CSE 331: Software Design & Engineering Fall 2024

Homework 5

Due: Saturday, November 2nd, 11pm

This worksheet contains only the *written* parts of HW5. The coding instructions will be released Thursday (10/31) and be due Monday (11/4).

While working on this homework, it may be useful to refer to the first page of the Section 5 worksheet where we have a list of function definitions for easy access. As usual, the section tasks are also good practice for the tasks in this homework.

In all proofs, you must use the '::' instead of 'cons'.

Submission

After completing all parts below, submit your solutions on Gradescope. The collection of all written answers to problems described in this worksheet should be submitted as a PDF to **"HW5 Written"**.

Don't forget to check that the submitted file is up-to-date with all written work you completed!

You may handwrite your work (on a tablet or paper) or type it, provided it is legible and dark enough to read. If you're using LaTeX, please make sure your file compiles correctly. When you turn in on Gradescope, please match each HW problem to the page with your work on it. If you fail to have readable work or assign pages, you will receive a point deduction.

Task 1 – Twice to Meet You [18 pts]

The functions twice-evens and twice-odds, both of type List → List, take a list as input and return the list where the numbers at even and odd indexes, respectively, are doubled and the others are left as is. We can define these recursively, in terms of each other ("mutual recursion"), as follows:

$$\begin{aligned} \text{twice-evens}(\text{nil}) &:= \text{nil} \\ \text{twice-evens}(x :: L) &:= 2x :: \text{twice-odds}(L) \\ \\ \text{twice-odds}(\text{nil}) &:= \text{nil} \\ \text{twice-odds}(x :: L) &:= x :: \text{twice-evens}(L) \end{aligned}$$

Now, suppose that you see the following snippet of TypeScript code in some large TypeScript program. The code in the snippet uses `len`, sum, `twice_evens`, and twice_odds, all of which are TypeScript implementations of the mathematical functions with the same names.

```
const x = sum(twice_evens(L));
const y = sum(twice_odds(L));

if (len(L) === 2)
  return x + y;  // = 3 * sum(L)
```

The comment shows the definition of what should be returned, but the code is not a direct translation of that. Below, we will use reasoning to prove that the code is correct.

Note that, if $\text{len}(L) = 2$, then $L = a :: b :: \text{nil}$ for some integers a and b, as we have previously proven in section. We will use that below. (Be sure to use "::" notation in your answers.)

(a) Using the fact that $L = a :: b :: \text{nil}$, prove by calculation that $\text{sum}(L) = a + b$.

(b) Using the same fact, prove by calculation that $\text{sum}(\text{twice-evens}(L)) = 2a + b$.

(c) Using the same fact, prove by calculation that $\text{sum}(\text{twice-odds}(L)) = a + 2b$.

(d) Prove that the code is correct by showing that $x + y = 3\,\text{sum}(L)$, i.e., that

$$\text{sum}(\text{twice-evens}(L)) + \text{sum}(\text{twice-odds}(L)) = 3\,\text{sum}(L)$$

You are free to cite parts (a-c) in your calculation since we know that $L = \text{cons}(a, \text{cons}(b, \text{nil}))$ holds on the line with the return statement. (You can write, e.g., "part (a)" as your explanation on the line that uses the fact proven in part (a).)

Task 2 – Swap 'Til You Drop [16 pts]

This problem uses the following function, swap : List → List, that swaps adjacent values in a list:

$$\begin{aligned} \text{swap}(\text{nil}) &:= \text{nil} \\ \text{swap}(a :: \text{nil}) &:= a :: \text{nil} \\ \text{swap}(a :: b :: L) &:= b :: a :: \text{swap}(L) \end{aligned}$$

Lists of length 0 and 1 are left as is, whereas if the list has length 2 or more, the order of the first two elements are swapped before we recurse on the rest of the list after those two elements.

Suppose you see the following snippet in some TypeScript code. It uses `len` and swap, which are TypeScript implementations of the mathematical functions with the same names.

```
if (len(L) === 3)
  return cons(1, cons(2, L));  // = swap(swap(cons(1, cons(2, L))))
```

The comment shows the definition of what should be returned, but the code is not a direct translation of those. Below, we will use reasoning prove that the code is correct.

The code above uses `cons`, but be sure to use "::" instead in your calculations.

(a) Let x be an integer. Prove that $\text{swap}(\text{swap}(x :: \text{nil})) = x :: \text{nil}$.

(b) Let x and y be integers and R be a list. Prove that

$$\text{swap}(\text{swap}(x :: y :: R)) = x :: y :: \text{swap}(\text{swap}(R))$$

(c) Let a, b, c, d, e be integers and $L = a :: b :: c :: d :: e :: \text{nil}$, i.e., L is some list of length 5. Prove that $\text{swap}(\text{swap}(L)) = L$.

