CMSC 330: Organization of Programming Languages

Tail Recursion
Reverse

let rec rev l = match l with
    [] -> []
  | (x::xs) -> (rev xs) @ [x]

- Pushes a stack frame on each recursive call

\[
\text{rev } [1;2;3] \\
\rightarrow (\text{rev } [2;3]) @ [1] \\
\rightarrow (((\text{rev } [3]) @ [2]) @ [1] \\
\rightarrow ((((\text{rev } []) @ [3]) @ [2]) @ [1] \\
\rightarrow ((([]) @ [3]) @ [2]) @ [1] \\
\rightarrow ([3] @ [2]) @ [1] \\
\rightarrow [3;2] @ [1] \\
\rightarrow [3;2;1]
\]
A Clever Version of Reverse

```ocaml
let rec rev_helper l a = match l with
    []       -> a
  | (x::xs) -> rev_helper xs (x::a)
let rev l = rev_helper l []
```

- No need to push a frame for each call!

```
rev [1;2;3] →
rev_helper [1;2;3] [] →
rev_helper [2;3] [1] →
rev_helper [3] [2;1] →
rev_helper [] [3;2;1] →
[3;2;1]
```

Stack: values of 1
Tail Recursion

• Whenever a function ends with a recursive call, it is called tail recursive
  – Its “tail” 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 rev and rev_helper

```ocaml
let rec rev l =
    match l with
    [] -> []
    | (x::xs) -> (rev xs) @ [x]
```

Waits for recursive call’s result to compute final result

```ocaml
let rec rev_helper l a =
    match l with
    [] -> a
    | (x::xs) -> rev_helper xs (x::a)
```

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
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 #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_left` is tail-recursive

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

A. True
B. False
Quiz #2

True/false: fold_left is tail-recursive

```
let rec fold_left 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

```ocaml
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
Tail Recursion Pattern (1 argument)

let func x =
  let rec helper arg acc =
    if (base case) then acc
    else
      let arg’ = (argument to recursive call)
      let acc’ = (updated accumulator)
      helper arg’ acc’ in (* end of helper fun *)
  in helper x (initial val of accumulator)
;;
Tail Recursion Pattern with factor

let fact x =
  let rec helper arg acc =
    if arg = 0 then acc
    else
      let arg’ = arg – 1 in
      let acc’ = acc * arg in
      helper arg’ acc’ in (* end of helper fun *)
  helper x 1
;;
Tail Recursion Pattern with \text{rev}

\begin{verbatim}
let \text{rev} \ x =
  let rec \text{rev}\_helper \ arg \ acc =
    match \ arg \ with \ [] \ -> \ acc
    | \ h::t \ ->
      let \ arg’ = \ t \ in
      let \ acc’ = \ h::acc \ in
      \text{rev}\_helper \ arg’ \ acc’ \ in
      (* end of helper fun *)
  \text{rev}\_helper \ x \ []
\end{verbatim}

Can generalize to more than one argument, and multiple cases for each recursive call
 Quiz #4

True/false: this is a tail-recursive map

```
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
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

```ml
let map f l = 
  let 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)
Outlook: Is Tail Recursion General?

• A function that is tail-recursive returns at most once (to its caller) when completely finished
  – The final result is exactly the result of a recursive call; no stack frame needed to remember the current call

• Is it possible to convert an arbitrary program into an equivalent one, except where no call ever returns?
  – Yes. This is called continuation-passing style
  – We will look at this later, if we have time