Patterns In Function Definitions

The following defines a predicate that tests whether a list, L is null (the predefined null function already does this).

```haskell
fun isNull L = 
  if L=[] then true 
  else false;
val isNull = fn : 'a list -> bool
```

However, we can decompose the definition using patterns to get a simpler and more elegant definition:

```haskell
fun isNull [] = true
| isNull(_::_) = false;
val isNull = fn : 'a list -> bool
```

The “|” divides the function definition into different argument patterns; no explicit conditional logic is needed. The definition that matches a particular actual parameter is automatically selected.

```haskell
fun fact(1) = 1
  | fact(n) = n*fact(n-1);
val fact = fn : int -> int
```

If patterns that cover all possible arguments aren't specified, you may get a run-time Match exception. If patterns overlap you may get a warning from the compiler.

```haskell
fun append([],L) = L
| append(hd::tl,L) = hd::append(tl,L);
val append = fn :
  'a list * 'a list -> 'a list
```

But a more precise decomposition is fine:

```haskell
fun append([],L) = L
  | append(hd::tl,L) = hd::append(tl,L)
  | append(L,[]) = L;
val append = fn :
  'a list * 'a list -> 'a list
```

If we add the pattern

```haskell
append(L,[]) = L
```
we get a redundant pattern warning (Why?)

```haskell
fun append([],L) = L
  | append(hd::tl,L) = hd::append(tl,L)
  | append(L,[]) = L;
stdIn:151.1-153.20 Error: match redundant
  (nil,L) => ...
  (hd :: tl,L) => ...
  --> (L,nil) => ...
```

But a more precise decomposition is fine:

```haskell
fun append([],L) = L
  | append(hd::tl,hd2::tl2) = hd::append(tl,hd2::tl2)
  | append(hd::tl,[]) = hd::tl;
val append = fn :
  'a list * 'a list -> 'a list
```
Function Types Can be Polytypes

Recall that 'a, 'b, ... represent type variables. That is, any valid type may be substituted for them when checking type correctness.

ML said the type of append is

```ml
val append = fn :
'a list * 'a list -> 'a list
```

Why does 'a appear in three places?

We can define eitherNull, a predicate that determines whether either of two lists is null as

```ml
fun eitherNull(L1,L2) =
null(L1) orelse null(L2);
```

val eitherNull =

```ml
fn : 'a list * 'b list -> bool
```

Why are both 'a and 'b used in eitherNull's type?

Currying

ML chooses the most general (least-restrictive) type possible for user-defined functions.

Functions are first-class objects, as in Scheme.

The function definition

```ml
fun f x y = expression;
```

defines a function f (of x) that returns a function (of y).

Reducing multiple argument functions to a sequence of one argument functions is called currying (after Haskell Curry, a mathematician who popularized the approach).

Thus

```ml
fun f x y = x :: [y];
val f = fn : 'a -> 'a -> 'a list
```

says that f takes a parameter x, of type 'a, and returns a function (of y, whose type is 'a) that returns a list of 'a.

Contrast this with the more conventional

```ml
fun g (x,y) = x :: [y];
val g = fn : 'a * 'a -> 'a list
```

Here g takes a pair of arguments (each of type 'a) and returns a value of type 'a list.

The advantage of currying is that we can bind one argument and leave the remaining argument(s) free.

For example

```ml
f(1);
```

is a legal call. It returns a function of type

```ml
fn : int -> int list
```

The function returned is equivalent to

```ml
fun h b = 1 :: [b];
val h = fn : int -> int list
```
Map Revisited

ML supports the map function, which can be defined as

\[
\text{fun map}(f, [\]) = [\]
\|
\text{map}(f, x::y) =
(f \, x) \, :: \, \text{map}(f, y);
\]

val map =
fn : ('a -> 'b) * 'a list -> 'b list

This type says that map takes a pair of arguments. One is a function from type \('a\) to type \('b\). The second argument is a list of type \('a\). The result is a list of type \('b\).

