Loops & Recursion

James Wilcox and Kevin Zatloukal

October 2023

In many cases, translating recursion into a more efficient loop is a complicated problem. Fortunately, this translation is straight-forward for some large classes of problems. In those cases, described below, we can provide simple "templates" for how to implement the recursive function with a loop.

"Bottom Up" with Natural Numbers

Our first template applies to functions that are structurally recursive on a natural number argument, that is, any function that looks like the following:

$$\begin{aligned} \textbf{func } & f(0) && := h() \\ & f(n+1) && := g(n+1, f(n)) \qquad \text{for any } n : \mathbb{N} \end{aligned}$$

where g and h are just simple expressions, e.g., $g(n,s) := 2n + s$ and $h() := 0$.

Such a function can be translated into the following code

```
const f = (n: number): number => {
  let i: number = 0;
  let s: number = h();
  // Inv: s = f(i)
  while (i !== n) {
    i = i + 1;
    s = g(i, s);
  }
  return s; // = f(n)
};
```

where `g(s)` is replaced with the expression in the definition of f, e.g., "`2s`".

Correctness

We can check correctness of the three triples in the function body as follows:

Initialization: We enter the loop with $i = 0$ and $s = h()$, so we can see that $s = h() = f(0) = f(i)$.

Loop Body: Reasoning backward through the loop, the invariant becomes $g(i+1, s) = f(i+1)$. We can see that this holds since

$$\begin{aligned} f(i+1) &= g(i+1, f(i)) && \text{def of } f \\ &= g(i+1, s) && \text{by Inv} \end{aligned}$$

Postcondition: We exit with $s = f(i)$ (Inv) and $i = n$ (loop condition). Together, that says $s = f(n)$, which is the postcondition of the function.

“Bottom Up” with Lists

Our next template applies to functions that are structurally recursive on a list argument, that is, any function that looks like the following:

$$\begin{aligned} &\textbf{func } f(\mathsf{nil}) && := h() \\ &f(\mathsf{cons}(x, L)) && := g(x, f(L)) \qquad \text{for any } x : \mathbb{Z} \text{ and } L : \mathsf{List} \end{aligned}$$

where g and h are just simple expressions, e.g., $g(\mathsf{cons}(x, L)) := \mathsf{cons}(2x, L)$ and $h() := \mathsf{nil}$.

Such a function can be translated into the following code

```
const f = (L: List): List => {
  let R: List = rev(L);
  let S: List = nil;
  let T: List = h();
  // Inv: L = concat(rev(R), S) and T = f(S)
  while (R !== nil) {
    T = g(R.hd, T);
    S = cons(R.hd, S);
    R = R.tl;
  }
  return T; // = f(L)
};
```

where `g(x, L)` is replaced with the expression in the definition of f, e.g., “`cons(2*x, L)`”.

Note that `S` does not influence the return value — it is included only to explain the invariant – so this function can be removed (ghosted) without affecting correctness.

Correctness

We can check correctness of the three triples in the function body as follows:

Initialization: We enter the loop with $R = \mathsf{rev}(L)$, $S = \mathsf{nil}$, and $T = h()$, so we can see that

$$\begin{aligned} \mathsf{concat}(\mathsf{rev}(R), S) &= \mathsf{concat}(\mathsf{rev}(\mathsf{rev}(L)), S) && \text{since } R = \mathsf{rev}(L) \\ &= \mathsf{concat}(\mathsf{rev}(\mathsf{rev}(L)), \mathsf{nil}) && \text{since } S = \mathsf{nil} \\ &= \mathsf{concat}(L, \mathsf{nil}) && \text{Lemma 3} \\ &= L && \text{Lemma 2} \end{aligned}$$

and also that $T = h() = f(\mathsf{nil}) = f(S)$, using $T = h()$, the def of f, and $S = \mathsf{nil}$, respectively.

Loop Body: Reasoning backward through the loop body, we need $L = \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}), \mathsf{cons}(R.\mathsf{hd}, S))$ and $g(R.\mathsf{hd}, T) = f(\mathsf{cons}(R.\mathsf{hd}, S))$ to follow from Inv and the fact that $R \neq \mathsf{nil}$, i.e., that $R = \mathsf{cons}(R.\mathsf{hd}, R.\mathsf{tl})$. We can see the second required fact as follows

$$\begin{aligned} f(\mathsf{cons}(R.\mathsf{hd}, S)) &= g(R.\mathsf{hd}, f(S)) && \text{def of } f \\ &= g(R.\mathsf{hd}, T) && \text{since } T = f(S) \end{aligned}$$

