CSE341, Fall 2011, Lecture 5 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

While the last lecture introduced the basics of datatype bindings and case expressions, this lecture:

- Gives a more precise semantics
- Demonstrates that lists and options are just (polymorphic) datatypes
- Shows that pattern-matching can also be used for each-of types
- Shows that val bindings and function arguments also use patterns, and in fact every function takes exactly one argument
- Patterns can be nested inside other patterns, which generalizes the definition of pattern-matching
- Gives examples showing the benefits of all these features

A precise definition of things so far

We can summarize what we know about datatypes and pattern matching from the previous lecture as follows: The binding

```
  datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

introduces a new type `t` and each constructor `Ci` is a function of type `ti->t`. One omits the "`of ti`" for a variant that "carries nothing." To "get at the pieces" of a `t` we use a case expression:

```
  case e of p1 => e1 | p2 => e2 | ... | pn => en
```

A case expression evaluates `e` to a value `v`, finds the first pattern `pi` that *matches* `e`, and evaluates `ei` to produce the result for the whole case expression. So far, patterns have looked like `Ci(x1,...,xn)` where `Ci` is a constructor of type `t1 * ... * tn -> t` (or just `Ci` if `Ci` carries nothing). Such a pattern matches a value of the form `Ci(v1,...,vn)` and binds each `xi` to `vi` for evaluating the corresponding `ei`.

Lists and options are datatypes

Because datatype definitions can be recursive, we can use them to create our own types for lists. For example, this binding works well for a linked list of integers:

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list
```

We can use the constructors `Empty` and `Cons` to make values of `my_int_list` and we can use case expressions to use such values:

```
val one_two_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append_mylist (l1,l2) =
    case l1 of
        Empty => l2
      | Cons(hd,tl) => Cons(hd, append_mylist(tl,l2))
```

It turns out the lists and options "built in" (i.e., predefined with some special syntactic support) are just datatypes. As a matter of style, it is better to use the built-in widely known feature than to invent your own. More importantly, it is better style to use pattern-matching for accessing list and option values, ***not*** the functions `null`, `hd`, `tl`, `isSome`, and `valOf` we saw previously. (We used them because we had not learned pattern-matching yet and we didn't want to delay practicing our functional-programming skills.)

For options, all you need to know is `SOME` and `NONE` are constructors, which we use to create values (just like before) and in patterns to access the values. Here is a short example of the latter:

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
```

The story for lists is similar with a few convenient syntactic peculiarities: `[]` really is a constructor that carries nothing and `::` really is a constructor that carries two things, but `::` is unusual because it is an infix operator (it is placed between its two operands), both when creating things and in patterns:

```
fun sum_list intlist =
    case intlist of
        [] => 0
      | head::tail => head + sum_list tail

fun append (l1,l2) =
    case l1 of
        [] => l2
      | head::tail => head :: append(tail,l2)
```

Notice here `head` and `tail` are nothing but local variables introduced via pattern-matching. We can use any names for the variables we want. We could even use `hd` and `tl` — doing so would simply shadow the functions predefined in the outer environment.

The reasons why you should usually prefer pattern-matching for accessing lists and options instead of functions like `null` and `hd` is the same as for datatype bindings in general: you can't forget cases, you can't apply the wrong function, etc. So why does the ML environment predefine these functions if the approach is inferior? In part, because they are useful for passing as arguments to other functions, a major topic still a couple lectures ahead of us.

Other than the strange syntax of `[]` and `::`, the only thing that distinguishes the built-in lists and options from our example datatype bindings is that the built-in ones are *polymorphic* – they can be used for carrying values of *any* type, as we have seen with `int list`, `int list list`, `(bool * int) list`, etc. You can do this for your own datatype bindings too, and indeed it is very useful. While we will not focus on using this feature (i.e., you're not responsible for knowing how to do it), there is nothing very complicated about it. For example, this is *exactly* how options are defined in the built-in environment:

```
datatype 'a option = NONE | SOME of 'a
```

Pattern-matching for each-of types, and the truth about val-bindings and function arguments

So far we have used pattern-matching for one-of types, but we can use them for each-of types also. Given a record value `{f1=v1,...,fn=vn}`, the pattern `{f1=x1,...,fn=xn}` matches and binds `xi` to `vi`. As you might expect, the order of fields in the pattern does not matter. As before, tuples are syntactic sugar for records, so the tuple value `(v1,...,vn)` matches the pattern `(x1,...,xn)`. So we could write this function for summing the three parts of an `int * int * int`:

```
fun sum_triple triple =
    case triple of
      (x,y,z) => z + y + x
```

And a similar example with records could look like this:

```
fun sum_stooges triple =
    case triple of
        {larry=x,curly=y,moe=z} => z + y + x
```

However, a case-expression with one branch is poor style — it just looks odd since the purpose of such expressions is to distinguish cases, plural. So how should we use pattern-matching for each-of types, when we know that a single pattern will definitely match so we are using pattern-matching just for the convenient extraction of values? It turns out you can use patterns in val-bindings too! So this approach is better style:

```
fun sum_stooges triple =
    let val {larry=x,curly=y,moe=z} = triple
    in
        x + y + z
    end
fun sum_triple triple =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

But actually we can do better still: Just like a pattern can be used in a val-binding to bind variables (e.g., `x`, `y`, and `z`) to the various pieces of the expression (e.g., `triple`), we can use a pattern when defining a function binding and the pattern will be used to introduce bindings by matching against the value passed when the function is called. So here is the third and best versions of our example functions:

```
fun sum_stooges {larry=x,curly=y,moe=z} =
    x + y + z
fun sum_triple (x,y,z) =
    x + y + z
```

This version of `sum_triple` should intrigue you: It takes a triple as an argument and uses pattern-matching to bind three variables to the three pieces for use in the function body. But it looks exactly like a function that takes three arguments of type `int`. Indeed, is the type `int*int*int->int` for three-argument functions or for one argument functions that take triples?

It turns out we have been basically lying: There is no such thing as a multi-argument function in ML: ***Every function in ML takes exactly one argument!*** Every time we write a multi-argument function, we are really writing a one-argument function that takes a tuple as an argument and uses pattern-matching to extract the pieces. This is such a common idiom that it is easy to forget about and it is totally fine to talk about "multi-argument functions" when discussing your ML code with friends. But in terms of the actual language definition, it really is a one-argument function: syntactic sugar for expanding out to the first version of `sum_triple` with a one-arm case expression.

This flexibility is sometimes useful. In languages like C and Java, you cannot have one function/method compute the results that are immediately passed to another multi-argument function/method. But with one-argument functions that are tuples, this works fine. Here is a silly example where we "rotate a triple to the right" by "rotating it to the left twice":

```
fun rotate_left (x,y,z) = (y,z,x)
fun rotate_right triple = rotate_left(rotate_left triple)
```

More generally, you can compute tuples and then pass them to functions you were thinking of as taking multiple arguments.

What about zero-argument functions? They do not exist either. The binding `fun f () = e` is using the unit-pattern `()` to match against calls that pass the unit value `()`, which as we know is the only value of type `unit`. The type unit is just a datatype with only one constructor, which takes no arguments and uses the unusual syntax `()`.

Digression: Type inference

By using patterns to access values of tuples and records rather than `#foo`, you will find it is no longer necessary to write types on your function arguments. In fact, it is conventional in ML to leave them off — you can always use the REPL to find out a function's type. The reason we needed them before is that `#foo` does not give enough information to type-check the function because the type-checker does not know what other fields the record is supposed to have, but the record/tuple patterns introduced above provide this information.

Digression: Multiple cases in a function binding

So far, we have seen pattern-matching on one-of types in case expressions and each-of types in case expressions (poor style) and val/function bindings (good style, in fact, you have been doing it since lecture 2 without knowing it). But is there a way to use one-of patterns in val/function bindings? This seems like a bad idea since we need multiple possibilities. But it turns out ML has special syntax for doing this in function definitions. Here are two examples, one for our own datatype and one for lists:

```
datatype exp = Constant of int | Negate of exp | Add of exp * exp | Multiply of exp * exp

fun eval (Constant i) = i
  | eval (Negate e2) = ~ (eval e2)
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Multiply(e1,e2)) = (eval e1) * (eval e2)

fun append ([],lst) = lst
  | append (head::tail,lst2) = head :: append(tail,lst2)
```

As a matter of *taste*, your instructor has never liked this style very much, and you have to get parentheses in the right places. But it is common among ML programmers and the textbook uses it, so you are welcome to as well. As a matter of *semantics*, it is just syntactic sugar for a single function body that is a case expression:

```
fun eval e =
    case e of
        Constant i => i
      | Negate e2  => ~ (eval e2)
      | Add(e1,e2) => (eval e1) + (eval e2)
      | Multiply(e1,e2) => (eval e1) * (eval e2)

fun append e =
    case e of
       ([],lst) => lst
     | (head::tail,lst2) => head :: append2(tail,lst2)
```

In general, the syntax

```
fun f p1 = e1
|   f p2 = e2
...
|   f pn = en
```

is just syntactic sugar for:

```
fun f x =
   case x of
     p1 => e1
   | p2 => e2
...
   | pn => en
```

The `append` example is our first example of *nested patterns*: each branch matches a pair of lists, by putting patterns (e.g., `[]` or `head::tail`) inside other patterns. This is discussed more generally next.