In curried form map is defined as

\[
\text{fun map} \, f \, [\] = [\]
\|
\text{map} \, f \, (x::y) =
(f \, x) \, :: \, \text{map} \, f \, y;
\]

val map =
fn : ('a -> 'b) -> 'a list -> 'b list

This type says that map takes one argument that is a function from type \('a\) to type \('b\). It returns a function that takes an argument that is a list of type \('a\) and returns a list of type \('b\).

The advantage of the curried form of map is that we can now use map to create “specialized” functions in which the function that is mapped is fixed.

For example,

val neg = map not;
val neg =
fn : bool list -> bool list
neg [true,false,true];
val it = [false,true,false] : bool list

Power Sets Revisited

Let’s compute power sets in ML.

We want a function \texttt{pow} that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

\[
\text{pow} \, [1,2] = \, [[1,2],[1],[2],[\]]
\]

We first define a version of \texttt{cons} in curried form:

\[
\text{fun cons} \, h \, t = h::t;
\]

val cons = fn :
\ 'a -> 'a list -> 'a list

Now we define \texttt{pow}. We define the powerset of the empty list, [\], to be [[]]. That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of \(h::t\), we compute the power set of \(t\), which we call \texttt{pset}. Then the power set for \(h::t\) is just \(h\) distributed through \texttt{pset} appended to \texttt{pset}.

We distribute \(h\) through \texttt{pset} very elegantly: we just map the function \((\text{cons} \, h)\) to \texttt{pset}. \((\text{cons} \, h)\) adds \(h\) to the head of any list it is given. Thus mapping \((\text{cons} \, h)\) to \texttt{pset} adds \(h\) to all lists in \texttt{pset}.
The complete definition is simply
\[
\text{fun pow \[1\] = [[]]} \\
\text{| pow (h::t) =} \\
\text{let} \\
\text{val pset = pow t} \\
\text{in} \\
\text{(map (cons h) pset) @ pset} \\
\text{end;} \\
\text{val pow =} \\
\text{fn : 'a list -> 'a list list}
\]
Let’s trace the computation of pow \[1,2\].
Here \(h = 1\) and \(t = [2]\). We need to compute pow \[2\].
Now \(h = 2\) and \(t = []\).
We know pow \[1\] = [[]],
so pow \[2\] =
\[
(\text{map (cons 2) [[]]} \)@[] = \\
( [[]] )@[] = [2,[]]
\]
Therefore pow \[1,2\] =
\[ (\text{map (cons 1) []} \)@[] = \\
[]@[] = []@[] = [[]]
\]

Composing Functions
We can define a composition function that composes two functions into one:
\[
\text{fun comp \( (f,g) \) \( (x) = f(g(x)) \);} \\
\text{val comp = fn :} \\
\text{('a -> 'b) * ('c -> 'a) ->} \\
\text{'c -> 'b}
\]
In curried form we have
\[
\text{fun comp \( f,g \) \( x = f(g(x)) \);} \\
\text{val comp = fn :} \\
\text{('a -> 'b) ->} \\
\text{('c -> 'a) -> 'c -> 'b}
\]
For example,
\[
\text{fun sqr \( x : \text{int} = x * x \);} \\
\text{val sqr = fn : int -> int} \\
\text{comp sqr sqr} \\
\text{val it = fn : int -> int}
\]
comp sqr sqr 3;
\text{val it = 81 : int}
In SML \( o \) (lower-case \( O \)) is the infix composition operator.
Hence
\[
\text{sqr o sqr} \equiv \text{comp sqr sqr}
\]
Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses.

That notation is

\[ \text{fn arg => body;} \]

For example,

\[ \text{val sqr = fn x : int => x*x;} \]
\[ \text{val sqr = fn : int -> int} \]

In fact the notation used to define functions,

\[ \text{fun name arg = body;} \]

is actually just an abbreviation for the more verbose

\[ \text{val name = fn arg => body;} \]

An anonymous function can be used wherever a function value is needed. For example,

\[ \text{map (fn x => [x]) [1,2,3];} \]
\[ \text{val it =} \]
\[ \text{[[1],[2],[3]] : int list list} \]

We can use patterns too:

\[ \text{(fn [] => [])} \]
\[ \text{| (h::t) => h::h::t);} \]
\[ \text{val it = fn : 'a list -> 'a list} \]

(What does this function do?)