CSE 341:
Programming Languages

Spring 2006

Lecture 7 — More on Tail Recursion & Accumulators; Deep Patterns

Room Change, OH & Travel

Fri. 5/19 *only*, we will meet in THO 125 (at usual time)

My Office Hour: Fridays at ?

But not this week—out of town. (Guest Lectures by Jonah W & F.)

Where we are

Two implementation tidbits: call stack & 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.
- first-class functions (closures, functions as values)

Tail calls

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.
- ...

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

```
fun rev1(nil) = nil
|   rev1(x::xs) = rev1(xs) @ [x];
```

Run time?

A Classic—Reversing a List II

```
fun rev1(nil) = nil
|   rev1(x::xs) = rev1(xs) @ [x];
```

Run time?

$O(n^2)$!

L1 @ L2 must *copy* L1:

```
fun append([],l2) = l2
|   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

```
fun rev1(nil) = nil
|   rev1(x::xs) = rev1(xs) @ [x];

fun rev2 lst =
    let fun f (nil, acc) = acc
        |   f (x::xs, acc) = f(xs,x::acc)
    in
        f(lst,nil)
    end
```

The standard trick: instead of operating on recursive result, push operation into the recursive call.

Run time?

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 (`(`$\boldsymbol{p1}$`,`$\boldsymbol{p2}$`)`) (matches a pair if $\boldsymbol{p1}$ matches the first component and $\boldsymbol{p2}$ 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)`