1 Tail recursion

Consider these functions for summing the elements of a list:

```ml
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 result, 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.

So why are we telling you this? Why would you ever want to write `sum2`, 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. Functional languages like ML typically include an hugely useful 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? Because with it, recursion can sometimes be as efficient as a `while` loop from languages like Java. (While loops don’t make the call-stack bigger.) The “sometimes” is exactly when calls are tail calls—something both you and the compiler can (often) figure out.

Tail calls do not need to be to the same function (`f` can tail-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 in which all recursive calls are tail calls).

**More examples of tail recursion.** 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, such that it needs only a small constant amount of call-stack space:

```math
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 noting that `fact1 4` and `fact2 4` produce the same answer even though the former performs \(4 \times (3 \times (2 \times (1 \times 1)))\) and the latter performs \(((1 \times 4) \times 3) \times 2 \times 1\). We are relying on the fact that multiplication is associative \((a \times (b \times c)) = ((a \times b) \times c)\) and that multiplying by 1 is the identity function \((1 \times x = x \times 1 = x)\). 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, so this isn’t a tremendous burden.

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

```math
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 might be the the most natural algorithm for recursively reversing a list, its inefficiency is caused by more than just 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 \(n^2\)—that is, 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` above. Over all the recursive invocations, `rev1` calls @ with first arguments of length \(n-1, n-2, \ldots, 1\). The sum of the integers from 1 to \(n-1\) is \(n \times (n - 1)/2\), which is proportional to \(n^2\).

As you learned in data structures, quadratic algorithms 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. But otherwise, fortunately, using the accumulator idiom leads to an almost-as-simple linear algorithm.

```math
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) the function does only a constant amount of work for each recursive call because :: does not have to traverse either of its arguments.

A definition of tail position. Although you can usually rely on intuition to determine which calls are tail calls, we can be more precise by defining tail position recursively, and defining a call as a tail call if it is in tail position. The definition has one part for each kind of expression; here are several parts:

- In \( \text{fun } f(x) = e \), expression \( e \) is in tail position.
- If \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) is in tail position, then \( e_2 \) and \( e_3 \) are in tail position (but not \( e_1 \)). (Case-expressions are similar.)
- If \( \text{let } b_1 \ldots b_n \text{ in } e \text{ end} \) 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.
- If an expression is not in tail position, then neither are any of its subexpressions.

2 Type variables

Suppose we want to compute the length of an integer list. We could write the following function:

\[
\text{fun len (lst : int list) = case lst of}
\]
\[
\text{[]} \Rightarrow 0
\]
\[
\text{| } _::t \Rightarrow 1 + (\text{len } t)
\]

But now suppose we want to compute the length of a string list. We can’t just apply \( \text{len} \) to a \text{string list}; that’s the wrong type. So we’d have to copy-and-paste and create a new version of \( \text{len} \):

\[
\text{fun len2 (lst : string list) = case lst of}
\]
\[
\text{[]} \Rightarrow 0
\]
\[
\text{| } _::t \Rightarrow 1 + (\text{len } t)
\]

That just doesn’t seem right. Duplicating code like that is bad practice. Can’t we somehow make the two functions share the same code?

Yes, we can. We can use a \text{type variable}, which stands for an unknown type:

\[
\text{fun len (lst : 'a list) = case lst of}
\]
\[
\text{[]} \Rightarrow 0
\]
\[
\text{| } _::t \Rightarrow 1 + (\text{len } t)
\]

(We’ve actually seen type variables a couple times already.) In this code, \( 'a \) (pronounced “alpha”) is a type variable. Function \( \text{len} \) could be invoked on a \text{int list} or a \text{string list} or a \text{t list} for any type \( t \). Type variables abstract from types in the same way that that variables we’re accustomed to abstract from values. Type variables enable a language feature known as \text{parametric polymorphism}. Roughly speaking, in ML that means functions can be parameterized with type variables, which means a given function could apply to many forms of values. (\text{“poly”} = \text{many}, \text{“morph”} = \text{form}) This is closely related to the idea of \text{generics} that
show up in Java or C++. Parametric polymorphism is an essential feature of ML. Without it, we would have to redefine lists for every type of element that a list might have.

Were we to leave off the argument type in `len`, the compiler would automatically infer the most general type that it can for the function:

```ml
fun len lst = 
  case lst of
    [] => 0
  | _::t => 1 + (len t)

val len = fn : 'a list -> int
```

So if you see type variables showing up in the REPL, know that the compiler is inferring types that are as general as possible. If you want them to be more specific, put in explicit type annotations.

For fun, consider the type `'a -> 'a`. What functions can we write with this type? Here’s one:

```ml
fun identity (x : 'a) = x
```

We challenge you to write any function that is not equivalent to this one but has the same type.

