CSE 331: Software Design & Engineering Fall 2024

Homework 7

Due: Friday, November 15th, 11pm

This worksheet contains only the *written* parts of HW7. The coding instructions will be released Thursday (11/14) and be due Monday (11/18). **The coding portion of this assignment is longer than in recent weeks, so you should aim to finish the written part early (Thursday night is a good goal) to have plenty of time to work on the code.**

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW7 Written**".

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

Make sure your work is legible and scanned clearly if you handwrite it, or compiled correctly if you choose to use LaTeX. Match each HW problem to the page with your work when you turn in. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

Task 1 – List and Shout [20 pts]

The following problems involve translation of loops into tail recursive functions.

1. The following TypeScript function calculates concat(L, R) using a loop.

```
const concat = <A>(L: List<A>, R: List<A>): List<A> => {
  S = rev(L);
  while (S.kind !== "nil") {
    R = cons(S.hd, R);
    S = S.tl;
  }
  return R;
};
```

Define a mathematical definition for a tail-recursive function concat-acc that has identical behavior to the loop body. Then, give a mathematical definition of a function concat that calls concat-acc in such a manner that it matches the overall behavior of the concat code.

Don't forget to add type declarations for your functions.

2. In HW5 Task 5, you defined a recursive function $f : \text{List}\langle\mathbb{Z}\rangle \to \mathbb{Z}$ such that

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

The following function g calculates the same value using a loop:

```
const g = (L: List<bigint>): bigint => {
  if (L.kind === "nil")
    return 0;

  let s = 0;
  let w = L.hd;
  L = L.tl
  // Inv: f(L_0) = s + f(w :: L)
  while (L.kind !== "nil") {
    s = w * L.hd + w + s;
    w = L.hd;
    L = L.tl
  }
  return s + w;
};
```

While your recursive version (probably) worked by looking at the *first two* elements in the list to calculate the answer, this version keeps track of the previous element, before the part in L, in a separate variable w. That allows it to calculate the answer using the head of L and the previous element, w, rather than looking at first two elements of L.

This loop only makes sense, however, if the list is non-empty (so there is a first element). The code handles the case of a nil list in the first if statement. We only reach the loop when the list is not empty.

Define a tail-recursive function g-acc that has identical behavior to the loop. Then, give a definition of g that matches the overall behavior of the function g, calling g-acc in the cases where it would enter the loop and returning the answer directly in the other case(s).

Don't forget to add type declarations for your functions.

Task 2 – Loops, I Did it Again [20 pts]

In HW6 Task 5, we looked a function "even" that takes a list of base-3 digits, List⟨Digit⟩, and determines if the value they represent is even or odd. Recall the definition of a Digit in base-3:

$$\textbf{type } \text{Digit} := 0 \mid 1 \mid 2$$

That function even was defined as follows:

$$\begin{aligned}
\text{even}(\text{nil}) &:= \text{true} \\
\text{even}(0 :: \text{ds}) &:= \text{even}(\text{ds}) \\
\text{even}(1 :: \text{ds}) &:= \text{not even}(\text{ds}) \\
\text{even}(2 :: \text{ds}) &:= \text{even}(\text{ds})
\end{aligned}$$

In this problem, we will do the same calculation using tail recursion. To do so, we first define a tail-recursive function, even-acc, of two arguments as follows:

$$\begin{aligned}
\text{even-acc}(\text{nil}, b) &:= b \\
\text{even-acc}(0 :: L, b) &:= \text{even-acc}(L, b) \\
\text{even-acc}(1 :: L, b) &:= \text{even-acc}(L, \text{not } b) \\
\text{even-acc}(2 :: L, b) &:= \text{even-acc}(L, b)
\end{aligned}$$

That would allow us to *re*-define even by invoking even-acc:

$$\text{even}(L) := \text{even-acc}(L, \text{true})$$

giving us a potentially more memory efficient implementation.

1. Translate the mathematical definitions for even-acc and the re-definition of even into one TypeScript function, even(L), that behaves identically to a tail-call optimized version of these definitions by using a loop.

   Be sure to include the invariant of the loop. Your invariant should be in the form shown in class for loops translated from tail recursion, and your code must be correct with that invariant.

2. Prove the following, which we'll refer to as "equation (1)," by induction

$$\text{even-acc}(L, b) = (\text{even}(L) = b) \tag{1}$$

   where the "=" on the right is the usual equals operator on booleans, which is true if both have the same value and false if they have different values.

   Note that $(b = \text{true})$ is equivalent to just b, and that $(\text{not } a = b)$ is equivalent to $(a = \text{not } b)$ since both mean $(a \neq b)$.

   Hint: You will likely need a proof by cases within your inductive proof.