and the first required fact like this

$$\begin{aligned} L &= \mathsf{concat}(\mathsf{rev}(R), S) && \text{Inv} \\ &= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{hd}, R.\mathsf{tl})), S) && \text{since } R \neq \mathsf{nil} \\ &= \mathsf{concat}(\mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}), \mathsf{cons}(R.\mathsf{hd}, \mathsf{nil})), S) && \text{def of } \mathsf{rev} \\ &= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}), \mathsf{concat}(\mathsf{cons}(R.\mathsf{hd}, \mathsf{nil}), S)) && \text{Lemma 4} \\ &= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}), \mathsf{cons}(R.\mathsf{hd}, \mathsf{concat}(\mathsf{nil}, S))) && \text{def of } \mathsf{concat} \\ &= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}), \mathsf{cons}(R.\mathsf{hd}, S)) && \text{def of } \mathsf{concat} \end{aligned}$$

Postcondition: We exit with $L = \mathsf{concat}(\mathsf{rev}(R), S)$ and $T = f(S)$ [Inv] and $R = \mathsf{nil}$ [loop condition]. Putting together the first and last of these we have $L = \mathsf{concat}(\mathsf{rev}(\mathsf{nil}), S) = \mathsf{concat}(\mathsf{nil}, S) = S$ using the def of rev and concat, respectively. Combining that with the second fact, we have $T = f(S) = f(L)$, which is the postcondition.

Well-Known Facts About Lists

Above, we used the following facts about lists. All are easily proven by structural induction.[1]

- Lemma 2: $\mathsf{concat}(L, \mathsf{nil}) = L$ for any list L
- Lemma 3: $\mathsf{rev}(\mathsf{rev}(L)) = L$ for any list L
- Lemma 4: $\mathsf{concat}(\mathsf{concat}(L, R), S) = \mathsf{concat}(L, \mathsf{concat}(R, S))$ for any lists L, R, S.

"Top Down" with Lists

Our next template applies to functions that are properly *tail recursive*, i.e., that directly return whatever value is returned from the recursive call they make. In simple cases, such a function looks like this:

$$\begin{array}{llr} \textbf{func}\ f(\mathsf{nil}, R) & := h(R) & \text{for any } R : \mathsf{List} \\ f(\mathsf{cons}(x, L), R) & := f(L, g(x, R)) & \text{for any } x : \mathbb{Z} \text{ and } L, R : \mathsf{List} \end{array}$$

where g and h are just simple expressions, e.g., $g(x, R) := \mathsf{cons}(x, R)$ and $h(R) := R$. In general, these functions can have more arguments and can use more than one element of the first argument before recursing.

Functions like those above can be translated into the following code

```
const f = (L: List, R: List): List => {
  // Inv: f(L0, R0) = f(L, R)
  while (L !== nil) {
    R = g(L.hd, R);
    L = L.tl;
  }
  return h(R); // = f(L0, R0)
};
```

where `g(x, L)` is replaced with the expression in the definition of f, e.g., "`cons(x, L)`".

We can check correctness of the three triples in the function body as follows:

Initialization: We can see that $f(L, R) = f(L_0, R_0)$ holds since neither L nor R has yet been changed.

Loop Body: Reasoning backward through the loop body, we need $f(L_0, R_0) = f(L.\mathsf{tl}, g(L.\mathsf{hd}, R))$ to hold for the code to be correct. We can see that this follows from Inv and $L \neq \mathsf{nil}$ as follows

$$\begin{aligned} f(L_0, R_0) &= f(L, R) && \text{Inv} \\ &= f(\mathsf{cons}(L.\mathsf{hd}, L.\mathsf{tl}), R) && \text{since } L \neq \mathsf{nil} \\ &= f(L.\mathsf{tl}, g(L.\mathsf{hd}, R)) && \text{def of } f \end{aligned}$$

Postcondition: From Inv and the fact that $L = \mathsf{nil}$, we get $f(L_0, R_0) = f(L, R) = f(\mathsf{nil}, R) = h(R)$, with the last step being the definition of f.

[1]Doing so is frequently a homework problem in 311, so we will not write out the proofs here.

Variations

The templates above can often be adapted to functions that do not strictly fit the templates above. For example, consider the following function that operates only on lists of even length, returning the elements at even indexes in the list with each of their values doubled:

$$\begin{array}{lll}
\textbf{func}\ \mathsf{twice\text{-}evens}(\mathsf{nil}) & := \mathsf{nil} & \\
\quad\ \ \mathsf{twice\text{-}evens}(\mathsf{cons}(x, \mathsf{nil})) & := \mathsf{undefined} & \text{for any } x : \mathbb{Z} \\
\quad\ \ \mathsf{twice\text{-}evens}(\mathsf{cons}(x, \mathsf{cons}(y, L))) & := \mathsf{cons}(2x, f(L)) & \text{for any } x, y : \mathbb{Z} \text{ and } L : \mathsf{List}
\end{array}$$

Rather than recursing on a list one element shorter than its argument, it recurses on a list two elements shorter, so it does not strictly fit the pattern above.

