CSE 331: Software Design & Implementation

Section 6

1. It's Forward Against Mine

In this problem, we will practice using forward reasoning to check the correctness of assignments. Assume that all variables are `bigints`. Do not use subscripts for this problem (they are not necessary), instead write assertions in terms of the current values of variables.

(a) Use forward reasoning to fill in the missing assertions (strongest postconditions) in the following code. Then prove that the stated postcondition holds.

$\{\{\, x \geq 4 \,\}\}$
`y = x - 2n;`
$\{\{$ _____ $\}\}$
`z = 2n * y;`
$\{\{$ _____ $\}\}$
`z = z - 2n;`
$\{\{$ _____ $\}\}$
$\{\{\, z \geq 0 \,\}\}$

(b) Use forward reasoning to fill in the missing assertions (strongest postconditions) in the following code. Then prove that the stated postcondition holds. (Reminder that with `bigint`, division is truncating division.)

$\{\{\, x < 4 \,\}\}$
`y = x + 4n;`
$\{\{$ _____ $\}\}$
`x = 2n * x;`
$\{\{$ _____ $\}\}$
`y = y + x;`
$\{\{$ _____ $\}\}$
$\{\{\, y < 16 \,\}\}$

2. Not For a Back of Trying

In this problem, we will practice using backward reasoning to check the correctness of assignments. Assume that all variables are `bigints`.

(a) Use backward reasoning to fill in the missing assertions (weakest preconditions) in the following code. Then prove that the stated precondition implies what is required for the code to be correct.

Feel free to simplify the intermediate assertions (i.e., rewrite them in an equivalent, but simpler, way). However, the assertions you write must be equivalent to still be weakest preconditions.

$\{\{\, x < w + 1 \,\}\}$
$\{\{$ _____ $\}\}$
`y = 3n * w;`
$\{\{$ _____ $\}\}$
`x = x * 3n;`
$\{\{$ _____ $\}\}$
`z = x - 9n;`
$\{\{\, z < y \,\}\}$

(b) Use backward reasoning to fill in the missing assertions (weakest preconditions) in the following code. Then prove that the stated precondition implies what is required for the code to be correct.

Feel free to simplify the intermediate assertions (i.e., rewrite them in an equivalent, but simpler, way). However, the assertions you write must be equivalent to still be weakest preconditions.

$\{\{\, x > 1 \,\}\}$
$\{\{$ _____ $\}\}$
`y = x - 4n;`
$\{\{$ _____ $\}\}$
`z = 3n * y;`
$\{\{$ _____ $\}\}$
`z = z + 6n;`
$\{\{\, z \geq y \,\}\}$

3. Nothing to Be If-ed At

In this problem, we will practice using forward reasoning to check correctness of `if` statements. Assume that all variables are `bigints`.

(a) Use forward reasoning to fill in the assertions. Then, combine the branches to assert the invariant we know at the end of the conditional and complete an argument by cases that this invariant implies $\{\{y \geq 2\}\}$.

Assume that `x` and `y` are both integers.

$\{\{\, x \geq 0 \,\}\}$
```
if (x >= 6n) {
```
$\{\{$ _____ $\}\}$
```
  y = 2n * x - 10n;
```
$\{\{$ _____ $\}\}$
```
} else {
```
$\{\{$ _____ $\}\}$
```
  y = 20n - 3n * x;
```
$\{\{$ _____ $\}\}$
```
}
```
$\{\{$ _____ or _____ $\}\}$

$\{\{\, y \geq 2 \,\}\}$

(b) Use forward reasoning to fill in the assertions. Then, combine the branches to assert the invariant we know at the end of the conditional and complete an argument by cases that this invariant implies $\{\{s \geq 1\}\}$.

Assume that `s` and `t` are both integers.

$\{\{\, s \neq t$ and $t > 0 \,\}\}$
```
if (s >= t) {
```
$\{\{$ _____ $\}\}$
```
  s = s / t;
```
$\{\{$ _____ $\}\}$
```
} else {
```
$\{\{$ _____ $\}\}$
```
  s = t - s;
```
$\{\{$ _____ $\}\}$
```
}
```
$\{\{$ _____ or _____ $\}\}$

$\{\{\, s \geq 1 \,\}\}$

4. Chicken Noodle Loop

The function sum-abs calculates the sum of the absolute values of the numbers in a list. We can give it a formal definition as follows:

$$
\begin{array}{lcll}
\textbf{func}\ \text{sum-abs}(\text{nil}) & := & 0 & \\
\quad \text{sum-abs}(\text{cons}(x, L)) & := & -x + \text{sum-abs}(L) & \text{if } x < 0 \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List} \\
\quad \text{sum-abs}(\text{cons}(x, L)) & := & x + \text{sum-abs}(L) & \text{if } x \geq 0 \quad \text{for any } x : \mathbb{Z} \text{ and } L : \text{List}
\end{array}
$$

