CSE 331
Software Design & Implementation

Spring 2023

Section 5 – Imperative Programming I

Administrivia

- HW5 released later today
  - Due Wednesday (5/3) @ 11:00pm

Hoare Triples – Review

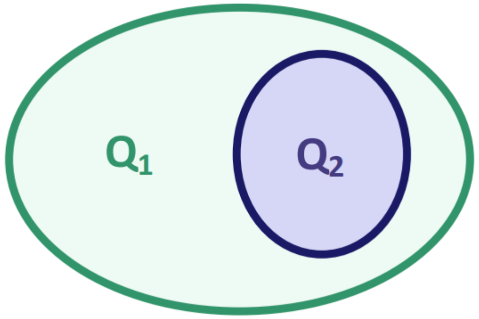
- A Hoare Triple has 2 assertions and some code

```
{{ P }}
   S
{{ Q }}
```

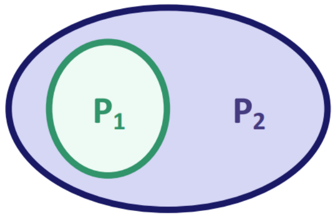
  - P is a precondition, Q is the postcondition
  - S is the code
- Triple is “valid” if the code is correct:
  - S takes any state satisfying P into a state satisfying Q
    - Does not matter what the code does if P does not hold initially

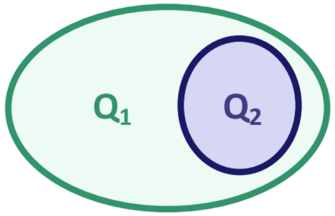
Stronger Assertion vs Spec – Review

- Assertion is stronger iff it holds in a subset of states



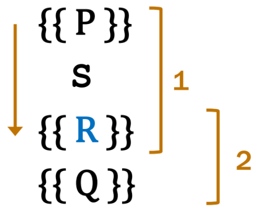
- Specification is stronger iff
  - Postcondition is stronger
    - Guarantees more specific output
  - Precondition is weaker
    - Allows more input





Forward Reasoning – Review

- Forwards reasoning fills in the postcondition
  - Gives strongest postcondition making the triple valid
- Apply forward reasoning to fill in R



  - First triple is always valid
  - Only need to check second triple (only proving implication since no code is present)

Mutations in Forward Reasoning – Review

- With variable mutations, we need to give new names to initial values
  - Will use "x" and "y" to refer to current values and "x_0" and "y_0" for earlier values

```
{{ w = x + y }}
  x = 4;
{{ w = x₀ + y and x = 4 }}
  y = 3;
{{ w = x₀ + y₀ and x = 4 and y = 3 }}
```

- If $x_0 = f(x)$, then we can simplify this to

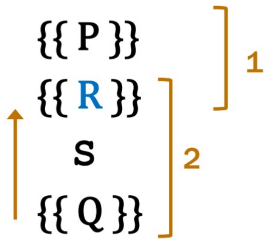
```
{{ P }}
  x = … x …;
{{ P[x ↦ f(x)] }}
```

no need for, e.g., "and $x = x_0 + 1$"

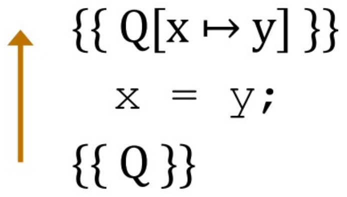
  - If assignment is "$x = x_0 + 1$", then "$x_0 = x - 1$"

Backward Reasoning – Review

- Backwards reasoning fills in preconditions
  - Gives weakest precondition making the triple valid
- Apply backwards reasoning to fill in R



  - Second triple is always valid
  - Only need to check first triple (implication only)
- Backwards reasoning is just substitution (so mechanically simpler than forward reasoning)



Conditionals – Review

- Want to use both forward reasoning and backwards reasoning to avoid "or"

```
{{}}
if (n >= 0) {
    {{ P1: n >= 0 }}
    {{ Q1: 2*n + 1 > n }}
    m = 2*n + 1;
    {{ m > n }}
} else {
    {{ P2: n < 0 }}
    {{ Q2: 0 > n }}
    m = 0
    {{ m > n }}
}
{{ m > n }}
```

Check this:

$$\begin{aligned} 2*n+1 &= n + n + 1 & \\ &\geq n + 1 & \text{since } n \geq 0 \\ &> n & \text{since } 1 > 0 \end{aligned}$$

1) Use forward reasoning to push the initial assertion from the conditional (red)
2) Use backwards reasoning from the given postcondition and push it to the top of the conditional (orange)
3) Prove the implication from the forward reasoned assertion (P1/P2) to the backwards reasoned assertion (Q1/Q2) (blue)

Loop Invariant – Review

- Loop invariant is true **every time** at the top of the loop

```
{{ Inv: I }}
while (cond) {
   S
}
```

  - Must be true when we get to the top the first time
  - Must remain true each time executes S and look back up
- Use "Inv:" to indicate loop invariant
  - Otherwise, this only claims to be true the first time at the loop

Proving Correctness – Review

```
{{ P }}
{{ Inv: I }}
while (cond) {
  {{ I and cond }}
    S
  {{ I }}
}
{{ I and not cond }}
{{ Q }}
```

1. I holds initially
2. S preserves I
3. Q holds when loop exits

Splits correctness into three parts

1.	**I holds initially**	implication
2.	**S preserves I**	forward/back then implication
3.	**Q holds when loop exits**	implication

Question 1b

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

```
{{ $x < 3$ }}
y = x + 4;
{{ ________________________________ }}
x = 2 * x;
{{ ________________________________ }}
y = y + x;
{{ ________________________________ }}
{{ $y < 14$ }}
```

Question 2a

(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 = 3 * w;
```

$\{\{$ ______________________________ $\}\}$

```
x = x * 3;
```

$\{\{$ ______________________________ $\}\}$

```
z = x - 6;
```

$\{\{\, z < y \,\}\}$

Question 3b

(b) Use forward reasoning to fill in P_1 and P_2 with the strongest postconditions at these two lines. Then, use backward reasoning to fill in Q_1 and Q_2 with the weakest preconditions at those lines. Finally, complete the correctness proof by showing that P_1 implies Q_1 and P_2 implies Q_2.

```
{{ $x \geq 0$ }}
if (x >= 6) {
   {{ $P_1$ : ______________________________ }}
   {{ $Q_1$ : ______________________________ }}
    y = 2*x - 10;
   {{ ___________________________________ }}
} else {
   {{ $P_2$ : ______________________________ }}
   {{ $Q_2$ : ______________________________ }}
    y = 20 - 3*x;
   {{ ___________________________________ }}
}
{{ $y > 1$ }}
```

Question 4

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: number = 0;
{{ Inv: $s + \text{sum-abs}(L) = \text{sum-abs}(L_0)$ }}
while (L !== nil) {
    if (L.hd < 0) {
        s = s + -L.hd;
    } else {
        s = s + L.hd;
    }
    L = L.tl;
}
{{ $s = \text{sum-abs}(L_0)$ }}
```

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

(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.

Question 4 – continued

(c) Prove that the invariant is preserved by the body of the loop. You can do this by any combination of forward and backward reasoning.

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.

Question 6

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

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

This function is defined recursively on a list argument so it fits the "top-down" template from lecture.

For simplicity, we will require the list to always have an even number of elements. Thus, the middle case in the above definition will not occur.

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

As in the template, we will have a variable "R" that stores the partially-completed answer (reversed).

(b) How do we initialize R 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! Remember that R stores the partial answer in *reverse* order!