You should apply part (a) once and part (b) multiple times (with different choices of x and y) rather than performing the same calculation again here. (Remember, that those facts we proved hold for *any* values of x and y.)

(d) Prove that the code is correct by showing that $\text{swap}(\text{swap}(\text{cons}(1, \text{cons}(2, L)))) = \text{cons}(1, \text{cons}(2, L))$, using the fact that L has length 3, i.e., that $L = u :: v :: w :: \text{nil}$ for some integers u, v, w.

Feel free to apply prior parts, if useful, rather than performing calculations again.

Task 3 – I Neg To Differ [11 pts]

In Task 3 of Homework 4, we defined a function ns : $\mathbb{Z} \to \mathbb{Z}$ that encoded characters (stored as integer values in the range 0–25) as other characters. In that problem, we claimed that our cipher, which encodes characters, also decodes them. In this problem, you will prove that is the case.

(a) For convenience, repeat your definition of ns from Task 3.

If you know there was an error in your answer from Homework 4, you are free to correct it here. (Be sure to explain what the problem was if you give a different answer.)

(b) Let j be an integer. Prove by cases that $\text{ns}(\text{ns}(j)) = j$.

Hint: If we know that $a \leqslant j \leqslant b$ and c is any integer, then we know that $c-b \leqslant c-j \leqslant c-a$. (The values a and b switch sides because they are negated.) Also, note that depending on how you wrote your definition for ns, your proof may be easier if your cases are more fine-grained than those in your definition.

Task 4 – Fish and Skips [15 pts]

The functions skip and keep, both of type List → List, both drop every other element of the list, with skip skipping the first element (and keeping the second) and keep keeping the first element (and skipping the second). They are defined via mutual recursion as follows:

$$\begin{aligned} \text{skip}(\text{nil}) &:= \text{nil} \\ \text{skip}(x :: L) &:= \text{keep}(L) \end{aligned}$$

$$\begin{aligned} \text{keep}(\text{nil}) &:= \text{nil} \\ \text{keep}(x :: L) &:= x :: \text{skip}(L) \end{aligned}$$

For example, with these definitions, we have $\text{skip}(1 :: 2 :: 3 :: 4 :: \text{nil}) = 2 :: 4 :: \text{nil}$, and we also have $\text{keep}(1 :: 2 :: 3 :: 4 :: \text{nil}) = 1 :: 3 :: \text{nil}$.

We will also need the following function, echo : List → List, which returns a list with an extra copy of every element, producing a list of twice the original length:

$$\begin{aligned} \text{echo}(\text{nil}) &:= \text{nil} \\ \text{echo}(x :: L) &:= x :: x :: \text{echo}(L) \end{aligned}$$

For example, we have $\text{echo}(1 :: 2 :: \text{nil}) = 1 :: 1 :: 2 :: 2 :: \text{nil}$.

(a) Prove, by structural induction, that $\text{skip}(\text{echo}(S)) = S$ for any list S.

(b) You see the following snippet in some TypeScript code:

```
// Return skip(1 :: 2 :: echo(L)), where + is concat
return cons(2, L);  // much faster!
```

The comment tells us what it should return, but the code does not return that, so we will need to use reasoning to check that it is correct.

Show that this code is correct by proving that $\text{skip}(1 :: 2 :: \text{echo}(L)) = 2 :: L$. Feel free to cite part (a).

The next problem will make use of some lists that do not contain integers. We can generalize our inductive List data type to allow it to store any type of data as follows:

$$\textbf{type}\ \mathsf{List}\langle T\rangle := \mathsf{nil} \mid \mathsf{cons}(\mathsf{hd}\colon T,\ \mathsf{tl}\colon \mathsf{List}\langle T\rangle)$$

A declaration like this is called a "generic" (or "parameterized") type. T is a *type* parameter, which we can fill in with any type we want. Filling in a different value for T gives us a different type. Hence, this one definition is creating infinitely many new types.

The type $\mathsf{List}\langle T\rangle$ defines a list that stores elements of type T. The "hd" argument of cons is now a T rather than $\mathbb{Z}$. If we wish to have a list of integers, we would now write that as $\mathsf{List}\langle\mathbb{Z}\rangle$.

The next problem will make use of the following functions that operate on the generic list type.

