Tail calls

If the result of \( f(x) \) is the result of the enclosing function body, then \( f(x) \) is a \textit{tail call}.

More precisely, a tail call is a call in \textit{tail position}:

- In \textit{fun} \( f(x) = e \), \textit{e} is in tail position.
- If \textit{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 (not \( e_1 \)). (Similar for \textit{case}).
- If \textit{let} \( b_1 \ldots b_n \ \text{in} \ e \) and is in tail position, then \( e \) is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

Where we are

Some implementation tidbits: ARs, call stacks & cons cells
Tail recursion avoids call stack overhead
Accumulator-style recursion typically tail-recursive
Today:

- more tail/accumulator examples
- more on pattern-matching as an elegant generalization of variable binding.

So what?

Why does this matter?

- Implementation takes space proportional to depth of function calls ("call stack" must "remember what to do next")
- But in functional languages, implementation must ensure tail calls eliminate the caller’s space
- Accumulators are a systematic way to make some functions tail recursive
- "Self" tail-recursive is very loop-like because space does not grow.
A Classic—Reversing a List I

\[
\text{fun rev1(nil) = nil}
\]
\[
| \text{rev1(x::xs) = rev1(xs) @ [x];}
\]

Run time?

A Classic—Reversing a List II

\[
\text{fun rev1(nil) = nil}
\]
\[
| \text{rev1(x::xs) = rev1(xs) @ [x];}
\]

Run time?

\[
O(n^2)
\]

L1 @ L2 must copy L1:

\[
\text{fun append([],L2) = L2}
\]
\[
| \text{append(x::xs, L2) = x:append(xs, L2);}
\]

So rev1([1,2,...,n]) takes time

\[1 + 2 + \cdots + n = O(n^2).
\]

A Classic—Reversing a List III

\[
\text{fun rev1(nil) = nil}
\]
\[
| \text{rev1(x::xs) = rev1(xs) @ [x];}
\]

\[
\text{fun rev2 list =}
\]
\[
| \text{let fun f (nil, acc) = acc}
\]
\[
| \text{f (x::xs, acc) = f(xs,x::acc)}
\]
\[
| \text{in}
\]
\[
| \text{f(list,nil)}
\]
\[
| \text{end}
\]

The standard trick: Do ops on way in, not way out. Instead of operating on recursive result, move operation into the recursive call.

Run time, now?

Deep patterns

Patterns are much richer than we have let on. A pattern can be:

- A variable (matches everything, introduces a binding)
- _ (matches everything, no binding)
- A constructor and a pattern (e.g., C p) (matches a value if the value "is a C" and p matches the value it carries)
- A pair of patterns ((p1,p2)) (matches a pair if p1 matches the first component and p2 matches the second component)
- A record pattern...
- An integer constant...
- ...

The truth, the whole truth, and nothing but

It's really:

- `val p = e`
- `fun f p1 = e1 | f p2 = e2 ... | f pn = en`
- `case e of p1 => e1 | ... | pn => en`

Inexhaustive matches may raise exceptions and are bad style.

Example: could write `Rope pr` or `Rope (r1,r2)`

Fact: Every ML function takes exactly one argument!

Some function examples

- `fun f1 () = 34`
- `fun f2 (x,y) = x + y`
- `fun f3 pr = let val (x,y) = pr in x + y end`

Is there any difference to callers between `f2` and `f3`?

In most languages, “argument lists” are syntactically separate, second-class constructs.

Can be useful: `f2 (if e1 then (3,2) else pr)`