### Pragmatics

Efficiency concerns:

- Time
- Space (= Time!)

Reasonable assumptions for tracking resource use:

- Applying a constructor allocates heap space.

```
fun id nil = nil
  | id l = l

fun id' nil = nil
  | id' (h::t) = h::t
```

- Evaluating a `fn` expression allocates heap space for **free variables**.

```
fun f (x,y,z,w) =
   fn a => x + y + z + w + a
val g = f (1,2,3,4)
g 5;
```

- Applying a function takes time.

Only recursive applications really matter. But not all recursive applications are equally expensive!

### Tail Recursion

Compare this version of `sum`:

```
fun sum l =
  let fun sum' (nil,a) = a
        | sum' (h::t,a) = sum'(t,h+a)
in sum' (l,0)
end
```

Notice that the recursive call to `sum'` is the **last** thing that happens in the function body. Such a call is said to be a **tail** call, and `sum'` is said to be **tail-recursive**.

In performing the tail-recursive call to `sum'`, the same steps could be performed as before, but this is not necessary.

- Because the caller does nothing after the return (except return itself), there is really no need for the recursive call to return control to the caller.
- Instead, the callee can return its value directly to the top of the recursion (the call from `sum`).
- So there's no need for the recursive call to save any local variables nor to store the current pc.

In short, a tail-recursive call can be performed just by passing the argument values and then performing a simple jump to the beginning of the called function.

### Costs of Recursion

Consider this version of the `sum` function:

```
fun sum nil = 0
  | sum (h::t) = h + (sum t)
```

Or, equivalently,

```
fun sum nil = 0
  | sum (h::t) =
        let val r = sum t
        in h + r
        end
```

What's involved in evaluating this function on a large list?

At each recursive call `sum t`, must:
- store the value of local variable `h`
- store the current pc
- pass the argument `t`
- jump to the beginning of `sum`

On return, must:
- pass back the result value `r`
- retrieve the old pc and jump there
- restore the value of `h`

Only then can the addition `h+r` be performed.

### Tail Recursion and Iteration

This is in every way much cheaper than an ordinary call. Indeed, it's no more expensive than jumping to the top of an **iterative** loop in an imperative language.

In fact, it's essentially the same thing:

```
sum(l) =
  var a = 0
  sum': if l = nil then
          a
        else
          a := hd(l) + a;
          l := tl(l);
          goto sum'
```

### Accumulation Parameters

The difference between `sum` and `sum'` is that the latter uses an extra parameter to `accumulate` the result, which is then returned by the `nil` case.

Adding an accumulation parameter is a common technique for changing non-tail-recursive functions to tail-recursive ones.

Contrast:

```
fun reverse nil = nil
  | reverse (h::t) = (reverse t) @ [h]

fun reverse l =
  let fun reverse' (nil,a) = a
        | reverse' (h::t,a) = reverse'(t,h::a)
  in reverse' (l,nil)
  end
```

The latter also avoids doing expensive append operations.

### Auxiliary Parameters

In general, it is more work to get rid of non-tail recursion in operations over trees. Consider

```
datatype 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
fun flatten (Leaf x) = [x]
  | flatten (Node(l,r)) = (flatten l) @ (flatten r)
```

We can use an accumulator to get rid of one non-tail call (and the append operation) by passing the accumlated flattening of all right-neighbors:

```
fun flatten t =
  let fun flatten' (Leaf x, a) = x::a
        | flatten' (Node(l,r),a) =
                flatten'(l,flatten'(r,a))
  in flatten' (t,nil)
  end
```

Note that there is still one non-recursive call. We can get rid of it as well, by adding **another** extra parameter that keeps an explicit stack of left-neighbor trees to be flattened:

```
fun flatten t =
  let fun flatten' (Leaf x, nil, done) =
                        x::done
        | flatten' (Leaf x, h::t, done) =
                        flatten' (h, t, x::done)
        | flatten' (Node(l,r), todo, done) =
                        flatten' (r, l::todo, done)
  in flatten'(t,nil,nil)
  end
```

### Reduce vs. Accumulate

Recall the general-purpose right-to-left `reduce` function

```
fun reduce f a nil = a
  | reduce f a (h::t) = f h (reduce f a t)
```

which computes

$(x_1$ `op` $(x_2$ `op` $(\ldots$ $(x_n$ `op a)` $\ldots)))$

Note that it is **not** tail-recursive. But the similar left-to-right **accumulate** function **is**:

```
fun accumulate f a nil = a
  | accumulate f a (h::t) = accumulate f (f a h) t
```

which computes

$((\ldots((a$ `op` $x_1)$ `op` $x_2)\ldots)$ `op` $x_n)$

We can always rewrite a `reduce` as a (perhaps more complicated) **accumulate**. In the special case where `op` is associative and commutative, `reduce` and `accumulate` calculate the same thing.

### Continuation functions

It turns out that we can always rewrite arbitrary recursive algorithms as tail-recursive ones by using explicit stack arguments like this. This is true in an ordinary imperative language too. (It's not clear that doing so will be a "win," however, unless we can also make other improvements–such as getting rid of appends–in the process.)

There's another, remarkable, method by which we can turn a functional program into one that is completely tail-recursive using extra arguments of **function** type.

Consider `sum` again:

```
fun sum nil = 0
  | sum (h::t) = h + (sum t)
```

And this version:

```
fun sum l =
  let fun sum' (nil,k) = k 0
        | sum' (h::t,k) = sum'(t, fn s => k(h+s))
  in sum' (l,fn s => s)
  end
```

Here `sum'` takes an extra argument `k`, called a **continuation**, which says "what to do with the result." Although tail-recursive, this program isn't really any more efficient than the original one. (Any ideas why not?)