CSE 331: Software Design & Engineering Fall 2024

Homework 6

Due: Saturday, November 9th, 11pm

This worksheet contains only the *written* parts of HW6. The coding instructions will be released Thursday (11/07) and be due Monday (11/11).

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW6 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. Make sure you match each HW problem to the page with your work. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

Reasoning rules

Apply the following rules to all reasoning problems unless stated otherwise.

- Assume that all code is TypeScript, and number variables are `bigints`.
- **You should not use subscripts** (unless we explicitly tell you otherwise or introduce subscriptted variables in the problem to refer to a variable's value when the function was initially called). Instead, write all assertions in terms of the current value of variables.
- Arithmetic simplification is not required, but if you choose to do so, you are always permitted and encouraged to show your work for any simplification or combination of facts, but please do so clearly to the side of your final assertions.
- All assertions should use math notation.
- **Use '::' instead of 'cons'.** (This also applies within proofs.)
- If you choose to abbreviate any function names within assertions, you *must* clearly define that abbreviation at the top of the problem.
- If not explicitly stated in the instructions, you may reason forward or backward.
- There's no need to include the value of *constant* variables in assertions.

Task 1 – The House That Back Built [15 pts]

After filling in the assertions, prove the implication that remains. These do not need to be formal proofs by calculation, English sentences explaining the facts can be sufficient and more intuitive (see the section 6 worksheet solutions for an example). Though we recommend you try to write a proof by calculation to make sure you don't forget any steps.

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

$\{\{\ y > 7 \text{ and } z > 1\ \}\}$
```
x = 3n * y + 1n;
```
$\{\{$ _____ $\}\}$
```
y = y - 7n;
```
$\{\{$ _____ $\}\}$
```
z = z * y;
```
$\{\{$ _____ $\}\}$
$\{\{\ x < 3z + 24\ \}\}$

2. Use **backward reasoning** to fill in the missing assertions (weakest preconditions) in the following code. Then, prove that the stated precondition implies the result from reasoning.

$\{\{\ z > 0\ \}\}$
$\{\{$ _____ $\}\}$
```
x = -z;
```
$\{\{$ _____ $\}\}$
```
z = z - 1n;
```
$\{\{$ _____ $\}\}$
```
y = x - 1n;
```
$\{\{\ y \leqslant z\ \}\}$

3. Use forward reasoning to fill in the assertions. Then, prove, by cases, that what we know at the end of the conditional implies the post condition.

   Note that because we have 3 branches in our conditional, the fact we know just after the conditional will have 3 cases "or"ed together.

$\{\!\{ x \geqslant 1 \text{ and } y = x^2 \}\!\}$

```
if (y < 20n) {
```

$\{\!\{$ _____ $\}\!\}$

```
  y = y + 4n;
```

$\{\!\{$ _____ $\}\!\}$

```
} else if (y < 28n) {
```

$\{\!\{$ _____ $\}\!\}$

```
  y = (y / x) + 1n;
```

$\{\!\{$ _____ $\}\!\}$

```
} else {
```

$\{\!\{$ _____ $\}\!\}$

```
  y = y / x;
```

$\{\!\{$ _____ $\}\!\}$

```
}
```

$\{\!\{$ _____ or _____ or _____ $\}\!\}$

$\{\!\{ y > 4 \}\!\}$

Task 2 – Hula-Loop [10 pts]

In this problem, we will prove the correctness of a loop that finds the quotient of x divided by 5, i.e., the *largest* value y such that $5y \leqslant x$. To say that y is the largest such value means that any larger value would not work, i.e., that $5(y+1) > x$.

We denote the initial value of x at the top by x_0. This is explicitly stated in the precondition as the fact "$x = x_0$". The first two facts of the postcondition say that y is the quotient of x_0 divided by 5. The third fact says that x is the remainder, i.e., the remaining amount not divisible by 5.

This loop calculates the quotient without division. Instead, it just uses subtraction. It operates by increasing `y` and decreasing `x` each time around. The first part of the invariant says that the distance from x_0 down to $5y$ (i.e., $x_0 - 5y$) is the same as the distance from x down to 0 (i.e., $x - 0 = x$). The second part of the invariant says that x has not moved below 0 (i.e., $x \geqslant 0$).

$\{\!\{\, x = x_0 \text{ and } x_0 \geqslant 0 \,\}\!\}$

```
let y: bigint = 0n;
```

$\{\!\{\,$ Inv: $x_0 - 5y = x$ and $x \geqslant 0 \,\}\!\}$

```
while (x >= 5n) {
    y = y + 1n;
    x = x - 5n;
}
```

$\{\!\{\, 5y \leqslant x_0$ and $x_0 < 5(y+1)$ and $x = x_0 - 5y \,\}\!\}$

1. Prove that the invariant is true when we get to the top of the loop the first time.
2. Prove that, when we exit the loop, the postcondition holds.
3. Prove that the invariant is preserved by the body of the loop. Use either forward or backward reasoning (your choice) to reduce the body to an implication and then check that it holds.