Nonetheless, the same "bottom up" approach can be used. In fact, we can use exactly the same invariant. The only difference is that the body of the loop now processes the next two elements from the list rather than just one.

```
const twice_evens = (L: List): List => {
  if (len(L) % 2 == 1) throw new Error('list has odd length');
  let R: List = rev(L);
  let S: List = nil;
  let T: List = h();
  // Inv: L = concat(rev(R), S) and T = twice-evens(S)
  while (R !== nil && R.tl !== nil) {
    T = cons(2 * R.tl.hd, T);          // R is reversed, so R.tl.hd is an even index
    S = cons(R.tl.hd, cons(R.hd, S));  // careful! note the order
    R = R.tl.tl;
  }
  return T; // = f(L)
};
```

Since the invariant is unchanged, our earlier correctness arguments continue to work for the initialization and postcondition. Only the loop body needs to be reconsidered.

Reasoning backward through the body, we get the facts that need to follow from Inv and loop condition:

$$L = \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}.\mathsf{tl}), \mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{cons}(R.\mathsf{hd}, S))) \text{ and}$$
$$\mathsf{cons}(2R.\mathsf{tl}.\mathsf{hd}, T) = \mathsf{twice\text{-}evens}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{cons}(R.\mathsf{hd}, S)))$$

We can verify the second required fact as follows:

$$\begin{array}{rll}
\mathsf{twice\text{-}evens}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{cons}(R.\mathsf{hd}, S))) &= \mathsf{cons}(2\,R.\mathsf{tl}.\mathsf{hd}, \mathsf{twice\text{-}evens}(S)) & \text{def of } f \\
&= \mathsf{cons}(2\,R.\mathsf{tl}.\mathsf{hd}, T) & \text{since } T = \mathsf{twice\text{-}evens}(S)
\end{array}$$

Verification for the first fact is longer, because the loop body now processes two elements rather than one, but we are essentially just repeating the earlier argument twice:

$$\begin{array}{rll}
L &= \mathsf{concat}(\mathsf{rev}(R), S) & \text{Inv} \\
&= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{hd}, R.\mathsf{tl})), S) & \text{since } R \neq \mathsf{nil} \\
&= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{hd}, \mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, R.\mathsf{tl}.\mathsf{tl}))), S) & \text{since } R.\mathsf{tl} \neq \mathsf{nil} \\
&= \mathsf{concat}(\mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, R.\mathsf{tl}.\mathsf{tl})), \mathsf{cons}(R.\mathsf{hd}, \mathsf{nil})), S) & \text{def of } \mathsf{rev} \\
&= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, R.\mathsf{tl}.\mathsf{tl})), \mathsf{concat}(\mathsf{cons}(R.\mathsf{hd}, \mathsf{nil}), S)) & \text{Lemma 4} \\
&= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, R.\mathsf{tl}.\mathsf{tl})), \mathsf{cons}(R.\mathsf{hd}, \mathsf{concat}(\mathsf{nil}, S))) & \text{def of } \mathsf{concat} \\
&= \mathsf{concat}(\mathsf{rev}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, R.\mathsf{tl}.\mathsf{tl})), \mathsf{cons}(R.\mathsf{hd}, S)) & \text{def of } \mathsf{concat} \\
&= \mathsf{concat}(\mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}.\mathsf{tl}), \mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{nil})), \mathsf{cons}(R.\mathsf{hd}, S)) & \text{def of } \mathsf{rev} \\
&= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}.\mathsf{tl}), \mathsf{concat}(\mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{nil}), \mathsf{cons}(R.\mathsf{hd}, S))) & \text{Lemma 4} \\
&= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}.\mathsf{tl}), \mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{concat}(\mathsf{nil}, \mathsf{cons}(R.\mathsf{hd}, S)))) & \text{def of } \mathsf{concat} \\
&= \mathsf{concat}(\mathsf{rev}(R.\mathsf{tl}.\mathsf{tl}), \mathsf{cons}(R.\mathsf{tl}.\mathsf{hd}, \mathsf{cons}(R.\mathsf{hd}, S))) & \text{def of } \mathsf{concat}
\end{array}$$