Programming ~~Paradigms~~ Languages F28PL, Lecture 3
Yet more on ML

Jamie Gabbay

November 12, 2015

Type variable

SML provides type variables to express unknown types. We use the Greek alphabet α, β, γ for these; in ASCII we write `'a`, `'b`, `'c`.

Functions with type variables are (parametrically) polymorphic—the 'parameters' here are the type variables.

```
fn x => x;                  (* polymorphic identity *)
val it = fn : 'a -> 'a
fn f => fn x => (f x); (* polymorphic application *)
val it = fn : ('a -> 'b) -> 'a -> 'b
fn x => fn y => x;      (* polymorphic first projection *)
val it = fn : 'a -> 'b -> 'a
fn (x:int) => fn (y:int) => x; (* non-polymorphic first projec
val it = fn : int -> int -> int
```

Type variables

You are required to be able to:

- deduce the types of such simple functions,
- generate functions to populate a polymorphic type,
- deduce what a function must do, just from its type.

Q. Write a function of type `'a -> bool -> ('a,bool)`.

Q. What are the types of `fn x => [x]` and `fn f => (f 0)`?

Q. What must the function
`map : ('a -> 'b) -> 'a list -> 'b list` do?

Your lecturer will provide answers at the modest cost of $10/question. PayPal accepted.

Type variables

Type variables must be used uniformly. For instance, `'a * 'a * 'a` is populated by `(1,2,3)` and `(true, false, true)`, and `("a", "b", "c")`, but not by `(1, false, "c")`.

Take a look:

```
> val f = fn (x:'a,y:'a,z:'a) => (x,y,z);
val f = fn : 'a * 'a * 'a -> 'a * 'a * 'a
> f (1,2,3);
val it = (1, 2, 3) : int * int * int
> f (true,false,true);
val it = (true, false, true) : bool * bool * bool
> f ("a","b","c");
val it = ("a", "b", "c") : string * string * string
> f (1,false,"c");
Error-Can't unify bool with string (Different type constructor
( 1, false, "c")
```

Polymorphic parametricity is quite good fun. You too can write parametric programs; have a go.

Tuple patterns

Patterns can include tuples (we have already seen this).

E.g. join two strings in tuple together with a space in between:

```
- fun tJoin (s1,s2) = s1^" "^s2;
> val tJoin =
    fn : string * string -> string
(* takes 2 strings so s1 and s2 must be string *)
- tJoin ("hello","there");
> "hello there" : string
```

Tuple patterns

This might seem obvious, but students often forget about patterns. Here's a really compact program to swap the elements of a list of pairs:

```
> fun swapmypairs [] = []
    | swapmypairs ((x,y)::tl) = (y,x)::(swapmypairs tl);
val swapmypairs = fn : ('a * 'b) list -> ('b * 'a) list
(* Let's have a go *)
> swapmypairs [("Sherlock","Holmes"),
               ("Philip","Marlowe"),
               ("Jane", "Marple")];
val it = [("Holmes", "Sherlock"),
          ("Marlowe", "Philip"),
          ("Marple", "Jane")]
: (string * string) list
```

Tuple patterns

Q. Does `swapmypairs` implement the same function as `map (fn (x,y) => (y,x))`? If so, why? If not, why not?

A. Yes. `map` just implements the traversal of the list.

Q. Does `swapmypairs` implement the same function as `map (fn (x,y) => if (x=y) then (x,x) else (y,x))`? If so, why? If not, why not?

A. No. The type is `(''a * ''a) list -> (''a * ''a) list`.

Lists

A list is an arbitrary (possibly 0) length sequence of the same type.

If `'a` is a type then `'a list` is a type populated by list of `'a`.

Lists are polymorphic. `'a` can be any type, including a list or function type:

```
> [];
val it = [] : 'a list
> nil;                (* Synonymous with [] *)
val it = [] : 'a list
> [#"a",#"b"];        (* List of chars *)
val it = [#"a", #"b"] : char list
> [[],[#"a"],[#"a",#"b"]];
                      (* List of lists of chars *)
val it = [[], [#"a"], [#"a", #"b"]] : char list list
> [fn x => x, fn f => fn x => f x];
                      (* List of polymorphic functions *)
val it = [fn, fn] : (('a -> 'b) -> 'a -> 'b) list
```

Lists

Non-examples of ML lists:

```
[1,"one"]
Error: must have same type
> [fn (x:int) => x, fn f => fn x => f x];
     (* Why precisely doesn't this work? *)
Error: must have same type
```

List constructors: `[]` or `nil` are synonymous for the empty list, and `(op ::):'a -> 'a list -> 'a list` generates a list from an `'a` and an `'a list`.

List patterns: `[]` and `hd::tl`.

