CSE 331
Software Design & Implementation

Spring 2024

Section 6 – Imperative Programming I

Administrivia

- HW6 released later today, due Wednesday (5/8) at **11pm**
- Can resubmit as many times as you'd like until the deadline.
  - Use the autograder as a tool if you're not sure if your code/tests have bugs

Proving Correctness of ADT

To prove the correctness of an ADT Implementation, you must show the correctness of the constructor and each method:

1) **Constructor**
    - must create a concrete state that satisfies the RI
    - must create the abstract state required by the spec
2) **Methods**
    - check value returned is the one stated by the spec
    - may need to use both RI and AF

Proving Correctness of ADT (Example)

```
interface NumberQueue {
    // @returns len(obj)
    size: () => bigint;
}
class ListQueue implements NumberQueue {
    // AF: obj = this.items
    readonly items: List;
    size = (): bigint => { return len(this.items); };
}
```

- In order to prove the correctness of size(), we need to use the AF which gives:

  len(this.items) = len(obj)   **By AF**

Hoare Triples – Review

- A Hoare Triple has 2 assertions and some code

  {{ P }}

  S

  {{ Q }}

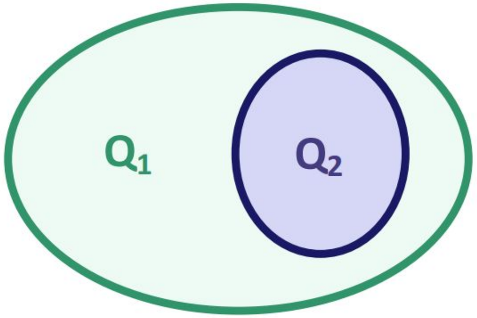
  – P is a precondition, Q is the postcondition

  – 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 vs Weaker – Review

- Assertion is stronger iff it holds in a subset of states
  - **Stronger** assertion implies the **weaker** one:
    If Q_2 is true, Q_1 must also be true, $Q_2 \rightarrow Q_1$

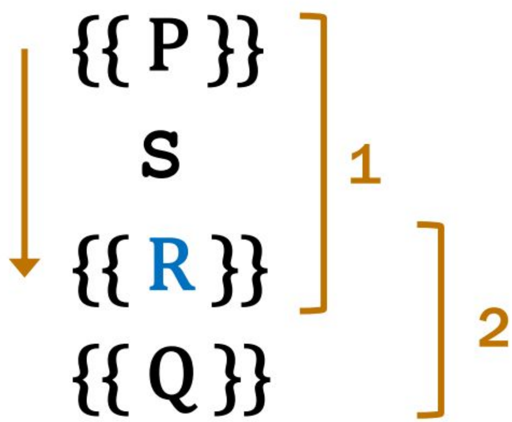


- Different from strength in *specifications*:
  - A stronger spec:
    - Stronger postcondition: guarantees more specific output
    - Weaker precondition: handles more allowable inputs

  compared to a weaker one

Forward Reasoning – Review

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



  - Check second triple by proving that R implies Q

Backward Reasoning – Review

- Backwards reasoning fills in preconditions
  - **Just use substitution!**
  - Gives weakest precondition making the triple valid
- Apply backwards reasoning to fill in R

{{ P }}

1

{{ R }}

S

2

{{ Q }}

  - Check first triple by proving that P implies R

- **Good example problems in section worksheet!**

Conditionals – Review

- Reason through "then" and "else" branches independently and combine last assertion of both branches with an "or" at the end
- Prove that each implies post condition by cases

```
const g = (n: number): number => {
  {{ }}
  let m;
  if (n >= 0) {
    m = 2*n + 1;
  } else {
    m = 0;
  }
  {{ m > n }}
  return m;
}
```

```
{{ }}
let m;
if (n >= 0) {
  m = 2*n + 1;
} else {
  m = 0;
}
{{ m > n }}
return m;
```

Loop Invariant – Review

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

true!

true!

true!

true!

