Lec 4 Tail Recursion?

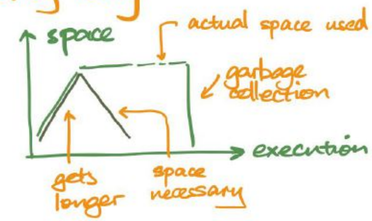
* Problem with non-tail recursion.

Consider [1,2,3,4]

```
length [1,2,3,4] => 1+ length [2,3,4]
                 => 1+(1+ length [3,4])
                 => 1 +(1+(1+ length [4]))
                 => 1 +(1 +(1 +(1 + length[])))
                 => 1 + (1+ (1+ (1+ 0)))
             ... => 4
```

Gets very long!
In fact unnecessarily long

Time O(n).
Space O(n).



* Save space → less garbage collection → faster run time

* Here comes tail recursion.

```
(* tlength : int list * int -> list
   REQ: true
   ENS: tlength(L, acc) ≅ length(L) + acc
*)
fun tlength ([]: int list , acc : int ) : int = acc
  | tlength (_ ::xs , acc ) = tlength ( xs, 1+acc )

fun length'( L: int list ) = tlength (L, 0).
```

accumulator

Notice it doesn't just take a list

Tail call: recursive call not adding anything.

```
length' [1,2,3,4] => tlength( [1,2,3,4], 0)
                  => tlength ( [2,3,4] , 1)
              ... => 4
```

Time O(n)
Space O(1)

* "tail recursive" if:
  - It's recursive
  - All recursive calls are tail calls

Prove that tlength works correctly

Theorem: $\forall L$: int list, acc: int, tlength (L, acc) $\cong$ length(L) + acc

Proof by structural induction on L.

Base case. When L=[ ].

WTS tlength(L, acc) $\cong$ length([ ]) +acc for any acc.

tlength ([ ], acc) $\Rightarrow$ acc [by clause 1 of tlength]

length ([ ]) + acc $\Rightarrow$ 0 + acc [by clause 1 of length]

$\Rightarrow$ acc [by math]

So tlength ([ ], acc) $\cong$ length ([ ]) + acc

* Reduction is equivalence

Inductive case.

Let L= x::xs for values x, xs

IH: tlength (xs, acc') $\cong$ length (xs) + acc' $\forall$ acc'

WTS: tlength (x::xs, acc) $\cong$ length (x:: xs) + acc $\forall$ acc

Well, tlength (x::xs, acc) $\cong$ tlength (xs, 1+acc) [by clause 2 of tlength]

$\cong$ length (xs) + (1+acc) [by IH with acc' = 1+acc]

Expression, not value, but we can treat it as value for $\cong$ proof plus is total.

and since length is total

we can always go backward since $\cong$ is symmetric

$\cong$ (1+ length (xs)) + acc [by math]

$\cong$ length (x::xs) + acc [by clause 2 of length]

Example

```
(* append  int list * int list -> int list *)
fun  append ( [ ]: int list , Y: int list ): int list  =  Y
     append ( x:: xs,  Y)  =  x :: append (xs, Y)
```

Time: O(length of 1st list)

reverse function

```
(* rev  int list -> int list *)
fun  rev ( [ ] :int list ): int list = [ ]
  |  rev ( x :: xs ) = rev xs @ [x]
```

Time: $O(n^2)$. Bad.

append ← O(len of list before it)

```
(* trev  int list * int list -> int list
   REQ : true
   ENS : trev (L, acc) ≅ rev (L) @ acc
*)
fun trev ([] : int list , acc : int list ) : int list = acc
  | trev ( x :: xs , acc ) = trev ( xs , x::acc )

fun rev' ( L: int list ) : int list = trev( L , [])
```

— Now time $O(n)$

Theorem: $\forall$ values L, acc : int list, trev(L, acc) ≅ rev(L)@acc

Proof by structural induction on L.

Base case. WTS: trev ([], acc) ≅ rev ([]) @ acc $\forall$ acc

trev ([] , acc) ≅ acc [trev 1]
≅ []@acc [append 1]
≅ rev([])@acc [rev 1]

Inductive case. Let $L = x::xs$ for values x, xs

IH: $trev(xs, acc') \cong rev\ xs\ @\ acc' \quad \forall acc'$

WTS: $trev(x::xs, acc) \cong rev\ rev(x::xs)\ @\ acc \quad \forall acc$

$$trev(x::xs, acc) \cong trev(xs, x::acc) \quad [trev2]$$
$$\cong rev\ xs\ @\ (x::acc) \quad [IH]$$

will get posted