Q. What type do `hd` and `tl` have in `fn (hd::tl) => (hd,tl)`?

A. `'a` and `'a list`.

Lists

Q. Are the following lists equal? `[1,2,3]` and `1::(2::[3])` and `1::(2::(3::[]))`?

A. Yes! Note in particular that `[3]` is equal to `3::[]`.

Q. Are the following lists equal? `[1,2,3]` and `[1]::[2]::[3]` and `1::(2::(3))`?

A. Trick question. The right-hand one is not a list, because `(3)` is not a list. The middle one is also not a list, because `[1]:int list` and `[2]:int list` and `[3]` does not have type `int list list`.

I would never ask a trick question like that in the exam, but I might ask this:

Q. `[1]::[2]::[3]` is not a well-formed ML expression. Give a full and detailed explanation of why.

A. It raises a type error, because `op ::` has type `'a -> 'a list -> 'a list` and `[2]` has type `int list` and `[3]` does not have type `(int list) list`.

Precedence

```
fn f => fn x => f x::[x];   (* Does this mean ... *)
fn f => fn x => (f x)::[x]; (* this?  Or ... *)
fn f => fn x => f(x::[x]);  (* this? *)
```

op `::` has lower precedence than function calls. Thus:

```
fn f => fn x => f x::[x];
val it = fn : ('a -> 'a) -> 'a -> 'a list
fn f => fn x => (f x)::[x];
val it = fn : ('a -> 'a) -> 'a -> 'a list
> fn f => fn x => f(x::[x]);
val it = fn : ('a list -> 'b) -> 'a -> 'b
```

Example use of lists

Very simple:

```
fun countdown 0 = [0]
  | countdown n = n::(countdown (n-1));
```

Generates a countdown. Equivalent to `list(range(n,0,-1))` in Python.

What is a list?

You must fully appreciate the significance of the following slogan:

A list is either empty or a head and a tail.

This is more than just a structural fact of lists. It implies a programming principle. To write a function on lists is, simply, to account for these two possibilities.

Thus:

Example list programs

```
fun sumlist [] = 0
  | sumlist (hd::tl) = hd+(sumlist tl);
fun joinlist [] = ""
  | joinlist (hd::tl) = hd^(joinlist tl);
fun doublelist [] = []
  | doublelist (hd::tl) = (2*hd)::(doublelist tl);
fun ziplists [] [] = [] (* Raises warning.  Why? *)
  | ziplists (h1::t1) (h2::t2) = (h1,h2)::(ziplists t1 t2);
fun interleave [] l2 = l2
  | interleave l1 [] = l1
  | interleave (h1::t1) (h2::t2) = h1::h2::(interleave t1 t2);
fun splitlist [] = ([],[])
  | splitlist [hd] = ([hd],[])
  | splitlist (h1::h2::tl) =
      let val (l1,l2) = (splitlist tl) in (h1::l1,h2::l2) end
fun concat [] l = l
  | concat (hd::tl) l = hd::(concat hd tl);
```

You don't need to pattern-match every argument (e.g. `concact`).

Note local definition in `splitlist`; see Slide 20.

Equality types

Suppose we want to count how often v appears in a list.

```
fun count v [] = 0
  | count v (hd::tl) =
        if (hd=v) then 1+(count v tl) else (count v tl);
> val countzero = fn : ''a -> ''a list -> int
```

Note the double dash here. This is because v must have equality type; it must support equality comparison.

Integers are an equality type. The canonical example of a non-equality type is a function type. Functions cannot be compared for equality, in general (cf. halting problem).

```
> 0 = 0;
val it = true : bool
> (fn x => x) = (fn x => x);
Error-Can't unify ''a with 'b -> 'b (Requires equality type)
```

Equality types

You are expected to know about equality types, and to recognise when they will appear. For instance:

Q. State the ML types of the following two programs, and explain their similarities and differences.

```
fun easyA (x,y) = true;
fun easyB (x,y) = (x=x) andalso (y=y);
```

A. The types are `easyA:'a*'b -> bool` and `easyB:''a*''b -> bool`. Both functions calculate the function 'map `(x,y)` to true', however, `easyB` performs equality tests on `x` and `y` and so restricts the polymorphism to equality types.

Accumulation variable

An extra variable added to a recursive function, used to pass information from one stage of the recursion to another.

Suppose we want to count the number of negative, zero, or positive numbers in a list of integers. We can use a 3-tuple to accumulate our count as we traverse the list:

```
fun counts (n,z,p) [] = (n,z,p)
  | counts (n,z,p) (0::t) = counts (n,z+1,p) t
  | counts (n,z,p) (h::t) =
      if h<0
      then counts (n+1,z,p) t
      else counts (n,z,p+1) t;
val counts = fn : int * int * int ->
                  int list -> int * int * int
```

Counts is tail recursive so a sufficiently smart compiler will re-use local stack in the function call and produce code effectively identical to an iterative loop (i.e. a 'for-next' loop). You're supposed to know that, too.

