CS 321 Programming Languages

Intro to OCaml – Recursion (tail vs forward)

Baris Aktemur

Özyeğin University

Last update made on Thursday 12th October, 2017 at 11:25.

Much of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at `http://courses.engr.illinois.edu/cs421`

Forward recursion

In forward recursion, you first call the function recursively on all the recursive components, and then build the final result from the partial results.

Wait until the whole structure has been traversed to start building the answer.

```
# let rec tally lst =
    match lst with
    | [] -> 0
    | x::xs -> x + tally xs;;

# let rec squareUp lst =
    match lst with
    | [] -> []
    | x::xs -> (x*x)::squareUp xs;;
```

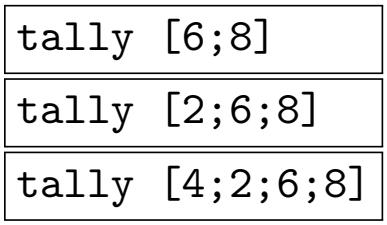
Functions calls and the stack

```
tally [4;2;6;8]
```

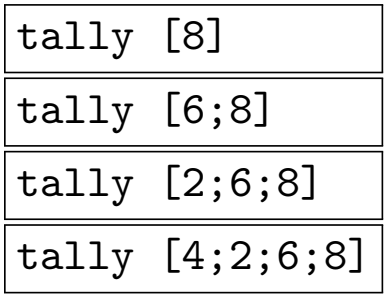

Functions calls and the stack

```
tally [2;6;8]
tally [4;2;6;8]
```

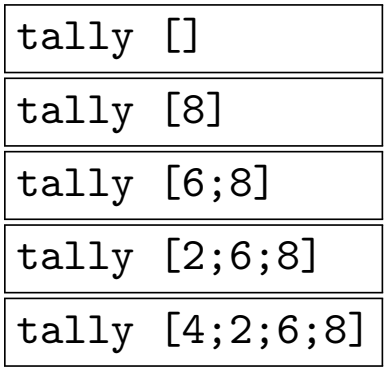
Functions calls and the stack



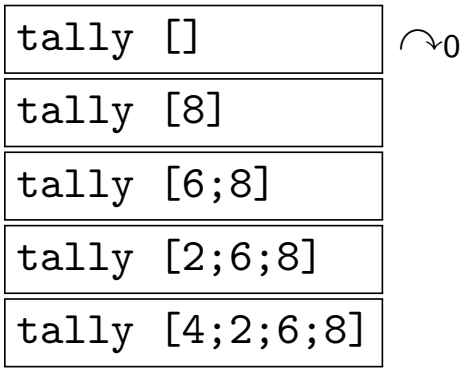
Functions calls and the stack



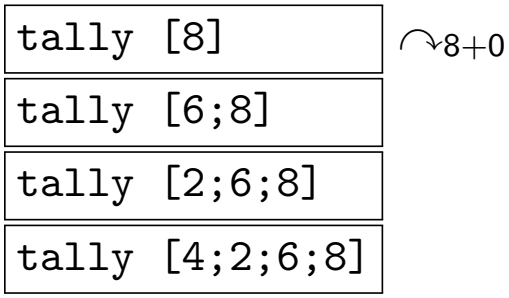
Functions calls and the stack



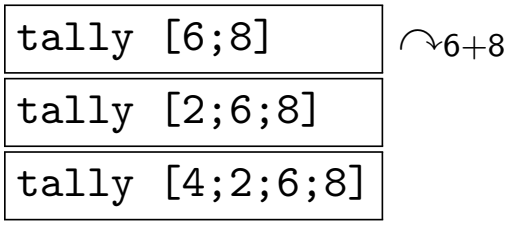
Functions calls and the stack



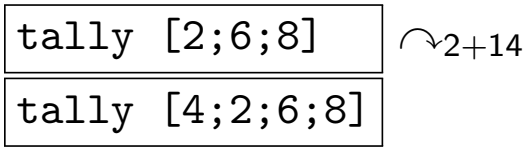
Functions calls and the stack



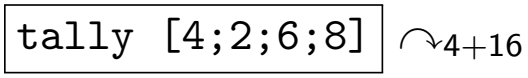
Functions calls and the stack



Functions calls and the stack



Functions calls and the stack



Functions calls and the stack

20

Tail recursion

A recursive function is tail-recursive if all recursive calls are the last thing that the function does.

