A Second Look At ML

Outline

- Patterns
- Local variable definitions
- A sorting example
Two Patterns You Already Know

- We have seen that ML functions take a single parameter:
  \[
  \text{fun } f \ n = n*n;
  \]
- We have also seen how to specify functions with more than one input by using tuples:
  \[
  \text{fun } f \ (a, b) = a*b;
  \]
- Both \( n \) and \( (a, b) \) are patterns. The \( n \) matches and binds to any argument, while \( (a, b) \) matches any 2-tuple and binds \( a \) and \( b \) to its components.

Underscore As A Pattern

- \[
  \text{fun } f _ = "yes";
  \]
- \[
  \text{val } f = \text{fn} : \text{'}a -> \text{string}
  \]
- \[
  f 34.5;
  \]
- \[
  \text{val } \text{it} = "yes" : \text{string}
  \]
- \[
  f [];
  \]
- \[
  \text{val } \text{it} = "yes" : \text{string}
  \]

- The underscore can be used as a pattern
- It matches anything, but does not bind it to a variable
- Preferred to:
  \[
  \text{fun } f \ x = "yes";
  \]
Constants As Patterns

- fun f 0 = "yes";

Warning: match nonexhaustive
  0 => ...
val f = fn : int -> string
- f 0;
val it = "yes" : string

- Any constant of an equality type can be used as a pattern
- But not:

  fun f 0.0 = "yes";

Non-Exhaustive Match

- In that last example, the type of f was int -> string, but with a “match non-exhaustive” warning
- Meaning: f was defined using a pattern that didn’t cover all the domain type (int)
- So you may get runtime errors like this:

  - f 0;
  val it = "yes" : string
  - f 1;
  uncaught exception nonexhaustive match failure
Lists Of Patterns As Patterns

- fun f [a,_] = a;
  Warning: match nonexhaustive
  a :: _ :: nil => ...
  val f = fn : 'a list -> 'a
- f ['#"f"','#"g"'];
  val it = '#"f"' : char

- You can use a list of patterns as a pattern
- This example matches any list of length 2
- It treats `a` and `_` as sub-patterns, binding `a` to the first list element

Cons Of Patterns As A Pattern

- fun f (x::xs) = x;
  Warning: match nonexhaustive
  x :: xs => ...
  val f = fn : 'a list -> 'a
- f [1,2,3];
  val it = 1 : int

- You can use a cons of patterns as a pattern
- `x::xs` matches any non-empty list, and binds `x` to the head and `xs` to the tail
- Parens around `x::xs` are for precedence
ML Patterns So Far

- A variable is a pattern that matches anything, and binds to it
- A `_` is a pattern that matches anything
- A constant (of an equality type) is a pattern that matches only that constant
- A tuple of patterns is a pattern that matches any tuple of the right size, whose contents match the sub-patterns
- A list of patterns is a pattern that matches any list of the right size, whose contents match the sub-patterns
- A cons (::) of patterns is a pattern that matches any non-empty list whose head and tail match the sub-patterns

Multiple Patterns for Functions

```ml
val f = fn : int -> string
- f 1;
val it = "one" : string
```

You can define a function by listing alternate patterns
Syntax

\[
\text{fun-def} ::= \text{fun} \ <\text{fun-bodies}> ; \\
<\text{fun-bodies}> ::= <\text{fun-body}> \\
\quad | \ <\text{fun-body}> \ '| ' <\text{fun-bodies}> \\
<\text{fun-body}> ::= <\text{fun-name}> <\text{pattern}> = <\text{expression}>
\]

- To list alternate patterns for a function
- You must repeat the function name in each alternative

Overlapping Patterns

- fun f 0 = "zero"
  = | f _ = "non-zero";
val f = fn : int -> string;
- f 0;
val it = "zero" : string
- f 34;
val it = "non-zero" : string

- Patterns may overlap
- ML uses the first match for a given argument
Pattern-Matching Style

