : int list
```

Two differences from our function `append`: 1. `op @` is probably optimised to the underlying list representation of the specific ML implementation; and 2. the types are different.

Make sure you understand how. Make sure you can write an append function of type `'a list * 'a list -> 'a list`.

List member

Let's pick up the pace:

1. Intended type is `'a -> 'a list -> bool` (but see below).
2. `e1` is not in `[]`, and
3. `e1` is in `e1::tl`, and
4. otherwise `e1` is in `e1::tl` if `e1` is in `tl`.

Patterns in ML are affine—a variable may occur at most once. So we can't write clause 2 above in ML as `member e1 (e1::tl) = ...`.

No matter:

```
- fun member _ [] = false
    | member e1 (e2::t) =  e1=e2 orelse member e1 t;
> val member =
   fn : ''a -> ''a list -> bool
```

List member

Note the difference in types: we wanted `'a -> 'a list -> bool` but we got.

```
> val member =
   fn : ''a -> ''a list -> bool
```

Thus `member` is polymorphic over equality types.

This is natural: in order to check `member e1 l` we need to be able to check whether elements of `l` are equal to `e1`.

(Python would be more optimistic: it would check mathematical equality where it can, and fall back to comparison of pointers where it can't. But that relies on having a von Neumann machine in the background.)

Add (push to queue)

When you get into this, the ML becomes easier to read than the English:

```
- fun add e [] = [e] |
      add e1 (e2::t) = if e1=e2
        then e2::t
        else e2::add e1 t;
> val add =
   fn : ''a -> ''a list -> ''a list
> add 5 [1,2,3,4];
val it = [1, 2, 3, 4, 5] : int list
> add "forty-two" ["And","the","answer","is"];
val it = ["And", "the", "answer", "is", "forty-two"]
       : string list
```

List remove/delete first occurrence

```
- fun delete _ [] = [] |
      delete e1 (e2::t) = if e1=e2
        then t
        else e2::delete e1 t;
> val delete =
   fn : ''a -> ''a list -> ''a list
> delete 0 [1,0,~1,2,0,~2];
val it = [1, ~1, 2, 0, ~2] : int list
```

Exercise: Write yourself a ‘delete all occurrences’ function, please.

Higher-order functions

A higher-order function is a function that inputs a function. In other languages, this may be called a functor.

In ML, no such distinction is made: functors are just a particular special case of functions.

Here are some examples of types that higher-order functions might have:

```
('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
('a -> bool) -> 'a list -> 'a list
('a -> 'b) -> 'a list -> 'b list
(('a -> 'b) -> 'b) -> 'a
```

It is often possible to deduce what a function must do, just from its type. For instance, the only thing that could populate `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)` is `fn f => fn g => g o f`.

Higher-order functions

Great for abstraction and code-reuse. Consider `('a -> bool) -> 'a list -> 'a list`. The only reasonable function that populates this type is `filter`:

```
- fun filter _ [] = [] |
    filter p (h::t) = if p h then h::filter p t
                              else filter p t;
> val filter =
   fn : ('a -> bool) -> 'a list -> 'a list
```

In words `filter p l` traverses `l` throwing out elements that don't satisfy p.

Very common operation, that.

Lists

```
- fun isPos x = x>0; (* Our p: ‘is positive’ *)
> val isPos = fn : int -> bool
- filter isPos [~2,1,0,2];
> [1,2] : int list
(* Execution:
filter isPos [~2,1,0,2]
==> filter isPos [1,0,2]
==> 1::filter isPos [0,2]
==> 1::filter isPos [2]
==> 1::2::filter isPos []
==> 1::2::[] ==> [1,2] *)
```

We note that `filter` is tail-recursive, so an optimising compiler will optimise this to the same low-level code as might be written by a programmer-optimised interative program. But the ML code is, I believe, cleaner (even for this simple example).

Map

The other big list operation is 'apply a function f to every element of a list'.

```
- fun map _ [] = []
    | map f (h::t) = f h::map f t;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

E.g. to find the list of sizes of a list of strings:

```
> size;
val it = fn : string -> int
> map size; (* Nice bit of partial application, here *)
val it = fn : string list -> int list
> map size ["a","bc","def"];
val it = [1,2,3] : int list
```

Examples

I studied maths as an undergrad so for me, the obvious example of `map` is for Taylor series like these:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

The components of this sum are an instance of `map` applied to a function `f` such that $n \mapsto x^n/n!$.

If you are into gaming and understand GPU architecture, then you might recognise that stream processors are nothing more than a hardware optimised for applying `map` in parallel to a list or an array of data.

Other examples abound. ML is very good at expressing this kind of thing.

Examples

Another example:

```
- fun powers (x:int) = (x,x*x,x*x*x);
> val powers = fn : int -> int * int * int
- map powers [1,2,3];
> [(1,1,1),(2,4,8),(3,9,27) :  (int * int * int) list
```

Evaluation is:

```
map powers [1,2,3]
==> powers 1::map powers [2,3]
==> powers 1::powers 2::map powers [3]
==> powers 1::powers 2::powers 3::map powers []
==> powers 1::powers 2::powers 3::[]
==> [(1,1,1),(2,4,8),(3,9,27)]
```

List insert

Can we write a program `insert` that if given an integer `i1` and a list of integers ordered in ascending order, will insert `i1` into the list so the result is also a list of integers ordered in ascending order?

This is pretty much what you do when you put a library book in a shelf: you're given a list in order and an element and you want to put the element in 'the right place'.

Why not have a go at this yourself before reading the answer?

First, write an inductive specification; then convert it to ML code.

List insert

Bet you just skipped to this slide without trying it yourself.