The function $\mathsf{sum} : \mathsf{List}\langle\mathbb{Z}\rangle \to \mathbb{Z}$, which adds up the numbers in a list, is defined as follows:

$$\begin{array}{lcl} \mathsf{sum}(\mathsf{nil}) & := & 0 \\ \mathsf{sum}(x :: \mathsf{xs}) & := & x + \mathsf{sum}(\mathsf{xs}) \end{array}$$

The function $\mathsf{tail} : \mathsf{List}\langle\mathbb{Z}\rangle \to \mathsf{List}\langle\mathbb{Z}\rangle$, which returns all of the elements in the list *except* for the first one, is defined as follows:

$$\begin{array}{lcl} \mathsf{tail}(\mathsf{nil}) & := & \mathsf{nil} \\ \mathsf{tail}(x :: \mathsf{xs}) & := & \mathsf{xs} \end{array}$$

The function $\mathsf{zip} : (\mathsf{List}\langle\mathbb{Z}\rangle, \mathsf{List}\langle\mathbb{Z}\rangle) \to \mathsf{List}\langle(\mathbb{Z},\mathbb{Z})\rangle$, which turns a pair of lists into a (single) list of pairs of numbers at the same positions in the two lists, is defined as follows:

$$\begin{array}{lcl} \mathsf{zip}(\mathsf{nil}, \mathsf{ys}) & := & \mathsf{nil} \\ \mathsf{zip}(\mathsf{xs}, \mathsf{nil}) & := & \mathsf{nil} \\ \mathsf{zip}(x :: \mathsf{xs}, y :: \mathsf{ys}) & := & (x, y) :: \mathsf{zip}(\mathsf{xs}, \mathsf{ys}) \end{array}$$

The function $\mathsf{mult} : \mathsf{List}\langle(\mathbb{Z},\mathbb{Z})\rangle \to \mathsf{List}\langle\mathbb{Z}\rangle$, which turns a list of pairs into a list of the products of the paired numbers, is defined as follows:

$$\begin{array}{lcl} \mathsf{mult}(\mathsf{nil}) & := & \mathsf{nil} \\ \mathsf{mult}((x, y) :: r) & := & x \times y :: \mathsf{mult}(r) \end{array}$$

Task 5 – Sum As You Are [20 pts]

We wish to calculate the following expression, where xs : List$\langle\mathbb{Z}\rangle$ is a list of integers:

$$\text{sum}(\text{mult}(\text{zip}(\text{xs}, \text{tail}(\text{xs})))) + \text{sum}(\text{xs})$$

The expression "mult(zip(xs, tail(xs)))" creates a list containing the product of each number and the one after it. For example, if xs $= 1 :: 2 :: 3 ::$ nil, then we have:

$$\begin{aligned}
&\text{mult}(\text{zip}(1 :: 2 :: 3 :: \text{nil}, \text{tail}(1 :: 2 :: 3 :: \text{nil}))) \\
&\quad = \text{mult}(\text{zip}(1 :: 2 :: 3 :: \text{nil}, 2 :: 3 :: \text{nil})) && \text{def of tail} \\
&\quad = \text{mult}((1,2) :: \text{zip}(2 :: 3 :: \text{nil}, 3 :: \text{nil})) && \text{def of zip} \\
&\quad = \text{mult}((1,2) :: (2,3) :: \text{zip}(3 :: \text{nil}, \text{nil})) && \text{def of zip} \\
&\quad = \text{mult}((1,2) :: (2,3) :: \text{nil}) && \text{def of zip} \\
&\quad = (1 \times 2) :: \text{mult}((2,3) :: \text{nil}) && \text{def of mult} \\
&\quad = (1 \times 2) :: (2 \times 3) :: \text{mult}(\text{nil}) && \text{def of mult} \\
&\quad = (1 \times 2) :: (2 \times 3) :: \text{nil} && \text{def of mult}
\end{aligned}$$

A "straight from the spec" implementation of this expression would make four passes over the list, each running in $O(n)$ time, where $n = \text{len}(xs)$, so the overall running time is $O(n)$. However, the calls to zip and mult both create new lists, so the heap memory used is also $O(n)$ even though we are only returning one number. In this problem, you will write a more memory efficient implementation.

(a) Define a single recursive function f : List$\langle\mathbb{Z}\rangle \to \mathbb{Z}$ so that f(xs) calculates the expression sum(mult(zip(xs, tail(xs)))) + sum(xs).

Use the above example as a guide for finding the pattern needed in your recursive case, but remember that the example only goes over the mult portion on the expression, don't forget the rest of the expression!

(b) Prove that your code is correct by structural induction, i.e., that this holds for any list xs : List$\langle\mathbb{Z}\rangle$:

$$f(\text{xs}) = \text{sum}(\text{mult}(\text{zip}(\text{xs}, \text{tail}(\text{xs})))) + \text{sum}(\text{xs})$$

Hint: You will likely need to handle the case of a list of length 1 separately from the others.