CSE 341, Spring 2008, Lecture 5 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This lecture covers two topics:

- It continues our investigation of datatypes and pattern-matching for defining and analyzing data
- It describes tail recursion and how to use accumulators to make more functions tail recursive

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`.

A recursive datatype definition lets us define a kind of tree where the different variants are different kinds of tree nodes (with potentially different numbers of children). We considered this example in depth:

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

It defines trees that describe arithmetic expressions.

It turns out that options and lists are essentially just datatypes and stylistically one should generally use pattern-matching for them, *not* the functions `null`, `hd`, `tl`, `isSome`, and `valOf` we saw previously. (We used them because we had not learned pattern-matching yet.)

For options, the constructors are `NONE`, which carries nothing, and `SOME`, which carries one value of any type you want. For example, `SOME 3` has type `int option` and `NONE` can have type `int option` or `(int * int) option` or `'a option`. You can define your own datatypes that work for "any type" like this, but we won't discuss how here.

Lists are similar except the constructor syntax is unusual. The constructor `[]` carries nothing. The constructor `::` carries two things, one of type `t` and one of type `t list` for any `t`, and `::` is written between the two things.

Since `NONE`, `SOME`, `[]`, and `::` are constructors, we use them in patterns just like any other constructors. For example:

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
fun sum_list intlist =
    case intlist of
        [] => 0
      | hd::tl => hd + sum_list tl
fun append (l1,l2) =
    case l1 of
        [] => l2
      | hd::tl => hd :: append(tl,l2)
```

There are several reasons to prefer pattern-matching over functions that test (like `null`) and extract values (like `hd` and `tl`). With a case expression, it's easier to check for missing and redundant cases; in fact, the type-checker does it for you. The syntax is more concise. If you want to define functions like `null` or `hd`, you can easily do so using pattern-matching. (This is exactly what the standard library does. The functions are useful for passing to other functions, as we will see in a few lectures.)

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
```

However, case expressions with one branch are strange-looking and poor style. Instead, a `val` binding can have a pattern in it. This is just syntactic sugar for a one-branch case-expression; the semantics is to do the match and raise an exception if it fails. So better style would be:

```
fun sum_triple_better triple =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

But we can do even better. Notice that `sum_triple` and `sum_triple_better` both have type `int * int * int -> int`; they take a triple of ints and produce an int. This function has the same type and is the best style:

```
fun sum_triple_best (x,y,z) =
    x + y + z
```

Previously we said such a function took three arguments. That's how we think and talk about it typically, but it turns out *every function in ML takes one argument*. It is just that we can put a *pattern* where the argument goes and the semantics is to match the value passed to the function with the pattern and use the new bindings in the function body. So `sum_triple_best` is really using syntactic sugar for how `sum_triple_better` is written, which in turn is syntactic sugar for how `sum_triple` is written.

So using a tuple-pattern as a function's one argument (recall every function takes exactly one argument) is "just an idiom" albeit a very, very common one. This is an elegant and useful thing. For example, one might usually have calls like `sum_triple_best(1,2,3)`, but now we know this is just calling `sum_triple_best` with the triple `(1,2,3)`, which matches the pattern `(x,y,z)`, binding `x` to `1`, `y` to `2`, and `z` to 3. Moreover, we can call `sum_triple_best` with any expression that evaluates to an `int * int * int`. For example, if `g` has type `bool -> int * int * int`, then we can write `sum_triple_best(g(true))`. You cannot do something like that as conveniently in Java or C because functions there actually take multiple arguments and there is no way to write a function like `g` that returns a, "collection of arguments for another function."

To understand tail recursion and accumulators, consider these two functions for summing the elements of a list:

```
fun sum_list_1 lst =
    case lst of
        [] => 0
      | i::lst1 => i + sum_list_1 lst1
```

```
fun sum_list_2 lst =
    let fun f (lst,acc) =
          case lst of
                [] => acc
              | i::lst1 => f(lst1,i+acc)
    in
        f(lst,0)
    end
```

Both functions compute the same results, but `sum_list_2` is more complicated, using a local helper function that takes an extra argument, called `acc` for "accumulator." In the base case of `f` we return `acc` and the value passed for the outermost call is 0, the same value used in the base case of `sum_list_1`. This pattern is common: The base case in the non-accumulator style becomes the initial accumulator and the base case in the accumulator style just returns the accumulator.

Why might `sum_list_2` be preferred when it is clearly more complicated? To answer, we need to understand a little bit about how function calls are implemented. At its simplest, there is a *call stack*, which is a stack (the data structure with push and pop operations) with one element for each function call that has been started but has not yet completed. So when the evaluation of one function body calls another function, a new element is pushed on the call stack and it is popped off when the called function completes.

So for `sum_list_1`, there will be one call-stack element (sometimes just called a "stack frame") for each recursive call to `sum_list_1`, i.e., the stack will be as big as the list. This is necessary because after each stack frame is popped off the caller has to, "do the rest of the body" – namely add `i` to the recursive result and return.

Given the description so far, `sum_list_2` is no better: `sum_list_2` makes a call to `f` which then makes one recursive call for each list element. However, when `f` makes a recursive call to `f`, *there is nothing more for the caller to do after the callee returns except return the callee's result.* This situation is called a *tail call* (let's not try to figure out why it's called this) and functional languages like ML typically include an optimization: When a call is a tail call, the caller's stack-frame is popped before the call – the callee's stack-frame just replaces the caller's. This makes sense: the caller was just going to return the callee's result anyway. For `sum_list_2` that means the call stack needs just 2 elements at any point (one for `sum_list_2` and one for the current call to `f`).

Why do implementations of functional languages include this optimization? It is because this way recursion can sometimes be as efficient as a while-loop, which also does not make the call-stack bigger. The "sometimes" is exactly when calls are tail calls, but tail calls do not need to be to the same function (`f` can call `g`), so they are more flexible than while-loops that always have to "call" the same loop. Using an accumulator is a common way to turn a recursive function into a "tail-recursive function" (one where all recursive calls are tail calls), but not always. For example, functions that process trees (instead of lists) typically have call stacks that grow as big as the depth of a tree, but that's true in Java too.

While most people rely on intuition for "which calls are tail calls," we can be more precise by defining *tail position* recursively and saying a call is a tail call if it is in tail position. The definition has one part for each kind of expression; here are several parts:

- In `fun f(x) = e`, `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (not `e1`). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (not any binding expressions).
- Function-call arguments are not in tail position.
- ...