No really; you'll learn more if you have a go first.

List insert

```
- fun insert(i:int) [] = [i]
    | insert i1 (i2::t) =
        if i1<i2 then i1::(i2,e2)::t
                 else i2::insert i1 t;
> fn : int -> int  list -> int  list
- insert 7 [5,9];
> [5,7,9] :int list
(* insert 7 [5,9]
==> 5::insert 7 [9]
==> 5::7::[9]
==> [5,7,9] *)
```

Sorting

We can now write an easy sorting algorithm:

```
- fun sort [] = []
    | sort (h::t) = insert h (sort t);
> fn : int list -> int list
- sort [7,9,5];
> [5,7,9] : int list
```

Isn't that beautiful, compact, and elegant code?

foldr

Consider summing a list of integers:

```
- fun sum [] = 0
    | sum (h::t) = h+sum t;
> val sum = fn: int list -> int
- sum [1,2,3];
> 6 : int
sum [1,2,3]
==> 1+sum [2,3]
==> 1+(2+sum [3])
==> 1+(2+(3+sum []))
==> 1+(2+(3+0))
==> 1+2+3+0
```

Intuitively, `sum` inserts + between the list elements (and calculates the result).

foldr

Consider joining a list of strings:

```
- fun join [] = ""
    | join (h::t) = h^join t;
> val join = fn: string list -> string
- string ["1","2","3"];
> "123" : int
sum ["1","2","3"]
==> "1"^join ["2","3"]
==> "1"^("2"^join ["3"])
==> "1"^("2"^("3"^join []))
==> "1"^("2"^("3"^""))
==> "1"^"2"^"3"^""
```

Intuitively, `join` inserts ^ between the list elements (and calculates the result).

foldr

Suppose we have a list of functions in `'a -> 'a` that we want to compose:

```
- fun bigo [] = (fn x => x)
    | bigo (h::t) = h o (bigo t);
> val bigo = fn: ('a -> 'a) list -> ('a -> 'a)
bigo [h,g,f]
==> h o bigo [g,f]
==> h o (g o bigo [f])
==> h o (g o f o (bigo []))
==> h o (g o f o (fn x => x))
==> h o g o f
```

Intuitively, `bigo` inserts o between the list elements.

We have a schema here:

foldr

- Base case: `[] ==> base value b`.
- Recursion case:
  `(h::t) ==> apply f to h and result of folding f over`

```
- fun foldr f acc [] = acc
    | foldr f acc (h::t) = f h (foldr f acc t);
> val foldr =
   fn: ('a->'b->'b) -> 'b -> 'a list -> 'b
```

I find this easiest to understand as follows:

```
foldr f acc [x,y,z] = f(x,f(y,f(z,acc)))
```

This is a surprisingly general recipe, because it captures the essence of iteration. This is a for-next loop. To be precise: `foldr` captures the essence of tail-recursion.

foldr

```
sum = foldr (fn x => fn acc => x+acc) 0
join = foldr (fn x => fn acc => x^acc) ""
bigo = foldr (fn x => fn acc => x o acc) (fn x => x)
sort = foldr insert [] (* do "insert" between elements of list
```

Now sum and join are kind of trivial, and bigo isn't trivial but it's easy to underestimate it; but you should know that sorting is a canonical non-trivial program.

foldl

Much like `foldr` but starts from the head of the list:

```
- fun foldl f acc [] = acc
    | foldl f acc (h::t) = foldl f (f h acc) t;
> val foldl =
   fn: ('a->'b->'b) -> 'b -> 'a list -> 'b
```

I understand it like this:

```
foldl f acc [x,y,z] = f(z,f(y,f(x,acc)))
foldr f acc [x,y,z] = f(x,f(y,f(z,acc)))  (* for comparison *)
```

`foldl` is natural because "we read the list from left-to-right and start computing on the x first".

Clearly, `foldl` is equal to
`fn f => fn acc => fn l => foldr f acc (rev l)`.

fold

Oh yes, and `rev` can, of course, be implemented using `foldl` and `foldr`:

```
foldr (fn x => fn acc => acc@[x]) []
foldl (fn x => fn acc => x::acc) []
```

Higher-order sort

Recall sorting using `foldr`:

```
sort [3,2,1]
==> foldr insert [] [3,2,1]
==> insert 3 (foldr insert [] [2,1])
==> insert 3 (insert 2 (foldr insert [] [1]))
==> insert 3 (insert 2 (insert 1 (foldr insert [] [])))
==> insert 3 (insert 2 (insert 1 []))
==> [1,2,3]
```

This invites generalisation.

Higher-order insert

Generalise insert to work with list of arbitrary type:

```
- fun gInsert p v [] = [v]
    | gInsert p v (h::t) =
      if p v h
      then v::h::t
      else h::gInsert p v t
> val gInsert = fn : ('a ->'a->bool)->
                   'a -> 'a list -> 'a list
```

- If `p` holds between `v` and `h` then put `v` on front of list.
- Otherwise put `h` on front of inserting `v` into `t` with `p`.

Higher-order sort

Our previous `sort` can be implemented as
`val sort = gInsert (fn x => fn y => x<y).`

A generalised sorting algorithm is just
`val gSort = fn p => foldr (gInsert p).`

Unpacking the fold explicitly:

```
- fun gSort p [] = []
    | gSort p (h::t) = gInsert p h (gSort p t);
> val gSort = fn : ('a -> 'a -> bool) ->
                   'a list -> 'a list
```

In OO programming you'd probably recognise a generalised sorting algorithm over objects with 'compare' and 'insert' methods. Where do you think these ideas come from?