Tail recursion generally requires extra "accumulator" arguments to pass partial results.

- May require an auxiliary function.

```
# let rec tally lst acc =
    match lst with
    | [] -> acc
    | x::xs -> tally xs (x+acc);;

val tally : 'a list -> int -> int

# tally [1;2;3;4;5] 0;;
(* Have to give an initial accumulated value *)
val it : int = 15
```

Functions calls and the stack

```
tally [4;2;6;8] 0
```


Functions calls and the stack

```
tally [2;6;8] 4
tally [4;2;6;8] 0
```

Functions calls and the stack

```
tally [6;8] 6
tally [2;6;8] 4
tally [4;2;6;8] 0
```


Functions calls and the stack

```
tally [8] 12
tally [6;8] 6
tally [2;6;8] 4
tally [4;2;6;8] 0
```

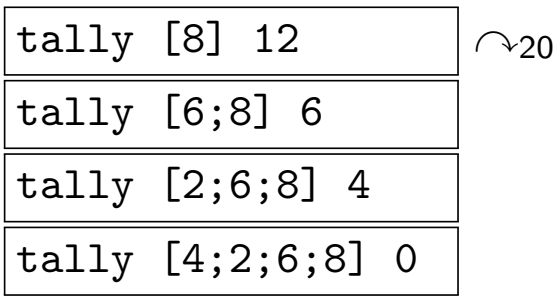
Functions calls and the stack

```
tally [] 20
tally [8] 12
tally [6;8] 6
tally [2;6;8] 4
tally [4;2;6;8] 0
```

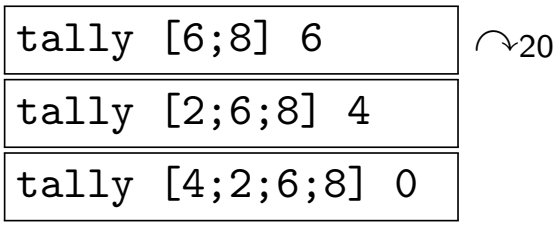

Functions calls and the stack

```
tally [] 20
tally [8] 12
tally [6;8] 6
tally [2;6;8] 4
tally [4;2;6;8] 0
```

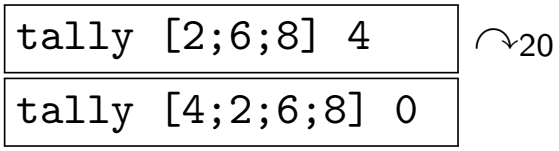
Functions calls and the stack



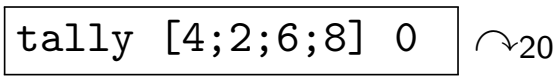
Functions calls and the stack



Functions calls and the stack



Functions calls and the stack



Özyeğin University — CS 321 Programming Languages 5

Functions calls and the stack

20

An optimization idea

Observation

When we have tail recursion, a function that is waiting for the called function to return a value simply propagates the return value to its caller.

Idea

We don't need to keep the frame of a tail-recursive function on the stack. Simply replace the tail-recursive function's stack frame with the called function.

Functions calls and the stack

```
tally [4;2;6;8] 0
```


Functions calls and the stack

```
tally [2;6;8] 4
```

Functions calls and the stack

```
tally [6;8] 6
```

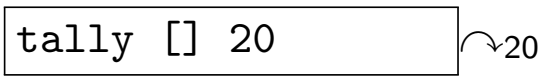

Functions calls and the stack

```
tally [8] 12
```

Functions calls and the stack

```
tally [] 20
```


Functions calls and the stack



Functions calls and the stack

20

Why do we care?

Reusing the stack frame of the tail-recursive function is known as the **tail call optimization**. It is an automatic optimization applied by the compilers and interpreters.

