"Bottom Up" Recursion With Loops

James Wilcox and Kevin Zatloukal

August 2024

Consider the following function $\mathsf{twice} : \mathsf{List}\langle\mathbb{N}\rangle \to \mathsf{List}\langle\mathbb{N}\rangle$, which returns a list with each value doubled:

$$\begin{aligned} \mathsf{twice}(\mathsf{nil}) &:= \mathsf{nil} \\ \mathsf{twice}(x :: L) &:= (2x) :: \mathsf{twice}(L) \end{aligned}$$

This function is not tail recursive, so it is not obvious how to implement it with a loop.

Nonetheless, suppose that we naively tried to implement with a loop as follows:

```
const twiceLoop = (L: List<bigint>): List<bigint> => {
  let S = nil;
  while (L.kind !== "nil") {
    S = cons(2 * L.hd, S);
    L = L.tl;
  }
  return S;
};
```

Like any loop, this is equivalent to some tail recursion. In this case, it implements the following:

$$\begin{aligned} \mathsf{twice\text{-}acc}(\mathsf{nil}, S) &:= S \\ \mathsf{twice\text{-}acc}(x :: L, S) &:= \mathsf{twice\text{-}accc}(L, 2x :: S) \end{aligned}$$

as the reader can easily check. Thus, the function `twiceLoop` calculates $\mathsf{twice\text{-}acc}(L, \mathsf{nil})$.

Let's look at what that function does on an example list. This loop calculates

$$\begin{aligned} \mathsf{twice\text{-}acc}(1 :: 2 :: 3 :: \mathsf{nil}, \mathsf{nil}) &= \mathsf{twice\text{-}acc}(2 :: 3 :: \mathsf{nil}, 2 :: \mathsf{nil}) \\ &= \mathsf{twice\text{-}acc}(3 :: \mathsf{nil}, 4 :: 2 :: \mathsf{nil}) \\ &= \mathsf{twice\text{-}acc}(\mathsf{nil}, 6 :: 4 :: 2 :: \mathsf{nil}) \\ &= 6 :: 4 :: 2 :: \mathsf{nil} \end{aligned}$$

whereas we were hoping to compute

$$\begin{aligned} \mathsf{twice}(1 :: 2 :: 3 :: \mathsf{nil}) &= 2 :: \mathsf{twice}(2 :: 3 :: \mathsf{nil}) \\ &= 2 :: 4 :: \mathsf{twice}(3 :: \mathsf{nil}) \\ &= 2 :: 4 :: 6 :: \mathsf{twice}(\mathsf{nil}) \\ &= 2 :: 4 :: 6 :: \mathsf{nil} \end{aligned}$$

In this case, we can see that

$$\mathsf{twice\text{-}acc}(1 :: 2 :: 3 :: \mathsf{nil}) = \mathsf{rev}(\mathsf{twice}(1 :: 2 :: 3 :: \mathsf{nil}))$$

and indeed this is true in general, as we will see below. Thus, if we changed the last line from `return S` to `return rev(S)`, then it would be correct!

More generally, consider any function of the form

$$\begin{aligned} f(\mathsf{nil}) &:= \mathsf{nil} \\ f(x :: L) &:= g(x) :: f(L) \end{aligned}$$

where g is just simple expressions like "$2x$".

We can try to implement this with the following tail recursion:

$$\begin{aligned} \mathsf{f\text{-}acc}(\mathsf{nil}, S) &:= S \\ \mathsf{f\text{-}acc}(x :: L, S) &:= \mathsf{f\text{-}acc}(L, g(x) :: S) \end{aligned}$$

To do so, we must explain the relationship between f and $\mathsf{f\text{-}acc}$. From our example of $\mathsf{twice\text{-}acc}$, we might guess that f and $\mathsf{f\text{-}acc}$ are related as follows:

$$\mathsf{f\text{-}acc}(L, S) = \mathsf{rev}(f(L)) +\!\!+ S \tag{1}$$

We can prove this holds by induction:

Base Case: We can see that

$$\begin{aligned} \mathsf{f\text{-}acc}(\mathsf{nil}, S) &= S && \text{def of } \mathsf{f\text{-}acc} \\ &= \mathsf{nil} +\!\!+ S \\ &= \mathsf{rev}(\mathsf{nil}) +\!\!+ S && \text{def of } \mathsf{rev} \end{aligned}$$

Inductive Hyp: Suppose that $\mathsf{f\text{-}acc}(L, S) = \mathsf{rev}(f(L)) +\!\!+ S$ for some L and any S.

Inductive Step: Let x be arbitrary. Then, we can see that

$$\begin{aligned} \mathsf{f\text{-}acc}(x :: L, S) &= \mathsf{f\text{-}acc}(L, g(x) :: S) && \text{def of } \mathsf{f\text{-}acc} \\ &= \mathsf{rev}(f(L)) +\!\!+ (g(x) :: S) && \text{Inductive Hyp.} \\ &= \mathsf{rev}(f(L)) +\!\!+ [g(x)] +\!\!+ S) \\ &= \mathsf{rev}(g(x) :: f(L)) +\!\!+ S) && \text{def of } \mathsf{rev} \\ &= \mathsf{rev}(f(x :: L)) +\!\!+ S) && \text{def of } f \end{aligned}$$

Applying equation (1) to the usual tail recursion invariant $\mathsf{f\text{-}acc}(L_0, S_0) = \mathsf{f\text{-}acc}(L, S)$ gives us:

$$\begin{aligned} \mathsf{rev}(f(L_0)) &= \mathsf{rev}(f(L_0)) +\!\!+ S_0 && \text{since } S_0 = \mathsf{nil} \\ &= \mathsf{f\text{-}acc}(L_0, S_0) && \text{by (1)} \\ &= \mathsf{f\text{-}acc}(L, S) && \text{Inv} \\ &= \mathsf{rev}(f(L)) +\!\!+ S && \text{by (1)} \end{aligned}$$

which is a version of the invariant with no reference to $\mathsf{f\text{-}acc}$.

The following loop calculates $\mathsf{f\text{-}acc}(L_0, \mathsf{nil})$ by tail recursion. Its invariant has been rewritten, as described on the previous page, so that it longer mentions $\mathsf{f\text{-}acc}$ and instead talks just about f:

```
const fLoop = (L: List<T>): List<T> => {
  let S: List<T> = nil;
  // Inv: rev(f(L_0)) = rev(f(L)) ++ S
  while (L.kind !== "nil") {
    S = cons(g(L.hd), S);
    L = L.tl;
  }
  return rev(S);  // = f(L_0)
};
```

When we exit the loop, we have $L = \mathsf{nil}$, so the invariant tells us that

$$\begin{aligned}
\mathsf{rev}(f(L_0)) &= \mathsf{rev}(f(L)) \mathbin{+\!\!+} S && \text{Inv} \\
&= \mathsf{rev}(f(\mathsf{nil})) \mathbin{+\!\!+} S && \text{since } L = \mathsf{nil} \\
&= \mathsf{rev}(\mathsf{nil}) \mathbin{+\!\!+} S && \text{def of } f \\
&= \mathsf{nil} \mathbin{+\!\!+} S && \text{def of } \mathsf{rev} \\
&= S
\end{aligned}$$

So we return $\mathsf{rev}(S) = \mathsf{rev}(\mathsf{rev}(f(L_0))) = f(L_0)$.

This completes the proof of correctness, and demonstrates how all "bottom up" recursive functions can be implemented with loops by reversing the answer at the end.