3. Using equation (1), which we proved holds in the last part, we can now rewrite our invariant without reference to even-acc ("Destroy the evidence"). This is useful to us because we can describe the behavior of our loop without reference to our intermediate tail-optimized version of the math definitions.

   Prove by calculation that $\text{even}(L_0) = (\text{even}(L) = b)$ holds.

Task 3 – In One Fell Loop [20 pts]

We have seen the following definition of a function contains(L, y) that checks whether the value y appears in the list L:

$$\mathsf{contains}(\mathsf{List}\langle\mathbb{Z}\rangle, \mathbb{Z}) \to \mathbb{B}$$

$$\begin{aligned}
\mathsf{contains}(\mathsf{nil}, y) &:= \mathsf{false} \\
\mathsf{contains}(x :: L, y) &:= \mathsf{true} && \text{if } x = y \\
\mathsf{contains}(x :: L, y) &:= \mathsf{contains}(L, y) && \text{if } x \neq y
\end{aligned}$$

This function is already tail recursive.

1. Translate it into a TypeScript function, contains(L, y), that behaves identically to a tail-call optimized version of these definitions by using a loop.

   Be sure to include the invariant of the loop. Your invariant should be in the form shown in class for loops translated from tail recursion, and your code must be correct with that invariant.

2. It is also possible to define contains as follows

   $$\mathsf{cont}(\mathsf{List}\langle\mathbb{Z}\rangle, \mathbb{Z}) \to \mathbb{B}$$

   $$\begin{aligned}
   \mathsf{cont}(\mathsf{nil}, y) &:= \mathsf{false} \\
   \mathsf{cont}(x :: L, y) &:= (x = y) \text{ or } \mathsf{cont}(L, y)
   \end{aligned}$$

   This definition is more concise and uses pattern matching which may make it more obviously correct to someone else reading the code, but since it is not tail recursive, a direct implementation would be less memory efficient. We need to show that our tail-recursive function definition is equivalent to this alternate definition that we'd like to use to document the function instead.

   Prove that the following, which we'll refer to as "equation (2),"

   $$\mathsf{cont}(L, y) = \mathsf{contains}(L, y) \tag{2}$$

   holds for all lists L and values y. Your proof should be by structural induction on L.

   Note that, for all boolean values b, we have "false or $b = b$" and "true or $b =$ true".

3. Using equation (2), rewrite your loop invariant to use cont, instead of the original contains.

Task 4 – Extra Credit: 1, 2, 3, 4, I Declare a Num War [0 pts]

The following function converts a natural number into a list of digits with base b.

$$\begin{aligned}\text{num-to-digits}(0, b) &:= \text{nil} \\ \text{num-to-digits}(n+1, b) &:= ((n+1) \bmod b) :: \text{num-to-digits}((n+1) \operatorname{div} b)\end{aligned}$$

where "$a \operatorname{div} b$" is integer division and "$a \bmod b$" is remainder.

The "Division Equation" states that $a = (a \bmod b) + b \cdot (a \operatorname{div} b)$ holds for all natural numbers a and b. That, along with the fact that $0 \leqslant a \bmod b < b$, is essentially the *declarative* specification for the two functions, mod and div.

(a) We want to prove that the list returned by num-to-digits(n) represent the number n, i.e., that

$$\text{value}(\text{num-to-digits}(n, b), b) = n$$

where value is the function that converts a list of digits in base $b \geqslant 2$ into a number:

$$\begin{aligned}\text{value}(\text{nil}, b) &:= 0 \\ \text{value}(x :: L, b) &:= x + b \cdot \text{value}(L, b)\end{aligned}$$

The function num-to-digits is not a simple structural recursion, for which a call on $n+1$ would recurse on n. Instead, this recurses on $(n+1) \operatorname{div} b$, which could be much smaller than n.

To do this proof, we need to use "strong" structural induction, where the inductive hypothesis assumes that the claim holds for all objects built in the process of producing $n+1$. That includes all numbers from 0 up to n. For this proof, you are allowed to use that form of induction.

Suppose that we try to implement num-to-digits with the following loop.

```
/** @returns a list R such that value(R, b) = n */
const num_to_digits = (n: bigint, b: bigint): List => {
  let digits: List = nil;
  // Inv: rev(num-to-digits(n_0, b)) = rev(num-to-digits(n, b)) ++ digits
  while (n !== 0) {
    digits = cons(n % b, digits);  // n % b means n mod b
    n = n / b;                     // n / b means n div b
  }
  return rev(digits);
};
```

The strange invariant for this loop is provided.

(b) Prove that the loop invariant holds initially.

(c) Prove that the body of the loop preserves the invariant.

(d) Prove that the return value $R :=$ rev(digits) satisfies the specification "value$(R, b) = n_0$".

Note: you will need the fact proven in part 1. You will also need the fact that rev(rev$(L)) = L$ for any list L, a fact that we will call "Idempotency of rev".