# [SPOILERS] Practice Problems for Section 3 -- Solutions

You are subscribed. Unsubscribe PINNED

No tags yet. + Add Tag

Sort replies by: Oldest first Newest first Most popular

Pavel Lepin COMMUNITY TA · a month ago

```
(** High-Order Fun **)

(* There Can Be Only One *)

fun fold_map f = List.foldr (fn (x, acc) => f x :: acc) []

fun fold_filter p = List.foldr (fn (x, acc) => if p x then x :: acc else acc) []

(* The Evil Twin *)
fun unfold f state =
    case f state of
        NONE => []
      | SOME (state', x) => x :: unfold f state'

(* A Novel Approach *)
val factorial = (List.foldl (op * ) 1) o (unfold (fn x => if x = 0 then NONE else SOME (x
 - 1, x)))

(* Answer to bonus question: No. Simple tail-recursive
   factorial works in constant space, while unfold here
   constructs a list of size n. In later sessions we
   will study a data structure that would make
   virtually the same implementation efficient. *)

(* Unforeseen Developments *)
fun unfold_map f =
    let
        fun helper param =
            case param of
                [] => NONE
              | x :: xs => SOME (xs, f x)
    in
        unfold helper
    end

(* So Imperative *)
fun do_until f p x =
    if p x
    then x
    else do_until f p (f x)

(* Yet Another Factorial *)
fun imp_factorial n = #1 (do_until (fn (acc, x) => (acc * x, x - 1)) (fn (_, x) => x = 0)
 (1, n))

(* Fixed Point *)
fun fixed_point f = do_until f (fn x => f x = x)

(* Newton's Method *)
fun my_sqrt n =
    let
        fun fixed_point f x = do_until f (fn x => abs (f x - x) < 0.0001) x
    in
        fixed_point (fn x => 0.5 * (x + n / x)) n
    end

(* Deeper into the woods *)

(* provided definition *)
datatype 'a tree = leaf | node of { value : 'a, left : 'a tree, right : 'a tree }

fun tree_fold f base t =
    case t of
        leaf => base
      | node { value = value, left = left, right = right } => f (tree_fold f base left, v
alue, tree_fold f base right)

fun tree_unfold f state =
    case f state of
        NONE => leaf
      | SOME (lstate, value, rstate) => node { value = value, left = tree_unfold f lstate
, right = tree_unfold f rstate }

(** A Grand Challenge **)

(* provided definitions *)
datatype expr = literal_bool | literal_int | binary_bool_op of expr * expr | binary_int_o
p of expr * expr | comparison of expr * expr | conditional of expr * expr * expr
datatype expr_type = type_bool | type_int

exception TypeError
```

```
fun infer_type expr =
    case expr of
        literal_bool => type_bool
      | literal_int => type_int
      | binary_bool_op (x1, x2) =>
            if infer_type x1 = type_bool andalso infer_type x2 = type_bool
            then type_bool
            else raise TypeError
      | binary_int_op (x1, x2) =>
            if infer_type x1 = type_int andalso infer_type x2 = type_int
            then type_int
            else raise TypeError
      | comparison (x1, x2) =>
            if infer_type x1 = type_int andalso infer_type x2 = type_int
            then type_bool
            else raise TypeError
      | conditional (x1, x2, x3) =>
            let
                val t2 = infer_type x2
                val t3 = infer_type x3
            in
                if infer_type x1 = type_bool andalso t2 = t3
                then t2
                else raise TypeError
            end

(** Back To The Future! 2 **)

(* GCD -- Final Redux *)

(* provided helper function *)
fun gcd (a : int, b : int) =
    if a = b
    then a
    else
        if a < b
        then gcd (a, b - a)
        else gcd (a - b, b)

fun gcd_list xs = List.foldl gcd (hd xs) (tl xs)

(* Element Of A List -- Final Redux *)

(* provided helper function *)
fun is_divisible_by (a : int, b : int) = a mod b = 0

fun any_divisible_by (xs, divisor) = List.exists (fn x => is_divisible_by (x, divisor)) x
```

```
s

(* Quirky Addition -- Continued -- Final Redux *)
val add_all_opt =
    let
        fun helper param =
            case param of
                (SOME x, SOME acc) => SOME (acc + x)
              | (NONE, acc) => acc
              | (x, _) => x
    in
        List.foldl helper NONE
    end

(* Flip Flop -- Final Redux *)
val alternate = #2 o (List.foldl (fn (x, (factor, acc)) => (~factor, factor * x + acc)) (
1, 0))

(* Minimum/Maximum -- Final Redux *)
fun min_max (x :: xs) = List.foldl (fn (x, (min, max)) => (if x < min then x else min, if
 x > max then x else max)) (x, x) xs

(* Lists And Tuples, Oh My! -- Final Redux *)
fun unzip xs = List.foldr (fn ((x, y), (xs, ys)) => (x :: xs, y :: ys)) ([], []) xs

(* Lists And Tuples, Oh My! -- Continued (1) -- Final Redux *)
fun zip xs =
    let
        fun helper param =
            case param of
                (x :: xs, y :: ys) => SOME ((xs, ys), (x, y))
              | _ => NONE
    in
        unfold helper xs
    end

(* BBCA -- Final Redux *)
fun repeats_list xs =
    let
        fun helper param =
            case param of
                (_ :: xs, 0 :: reps) => helper (xs, reps)
              | (x :: xs, n :: reps) => SOME ((x :: xs, n - 1 :: reps), x)
              | _ => NONE
    in
        unfold helper xs
    end
```

```
(* 38 Cons Cells -- Final Redux *)
fun length_of_a_list xs = List.foldl (fn (_, x) => x + 1) 0 xs

(* Forest For The Trees -- Final Redux *)

fun tree_height t = tree_fold (fn (l, _, r) => 1 + Int.max (l, r)) 0 t

fun sum_tree t = tree_fold (fn (l, v, r) => l + v + r) 0 t

(* provided definition *)
datatype flag = leave_me_alone | prune_me

fun gardener t =
    let
        fun helper state =
            case state of
                node { value = prune_me, left = _, right = _ } => NONE
              | node { value = value, left = left, right = right } => SOME (left, value,
right)
              | _ => NONE
    in
        tree_unfold helper t
    end
```

