Programming ~~Paradigms~~ Languages F28PL, Lecture 2
More on ML

Jamie Gabbay

November 9, 2015

Declarations

Associate an identifier (variable) with a value.

```
- val identifier = expression;
> val identifier = value : type
```

An identifier is a sequence of letters, digits and _, starting with a letter. Use the identifier in subsequent expressions to return the value bound to it.

```
> val FALSE = false;
val FALSE = false : bool
> val ascii_a = ord #"a";
val ascii_a = 97 : int
> val identity = (fn x => x);
val identity = fn : 'a -> 'a
> val _asciia = ord #"a";
Error-Constructor in a pattern was not an identifier
   Found near (_)(asciia)
> (FALSE, identity, ascii_a);
val it = (false, fn, 97) : bool * ('a -> 'a) * int
```

Declarations

Can't assign new value to declared variable, but can redeclare variable by entering new definition.

```
- val toPound = 0.68;
> val toPound = 0.68 : real
- val toPound = 0.69;
> val toPound = 0.69 : real
```

Creates a new instance of `toPound`. Does not reassign existing variable.

This matters. For instance ...

New instances $\neq$ assignment

```
> type dollar = real;
type dollar (* type alias *)
> type pound = real;
type pound
> val toPound = 0.68;
val toPound = 0.68 : real  (* conversion rate *);
> val dollartopound = fn (x:dollar) => (toPound*x):pound;
val dollartopound = (* function ; see next slide *)
   fn : dollar -> pound (* conversion function *)
> dollartopound 1.0;
val it = 0.68 : pound
> val toPound = 0.69;
val toPound = 0.69 : real
> dollartopound 1.0;
val it = 0.68 : pound (* Still bound to old value! *)
> val dollartopound = fn x => toPound*x;
val dollartopound = fn : dollar -> pound
   (* So reassign. *)
> dollartopound 1.0;
val it = 0.69 : pound; (* Phew! *)
```

Function

A function maps from

- a domain type (the parameter of the function) to
- a range type (the result of the function).

Function type is displayed as

```
fn : domain type -> range type
```

Example functions

can display types of prefix system functions

```
- size;
> val it = fn : string -> int
(* takes a string and returns an int *)
- not;
> val it = fn : bool -> bool
(* takes a bool and returns a bool *)
- dollartopound;
> val it = fn : dollar -> pound
(* takes dollars and returns pounds *)
```

Higher-order functions possible (more on this later!):

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

Defining functions

Two ways to define functions:

- Nameless style with `fn`, and
- named style with `fun`.

Thus:

```
fn pattern => expression;
fun ident pattern => expression;
(* or equivalently *)
val ident = fn pattern => expression;
```

Example function definitions

For example:

```
fn x => x;                    (* nameless *)
fun identity x = x;           (* named *)
val identity = fn x => x;     (* named *)
(* *)
fn x => toPound*x;            (* nameless *)
val dollartopound = fn x => toPound*x;
fun dollartopound x = topound*x;
```

Multiple arguments are possible:

```
val application = fn f => fn x => f x;
fun application f x = f x;
> application;
val it = fn : ('a -> 'b) -> 'a -> 'b
```

Example function definition

```
val application = fn f => fn x => f x;
fun application f x = f x;
> application;
val it = fn : ('a -> 'b) -> 'a -> 'b
```

You are expected to understand that we can view `application` in two ways:

- as a function taking two arguments, one in `'a -> 'b` and one in `'a`, and returning a result in `'b`, or
- as a function taking one argument in `'a -> 'b` and returning a function in `'a -> 'b`.

Type inference

ML uses polymorphic type inference (the Hindley-Milner type inference algorithm) to deduce types from use of operators/values of known type. For instance, increment an integer:

```
- fun inc x = x+1;
> val it = fn : int -> int
```

- `1` has type `int` so
- `+` must be `int` addition so
- `x` must have type `int`.

You are expected to understand how this works & be able to infer types by hand.

Don't say "`1` is `int`", say "`1` has type `int`". It makes thinking easier.

Example function definition

Append "s" to a string:

```
- fun plural w = w^"s";
> val plural = fn : string -> string
```

Type inference: `^` has type `(string * string) -> string`.

```
> op ^;
val it = fn : string * string -> string
```

(We write `op ^` because `^` has binary infix notation; `op` says 'give me the function associated to this infix operator'.)

It follows that `w` must have type `string` (and `"s"` must be string, which it is), and so must the result of the calculation.