■ These definitions are equivalent:

\[
\begin{align*}
\text{fun } f \ 0 &= \ "zero" \\
| \ f \ \_ &= \ "non-zero"; \\
\text{fun } f \ n &= \\
\quad \text{if } n = 0 \ \text{then } \ "zero" \\
\quad \text{else } \ "non-zero";
\end{align*}
\]

■ But the pattern-matching style usually is preferred in ML

■ It often gives shorter and more legible functions

Pattern-Matching Example

Original (from Chapter 5):

\[
\begin{align*}
\text{fun } \text{fact} \ n &= \\
\quad \text{if } n = 0 \ \text{then } 1 \ \text{else } n \times \text{fact}(n-1);
\end{align*}
\]

Rewritten using patterns:

\[
\begin{align*}
\text{fun } \text{fact} \ 0 &= 1 \\
| \ \text{fact} \ n &= n \times \text{fact}(n-1);
\end{align*}
\]
Pattern-Matching Example

Original (from Chapter 5):

```plaintext
fun reverse L = 
  if null L then nil 
  else reverse(tl L) @ [hd L];
```

Improved using patterns:

```plaintext
fun reverse nil = nil 
| reverse (first::rest) = 
  reverse rest @ [first];
```

More Examples

This structure occurs frequently in recursive functions that operate on lists: one alternative for the base case (nil) and one alternative for the recursive case (first::rest).

Adding up all the elements of a list:

```plaintext
fun f nil = 0 
| f (first::rest) = first + f rest;
```

Counting the true values in a list:

```plaintext
fun f nil = 0 
| f (true::rest) = 1 + f rest 
| f (false::rest) = f rest;
```
More Examples

Making a new list of integers in which each is one greater than in the original list:

```
fun f nil = nil
| f (first::rest) = first+1 :: f rest;
```

A Restriction

- You can't use the same variable more than once in the same pattern
- This is not legal:

  ```
  fun f (a,a) = ... for pairs of equal elements
  | f (a,b) = ... for pairs of unequal elements
  ```

- You must use this instead:

  ```
  fun f (a,b) = 
  if (a=b) then ... for pairs of equal elements
  else ... for pairs of unequal elements
  ```
The **polyEqual** Warning

```plaintext
- fun eq (a,b) = if a=b then 1 else 0;
Warning: calling polyEqual
val eq = fn : ''a * ''a -> int
- eq (1,3);
val it = 0 : int
- eq ("abc","abc");
val it = 1 : int
```

- Warning for an equality comparison, when the runtime type cannot be resolved
- OK to ignore: this kind of equality test is inefficient, but can’t always be avoided

Patterns Everywhere

```plaintext
- val (a,b) = (1,2.3);
val a = 1 : int
val b = 2.3 : real
- val a::b = [1,2,3,4,5];
Warning: binding not exhaustive
  a :: b = ...
val a = 1 : int
val b = [2,3,4,5] : int list
```

- Patterns are not just for function definition
- Here we see that you can use them in a **val**
- More ways to use patterns, later
Local Variable Definitions

- When you use `val` at the top level to define a variable, it is visible from that point forward.
- There is a way to restrict the scope of definitions: the `let` expression

\[
\text{<let-exp> ::= let <definitions> in <expression> end}
\]
Example with `let`

- `let val x = 1 val y = 2 in x+y end;`
- `val it = 3 : int;`
- `x;`
  
  Error: unbound variable or constructor: x

- The value of a `let` expression is the value of the expression in the `in` part
- Variables defined with `val` between the `let` and the `in` are visible only from the point of declaration up to the `end`

Proper Indentation for `let`

```plaintext
let
  val x = 1
  val y = 2
in
  x+y
end
```

