1 Deep pattern matching

Earlier, we said that a pattern matches a value if the pattern “looks like” the value. Let’s make that more precise and, along the way, learn about fancier kinds of pattern matching.

First, it turns out that anywhere we have been putting a variable in patterns, we can instead put another pattern. So patterns nest—much like expressions nest. We can match against values deep inside of expressions. Examples:

- Pattern `a::(b::(c::d))` would match any list with at least three elements and would bind `a` to the first element, `b` to the second, `c` to the third, and `d` to the list of all the other elements (if any).
- Pattern `a::(b::(c::[]))`, however, would match only lists with exactly three elements.
- Pattern `(a,b,c)::d` would match 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.

Here is a more precise definition of pattern matching than our previous intuition of “looks like”:

- Constructor pattern `C` matches value `C`, assuming `C` is a constructor that carries no data. It introduces no bindings.
- Constructor pattern `C p`, where `C` is a constructor and `p` is a pattern, matches value `C v` (note: the constructors are the same) if `p` matches `v`—i.e., the nested pattern matches the carried value. It introduces any bindings that matching `p` against `v` introduces.
- Pattern `(p1,p2,...,pn)` matches tuple value `(v1,v2,...,vn)` if `pi` matches `vi`. It introduces all the bindings that the recursive matches introduce.
- Pattern `{f1=p1,f2=p2,...,fn=pn}` matches record value `{f1=v1,f2=v2,...,f2=vn}` if `pi` matches `vi`. It introduces all the bindings that the recursive matches introduce.
- Variable pattern `x` matches any value `v` and introduces a binding from `x` to `v`.
- Value pattern `v` matches exactly value `v` and introduces no bindings. Note that the first first item in this list (constructor pattern `C`) is just a special case of this item.
- Wildcard pattern `_` matches any value and introduces no bindings.

This recursive definition of pattern matching extends our previous understanding in two interesting ways. First, if a constructor `C` carries multiple values, we could either write pattern `C(x1,...,xn)`, or we could instead write pattern `C x`; this would bind `x` to the tuple `(v1,...,vn)` that 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.

Wildcard patterns can be useful in deep pattern matching. For example, consider this function for computing a list’s length:
fun len lst =  
cast lst of  
  [] => 0  
  x::tail => 1 + len tail

Variable \(x\) is not used in its branch expression. In such cases, it is better style not to introduce a variable. Instead, we should write:

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

2 Examples of nested patterns

Here’s an elegant example of using nested patterns, rather than an ugly mess of nested case-expressions: “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 determines the sign that a multiplication would have without performing the multiplication; it’s a bit silly but it 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
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.

It probably would be better to enumerate the remaining cases in this example:

  ... 
  | (N, P) => N
  | (P, N) => N

The type-checker can then determine that no cases are missing. That’s non-trivial, because it has to reason about the use \((Z, _)\) and \((_, Z)\) to figure out that there are no missing possibilities of type \(\text{sgn} \times \text{sgn}\).

3 Exceptions

One of our examples above used exceptions, and we’ve seen them crop up before, too. Let’s talk about them in more detail.

Exceptions can be raised (aka thrown) with the raise expression. For example, the \texttt{hd} function in the standard library raises the \texttt{Empty} exception when called with \texttt{[]}:

\begin{verbatim}
fun hd xs =
  case xs of
    [] => raise Empty
  | x::xs => x
\end{verbatim}

You can create your own kinds of exceptions with an exception binding. Exceptions can optionally carry values with them, which let the code raising the exception provide more information:

exception MyException
exception MyOtherException of int * int

In general, the syntax of an exception binding is as follows:

exception C [of t]

Here, the square brackets around \texttt{of t} indicates that part of the syntax is optional.

Exception bindings are very similar to constructors of a datatype binding. Indeed, exception constructors are functions (if they carry values) or values (if they don’t) that create values of type \texttt{exn} rather than the type of a datatype. So \texttt{Empty}, \texttt{MyException}, and \texttt{MyOtherException(3,9)} are all values of type \texttt{exn}, whereas \texttt{MyOtherException} has type \texttt{int*int->exn}. Usually we just use exception constructors as arguments to \texttt{raise}, such as \texttt{raise MyOtherException(3,9)}, but we can use them more generally to create values of type \texttt{exn}.
The type-checking rules for exception bindings are thus very similar to the type-checking rules for datatype bindings. Type-checking the exception binding causes `C` to be added to the current static environment with its type.