- Loop invariant must be true ***every time*** at the top of the loop
  - The first time (before any iterations) and for the beginning of each iteration
- Also true every time at the bottom of the loop
  - Meaning it's true immediately after the loop exits
- During the body of the loop (during **S**), it isn't true

- Must use "`Inv`" notation to indicate that it's not a standard assertion

Well-Known Facts About Lists

- Feel free to cite these in your proofs! They're easily proven by structural induction (and you don't have to do that again)

- Lemma 2: **concat(L, nil) = L for any list L**

- Lemma 3: **rev(rev(L)) = L for any list L**

- Lemma 4: **concat(concat(L, R), S)**
  = concat(L, concat(R, S)) for any lists L, R, S

Question 4

Prove that the following code correctly calculates $\mathrm{sum} - \mathrm{abs}(L)$

(a) Invariant is true at top of loop the first time

(c) Invariant is preserved by loop body

(b) Postcondition holds when we exit

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

$\{\{\ \text{Inv: } s + \text{sum-abs}(L) = \text{sum-abs}(L_0)\ \}\}$

$\{\{\ s = \text{sum-abs}(L_0)\ \}\}$

Question 1a

$\{\{\ x \geq 4\ \}\}$

```
y = x - 2;
```

$\{\{$ ______________________ $\}\}$

```
z = 2 * y;
```

$\{\{$ ______________________ $\}\}$

```
z = z - 2;
```

$\{\{$ ______________________ $\}\}$

$\{\{\ z \geq 0\ \}\}$

Question 1b

$\{\{\, x < 4 \,\}\}$

```
y = x + 4;
```

$\{\{\, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \,\}\}$

```
x = 2 * x;
```

$\{\{\, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \,\}\}$

```
y = y + x;
```

$\{\{\, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \,\}\}$

$\{\{\, y < 16 \,\}\}$

Question 2a

$\{\{\, x < w + 1 \,\}\}$

$\{\{$ ____ $\}\}$

```
y = 3 * w;
```

$\{\{$ ____ $\}\}$

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

$\{\{$ ____ $\}\}$

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

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

Question 2b

$\{\{\, x > 1 \,\}\}$

$\{\{$ ________________________________ $\}\}$

```
y = x - 4;
```

$\{\{$ ________________________________ $\}\}$

```
z = 3 * y;
```

$\{\{$ ________________________________ $\}\}$

```
z = z + 6;
```

$\{\{\, z \geq y \,\}\}$

Question 3b – “then” branch

$\{\{\ s \neq t \text{ and } t > 0\ \}\}$

```
if (s >= t) {
```

$\{\{$ ________________________________ $\}\}$

```
    s = s / t;
```

$\{\{$ ________________________________ $\}\}$

```
} else {
    s = t - s;
}
```

$\{\{$ ________________________________ $\}\}$

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

Question 3b – “else” branch

$\{\{\ s \neq t$ and $t > 0\ \}\}$

```
if (s >= t) {
    s = s / t;
} else {
```

$\{\{$ ____ $\}\}$

```
    s = t - s;
```

$\{\{$ ____ $\}\}$

```
}
```

$\{\{$ ____ $\}\}$

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

Question 5a

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:

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

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?

Question 5b

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:

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

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

(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.

Question 6

(a) Give the invariant for the loop, based on the "bottom-up" template for lists

L = concat(rev(R), S) and T = swap(S)

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

We can set R = rev(L), S = nil, and T = nil.

In that case, we have T = nil = swap(nil) = swap(S) which shows that the second half of Inv holds.

To prove the first half, take this calculation

concat(rev(R), S)	= concat(rev(rev(L)), S)	since R = rev(L)
	= concat(L, S)	Lemma 3: rev(rev(L) = L
	= concat(L, nil)	Since S = nil
	= L	Lemma 2

Question 6

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

We exit when $R = \text{nil}$. So we will loop `while (R !=== nil)`

(d) Generally, the template says we move down the list with L = L.tl. swap processes 2 elements of the list at at time, so our loop should do the same. Write the loop body that does this and maintains the invariant:

```
T = cons(R.hd, cons(R.tl.hd, T));
S = cons(R.tl.hd, cons(R.hd, S));
R = R.tl.tl;
```

Question 6

```
T = cons(R.hd, cons(R.tl.hd, T));
S = cons(R.tl.hd, cons(R.hd, S));
R = R.tl.tl;
```

We still need need to prove this code maintains the invariant:

First part (L = concat(rev(R))) is the same as the notes.

For T = swap(S), we can backward reason through the loop body from our result list, R to get:

```
cons(R.hd, cons(R.tl.hd, T))
      = swap(cons(R.tl.hd, cons(R.hd, S)))
```

We need to prove that the Inv implies this statement which can be done with this calculation:

swap(cons(R.tl.hd, cons(R.hd, S)))
= cons(R.hd, cons(R.tl.hd, swap(S))) Def of swap
= cons(R.hd, cons(R.tl.hd, T)) Since T = swap(S)