Now, consider the type `'a -> int`. What could a function with this type do? Since `'a` could be anything, the function can’t possibly do anything with the parameter of type `'a`. So this function must always return the same value, since it takes no inputs that it can actually understand. Here’s an example:

```ml
fun constant (x : 'a) = 9 * 73
```

Next, consider the type `'a -> 'b`. Is it possible to write a function of this type? It might seem like you can’t, but...

```ml
fun diverge (x : 'a) = diverge x
```

The compiler figures out that this function will never return when called—so it won’t `converge` to a value, which means that it will `diverge`, hence the name. Thus whenever returns, it could return a value of any type. (Here’s a closely related idea: All unicorns that live on the moon are purple.)

### 3 Equality types

The following function binding will be rejected by the compiler. Why?

```ml
fun equal (x:'a, y:'a) = (x = y)
```

The problem is that ML doesn’t know what concrete type `'a` is, so does not know whether it supports equality comparisons. Why worry about that? Consider the following code:

```ml
equal(fact1,fact2)
```

(Functions `fact1` and `fact2` are defined in the previous lecture; they are the recursive and tail-recursive implementations of factorial.) Is this expression true? To determine that, ML would have to figure out whether two functions always return the same result on every input. That’s not possible. (Take CSci 3313 to find out why.) The problem is that equality comparisons are not defined on function values. (Nor are they on `real` values.)
If you want to specify a type variable that stands for a type that does support equality comparisons, you instead write `''a. (Note the second quote mark.) This is an equality type variable. This function will then be accepted by the compiler:

```ml
fun equal (x: ''a, y: ''a) = (x = y)
```

However, the current version of the SML/NJ compiler will produce a warning: calling `polyEqual`. This is a performance warning—when the compiler doesn’t know anything about the types involved, the equality comparison it performs is actually rather slow.

Types that can instantiate equality type variables include the following: `int`, `bool`, `string`, tuples, records, options, and lists.

### 4 Type annotations and inference

We started off the class always writing the types of function arguments—for example,

```ml
fun pow (x:int, y:int) =
  if y = 0
  then 1
  else x * pow(x,y-1)
```

But as we observed above, the compiler can often infer types for us. We could, for example, write `pow` without argument types:

```ml
fun pow (x, y) =
  if y = 0
  then 1
  else x * pow(x,y-1)
```

The compiler will automatically figure out that the type of `pow` is `int * int -> int`. We’ll talk more about how it does so in a later lecture. But if you think about it, you’ll see it’s not hard for `pow`: Variable `y` is compared to 0, so `y` must have type `int`. The `then` branch returns 1, so the return type of the function must be `int`. The `else` branch multiplies the result of a recursive call with `x`, so `x` must also have type `int`.

There were two reasons we started off the semester with writing down argument types. First, it’s more familiar to you from Java. Second, selectors like `#1` and `#field` make type inference difficult. In the following function, what is the type of `triple`?

```ml
fun sum_triple triple =
  #1 triple + #2 triple + #3 triple
```

It could have type `int*int*int` or `int*int*int*int` or `int*int*string` or `int*int*int*int*string` or an infinity of other types. So the ML compiler can’t infer enough about the type of `triple`, and we get a compile-time error: “Error: unresolved flex record (need to know the names of ALL the fields in this context).” But if we pattern match against `triple`, we give the compiler enough information to figure out which type it must have:

```ml
fun sum_triple triple =
  case triple of
    (a, b, c) => a + b + c
```

In this case, `triple` has type `int*int*int`. But if we write this:
fun sum_triple triple =
    case triple of
      (a,b,c,d) => a+b+c

Then \texttt{triple} has type \texttt{int*int*int*'a}. That is, \texttt{triple} is actually a 4-tuple (despite the variable name) whose 4th component has an unknown type, represented by type variable \texttt{'a}.

Now that we’ve learned pattern matching, we’ll rarely need to write down types anymore. But we might do so anyway, because it’s a good way to make code readable and self-documenting.

Whenever we do want to write down types, we can do so pretty much anywhere. If \texttt{e} is an expression, and if \texttt{t} is a type, then \texttt{e:t} is also an expression. The :\texttt{t} is a \textit{type annotation}, aka \textit{constraint}, that is like a hint to the compiler: it tells the compiler that expression \texttt{e} must have type \texttt{t}. If \texttt{e} turns out not to have type \texttt{t}—for example, \texttt{1:string}—the compiler indicates an error.

In fact, type annotations can be written on, among other things, tuple components (because components of tuples are expressions). So instead of \texttt{(1,2)} we could write \texttt{(1:int,2:int)}. Or instead of \texttt{(x,y)} we could write \texttt{(x:int,y:int)}.

Patterns can contain type annotations, too. And that’s why we can write either \texttt{fun pow (x:int, y:int) = ...} or \texttt{fun pow (x, y) = ....} Both describe a function that takes a pair as its single argument; the former gives two annotations on the components of that pair, the latter omits the annotations.