Equations for len:

1. len [] = 0

2. len (x :: xs) = 1 + len xs for any x : 'a, xs : 'a list

THEOREM (append adds length).

For any lists l1, l2 : 'a list, we have:

len l1 + len l2 = len (app l1 l2)

PROOF. By induction on l1.

CASE. l1 = [].

WTS: len [] + len l2 = len (app [] l2)

LHS = len [] + len l2

= 0 + len l2 -- by defn of len

= len l2 -- by common sense

RHS = len (app [] l2)

= len l2 -- by defn of app

= LHS

CASE. l1 = x :: xs.

WTS: len (x :: xs) + len l2 = len (app (x :: xs) l2)

IH: len xs + len l2 = len (app xs l2)

LHS = len (x :: xs) + len l2

= 1 + len xs + len l2 -- by defn len

= 1 + len (app xs l2) -- by IH

= len (x :: app xs l2) -- by len, backwards

= len (app (x :: xs) l2) -- by app, backwards

= RHS QED.

THEOREM (map preserves length).

For any list l : 'a list, and any f : 'a -> 'b, we have:

len l = len (map f l)

PROOF. By induction on l.

CASE. l = [].

WTS: len [] = len (map f [])

LHS = len [] = 0 -- by defn of len

RHS = len (map f [])

= len [] -- by defn of map

= 0 -- by defn of len

CASE. l = x :: xs.

WTS: len (x :: xs) = len (map f (x :: xs))

IH: len xs = len (map f xs)

LHS = len (x :: xs)

= 1 + len xs -- by defn of len

= 1 + len (map f xs) -- by IH

Approach 1 : backwards definitions:

= len (f x :: map f xs) -- by len, backwards

= len (map f (x :: xs)) -- by map, backwards

= RHS QED.

THEOREM (Map distributes over append.)

For any l1, l2: 'a list and any f: 'a -> 'b, we have:

app (map f l1) (map f l2) = map f (app l1 l2)

CASE. l1 = []

NTS: app (map f []) (map f l2) = map f (app [] l2)

LHS = app (map f []) (map f l2)

= app [] (map f l2) -- by map

= mao f l2 -- by app

RHS = map f (app [] l2)

= map f l2

= LHS

CASE. l1 = x :: xs.

NTS: app (map f (x :: xs)) (map f l2) = map f (app (x :: xs) l2)

IH: app (map f xs) (map f l2) = map f (app xs l2)

LHS = app (map f (x :: xs)) (map f l2)

= app (f x :: map f xs) (map f l2) -- by map

= f x :: app (map f xs) (map f l2) -- by app

= f x :: map f (app xs l2) -- by IH

RHS = map f (app (x :: xs) l2)

= map f (x :: app xs l2)

= f x :: map f (app xs l2)

= LHS QED.

THEOREM. (sum and sum\_tr are equivalent. Take 2.Need to generalize problem!)

For any l: int list and any acc: int, we have:

Acc + sum l = sum\_tr l acc

PROOF. By induction on l

Case l = []

NTS: acc + sum [] = sum\_tr [] acc

LHS = acc + sum []

= acc + 0 -- by def of sum

= acc -- by common sense

RHS = sum\_tr [] acc

= acc - by def of sum\_tr

= LHS

Case l = x :: xs

NTS: acc + sum (x :: xs) = sum\_tr (x :: xs) acc

IH: for any acc: int, we have: acc + sum xs = sum\_tr xs acc

LHS = acc + sum (x :: xs)

= acc + (x + sum xs)

RHS = sum\_tr (x :: xs) acc

= sum\_tr xs (acc +x)

= (acc + x) + sum xs -- by IH with acc := acc + x

-- specialized IH: (acc + x) + sum xs = sum\_tr xs (acc + x)

= LHS -- by common sense

let rec change (coins : int list) (amt : int) : int list =

match coins with

| \_ when amt = 0 -> []

| [] -> raise Change (\* nonzero change to make but no more coins to try! \*)

| c :: cs when c > amt ->

change cs amt

| c :: cs -> try c :: change (c :: cs) (amt - c)

with | Change -> change cs amt

let rec change\_cps (coins : int list) (amt : int) (return : int list -> 'r) (backtrack : unit -> 'r) : 'r = match coins with

| \_ when amt = 0-> return []

| [] -> backtrack ()

| c :: cs when c > amt -> change\_cps cs amt return backtrack

| c :: cs ->

change\_cps (c :: cs) (amt - c)

(fun soln -> return (c :: soln))

(fun () -> change\_cps cs amt return backtrack)

let rec repeat (x : 'a) : 'a str = { hd = x;

tl = Susp (fun () -> repeat x) }

let rec nats\_from n = { hd = n;

tl = Susp (fun () -> nats\_from (n+1)) ; }

let rec iterate (f : 'a -> 'a) (x : 'a) : 'a str = {hd = x; tl = Susp (fun () -> iterate f (f x))}

let rec unfold (f : 'state -> 'elem \* 'state) (s : 'state) : 'elem str = let (x, s') = f s in { hd = x; tl = Susp (fun () -> unfold f s') }

fib : unfold (fun (a, b) -> (a, (b, a +b))) (0, 1)

let susp\_map (f : 'a -> 'b) (s : 'a susp) : 'b susp = Susp (fun () -> f (force s))

let rec str\_map (f : 'a -> 'b) (s : 'a str) : 'b str = { hd = f s.hd;

tl = susp\_map (str\_map f) s.tl; }

let rec add\_streams (s1 : int str) (s2 : int str) : int str = { hd = s1.hd + s2.hd;

tl = Susp (fun () -> add\_streams (force s1.tl) (force s2.tl)) }