Accumulation variable

There's a nice opportunity for some partial application here.

```
fun count l = counts (0,0,0) l;          (* Valid *)
val count = fn l => counts (0,0,0) l; (* Equivalent to above.
val count = counts (0,0,0);              (* Slick *)
```

The third and final program has the most style, to my mind.

Accumulation variable

Generate list of squares from `m` to `n` inclusive, in ascending order:

```
fun squares m n =
  if m>n then []
         else (m*m)::squares (m+1) n;
val squares =
   fn : int -> int -> int list
```

`m` can be viewed as an accumulation variable. For for instance:

```
val fromzerosquares = squares 0;
```

Local definitions

Local definitions let you bind a variable to a value in a local scope:

```
let val x = 12
  in x*x*x
  end;
> 1728 : int
```

Almost equivalent to function application:

```
(fn x => x*x*x) 12
> 1728 : int
```

But not quite: instantiation of type variables is per instance with `let` and per application with function application.

Thus ...

Local definitions

```
let val id = (fn x => x) in (id 1,id "one") end;
val it = (1, "one") : int * string
(* That works *)
(fn id => (id 1,id "one")) (fn x => x);
Error-Can't unify Int32.int/int with string
(* That doesn't! *)
```

`fn x => x` has type `'a -> 'a`. The type variable can be instantiated to `int` on the left and `string` on the right in the first case, but must be instantiated uniformly to a single type in the second case.

Most of the time, `let` is useful in combination with pattern-matching. The `splitlist` example from Slide 14 is typical.

Exceptions

Exceptions are impure. They step outside the functional programming paradigm.

ML is not a pure functional programming language. It is a programming language within which pure functional programming is easy to do. But, if you choose to stray to the dark side, you can.

Exceptions break the flow of control—typically after some error. When an exception is raised control is transferred to the innermost handler for that exception.

The canonical example is divide-by-zero, of course.

Exceptions

An exception is initiated by `raise`.

If no handler is present, then control passes to the system and the program stops.

```
- exception DIVIDE_BY_ZERO;  (* We declare our exception *)
> exception DIVIDE_BY_ZERO
- fun divide x y =
   if y=0
   then raise DIVIDE_BY_ZERO
   else x div y;
> val divide = fn: int -> int
- divide 3 0;
> exception DIVIDE_BY_ZERO
uncaught exception DIVIDE_BY_ZERO
```

Exceptions

'Pop' from a list:

```
exception LIST_EMPTY;
fun pop [] = raise LIST_EMPTY
  | pop (hd::tl) = hd;
> pop [3];
val it = 3 : int
> pop [];
Exception- LIST_EMPTY raised
```

Exceptions

Exceptions can carry data:

```
> exception MY_EXCEPTION of int;
exception MY_EXCEPTION of int
> raise MY_EXCEPTION 5;
Exception- MY_EXCEPTION 5 raised
```

Exceptions are handled as follows:

```
(raise MY_EXCEPTION 5) handle MY_EXCEPTION x => x+1;
```

Exceptions

Top-level exceptions must be monomorphic (having one type—i.e. not polymorphic) and types of local exceptions are frozen. So this won't compile:

```
exception LIST_ODD of 'a;
fun twotop (h1::h2::tl) = [h1,h2]
  | twotop [h1:'a] = raise LIST_ODD h1
  | twotop [] = raise LIST_EMPTY;
```

Why this restriction? To exclude this kind of thing:

```
exception EXCEPT of 'a;
(raise EXCEPT true) handle EXCEPT x => x+1;
```

Type aliases

```
type person_ID = int;
type name = string;
```

Now `person_ID` is a synonym for `int`. You can use explicit type annotations freely:

```
> type person_ID = int;
type person_ID
> type name = string;
type name
> fn (x:person_ID) => x;
val it = fn : person_ID -> person_ID
> fn (x:person_ID) => (x:int);
val it = fn : person_ID -> int
> fn (x:int) => x;
val it = fn : int -> int
```

Print depth

Consider a long list:

```
> List.tabulate(1000,fn x => x);
val it =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
   17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
   31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
   45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
   59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
   73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
   87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...]
   : int LIST.list
```

The pretty-printer gives up after 100 entries. This is controlled by a global variable for print depth.

Variable is compiler-dependent.

Print depth

Running PolyML on my system it's called `PolyML.Compiler.printDepth`.

(I found this out from `polyml.org/documentation/Reference/PolyMLStructure.html`.)

```
> PolyML.Compiler.printDepth;
val it = ref 100 : int ref
> PolyML.Compiler.printDepth := 1000;
val it = () : unit
> List.tabulate(1000,fn x => x);
(* Whole list printed out *)
```

Note the use here of global variables and thus global state. Exceptions and global variables are the two big 'impure' (i.e. non-functional) features of ML.