1 Review of datatypes and case expressions

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

\[
\text{datatype } t = C_1 \text{ of } t_1 \mid C_2 \text{ of } t_2 \mid \ldots \mid C_n \text{ of } t_n
\]

introduces a new type \(t\) and each constructor \(C_i\) is a function of type \(t_i \rightarrow t\). We omit “of \(t_i\)” for a variant that carries nothing. To access the data carried by a variant \(t\) we use a case expression:

\[
\text{case } e \text{ of } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2 \mid \ldots \mid p_n \Rightarrow e_n
\]

The evaluation rules for a case expression are: evaluate \(e\) to a value \(v\), find the first pattern \(p_i\) that matches \(e\), and evaluate \(e_i\) to produce the result for the whole case expression.

So far, patterns have had the form \(C_i(x_1, \ldots, x_n)\) where \(C_i\) is a constructor of type \(t_1 \times \ldots \times t_n \rightarrow t\) (or just \(C_i\) if \(C_i\) carries nothing). Such a pattern matches a value of the form \(C_i(v_1, \ldots, v_n)\) and binds each \(x_i\) to \(v_i\) for evaluating the corresponding \(e_i\).

The type checking rules are: if \(e\) and all the \(p_i\) have type \(t_a\), and if all \(e_i\) have type \(t_b\), then the case expression has type \(t_b\). Type \(t\) is in scope when type checking the \(t_i\). And the datatype binding adds \(t\) to the current static environment as a new type, along with bindings for all the constructors to their types.

Now that we know about case expressions, it is better style to use pattern-matching to access list and option values, rather than functions null, hd, tl, isSome, and valOf we previously learned.

2 Built-in datatypes

We could use datatypes to create our own types for lists. For example, this binding works well for a list of integers:

\[
\text{datatype } my\_\text{int}\_\text{list} = \text{Empty} \\
\mid \text{Cons } \text{of } \text{int} \times my\_\text{int}\_\text{list}
\]

We can use constructors \text{Empty} and \text{Cons} to make values of \text{my\_int\_list} and use case expressions to compute with such values:

\[
\text{val one\_two\_three} = \text{Cons}(1,\text{Cons}(2,\text{Cons}(3,\text{Empty})))
\]

\[
\text{fun append\_mylist}(l1, l2) = \\
\text{case } l1 \text{ of } \\
\quad \text{Empty } \Rightarrow l2 \\
\quad | \text{Cons}(hd, tl) \Rightarrow \text{Cons}(hd, \text{append\_mylist}(tl, l2))
\]

So we don’t actually need the built-in list type; we could have built it ourselves.

In fact, lists and options really are just datatypes that are “included” in every program before compilation. There’s nothing truly special about them. Nonetheless, it’s better to use built-in, widely-known features than to invent your own.
Options are datatypes. 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:

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

Lists are datatypes. Value [] is syntactic sugar for nil, which is really a constructor that carries nothing. Operator :: really is a constructor that carries two values, but :: is unusual because it is an infix operator—it is placed between its two operands.

```ml
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. Although we won’t focus on this feature (i.e., you’re not responsible for understanding it in any detail beyond what appears here), there is nothing very complicated about it. For example, this is exactly how lists and options are defined in the built-in environment:

```ml
datatype 'a list = nil | :: of 'a * 'a list

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

3 Pattern matching for tuples and records

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 it and binds xi to vi. As you might expect, the order of fields in the pattern does not matter.

Similarly 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:

```ml
fun sum_triple (triple:int*int*int) = 
```
case triple of
    (x,y,z) => z + y + x

And a similar example with records could look like this:

fun sum_stooges (triple:{larry:int,curly:int,moe:int}) =
    case triple of
        {larry=x,curly=y,moe=z} => z + y + x

However, a case expression with just one branch is poor style—after all, the purpose of such expressions is to distinguish multiple cases. There’s a better approach: it turns out you can use pattern matching in let expressions:

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

fun sum_triple (triple:int*int*int) =
    let
        val (x,y,z) = triple
    in
        x + y + z
    end

But we can do even better. Patterns can be used when defining function bindings!

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. So is int*int*int->int the type of three-argument functions, or the type of single-argument functions that take triples as their argument? We’ll return to this question in the next lecture.

**Function type inference.** Note that we didn’t write any types on function arguments in our final versions of sum_stooges and sum_triple. By using patterns to access values of tuples and records, rather than selectors like #larry and #1, you will find it is no longer necessary to write types on your function arguments. 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 and tuple patterns introduced above provide this information.

Because the types of argument can be inferred, it is common in ML for programmers to omit them. People have different opinions as to whether this is good style, or not. For important functions that are likely to be called by code in other files, it can be a good idea to explicitly write the types. For helper functions that are only called by local code, it’s often okay to omit the types.
4 Type synonyms

One other kind of binding is a *type synonym* binding. Here’s an example:

```ocaml
type int_pair_list = (int*int) list
```

This binding doesn’t really do all that much. It just records `int_pair_list` as a type that means exactly the same thing as `(int*int) list`. The two can be used completely interchangeably. Sometimes you might even see the REPL switch back and forth between them.

The syntax of type synonym bindings is `type t1 = t2`. The type-checking rule is that `t1` is added to the static environment in type checking any further expressions, and moreover is recorded as being a synonym for `t2`.

One of the more useful things to do with type synonyms is to give names to record types, so that you don’t have to keep writing the (long) record type in your program.