- For readability, use multiple lines and indent `let` expressions like this
- Some ML programmers put a semicolon after each `val` declaration in a `let`
Long Expressions with `let`

```haskell
fun days2ms days = 
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

- The `let` expression allows you to break up long expressions and name the pieces
- This can make code more readable

Patterns with `let`

```haskell
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) = 
  let
    val (x, y) = halve cs
  in
    (a::x, b::y)
  end;
```

- By using patterns in the declarations of a `let`, you can get easy “deconstruction”
- This example takes a list argument and returns a pair of lists, with half in each
Again, Without Good Patterns

```plaintext
let
    val halved = halve cs
    val x = #1 halved
    val y = #2 halved
in
    (a::x, b::y)
end;
```

- In general, if you find yourself using `#` to extract an element from a tuple, think twice
- Pattern matching usually gives a better solution

---

**halve At Work**

```plaintext
- fun halve nil = (nil, nil)
  = |  halve [a] = ([a], nil)
  = |  halve (a::b::cs) =
    =     let
    =     val (x, y) = halve cs
    =     in
    =     (a::x, b::y)
    =     end;
val halve = fn : 'a list -> 'a list * 'a list
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```
Outline

- Patterns
- Local variable definitions
- A sort example

Merge Sort

- The `halve` function divides a list into two nearly-equal parts
- This is the first step in a merge sort
- For practice, we will look at the rest
Example: Merge

fun merge (nil, ys) = ys
|   merge (xs, nil) = xs
|   merge (x::xs, y::ys) =
    if (x < y) then x :: merge(xs, y::ys)
    else y :: merge(x::xs, ys);

■ Merges two sorted lists
■ Note: default type for < is int

Merge At Work

- fun merge (nil, ys) = ys
= |   merge (xs, nil) = xs
= |   merge (x::xs, y::ys) =
=   if (x < y) then x :: merge(xs, y::ys)
=   else y :: merge(x::xs, ys);
val merge = fn : int list * int list -> int list
- merge ([2],[1,3]);
val it = [1,2,3] : int list
- merge ([1,3,4,7,8],[2,3,5,6,10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
Example: Merge Sort

```ml
fun mergeSort nil = nil
| mergeSort [a] = [a]
| mergeSort theList = 
  let
    val (x, y) = halve theList
  in
    merge(mergeSort x, mergeSort y)
  end;
```

- Merge sort of a list
- Type is `int list -> int list`, because of type already found for `merge`

Merge Sort At Work

```ml
val mergeSort = fn : int list -> int list
val it = fn : int list -> int list
- mergeSort [4,3,2,1];
val it = [1,2,3,4] : int list
- mergeSort [4,2,3,1,5,3,6];
val it = [1,2,3,3,4,5,6] : int list
```
Nested Function Definitions

- You can define local functions, just like local variables, using a `let`
- You should do it for helper functions that you don't think will be useful by themselves
- We can hide `halve` and `merge` from the rest of the program this way
- Another potential advantage: inner function can refer to variables from outer one (as we will see in Chapter 12)

```ml
(* Sort a list of integers. *)
fun mergeSort nil = nil
| mergeSort [e] = [e]
| mergeSort theList = let
  (* From the given list make a pair of lists
   * (x,y), where half the elements of the
   * original are in x and half are in y. *)
  fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a::b::cs) = let
    val (x, y) = halve cs
    in
    (a::x, b::y)
    end;

continued...
```
(* Merge two sorted lists of integers into * a single sorted list. *)

fun merge (nil, ys) = ys

| merge (xs, nil) = xs
| merge (x::xs, y::ys) =
|   if (x < y) then x :: merge(xs, y::ys)
|     else y :: merge(x::xs, ys);

val (x, y) = halve theList

in
  merge(mergeSort x, mergeSort y)
end;

Commenting

■ Everything between (* and *) in ML is a comment
■ You should (at least) comment every function definition, as in any language
  – what parameters does it expect
  – what function does it compute
  – how does it do it (if non-obvious)
  – etc.