Experiment

Write a function that takes a value `x` and an integer `n`, and returns a list of length `n` whose elements are all `x`.

```
# let rec makeList x n =
```

```
val makeList : 'a -> int -> 'a list = <fun>

# makeList "a" 5;;
- : string list = ["a"; "a"; "a"; "a"; "a"]
# makeList 3 40;;
- : int list =
  [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3]
```


Experiment

Write a function that takes a value `x` and an integer `n`, and returns a list of length `n` whose elements are all `x`.

```
# let rec makeList x n =
    if n = 0 then []
    else x :: makeList x (n-1);;

val makeList : 'a -> int -> 'a list = <fun>

# makeList "a" 5;;
- : string list = ["a"; "a"; "a"; "a"; "a"]
# makeList 3 40;;
- : int list =
  [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3]
```

Experiment

```
# makeList 3 99999;;
- : int list =
  [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   ...]

# makeList 3 1234567;;

Stack overflow during evaluation (looping recursion?).
```


Experiment

```
# let rec makeList x n acc =
    if n = 0 then acc
    else makeList x (n-1) (x::acc);;

val makeList : 'a -> int -> 'a list -> 'a list = <fun>

# makeList 3 10 [];;
- : int list = [3; 3; 3; 3; 3; 3; 3; 3; 3; 3]

# makeList 3 12345678 [];;
- : int list =
  [3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3; 3;
   ...]
```

Observation

Hmm... So there really is a difference between tail vs. forward recursion.

How to write tail-recursive functions?

To write functions in tail-recursive form, answer the following question:
What information do I need to pass from the caller to the callee (i.e. from a lower stack frame to the upper stack frame) so that I won't need the caller again, and can simply throw it away?

Tail recursion

```
# let rec squareUp lst =
    match lst with
    | [] -> []
    | x::xs -> (x*x)::squareUp xs;;
val squareUp : int list -> int list = <fun>

# let rec squareUp lst acc =
    match lst with
    | [] -> acc
    | x::xs -> squareUp xs (acc@[x*x]);;
val squareUp : int list -> int list -> int list = <fun>

# squareUp [1;2;3;4;5] [];;
- : int list = [1; 4; 9; 16; 25]
```

Exercise

Convert the following functions to tail-recursive form.

```
# let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1);;

# let rec power x n =
    if n = 0 then 1
    else x * power x (n - 1);;

# let rec fib n =
    if n = 0 then 1 else
    if n = 1 then 1 else
    fib(n-1) + fib(n-2);;
```


Exercise solutions

```
# let rec factorial n acc =
    if n = 0 then acc
    else factorial (n-1) (n*acc);;

val factorial : int -> int -> int

# factorial 0 1;;
- : int = 1
# factorial 1 1;;
- : int = 1
# factorial 5 1;;
- : int = 120
# factorial 6 1;;
- : int = 720
```

Exercise solutions

```
# let rec fib n nm1 nm2 =
    if n = 0 then nm2
    else if n = 1 then nm1
    else fib (n-1) (nm1+nm2) nm1;;
val fib : int -> int -> int -> int = <fun>

# fib 0 1 1;; (* Initial values for nm1 and nm2 are 1 and 1 *)
- : int = 1
# fib 1 1 1;;
- : int = 1
# fib 2 1 1;;
- : int = 2
# fib 3 1 1;;
- : int = 3
# fib 4 1 1;;
- : int = 5
# fib 5 1 1;;
- : int = 8
# fib 6 1 1;;
- : int = 13
```


Better Programming

Use an auxiliary function to hide the accumulator from the user.

```
# let tally numbers =
    let rec sum lst acc =
      match lst with
      | [] -> acc
      | x::xs -> sum xs (x+acc)
    in sum numbers 0;;
val tally : int list -> int = <fun>

# let squareUp numbers =
    let rec aux lst acc =
      match lst with
      | [] -> acc
      | x::xs -> aux xs (acc@[x*x])
    in aux numbers [];;
val squareUp : int list -> int list = <fun>
```

