OCaml Tail Recursion

CSC 310 - Programming Languages

Factorial Example

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

fact 3 = 3 * fact 2
       = 3 * 2 * fact 1
       = 3 * 2 * 1 * fact 0
       = 3 * 2 * 1 * 1
       = 3 * 2 * 1
       = 3 * 2
       = 6
```

Factorial Example (Continued)

```
# 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?).
```

Another Factorial Example

```
let fact n =
  let rec aux x a =
    if x = 0 then a
    else aux (x-1) x*a
  in
  aux n 1
```

```
fact 3 = aux 3 1
       = aux 2 3
       = aux 1 6
       = 6
```

Tail Recursion

- When a function's result is completely computed by its recursive call, it is called tail recursive.
- Tail recursive function can be implemented without requiring a stack frame for each call.
- Typical patter is to use an accumulator to build up the result, and return it in the base case.

Example

- Recursive version

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

- Tail Recursive version

```
let sumlist lst =
  let rec helper l a =
    match l with
    | [] -> a
    | (x::xs) -> helper xs (x+a)
  in
  helper l 0
```

Tail Recursion is Important

- Pushing a stack frame for each recursive call when operation on a list is dangerous.
    - One stack frame for each element.
- Favor tail recursion when inputs could be large
    - Prefer `List.fold_left` to `List.fold_right`
    - Convert recursive functions to be tail recursive

Tail Recursive Pattern (One argument)

```
let <fun> x =
  let rec helper arg acc =
    if <base case> then acc
    else
      let arg' = <argument to recursive call> in
      let acc' = <updated accumulator> in
      helper arg' acc'
  in
  helper x <initial value of accumulator>
```

Tail Recursive Pattern with Factorial

```
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
  helper x 1
```

Tail Recursive Pattern with Reverse

```
let reverse x =
  let rec helper arg acc =
    match arg with
    | [] -> acc
    | (h::t) ->
      let arg' = t in
      let acc' = h::acc in
      helper arg' acc'
  in
  helper x []
```

Tail Recursive Map

```
let map f lst =
  let rec helper arg acc =
    match arg with
    | [] -> acc
    | (h::t) -> helper t ((f h)::acc)
  in
  reverse (helper lst [])
```

Generality of Tail Recursion

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