Lecture 36: Tail Recursion, continued & Analysis Review
10:00 AM, Nov 30, 2018

Contents

1 make_list 1
2 Justification 2
3 From append to reverse and back again 3
4 Review of analysis so far 4
   4.1 Big O and Big Omega 4

1 make_list

Write a procedure that, given an integer $n$, forms a list consisting of the integers 1 through $n$ (in some order).

The code for such a procedure using traditional recursion looks like the following:

```ocaml
let rec make_list : int -> int list = function
  0 -> []
| n -> n::(make_list (n-1));;
```

Let's see if it works:

```
# make_list 10;;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

It works! But what happens when we try to run this procedure on an absurdly high value for $n$?

```
# make_list 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Why is this happening? The reason is because the results of each recursive need to be saved in order to properly cons each value of $n$ onto the rest of the list. This overloads the memory and causes a 'stack overflow'.

How do we get around this? We can make it tail-recursive.

**Quiz:** Write a tail-recursive version of make_list. (Hint: the order of integers is allowed to be different.)

**Answer:**
let rec make_list_helper : int * int list -> int list = function
  (0, aloi) -> lst
| (n, aloi) -> make_list_helper(n-1, n::aloi) ;

let make_list : int -> int list = function
  n -> make_list_helper(n, []) ;

Now, we have no problem making lists even of length 100 million!

let l = make_list (100000000, []) ;;

Quiz: Write the spec for make_list_helper
Answer:

- **input:** pair consisting of nonnegative integer \( n \), and int list
- **output:** list consisting of the first \( n \) positive integers, followed by the elements of the input list

2 Justification

Last time we wrote reverse tail-recursively but perhaps we didn’t do a good job of convincing you that it is correct.

let rec rev_helper : 'a list * 'a list -> 'a list = function
  [], result -> result
| first::rest, result -> rev_helper (rest, first::result) ;;

let rec rev : 'a list -> 'a list = function
  lst -> rev_helper (lst, []) ;;

This is tail-recursive (the output from a call equals the recursive output)—yay!—and it also turns out to be a linear-time procedure—yay!

To understand how it works, let’s write the spec for rev_helper. To get you started, here is an example:

```# rev_helper ([1;2;3], [4;5;6]);;- : int list = [3; 2; 1; 4; 5; 6]```

Quiz: Write the spec for rev_helper
Answer:

- **input**: A pair consisting of a list `lst` and a list `partial`
- **output**: A list consisting of the elements of `lst` in reverse order followed by the elements of `partial`.

Is this consistent with the code we wrote?

First, is it consistent with the base case? `([], result -> result)`

Yes, if `lst` is empty then the spec says the output includes the elements of `partial` in the order in which they appear.

Is it consistent with the recursive case? `(first::rest, result -> rev_helper (rest, first::result))`

According to the spec, the value of the right-hand side is the list consisting of

- the elements of `rest` in reverse order, followed by
- the elements of `first::result`

This is the same as

- the elements of `first::rest` in reverse order, followed by
- the elements of `result`

This seems to indicate that the program code is consistent with the spec we chose.

Assuming that `rev_helper` correctly implements that spec, what about the procedure `rev` itself? Does it correctly return the reverse?

The procedure `rev` given input `lst` returns the value of `rev_helper (lst, [])`. According to the spec of `rev_helper`, this should be the list consisting of the elements of `lst` in reverse order, followed by the elements of the empty list. That is the same as just the list consisting of the elements of `lst` in reverse order, which is what `rev` is supposed to return on input `lst`.

### 3 From append to reverse and back again

Last time we

1. Noted that OCaml has a built-in `append` operator
2. Wrote our own `append`
3. Noted that it runs in linear time. (Some of you in the Monday quiz seemed to think that OCaml’s `append` operator takes constant time but it runs in linear time)

Next, we wrote `reverse`. Our `reverse` was based on `append` and, consequently, takes quadratic time.
Next, we wrote a tail-recursive reverse. It turned out to be linear time.

Now, let’s write a tail-recursive append. Why? What will happen if you try our own append procedure on a big list? Stack overflow.

**Quiz:** Write a tail-recursive version of append.

- Hint 1: You don’t need an extra argument
- Hint 2: Result might not come out the way you expect or want.

```ocaml
let rec tail_append_helper : 'a list * 'a list -> 'a list =
  function (alod1, alod2) ->
    match alod1 with
      [] -> alod2
    | hd::tl -> tail_append_helper(tl, hd::alod2) ;;
```

Try it out:

```
# tail_append_helper ([1;2;3], [4;5;6]);;
- : int list = [3; 2; 1; 4; 5; 6]
```

What is the spec of this procedure?

- Input: two lists, alod1 and alod2
- Output: the list consisting of the elements in alod1 in reverse order, followed by the elements of alod2

Can we write append in terms of this helper?

```ocaml
let append = function (alod1, alod2) ->
  tail_append_helper((rev alod1), alod2) ;;
```

4 Review of analysis so far

4.1 Big O and Big Omega

If there are constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \) then we say \( f \) is \( O(g) \).

Informally, “\( f(n) \) is at most a constant times \( g(n) \) as \( n \) grows” or “\( f(n) \) is eventually at most a constant times \( g(n) \).”

This only makes sense if \( n \) is allowed to be arbitrarily large.

If there are constants \( c \) and \( n_0 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \) then we say \( f \) is \( \Omega(g) \).

Informally, “\( f(n) \) is at least a constant times \( g(n) \) as \( n \) grows” or “\( f(n) \) is eventually at least a constant times \( g(n) \).”
“f(n) is O(g(n))” is the same as “g(n) is Ω(f(n)).”

If f(n) is O(g(n)) and is Ω(g(n)), we say f is Θ(g).

Example: f(n) = 2n and g(n) = 3n. Even though g(n) has a higher slope, we say f is Θ(g).

When writing a function (especially inside big-O or big-Ω or big-Θ), we often use an abbreviation for writing the function:

- The function g such that g(n) = n is denoted “n”

- The function g such that g(n) = n log n is denoted “n log n”.

In general, we denote the function by an expression in n (or whatever variable, but usually n) that represents the value of the function for a given value of n.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)