Better Programming

Use an auxiliary function to hide the accumulator from the user.

```
# let factorial n =
    let rec fact m acc =
      if m = 0 then acc
      else fact (m-1) (m*acc)
    in fact n 1;;
val factorial : int -> int = <fun>

# let fib m =
    let rec aux n nm1 nm2 =
      if n = 0 then nm2
      else if n = 1 then nm1
      else aux (n-1) (nm1 + nm2) nm1
    in aux m 1 1;;
val fib : int -> int = <fun>
```


Exercise

Convert the following function to tail-recursive form.

```
# let rec rev aList =
    match aList with
    | [] -> []
    | x::xs -> rev xs @ [x];;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3;4;5;6];;
- : int list = [6; 5; 4; 3; 2; 1]
```

Exercise

Convert the following function to tail-recursive form.

```
# let rec rev aList =
    match aList with
    | [] -> []
    | x::xs -> rev xs @ [x];;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3;4;5;6];;
- : int list = [6; 5; 4; 3; 2; 1]

# let rev aList =
    let rec aux lst acc =
      match lst with
      | [] -> acc
      | x::xs -> rev xs (x::acc)
    in aux aList [];;
val rev : 'a list -> 'a list = <fun>
```


Tail vs. Forward

Note:
In general, it is possible for a recursion to be neither forward nor tail. This is the case when a function does some stuff, makes the recursive call, then further processes the return value of the recursive call. For simplicity and without loss of the point we are making in this lecture, we consider only forward vs. tail recursion.

Exercise

Convert the following function to tail-recursive form.

```
# let rec numZeros lst =
    match lst with
    | [] -> 0
    | 0::xs -> 1 + numZeros xs
    | x::xs -> numZeros xs;;
```


How long will it take?

- Remember the big-oh notation from CS201?
- Question: given input of size n, how long to generate output?
- Express output time in terms of input size, omit constants and take the biggest power.

Common big-O times

- Constant time $O(1)$
  - input size doesn't matter
- Linear time $O(n)$
  - double input $\implies$ double time
- Quadratic time $O(n^2)$
  - double input $\implies$ quadruple time
- Exponential time $O(2^n)$
  - increment input $\implies$ double time

Linear time

- Expect most list operations to take linear time $O(n)$.
- Each step of the recursion can be done in constant time.
- Each step makes only one recursive call.
- List example: `length`, `squareUp`, `append`
- Integer example: `factorial`

```
# let rec length lst =
    match lst with
    | [] -> 0
    | x::xs -> 1 + length xs;;
```

Quadratic time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

```
let rec poorRev lst =
  match lst with
  | [] -> []
  | x::xs -> poorRev xs @ [x];;
(* Compare poorRev to the function below *)
let rec rev_aux lst acc =
  match lst with
  | [] -> acc
  | x::xs -> rev_aux xs (x::acc);;

let rev lst = rev_aux lst [];;
```


Comparison

```
poorRev [1;2;3] =
(poorRev [2;3])@[1] =
((poorRev[3])@[2])@[1] =
(((poorRev[])@[3])@[2])@[1] =
(([]@[3])@[2])@[1] =
([3]@[2])@[1] = (* append is linear *)
(3::([]@[2]))@[1] =
[3;2]@[1] =
3::([2]@[1]) =
3::(2::([]@[1])) =
[3;2;1]
```

Comparison

```
rev [1;2;3] =
rev_aux [1;2;3] [] =
rev_aux [2;3] [1] =
rev_aux [3] [2;1] =
rev_aux [] [3;2;1] =
[3;2;1]
```


Exponential time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naive code that is exponential for functions that can be linear

Exponential time

```
let rec naiveFib n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> naiveFib (n-1) + naiveFib (n-2);;

let fib n =
  let rec tailFib n nm1 nm2 =
    if n = 0 then nm2 else
    if n = 1 then nm1 else
    tailFib (n-1) (nm1+nm2) nm1
  in tailFib n 1 1;;
```

Experiment

Run the two versions with various big inputs. (e.g. 40) What difference do you see?