0 · flag

Nikolai Saeverud · a month ago

I guess I am busy doing practice problems today. I did the unfold_map and came up with two alternative solutions. I wondered if you could comment on the style and signature of the two alternatives. Why would the unfold_map2 have ''a instead of 'a?

val unfold_map2 = fn : (''a -> 'b) -> ''a list -> 'b list
fun unfold_map2 f = unfold (fn xs => if xs = [] then NONE else SOME (tl xs, f (hd xs)))

val unfold_map3 = fn : ('a -> 'b) -> 'a list -> 'b list
fun unfold_map3 f = unfold (fn xs => case xs of [] => NONE | x :: xs' => SOME (xs', f x))

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

Only the values in so-called "equality types" can be compared for equality by using `=`. A list type is an equality type if and only if the type of values contained in the list is also an equality type. This constraint is expressed as `''a list`. Since your first version of `unfold` compares `xs` to an empty list using `=`, SML's type system infers that `xs` must belong to an equality type. I'd recommend reviewing "Polymorphic and Equality Types" in section 2 if this doesn't make a whole lot of sense.

↑ 1 ↓ · flag

Nikolai Saeverud · a month ago

I will review it that lecture. It is a lot to take in, but is it ever rewarding when things start to come together. Is it ok, though to use the anonymous function in unfold_map3 instead of the let expression in the provided solution?

↑ 0 ↓ · flag

Pavel Lepin COMMUNITY TA · a month ago

Certainly, I just prefer to define a named function for expressions involving `case` / pattern matching.

↑ 1 ↓ · flag

Pavel Lepin COMMUNITY TA · a month ago

To clarify a little: overall, I was trying to make the reference solutions very close in style to examples on the grading rubric for peer assessment, to avoid sending fellow learners any conflicting signals. But certain personal idiosyncrasies do inevitably shine through, and one of the messages of peer assessments is the fact that there's a fairly wide variety of reasonable styles and approaches to solving the problems anyway.

If you do find anything in the solutions that seems not very readable to you, or could be otherwise "objectively" improved upon, please do let me know, I'll consider doing something about this at the very least.

↑ 1 ↓ · flag

Nikolai Saeverud · a month ago

Thank you Pavel. That's perfect. I am practicing doing all the different styles that we have learned, and I am not really at the stage of personal preferences of style. But thank you for the confirmation that one solution is more or less equal to the other. It gives a broader perspective on the possibilities of sml.

↑ 0 ↓ · flag

+ Comment

KL Tah · 22 days ago

the approach i'm trying to take with unfold_map is to pattern match on each element of the input array, have unfold operate only ONCE on that element, then return and take the next element etc. Problem is how do I craft such a condition so it stops after one execution?
I took a peek at the solutions and it involves a helper function but I'm trying to do it in a way that's more transparent for me.

↑ 0 ↓ · flag

KL Tah · 21 days ago

Also, it looks like unfold_map returns a partial application unfold helper, but helper takes in an array. Doesn't this break the typecheck in unfold which requires a single value, not an array?

↑ 0 ↓ · flag

Pavel Lepin COMMUNITY TA · 21 days ago

`unfold_map` takes two arguments -- a function to map over the collection and the collection itself. So does `unfold` -- it takes a function that produces successive states and elements and an initial state. So the initial state for `unfold` will be the collection we want to map over.

↑ 0 ↓ · flag

KL Tah · 21 days ago

Yes, however, initial state is supposed to be of type 'a, whereas the collection is of type 'a list.

```
('a -> ('a * 'b) option) -> 'a -> 'b list
```

So when I try to call

```
unfold some_function [1,2,3]
```

That doesn't work
However, when I call

```
unfold helper [1,2,3]
```

which is essentially what's being called when I call unfold_map, it works! Can you help me understand what I'm missing here? Surely, no function could be so special that the typechecker would want to make concessions as to what types it can receive (it is after all a machine with no feelings), but that's what it seems like for helper.

↑ 0 ↓ · flag

Pavel Lepin COMMUNITY TA · 20 days ago

> initial state is supposed to be of type 'a, whereas the collection is of type 'a list.

Those are `'a`s from *different* signatures.

`unfold`'s signature is:

```
('a -> ('a * 'b) option) -> 'a -> 'b list
```

`unfold_map`'s is (modulo type variable renaming):

```
('c -> 'd) -> 'c list -> 'd list
```

So the type of expression `unfold helper` must be *both* `'a -> 'b list` and `'c list -> 'd list`. We can unify those types by assuming `'a = 'c list` and `'b = 'd`. This also tells us that the type of helper must be `'a -> ('a * 'b) option`, or, in terms of `'c` and `'d`, `'c list -> ('c list * 'd) option`.

0 · flag

KL Tah · 20 days ago

hmm ... i see. i'm not quite understanding what you mean by signatures. it seems here that the typechecker has somehow made an association between a non-list variable and a list variable ('a = 'c list) which isn't usually allowed as far as i can tell (for example i can't simply feed in a list to a function that takes only a single number) so i'm still unsure as to what's special in this case that has allowed the typechecker to make that association in this particular instance.

0 · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B | I | Link | <code> | Pic | Math | Edit: Rich | Preview

☐ Make this post anonymous to other students

☑ Subscribe to this thread at the same time

Add post