You are expected to type `sSize`, just by looking at it:

```
> size;
val it = fn : string -> int
> fun sSize s = (s,size s);
```

Explicit types

You can give the compiler (or the reader of the program) type hints. These explicit type annotations can go anywhere:

```
> fn (x:int) => x*x;
val it = fn : int -> int
> fn (x:real) => x*x;
val it = fn : real -> real
> fn x => (x*x):int;
val it = fn : int -> int
> fn x => (x*x):real;
val it = fn : real -> real
> fn x => (x:int)*x;
val it = fn : int -> int
```

Explicit types

Have fun giving contradictory type annotations and watching the algorithm cry!

```
> fn x => (x:int)*(x:real);
Error-Can't unify int with real (Different type constructors)
Found near x => *( x : int, x : real)
```

Type annotations can be polymorphic. Quite subtle restrictions can be expressed:

```
fn (x:int,y) => (x,y);
val it = fn : int * 'a -> int * 'a
fn (x:''a,y) => (x,y); (* Apply to (1,1.0) and (1.0,1) *)
val it = fn : ''a * 'b -> ''a * 'b
fn (x:'a,y:'a) => (x,y);
val it = fn : 'a * 'a -> 'a * 'a
```

How does a function call work?

Must call `type1 -> type2` function with a `type1` argument to return a `type2` result.

Suppose `exp2:type1 -> type2` and `exp1:type1`. If we ask the system to evaluate `exp2 exp1` (`exp2` applied to `exp1`) then the system performs the following steps:

- Evaluate `exp2` to calculate some function value `(fn x => val2) : type1 -> type2`.
- Evaluate `exp1` to calculate some value `val1:type1` (note: `type1` may itself be a function type; doesn't matter).
- Bind `x` to `val1`, and evaluate `val2` to `val2':type2` in this environment.
- Return `val2'`.

In effect, `(fn x => e) s` evaluates $e[x \mapsto s]$ (e with x substituted for s). Cf. $(\lambda a.e)s$ in the lambda-calculus.

Multiple arguments & patterns

Best explained with examples:

```
fun sumsq x y = x*x + y*y; (* is shorthand for *)
val sumsq = fn x => fn y => x*x + y*y;
(* alternatively we can use a pattern *)
fun sumsq (x,y) (* <- a pattern *) = x*x + y*y;
val sumsq = fn (x,y) => x*x + y*y;
```

Patterns can be nested and get arbitrarily complex. Here's my best effort to create a complex pattern:

```
fun sumsqlist [] = 0
  | sumsqlist (x,y)::tl = (x*x+y*y)+(sumsqlist tl);
```

Try your own!

Patterns

You can pattern-match on numbers and strings, and it behaves like a case or switch statement:

```
fun isZero 0 = true
  | isZero n = false;
isZero = fn : int -> bool
fun translate "va␣te␣faire␣foutre" = "thank␣you␣for␣coming"
  | translate "ta␣geule" = "hush,␣please"
  | translate x = x;
censor = fn : string -> string
```

Can't do arithmetic, or use other functions, or repeat variables:

```
fun decrement (n+1) = n;
Error
(* fun decrement n = n-1 works, though *)
fun censor x^"shit"^y = x^"sugar"^y;
Error
fun equals x x = true
  | equals x y = false;
Error
```

Patterns

You are expected to understand the execution model of ML. Left-to-right, top-down. This means that in the second clause below, we know n cannot be bound to zero:

```
fun isZero 0 = true
  | isZero n = false;
```

So if I ask you "Explain why `isZero 0` returns `false`. (2 points)", I want you to tell me that

- the first clause says so (1 point); and also
- the first clause is attempted before the second clause (1 point).

Wildcard pattern

The wildcard pattern _ matches anything and ignores it. Saves you the price of one variable name:

```
fun isZero 0 = true |
      isZero _ = false;
fun AND true true = true
  | AND _ _ = false;
(* shorter, faster, and clearer than this *)
fun AND true true = true
  | AND true false = false
  | AND false true = false
  | AND false false = false;
(* and easier to write than this *)
fun AND true true = true
  | AND x y = false;
```

This AND is non-sequential, unlike `andalso`; AND always evaluates both arguments, even if the first one evaluates to `false`.

Patterns & wildcards

Here is non-sequential conjunction (so it evaluates both arguments, unlike `andalso`):

```
fun AND false false = false
  | AND false true = false
  | AND true false = false
  | AND true true = true;
val AND = fn : bool -> bool -> bool
- AND true false;
> false : bool
(* true false doesn't match false false
true false doesn't match false true
true false matches true false ==> false *)
```

Patterns & wildcards

Again, using variables and wildcards:

```
fun AND true true = true
  | AND _ _ = false;
fun AND false true = false
  | AND true y = y;
fun AND true true = true
  | AND x y = false;
```

Above are all equivalent.

But this doesn't compile:

```
fun AND x x = x
  | AND x y = false;
```

Why?

Patterns & wildcards

Why?

Patterns must be linear; must mention each variable at most once.
The closest we can come to the spirit of the program above is this:

```
fun AND x y = if (x=y) then x else false;
```

This is funky . . . but arguably it's poor code, because it's unnecessarily funky.

Recursion

A function is recursive when it calls itself. Canonical example (thanks to Phil):

```
fun recursive_joke x = recursive_joke x;
val recursive_joke = fn : 'a -> 'b
fun long_recursive_joke x = long_recursive_joke x@x;
(* Dangerous!  Why? *)
val long_recursive_joke = fn : 'a -> 'b
```

Recursion is to functional programming as iteration is to imperative programming.

