CMSC 330: Organization of Programming Languages

Tail Recursion

CMSC330 Spring 2024
Factorial

fact n = \[
\begin{cases} 
1 & \text{n=0} \\
 n \times \text{fact} \ (n-1) & \text{n>0}
\end{cases}
\]

let rec fact n = 
  if n = 0 then 1 
  else n \times \text{fact} \ (n-1)

fact 4 = 24
Factorial

\[
\text{fact } n = \begin{cases} 
1 & \text{n=0} \\
\text{n} \times \text{fact } (n-1) & \text{n>0}
\end{cases}
\]

\[
\text{fact } 3 = 3 \times \text{fact } 2 \\
= 3 \times 2 \times \text{fact } 1 \\
= 3 \times 2 \times 1 \times \text{fact } 0 \\
= 3 \times 2 \times 1 \times 1 \\
= 3 \times 2 \times 1 \\
= 3 \times 2 \\
= 6
\]
# let rec fact n = if n = 0 then 1 else n * fact (n-1);;
val fact : int -> int = <fun>

# fact 1000000 ;
Stack overflow during evaluation (looping recursion?).
Yet Another Factorial

\[
\text{aux } x \ a = \begin{cases} 
  a & x=0 \\
  \text{aux } (x-1) \ x*\ a & x>0 
\end{cases}
\]

\[
\text{fact } n \ = \ \text{aux } n \ 1
\]

let \text{fact } n =

\[
\text{let rec aux } x \ a =
\begin{align*}
  \text{if } x &= 0 \text{ then } a \\
  \text{else } \text{aux } (x-1) \ x*\ a
\end{align*}
\]

\[
\text{in}
\]

\[
\text{aux } n \ 1
\]
Yet Another Factorial

\[
\text{aux } x \ a = \begin{cases} 
  a & \text{x=0} \\
  \text{aux } (x-1) \ x*\text{a} & \text{x>0}
\end{cases}
\]

\[
\text{fact } n = \text{aux } n \ 1
\]

\[
\text{fact } 3 = \text{aux } 3 \ 1 \\
= \text{aux } 2 \ 3 \\
= \text{aux } 1 \ 6 \\
= 6
\]

No Stack!

No need to push a new frame on each call

- The result of the evaluation is just the result of the recursive call – nothing to remember
- *So: Reuse the current frame*
Tail Recursion

- Whenever a function’s result is completely computed by its recursive call, it is called **tail recursive**
  - Its “tail” – the last thing it does – is recursive

- Tail recursive functions can be implemented **without requiring a stack frame for each call**
  - No intermediate variables need to be saved, so the compiler overwrites them

- Typical pattern is to use an **accumulator** to build up the result, and return it in the base case
Compare fact and aux

let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)

Waits for recursive call’s result to compute final result

let fact n =
  let rec aux x acc =
    if x = 1 then acc
    else aux (x-1) (acc*x)
  in
  aux n 1

final result is the result of the recursive call
Exercise: Finish Tail-recursive Version

let rec sumlist l =
  match l with
  [] -> 0
  | (x::xs) -> (sumlist xs) + x

Tail-recursive version:

let sumlist l =
  let rec helper l a =
    match l with
    [] -> a
    | (x::xs) -> helper xs (x+a)
  in
  helper l 0
True/false: map is tail-recursive.

```
let rec map f = function
  | [] -> []
  | (h::t) -> (f h)::(map f t)
```

A. True  
B. False
Quiz #1

True/false: map is tail-recursive.

```
let rec map f = function
  | [] -> []
  | (h::t) -> (f h)::(map f t)
```

A. True
B. False
Quiz #2

True/false: fold is tail-recursive

```
let rec fold f a = function
    [] -> a
  | (h::t) -> fold f (f a h) t
```

A. True
B. False
Quiz #2

True/false: fold is tail-recursive

```
let rec fold f a = function
  | []   -> a
  | (h::t) -> fold f (f a h) t
```

A. True
B. False
Quiz #3

True/false: fold_right is tail-recursive

```
let rec fold_right f l a =
  match l with
  | []   -> a
  | (h::t) -> f h (fold_right f t a)
```

A. True
B. False
Quiz #3

True/false: fold_right is tail-recursive

let rec fold_right f l a =
  match l with
  | [] -> a
  | (h::t) -> f h (fold_right f t a)

A. True
B. False
Tail Recursion is Important

• Pushing a call frame for each recursive call when operating on a list is dangerous
  – One stack frame for each list element
  – Big list = stack overflow!

• So: favor tail recursion when inputs could be large (i.e., recursion could be deep). E.g.,
  – Prefer `List.fold_left` to `List.fold_right`
    • Library documentation should indicate tail recursion, or not
  – Convert recursive functions to be tail recursive
Quiz #4

True/false: this is a tail-recursive `map`

```ocaml
let map f l =
  let rec helper l a =
    match l with
    | [] -> a
    | h::t -> helper t ((f h)::a)
  in helper l []
```

A. True
B. False
Quiz #4

True/false: this is a tail-recursive map

```ocaml
let map f l =
    let rec helper l a =
      match l with
        [] -> a
      | h::t -> helper t ((f h)::a)
    in helper l []
```

A. True
B. False (elements are reversed)
A Tail Recursive `map`

```ocaml
define map f l =
define rec helper l a =
    match l with
    [ ] => a
    | h::t => helper t ((f h)::a)
in rev (helper l [ ])
```

Could instead change `(f h)::a` to be `a@(f h)`

**Q:** Why is the above implementation a better choice?

**A:** $O(n)$ running time, not $O(n^2)$ (where $n$ is length of list)