CSE 341: Programming Languages

Spring 2007
Lecture 6 — More on Tail Recursion & Accumulators
More on Bindings & Immutability

What does this do?

```ml
val x = 1;
val x = 2;
```

First binding to `x` is *hidden* by 2nd, but not overwritten, changed or erased.

You could still see it if you wanted, e.g.:

```ml
val x = 1;
fun oldx() = x;
val x = 2;
oldx();
```

Bindings are *immutable*. (Deleting inaccessible ones, e.g. the 1st `x` in the 1st example, is a performance issue, not a correctness issue.)
More...

A more subtle example:

```ml
val x = [3];
val y = 2 :: x;
val z = 1 :: y;
(* What’s z? *)
val x = [42];
(* What’s z now? *)
```

Or this:

```ml
val x = [1,2,3,4,...,999];
val y = 42 :: tl(x);
```

Did that allocate 1000 mem cells, or 2000?
Implementing lists

Want: null, hd, tl, ::

How: Arrays? Pointers? Other?

Costs: memory, time, code
Using Lists (Java)

Consider a linked list of integers, implemented in Java.

• What data structure (if you build it from scratch)?

How would you implement functions for:

• Test if a list is empty? (How fast?)
• Extract the hd of a lst? (How fast?)
• Extract the tl of a lst? (How fast?)
• Implement ::? (How fast? Semantics?)
• Find the last element of a list? (How fast? How much memory?)
• Find the length of a list? (How fast? How much memory?)
Implementing lists

Want: null, hd, tl, ::

How: Arrays? Pointers? Other?

Costs: memory, time, code

[1,2,3]
Using Lists (ML)

Consider

```ml
fun len []     = 0
  | len (x::xs) = 1 + len xs;
```

val theLength = len [1,2,3,4,5];

Q: How do you implement function call?

A: “Activation Records” and a “Call Stack”
Activation Records

What:

• Info about each activation of each procedure
• Dynamically created on call, destroyed (usually) on return
• Values of local variables
• Where was I called from/Where do I return to?
• (Housekeeping info: save state, registers, temp variables, partially evaluated exprs, etc. across function calls)
Activation Records (cont.)

Why:

• Esp. with recursion, there may be many simultaneous activations of a given procedure, each with different values for local vars, different return addresses, etc.

• The AR is a simple implementation trick to keep it all straight

Downsides:

• The main source of “function call overhead”, both space & time.
fun len [] = 0
  | len (x::xs) = 1 + len xs;
val theLength = len [1,2,3];
Implementing calls

Consider

fun len [] = 0
| len (x::xs) = 1 + len xs;

val theLength = len [1,2,3,4,5];

Compare:

fun last [x] = x
| last(x::xs) = last xs;

val theLast = last [1,2,3,4,5];
Tail calls

A call $f(x)$ is called a *tail call* if it appears at the “tail end” of $g$, and the value of $f(x)$ is returned as the value of $g$ without change.

Why care? Because they can be optimized! The usual call mechanism:

- Suspend activation of $g$
- Build AR for $f$, then run $f$
- Destroy AR for $f$, passing value of $f(x)$ back to $g$
- Destroy AR for $g$, passing value of $g(-) = f(x)$ back to $g$’s caller

Can be streamlined to:

- *Reuse* $g$’s AR for $f$
- Don’t “call” $f$, just jump to start of its code
- When $f$ returns, return its value directly $g$’s caller

A key special case: direct tail-recursion turns into a loop!
Accumulators: can turn non-tail calls into tail calls

\[
\text{fun } \text{len } [\ ] = 0 \\
| \text{len } (x::xs) = 1 + \text{len } xs;
\]

Becomes:

\[
\text{fun } \text{len2 } lst = \\
\text{let lenaux}([], \text{acc}) = \text{acc} \\
| \text{lenaux}(x::xs, acc) = \text{lenaux}(xs, acc+1); \\
in \\
\text{lenaux}(lst,0) \\
\text{end};
\]

The standard trick: Do ops on way in, not way out. Instead of operating on recursive \textit{result}, move operation into the recursive \textit{call}. 
Tail calls: definition

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

More precisely, a tail call is a call in tail position:

- 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 arguments are not in tail position.
- ...

CSE 341 Spring 2007, Lecture 6