Task 3 – Loop Dreams [15 pts]

In this problem, you will write a loop that finds the integer square root of x, i.e., the *smallest* integer v such that $x \leqslant v^2$. That v is the smallest such integer means that no smaller integer would work, i.e., that $(v-1)^2 < x$. These two facts are the postcondition of the loop below.

It is only possible to have a number smaller than x if $x > 0$, so that is required by the precondition.

Your loop should calculate the square root using *only addition*. It will operate by increasing v until it is the integer square root of x. In order to do this without multiplication or subtraction, we will need to keep track of two other values. The variable "w" stores $2v-1$, and the variable y stores v^2. The invariant states these two facts and the first part of the postcondition, namely, that $(v-1)^2 < x$.

The basic structure of the loop is as follows. You will fill in the missing pieces below.

$\{\{\, x > 0 \,\}\}$

```
let v: bigint = _____________;
let w: bigint = _____________;
let y: bigint = _____________;
```

$\{\{$ Inv: $(v-1)^2 < x$ and $w = 2v-1$ and $y = v^2$ $\}\}$

```
while (_____________________________________) {
    v = v + 1n;
    w = _____________________________________;
    y = _____________________________________;
}
```

$\{\{\, (v-1)^2 < x$ and $x \leqslant v^2 \,\}\}$

1. Fill in the initialization code above the loop. Then, prove that the invariant holds with your code.
2. Fill in the loop condition. Then, prove that the post condition holds when the loop exits.
3. The first line of the body of the loop increases `v` by 1. Fill in the updates to `w` and `y` so that the invariant remains true with a v that is one larger.

   Give the two lines of code. Then, use either forward or backward reasoning (your choice) to reduce to an implication and prove that it holds, showing that the invariant remains true.

Task 4 – If It Ain't Broke, Don't Mix It [20 pts]

We can define the set of primary colors as an enum-like inductive data type as follows:

$$\textbf{type}\ \text{Color} := \text{RED} \mid \text{YELLOW} \mid \text{BLUE}$$

Purple, green, and orange can be expressed as 50/50 mixtures of pairs of these colors. Specifically, purple is 50% red and 50% blue, and green is 50% red and 50% yellow.

The two functions, $\text{amt-purple}, \text{amt-green} : \text{List}\langle\text{Color}\rangle \to \mathbb{R}$, take lists of primary colors and return the amounts of purple and green, respectively, present in the list:

$$\begin{aligned}
\text{amt-purple}(\text{nil}) &:= 0 \\
\text{amt-purple}(\text{RED} :: \text{cs}) &:= 0.5 + \text{amt-purple}(\text{cs}) \\
\text{amt-purple}(\text{YELLOW} :: \text{cs}) &:= \text{amt-purple}(\text{cs}) \\
\text{amt-purple}(\text{BLUE} :: \text{cs}) &:= 0.5 + \text{amt-purple}(\text{cs})
\end{aligned}$$

$$\begin{aligned}
\text{amt-green}(\text{nil}) &:= 0 \\
\text{amt-green}(\text{RED} :: \text{cs}) &:= 0.5 + \text{amt-green}(\text{cs}) \\
\text{amt-green}(\text{YELLOW} :: \text{cs}) &:= 0.5 + \text{amt-green}(\text{cs}) \\
\text{amt-green}(\text{BLUE} :: \text{cs}) &:= \text{amt-green}(\text{cs})
\end{aligned}$$

In this problem, we will prove the correctness of a loop that finds the amount of purple and green present in a list L of primary colors. The loop operates by moving forward through the list, updating L at each point, to keep track of where we are, until the list is empty. As usual, L_0 refers to the initial value of the variable L, which is the full list.

The variables "p" and "g" keep track of the amount of purple and green, respectively, in the part of the list processed so far. The first part of the invariant says that the amount of purple in the full list is equal to p plus the amount remaining in the list L. The second part states a similar fact for green.

The postconditions states that p and g contain the full amount of purple and green, respectively, in the full list.