In this problem, we will prove that the following code correctly calculates sum-abs(L). The invariant for the loop is already provided. It references L_0, which is the initial value of L when the function starts.

```
let s: bigint = 0;
{{ Inv: s + sum-abs(L) = sum-abs(L_0) }}
while (L.kind !== ''nil'') {
   if (L.hd < 0n) {
      s = s + -L.hd;
   } else {
      s = s + L.hd;
   }
   L = L.tl;
}
{{ s = sum-abs(L_0) }}
```

(a) Prove that the invariant is true when we get to the top of the loop the first time.

(b) Prove that, when we exit the loop, the postcondition holds.

(c) Prove that the invariant is preserved by the body of the loop. To do this, use backward reasoning to reason through the last assignment statement "`L = L.tl;`". Then, use forward reasoning for each branch of the "`if`" statement (as in Problem 3). Finally, complete the correctness check by verifying that each of the assertions you produced with forward reasoning implies the assertion produced by backward reasoning immediately above the last assignment statement.

We have previously used the fact that, when $L \neq \text{nil}$, we know that $L = \text{cons}(x, R)$ for some $x : \mathbb{Z}$ and $R : \text{List}$. However, in the code, we know exactly what x and R are, namely, $x = L.\text{hd}$ and $R = L.\text{tl}$. Hence, when $L \neq \text{nil}$, we actually have $L = \text{cons}(L.\text{hd}, L.\text{tl})$. Feel free to use that in your proof.

5. The Only Game in Down

The function "countdown" takes an integer argument "n" and returns a list containing the numbers $n, \ldots, 1$. It can be defined recursively as follows:

$$\begin{array}{lcll} \textbf{func}\ \text{countdown}(0) & := & \text{nil} & \\ \quad \text{countdown}(n+1) & := & \text{cons}(n+1, \text{countdown}(n)) & \text{for any } n : \mathbb{N} \end{array}$$

This function is defined recursively on a natural number so it fits the natural number template from lecture.

(a) Using the template described in lecture, give the invariant for a loop implementation of this function.

Assume that the variable counting up to n is called "`i`" and the partial-result is stored in a variable called "`L`" (rather than "s").

(b) How do we initialize `i` and `L` so that the invariant is true initially?

(c) When do we exit the loop calculating "countdown(n)"? What should the condition of the `while` be?

(d) What code do we write in the body of loop so that the invariant remains true when i is increased by one?

Be careful! It's easy to make a mistake here.

6. Take It From the Swap

Recall the function "swap", which swaps adjacent elements in a list:

$$
\begin{array}{lcll}
\textbf{func}\ \text{swap}(\text{nil}) & := & \text{nil} & \\
\quad \text{swap}(\text{cons}(a, \text{nil})) & := & \text{undefined} & \text{for any } a : \mathbb{Z} \\
\quad \text{swap}(\text{cons}(a, \text{cons}(b, L))) & := & \text{cons}(b, \text{cons}(a, \text{swap}(L))) & \text{for any } a, b : \mathbb{Z} \text{ and } L : \text{List}
\end{array}
$$

This function is defined recursively on a list argument so we can use the "bottom up" template for lists.

Note that this function requires the list to always have an even number of elements. Thus, the middle case in the above definition will not occur (it is included only to maintain exhaustive pattern matching).

(a) Using the template described in lecture, give the invariant for a loop implementation of this function.

As in the template, assume that we have a variable "`R`" that stores the part of the input list `L` that we have not yet processed in *reverse order*, a variable "`S`" that stores the part of `L` that we have processed already, and a variable "`T`" that stores our calculation of swap(S).

Your loop should differ from the basic template, however, by processing **two elements** each time through the loop! That is, after all, how the recursive version of swap works above. Note that this does not require any changes to the invariant, initialization, or exit condition. Only the loop body differs.

(b) How do we initialize the variables so that the invariant is true initially?

(c) When do we exit the loop? What should the condition of the `while` be?

(d) The template tells us to move down the list by setting `L = L.tl`. However, swap makes its recursive call on a list that is shorter by two elements, so our loop body should try to move forward by two elements.

What code do we write so that the list gets shorter by two elements and the invariant remains true?

Be careful! There are lots of ways to make mistakes here.