Nested Patterns

It turns out the definition of patterns is recursive: anywhere we have been putting a variable in our patterns, we can instead put another pattern. Roughly speaking, the semantics of pattern-matching is that the value being matched must have the same “shape” as the pattern and variables are bound to the “right pieces.” (This is very hand-wavy explanation which is why a precise definition is described below.) For example, the pattern `a::(b::(c::d))` would match any list with at least 3 elements and it would bind `a` to the first element, `b` to the second, `c` to the third, and `d` to the list holding all the other elements (if any). The pattern `a::(b::(c::[]))` on the other hand, would match only lists with exactly three elements. Another nested patterns is `(a,b,c)::d`, which matches any non-empty list of triples, binding `a` to the first component of the head, `b` to the second component of the head, `c` to the third component of the head, and `d` to the tail of the list.

In general, pattern-matching is about taking a value and a pattern and (1) deciding if the pattern matches the value and (2) if so, binding variables to the right parts of the value. Here are some key parts to the elegant recursive definition of pattern matching:

- A variable pattern (`x`) matches any value `v` and introduces one binding (from `x` to `v`).
- The pattern `C` matches the value `C`, if `C` is a constructor that carries no data.
- The pattern `C p` where `C` is a constructor and `p` is a pattern matches a value of the form `C v` (notice the constructors are the same) if `p` matches `v` (i.e., the nested pattern matches the carried value). It introduces the bindings that `p` matching `v` introduces.
- The pattern `(p1,p2,...,pn)` matches a tuple value `(v1,v2,...,vn)` if `p1` matches `v1` and `p2` matches `v2`, ..., and `pn` matches `vn`. It introduces all the bindings that the recursive matches introduce.
- (A similar case for record patterns of the form `{f1=p1,...,fn=pn}` ...)

This recursive definition extends our previous understanding in two interesting ways. First, for a constructor `C` that carries multiple arguments, we do not have to write patterns like `C(x1,...,xn)` though we often do. We could also write `C x`; this would bind `x` to the tuple that the value `C(v1,...,vn)` carries. What is really going on is that all constructors take 0 or 1 arguments, but the 1 argument can itself be a tuple. So

`C(x1,...,xn)` is really a nested pattern where the `(x1,...,xn)` part is just a pattern that matches all tuples with n parts. Second, and more importantly, we can use nested patterns instead of nested case expressions when we want to match only values that have a certain "shape."

There are additional kinds of patterns as well. Sometimes we do not need to bind a variable to part of a value. For example, consider this function for computing a list's length:

```
fun len lst =
    cast lst of
       [] => 0
       x::tail => 1 + len tail
```

We do not use the variable `x`. In such cases, it is better style not to introduce a variable. Instead, the *wildcard pattern* `_` matches everything (just like a variable pattern matches everything), but does not introduce a binding. So we should write:

```
fun len lst =
    cast lst of
       [] => 0
       _::tail => 1 + len tail
```

In terms of our general definition, wildcard patterns are straightforward:

- A wildcard pattern (`_`) matches any value `v` and introduces no bindings.

Lastly, you can use integer constants in patterns. For example, the pattern `37` matches the value `37` and introduces no bindings.

Examples of Nested Patterns

An elegant example of using nested patterns rather than an ugly mess of nested case-expressions is "zipping" or "unzipping" lists (three of them in this example):

```
exception BadTriple

fun zip3 list_triple =
    case list_triple of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise BadTriple

fun unzip3 lst =
    case lst of
        [] => ([],[],[])
      | (a,b,c)::tl => let val (l1,l2,l3) = unzip3 tl
                       in
                           (a::l1,b::l2,c::l3)
                       end
```

This example checks that a list of integers is sorted:

```
fun nondecreasing intlist =
    case intlist of
        [] => true
      | x::[] => true
      | head::(neck::rest) => (head <= neck andalso nondecreasing (neck::rest))
```

It is also sometimes elegant to compare two values by matching against a pair of them. This example, for determining the sign that a multiplication would have without performing the multiplication, is a bit silly but demonstrates the idea:

```
datatype sgn = P | N | Z

fun multsign (x1,x2) =
  let fun sign x = if x=0 then Z else if x>0 then P else N
  in
      case (sign x1,sign x2) of
          (Z,_) => Z
        | (_,Z) => Z
        | (P,P) => P
        | (N,N) => P
        | _     => N (* many say bad style; I am okay with it *)
  end
```

The style of this last case deserves discussion: When you include a "catch-all" case at the bottom like this, you are giving up any checking that you did not forget any cases: after all, it matches anything the earlier cases did not, so the type-checker will certainly not think you forgot any cases. So you need to be extra careful if using this sort of technique and it is probably less error-prone to enumerate the remaining cases (in this case `(N,P)` and `(P,N)`). That the type-checker will then still determine that no cases are missing is useful and non-trivial since it has to reason about the use `(Z,_)` and `(_,Z)` to figure out that there are no missing possibilities of type `sgn * sgn`.