CSE341: Programming Languages

Section 3
Function Patterns
Tail Recursion

Winter 2018

Function Patterns

- Just a syntactic sugar: a pattern matching of function arguments

```
fun f x = e1
  case x of
    p1 => e1
  | p2 => e2
   …
```

- Can be written as

```
fun f p1 = e1
  | f p2 = e2
  …
  | f pn = en
```

- Nothing more powerful, it’s a matter of taste

Another example of tail recursion

```
fun sum xs =
   case xs of
       [] => 0
     | x::xs' => x + sum xs'
```

```
fun sum xs =
    let fun aux(xs,acc) =
          case xs of
             [] => acc
           | x::xs' => aux(xs',x+acc)
    in
       aux(xs,0)
    end
```

And another

```
fun rev xs =
   case xs of
          [] => []
        | x::xs' => (rev xs') @ [x]
```

```
fun rev xs =
    let fun aux(xs,acc) =
            case xs of
                 [] => acc
               | x::xs' => aux(xs',x::acc)
    in
       aux(xs,[])
    end
```

Actually much better

```
fun rev xs =
   case xs of
        [] => []
      | x::xs' => (rev xs') @ [x]
```

- For **`fact`** and **`sum`**, tail-recursion is faster but both ways linear time
- Non-tail recursive **`rev`** is quadratic because each recursive call uses append, which must traverse the first list
  - And 1+2+…+(length-1) is almost length*length/2
  - Moral: beware list-append, especially within outer recursion
- Cons constant-time (and fast), so accumulator version much better

To show you regular recursions do fail

- OCaml code
- Why SML works?
  - Hopefully we can talk about it in Section 8
  - Otherwise, if we don’t get a chance to talk about it and you are really curious, you should take 505

Always tail-recursive?

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Most obvious examples are functions that process trees

In these cases, the natural recursive approach is the way to go

- You could get one recursive call to be a tail call, but rarely worth the complication

Also beware the wrath of premature optimization

- Favor clear, concise code
- But do use less space if inputs may be large

What is a tail-call?

The "nothing left for caller to do" intuition usually suffices

- If the result of **`f x`** is the "immediate result" for the enclosing function body, then **`f x`** is a tail call

But we can define "tail position" recursively

- Then a "tail call" is a function call in "tail position"

…

Precise definition

A *tail call* is a function call in *tail position*

- If an expression is not in tail position, then no subexpressions are

- In **`fun f p = e`**, the body **`e`** is in tail position
- If **`if e1 then e2 else e3`** is in tail position, then **`e2`** and **`e3`** are in tail position (but **`e1`** is not). (Similar for case-expressions)
- If **`let b1 … bn in e end`** is in tail position, then **`e`** is in tail position (but no binding expressions are)
- Function-call *arguments* **`e1 e2`** are not in tail position
- …

A lot of tail recursion problems

- Problem 1: inc_all, increment all elements of the given list by 1
  - inc_all([1, 2, 3, 5]) = [2,3,4,6]
- Problem 2: repeat, repeat(x, n) returns a list with n repeated values of x
  - repeat(1, 5) = [1,1,1,1,1]
- Problem 3: range, range(lo, hi) returns a list of all values from lo to (hi - 1)
  - range(2, 5) = [2, 3, 4]

A lot of tail recursion problems

- Problem 4: pair_chain, (pair_chain l) returns a list of all pairs of consecutive elements in l in any order
  - pair_chain([1, 2, 3, 5]) = [(3,5),(2,3),(1,2)]
- Problem 5: triples, triples(xs, ys, zs) combines three lists into a triple list if they have equal length, otherwise raise a LengthMismatch exception
  - triples([1, 4], [2, 5], [3, 6]) = [(4,5,6),(1,2,3)]
  - triples([1, 4], [2, 5], [3]) should raise exception

A lot of tail recursion problems

- Problem 6: choose2, (choose2 l) returns a list of pairs using all combination of elements of l. The list can be in any order.
  - Write for normal recursion first
  - choose2_tail([1, 2, 3, 4, 5]) =

    [(4,5),(3,5),(3,4),(2,5),(2,4),(2,3),(1,5),(1,4),(1,3),(1,2)]