CSE341, Fall 2011, Lecture 6 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.

This lecture covers two unrelated topics:

- The notion of tail recursion, its relation to efficiency of recursion in functional languages like ML, and how to use accumulators as a general technique to make functions tail recursive — note this topic involves new idioms but no new language constructs
- Exceptions

Tail Recursion and Accumulators

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

```
fun sum1 xs =
    case xs of
        [] => 0
      | i::tl => i + sum1 tl

fun sum2 xs =
    let fun f (xs,acc) =
            case xs of
                [] => acc
              | i::tl => f(tl,i+acc)
    in
        f(xs,0)
    end
```

Both functions compute the same results, but `sum2` 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 `sum1`. 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 `sum2` be preferred when it is clearly more complicated? To answer, we need to understand a little bit about how function calls are implemented. Conceptually, 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. Each element stores things like the value of local variables and what part of the function has not been evaluated yet. 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 `sum1`, there will be one call-stack element (sometimes just called a "stack frame") for each recursive call to `sum1`, 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, `sum2` is no better: `sum2` 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 essential 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 `sum2` that means the call stack needs just 2 elements at any point (one for `sum2` and one for the current call to `f`).

Why do implementations of functional languages include this optimization? By doing so, 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, something you the programmer can reason about since you can look at the code and identify which calls are tail calls.

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 any language: while-loops are not very useful for processing trees.

More Examples

Tail recursion is common for functions that process lists, but the concept is more general. For example, here are two implementations of the factorial function where the second one uses a tail-recursive helper function so that it needs only a small constant amount of call-stack space:

```
fun fact1 n = if n=0 then 1 else n * fact1(n-1)

fun fact2 n =
    let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n)
    in
        aux(n,1)
    end
```

It is worth noticing that `fact1 4` and `fact2 4` produce the same answer even though the former performs $4*(3*(2*(1*1)))$ and the latter performs $(((1*4)*3)*2)*1$. We are relying on the fact that multiplication is associative ($a*(b*c) = (a*b)*c$) and that multiplying by 1 is the identity function ($1*x = x*1 = x$). The earlier `sum` example made analogous assumptions about addition. In general, converting a non-tail-recursive function to a tail-recursive function usually needs associativity, but many functions are associative.

A more interesting example is this inefficient function for reversing a list:

```
fun rev1 lst =
   case lst of
       [] => []
     | x::xs => (rev1 xs) @ [x]
```

We can recognize immediately that it is not tail-recursive since after the recursive call it remains to append the result onto the one-element list that holds the head of the list. Although this is the most natural way to reverse a list recursively, the inefficiency is caused by more than creating a call-stack of depth equal to the argument's length, which we will call n. The worse problem is that the total amount of work performed is proportional to n^2, i.e., this is a quadratic algorithm. The reason is that appending two lists takes time proportional to the length of the first list: it has to traverse the first list — see our own implementation of append in an earlier lecture. Over all the recursive calls to `rev1`, we call `@` with first arguments of length $n-1, n-2, ..., 1$ and the sum of the integers from 1 to $n-1$ is $n*(n-1)/2$.

As you learn in a data structures and algorithms course (CSE332), quadratic algorithms like this are much slower than linear algorithms for large enough n. That said, if you expect n to always be small, it may be

be worth valuing the programmer's time and sticking with a simple recursive algorithm. Else, fortunately, using the accumulator idiom leads to an almost-as-simple linear algorithm.

```
fun rev2 lst =
    let fun aux(lst,acc) =
            case lst of
                [] => acc
              | x::xs => aux(xs, x::acc)
    in
        aux(lst,[])
    end
```

The key differences are (1) tail recursion and (2) we do only a constant amount of work for each recursive call because `::` does not have to traverse either of its arguments.

A Precise Definition of Tail Position

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 an expression is not in tail position, then neither are any of its subexpressions.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (but not `e1`). (Case-expressions are similar.)
- If `let b1 ... bn in e end` is in tail position, then `e` is in tail position (but no expressions in the bindings are).
- Function-call arguments are not in tail position.
- ...

Exceptions

ML has a built-in notion of exception. You can *raise* (also known as *throw*) an exception by using the `raise` primitive. For example, the `hd` function in the standard library raises the `Empty` exception when called with `[]`:

```
fun hd xs =
    case xs of
        []   => raise Empty
      | x::_ => x
```

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 MyUndesirableCondition
exception MyOtherException of int * int
```

Kinds of exceptions are a *lot* like constructors of a datatype binding. Indeed, they are functions (if they carry values) or values (if they don't) that create values of type `exn` rather than the type of a datatype. So `Empty`, `MyUndeseriableCondition`, and `MyOtherException(3,9)` are all values of type `exn`, whereas `MyOtherException` has type `int*int->exn`. Usually we just use exception constructors as arguments to `raise`, such as `raise MyOtherException(3,9)`, but we can use them more generally to create values of type `exn`.

For example, here is a version of 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.)

```
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, which the function then does not *raise*.

The other feature related to exceptions is *handling* (also known as *catching*) them. For this, ML has handle-expressions, which look like `e1 handle p => e2` where `e1` and `e2` are expressions and `p` is a pattern that matches an exception. The semantics is to evaluate `e1` and have the result be the answer. But if an exception matching `p` is raised by `e1`, then `e2` is evaluated and that is the answer for the whole expression. If `e1` raises an exception that does not match `p`, then the entire handle-expression also raises that exception. Similarly, if `e2` raises an exception, then the whole expression also raises an exception.

As with case-expressions, handle-expression can also have multiple branches each with a pattern and expression, syntactically separated by `|`.