Typically:

- One or more base cases in which the function returns a value without calling itself.
- Some recursive cases in which the function calls itself (possibly more than once, and perhaps combining the results in interesting ways).

Recursions

Many interesting recursive functions; e.g.:

```
fun fib 0 = 1 (* Fibonacci sequence *)
  | fib 1 = 1
  | fib n = (fib(n-1))+(fib(n-2));
fun fact 0 = 1    (* Factorial *)
  | fact n = n*(fact(n-1));
fun logmap r 0 = 0.0  (* Logistic map; basis of chaos theory *
  | logmap r n = r*(logmap r (n-1))*(1.0-(logmap r (n-1)));
fun listupto f 0 = [f 0] (* List a function's values *)
  | listupto f n = (f n)::(listupto f (n-1));
fun listlogmap r = listupto (logmap r); (* List the logistic m
```

Exercise: What are the types of the functions above?

Exercise: Implement Bubblesort and Quicksort as recursive functions on lists.

Recursions

Recursion does not have to be over integers. However, integers are an excellent source of nontrivial examples, so let's stick to them.

```
fun sumto f 0 = 0.0  (* Summing a function over a range *)
  | sumto f n = (f n)+(sumto f (n-1));
val sum = sumto (fn x => x); (* sum n => 0+1+2+...+n *)
fun pow x 0 = 1.0 (* pow x n => x^n *)
  | pow x n = x*(pow x (n-1));
```

Then the following Taylor series for e^x

$$e^x = \sum_{k \geq 0} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

can be succinctly expressed in ML as an approximation function:

```
fun exp x = sumto (fn n => (pow x n)/(real (fact n)));
```

Question: why do we need the `real` in there?

Partial application

Consider this:

```
sumto (fn n => (pow 4.0 n)/(real (fact n)));
```

Let's look at the types:

```
> sumto (fn n => (pow 4.0 n)/(real (fact n)));
val it = fn : int -> real
> sumto;
val it = fn : (int -> real) -> int -> real
> fn n => (pow 4.0 n)/(real (fact n));
val it = fn : int -> real
```

Above, `sumto` is applied to a function to return . . . a function.

Partial application is when we apply a function of n parameters to $m \leq n$ arguments. This 'freezes' m arguments, and leaves $n - m$ arguments outstanding.

Partial application

```
- fun f a b c d = ...
> val f =
    fn : type1 -> type2 -> ... typeN
- val aval = ...;
> val aval = ...: type1
- val f' = f aval;
> val f' : type2 -> ... typeN
(* f' is like f with a frozen to the value of aval *)
```

Partial application

```
- fun power _ 0 = 1 |
      power x n = x*power x (n-1);
> val power = fn : int -> int  -> int
- val pow2 = power 2;
> val pow2 = fn : int -> int
(* pow2 is like power with x frozen to 2 *)
- pow2 3;
> 8 : int
pow2 3 ==> power 2 3 ==> ... 8
```

Partial application

```
- fun sJoin _ 0 = "" |
      sJoin s n = s^sJoin s (n-1);
> val sJoin = fn : string -> int -> string
- val stops = sJoin ".";
> val stops = fn : int -> string
stops is like sJoin with s frozen to "."
- stops 3;
> "..." : string
stops 3 ==> sJoin "." 3 ==> ... "..."
```

Partial application

I suggest that good practice when designing a function f of n arguments is to order the arguments from

- 'least likely to vary' to 'most likely to vary', or if you prefer
- 'arrives first' to 'arrives last'.

That way, we can conveniently partially apply f to its arguments 'as they arrive'.

I do this all the time when programming LaTeX macros, which are just functions. This can make a real difference.

Conditional expressions

The usual:

```
if exp1:bool then exp2:'a else exp3:'a

fun abs x = (* find absolute value of x *)
   if x>0 then x else ~x;
(* val abs = fn : int -> int; *)
fun max (x:int) y =
   if x>y then x else y;

datatype fingers = Little | Ring | Mid | Fore | Thumb;
fun isThumb x = if x=Thumb then true else false;
(* though simpler: *) fun isThumb x = (x=Thumb);
fun isZero x = if x=0 then true else false;
(* though simpler: *) fun isZero x = (x=0);
fun safeDiv x y = if y=0 then 0 else (x div y);
```

Question: why the explicit type annotation `x:int` above?

Conditional expressions

Following programs are in poor taste, like the brown corduroy suit that my maths teacher once wore to work (once, and we never let him forget it):

```
fun tastelessZero x = if (x=0)=true then true else false;
fun tastelessNot x = if (x=false)=true then true else false;
(* simplify to *)
fun tastefulZero x = if x=0 then true else false;
fun tastefulNot x = if x=false then true else false;
(* and in fact, these simplify further to *)
fun slickZero x = (x=0);
fun slickNot x = (x=false);
```

These latter programs are Keanu Reeves in a long leather coat and sunglasses; pointless, but nice to watch.