$\{\!\{ L = L_0 \}\!\}$
```
let p: bigint = 0n;
let g: bigint = 0n;
```
$\{\!\{$ Inv: amt-purple$(L_0) = p +$ amt-purple(L) and amt-green$(L_0) = g +$ amt-green$(L) \}\!\}$
```
while (L.kind !== "nil") {
   if (L.hd.kind === "RED") {
```
$\{\!\{ P_1 :$ Inv and $\_\_\_\_\_ \}\!\}$

$\{\!\{ Q_1 :$ $\_\_\_\_\_ \}\!\}$
```
    p = p + 0.5;
    g = g + 0.5;
   } else if (L.hd.kind === "YELLOW") {
```
$\{\!\{ P_2 :$ Inv and $\_\_\_\_\_ \}\!\}$

$\{\!\{ Q_2 :$ $\_\_\_\_\_ \}\!\}$
```
    g = g + 0.5;
   } else { // "BLUE"
```
$\{\!\{ P_3 :$ Inv and $\_\_\_\_\_ \}\!\}$

$\{\!\{ Q_3 :$ $\_\_\_\_\_ \}\!\}$
```
    p = p + 0.5;
   }
   L = L.tl;
}
```
$\{\!\{ p =$ amt-purple(L_0) and $g =$ amt-green$(L_0) \}\!\}$

1. Prove that the invariant is true when we get to the top of the loop the first time.
2. Prove that, when we exit the loop, the postcondition holds.
3. Use forward reasoning to fill in each of the P_i's above and backward reasoning to fill in each of the Q_i's above.

   Note that $L \neq$ nil means that we can write $L = L$.hd :: L.tl since "::" is the only non-nil constructor.

4. Prove that the invariant is preserved by the loop body by showing that each P_i implies each Q_i.

Task 5 – Barking Up the Wrong Three [20 pts]

Suppose we define the set of base-3 digits as

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

Then, we can represent number written in base 3 as a List$\langle$Digit$\rangle$.

The following function, non-zeros : List$\langle$Digit$\rangle \to \mathbb{N}$, counts the number of non-zero digits in a given base-3 number:

$$\begin{aligned}
\text{non-zeros}(\text{nil}) &:= 0 \\
\text{non-zeros}(0 :: \text{ds}) &:= \text{non-zeros}(\text{ds}) \\
\text{non-zeros}(1 :: \text{ds}) &:= 1 + \text{non-zeros}(\text{ds}) \\
\text{non-zeros}(2 :: \text{ds}) &:= 1 + \text{non-zeros}(\text{ds})
\end{aligned}$$

The next function, even : List$\langle$Digit$\rangle \to \mathbb{B}$, determines whether the given base-3 number is even:

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

For this problem, we can take this as the definition. In the extra credit problem, we will prove that this function correctly checks whether the number represented by a sequence of base-3 digits is indeed even.

In this problem, you will write a loop that, at the same time, calculates the number of non-zero digits, stored in a variable a, and whether the digits are even, stored in a variable b. The two facts of the postcondition state that these variables contain the values of these two functions on the full list.

Your loop should calculate these values by making a single pass through the list from front to back, exiting when you reach the end of the list.. The first fact of the invariant states that the number of non-zero digits in the whole list is a plus the number of non-zero digits remaining in L. The second fact of the invariant states that the number is even exactly when the evenness of the remaining digits matches the value of b (i.e., they are both true or both false).

The basic structure of the loop is as follows. You will fill in the missing pieces below.

$\{\{ L = L_0 \}\}$
```
let a: bigint = _____________;
let b: boolean = _____________;
```
{{ Inv: non-zeros(L_0) = a + non-zeros(L) and even(L_0) = (b = even(L)) }}
```
while (L.kind !== "nil") {
    ...
    // fill in the code here
    ...
    L = L.tl;
}
```
{{ a = non-zeros(L_0) and b = even(L_0) }}

1. Fill in the initialization code above the loop. Then, prove that the invariant holds with your code.

   Note that, if x is a boolean, then x = true is true exactly when x is, and x = false is true when not x is.

2. Prove that the post condition holds when the loop exits.

3. Fill in the missing code in the body of the loop so that the invariant is preserved when L moves forward to the next element of the list.

   Then, use either forward or backward reasoning (or both) to reduce correctness of the loop body to implication(s) and prove that they hold.

   Hint: note that, if b and c are booleans, then "not $b = c$" is the same as "b = not c". Both expressions are true exactly when the values of b and c are different (one is true and one is false).

Task 6 – Extra Credit [0 pts]

We can calculate the value represented by a given sequence of digits as follows:

$$\begin{aligned} \text{value}(\text{nil}) &:= 0 \\ \text{value}(d :: \text{ds}) &:= d + 3 \cdot \text{value}(\text{ds}) \end{aligned}$$

When we write a base-3 number like "120", the first digit "1" represents the 1×3^2, so it is the "highest order" digit. Storing the high order digits at the front of a list is a called "big endian" representation. The function above instead assumes that the "lowest order" digits are at the front of the list, so the same number would be represented as $0 :: 2 :: 1 ::$ nil. This is called a "little endian" representation. The two representations are equally valid, but little endian is easier for us in this instance.

Prove that the function even(ds) above correctly calculates whether value(ds) is even. Specifically, prove by structural induction that "value(ds) is even" = even(ds). Feel free to use standard facts about even and odd such as "even + odd = odd".