As an example, here is a function that returns the maximum element in a list of integers. Rather than return an `option` or raise a particular exception like `Empty` if called with `[]`, it takes an argument of type `exn` and raises it. So the caller can pass in the exception of its choice. (The type checker can infer that `ex` must have type `exn` because that is the type `raise` expects for its argument.)

```ml
fun maxlist (xs,ex) = 
    case xs of 
        [] => raise ex 
    | x::[] => x 
    | x::xs' => Int.max(x, maxlist(xs',ex))
```

Notice that calling `maxlist([3,4,0],Empty)` would not raise an exception; this call passes an exception value to the function, but the function never raises that value.

Another feature related to exceptions is `handling` (also known as `catching`) them. For this, ML has `handle` expressions. Their syntax is `e0 handle p1 => e1` where `e0` and `e1` are expressions and `p1` is a pattern. The type-checking rule is that if `e0` and `e1` both have type `t`, then the entire `handle` expression has type `t`. The evaluation rule—so long as no exceptions are thrown—is to evaluate `e0` to a value `v0` and return `v0` as the result of the `handle` expression. But if `e0` during its evaluation raises an exception matching `p1`, then `e1` is evaluated to a value `v1` and `v1` is returned as the result of the `handle` expression. And if `e1` raises an exception, then the `handle` expression also raises that exception.

More generally, `handle` expressions can have multiple branches each with a pattern and expression, syntactically separated by `|`. So the full syntax is

```ml
e0 handle p1 => e1 
    | p2 => e2 
    ... 
    | pn => en
```

In fact, every `handle` expression automatically has a final branch added by the compiler:

```ml
... 
    | exn => raise exn
```

That branch has the effect of raising any unhandled exceptions. So if `e0` raises an exception that does not match any `pi`, then the entire `handle` expression raises that exception.

4 The truth about functions

It turns out we’ve been lying to you about something for the last three weeks. There is no such thing as a multi-argument function in ML. Every function in ML takes exactly one argument!

Here’s an example:

```ml
fun sum_triple (x:int,y:int,z:int) = 
    x + y + z
```

Function `sum_triple` looks like it takes three arguments, and that’s how we’ve been describing it so far. But take another look at `(x:int,y:int,z:int)`: it’s really a pattern that matches against a triple and binds
the first part to \(x\), the second part to \(y\), and the third part to \(z\). You might ask, what about the types that are in the pattern? We haven’t seen those in patterns so far. It turns out that you can pretty much always add type annotations like this to ML code, and the compiler just uses them to help it do type inference. We could in fact write \texttt{sum\_triple} in the following completely equivalent way:

```ml
fun sum\_triple (x,y,z) = 
    x + y + z
```

And now \((x,y,z)\) looks exactly like patterns we saw in the last lecture.

So every time thought we were writing a multi-argument function, we were 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’s just fine to talk about “multi-argument functions” when discussing your ML code with friends. But in terms of the actual language definition, multi-argument functions are just syntactic sugar for one-argument functions that use pattern matching.

This flexibility is sometimes useful. For example, it makes it possible for a function to immediately pass its “multiple” results to another “multi-argument” function. Here is a silly example where we rotate a triple to the right by rotating it to the left twice:

```ml
fun rotate\_left (x,y,z) = (y,z,x) 
fun rotate\_right triple = rotate\_left(rotate\_left triple)
```

Try coding that up in Java or C and see if you can get something so concise! In ML, functions can compute tuples and then pass those tuples to other functions, in a way that only seem like passing multiple arguments.

### 5 Pattern matching with functions

Now that we know function arguments are really just a pattern, we might ask: can we use multiple patterns to match against function arguments? Yes we can.

Here’s how to write list append using function patterns:

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

The general syntax here is as follows:

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

And that code works exactly like the following code:

```ml
fun f x =
    case x of
      p1 => e1
    | p2 => e2
    | ...
    | pn => en
```
As a matter of style, some people have very strong opinions either way. Your instructor is more ambivalent: either can produce readable code that isn’t likely to contain errors, and that’s the most important thing.

You might wonder whether function patterns are just syntactic sugar for a case expression. It turns out that, although that would be an entirely reasonable choice in language design, the designers of ML instead made function patterns a primitive feature of the language.