Quiz Section 7: Tail Recursion

In HW6 Task 5, we worked with numbers represented as lists of base-3 digits. In HW7, we will work with the same representation of numbers, but now using an arbitrary base $b \geqslant 2$.

When we write a base-10 number like "120", the first digit "1" represents the 1×10^2, so it is the "highest order" digit. Storing the high order digits at the front of a list is a called "big endian" representation. While the big endian representation matches how we write numbers, it is less convenient for our calculations, so instead, we will store the "lowest order" digits are at the front of the list. For example, 120 would be stored as $0 :: 2 :: 1 ::$ nil rather than $1 :: 2 :: 0 ::$ nil. This is called a "little endian" representation, and it is what we will use.

We can formalize the discussion above by defining the the value of the base-b digits as

$$\begin{aligned} \mathsf{value}(\mathsf{nil}, b) &:= 0 \\ \mathsf{value}(d :: \mathsf{ds}, b) &:= d + b \cdot \mathsf{value}(\mathsf{ds}, b) \end{aligned}$$

This simple, recursive definition encodes the fact that the digits are in the little endian representation.[1] For example, we can see that

$$\begin{aligned} \mathsf{value}(0 :: 2 :: 1 :: \mathsf{nil}, 10) &= 0 + 10 \cdot \mathsf{value}(2 :: 1 :: \mathsf{nil}, 10) && \text{def of value} \\ &= 0 + 10 \cdot (2 + 10 \cdot \mathsf{value}(1 :: \mathsf{nil}, 10)) && \text{def of value} \\ &= 0 + 10 \cdot (2 + 10 \cdot (1 + 10 \cdot \mathsf{value}(\mathsf{nil}, 10))) && \text{def of value} \\ &= 0 + 10 \cdot (2 + 10 \cdot (1 + 10 \cdot 0)) && \text{def of value} \\ &= 0 + 10 \cdot (2 + 10 \cdot 1) \\ &= 0 + 10 \cdot 12 \\ &= 120 \end{aligned}$$

[1] A recursive function defining the value in the big endian representation would not be so simple!

Task 1 – A Tail Recursive Version

We would like to implement this with a loop, but the above definition is not tail recursive. So let's look at a tail recursive version of the definition which uses an accumulation parameter to calculate (as we move down the list) the factor $c = b^k$ to multiply by as we perform the sum:

$$\begin{aligned} \text{value-acc}(\text{nil}, b, c, s) &:= s \\ \text{value-acc}(d :: \text{ds}, b, c, s) &:= \text{value-acc}(\text{ds}, b, b \cdot c, s + c \cdot d) \end{aligned}$$

Write a function that calculates value-acc$(\text{digits}, b, 1, 0)$ with a **loop** using the approach taught in lecture. Your function should have the following signature:

```
const value = (digits: List<number>, base: number): number => { ... };
```

Be sure to include the invariant of the loop, using the approach taught in lecture for tail-recursive functions. Your code must be correct with that invariant.

Task 2 – Relating the Two Functions

Next, we need to describe how value and value-acc are related. Prove that the following holds:

$$\text{value-acc}(\text{ds}, b, c, s) = s + c \cdot \text{value}(\text{ds}, b) \tag{1}$$

for all ds, b, c, and s by structural induction on ds.

If true, this would tell us that $\text{value-acc}(\text{ds}, b, 1, 0) = 0 + 1 \cdot \text{value}(\text{ds}, b) = \text{value}(\text{ds}, b)$, which means that the `valueAcc` function we wrote correctly calculates value.

Task 3 – Another Invariant

Use equation (1) (i.e. value-acc(ds, b, c, s) = s + c * value(ds, b)) to rewrite your invariant so that it no longer mentions "value-acc".

(Once you've done this, you'll have erased all of the tracks of how you used tail recursion to solve this problem. When someone asks how you came up with that loop invariant, say "it just came to me".)

Task 4 – Back to Floyd Logic

Let's confirm that the final version of our code is correct without any use of value-acc.

Of course, you and I know that the loop is value-acc, but let's check that the code reviewer can see that our code is correct without any knowledge of that function.

a) Prove that the invariant holds when we first get to the top of the loop.

b) Prove that, when we exit, the function returns $\text{value}(\text{digits}_0, \text{base})$.

c) Prove that the invariant is preserved when we execute the loop body.

Task 5 – Factailrial

The following TypeScript function computes the factorial of a given number using a loop.

```
const factorial = (n: bigint): bigint => {
 let s = 1;
 while (n > 0) {
  s *= n;
  n--;
 }
 return s;
}
```

Define a mathematical definition for a tail-recursive function factorial-acc that has identical behavior to the loop body. Then, give a mathematical definition of a function factorial that calls factorial-acc in such a manner that it matches the overall behavior of the factorial code. Don't forget to add type declarations for your functions.