EECS 498 - 003 Lab 6

Keshav Singh

Agenda

- Midterm Review Session:
  - Great job on midterm! Any feedback?
  - Discuss general idea behind all questions
  - You are **strongly** encouraged to interrupt and ask questions at any point (provided it isn't "why is my solution wrong")
  - Regrades open from today to next Friday 11:59 pm
- If time permits, begin discussing refinement and example exercise that we will complete next week

Question 1

Respond True/False to each of the following statements

1. A precondition violation in a Dafny program is automatically detected during the execution of the program.

   ANSWER:

2. Let `S` be the set of all safe states and `D` be the set of all "dangerous" states (i.e. the safe states that have a direct transition to an unsafe state). Then the set `S-D` is not necessarily an **inductive invariant**.

   ANSWER:

3. Let `S` be the set of all safe states and `D` be the set of all "dangerous" states (i.e. the safe states that have a direct transition to an unsafe state). Then the set `S-D` is an **invariant**.

   ANSWER:

4. Given a state machine and a transition predicate `MyAction(v:Variables, v':Variables)`, for every reachable state `v`, there exists a single state `v'` that satisfies `MyAction(v, v')`.

   ANSWER:

5. The following lemma verifies successfully.

```
lemma foo(s: seq<int>, i: int)
    requires |s| > 0
    requires 0 <= i < |s|
    ensures s[..i]+s[i..]==s
{

}
```

ANSWER:

Question 2

2 Index of Max (15 minutes, 16 points)

Write a function (**NOT** a method) that takes in a non-empty sequence of integers and returns the index of the maximum integer on the sequence. You must write the entire function, including the signature, pre- and post-conditions, and body. Your code should not require any assertions to prove the post-conditions.

Question 2: Key Takeaways

- Functions in dafny refer to declarative functions, not imperative methods
- You may not use :| unless you have a witness
- Ensure you are exact about your pre and post-conditions to not permit any incorrect output/disallow correct output (remember, your spec is critical for verification!)
- Generally, you may check if your post-conditions will verify if you can perform a proof by induction

Question 3

3 The Jugs Riddle (30 minutes, 22 points)

Some of you may be familiar with this problem either from the "Die Hard 3" movie or from a popular riddle with the same essence. In the movie there are two jugs, one of size 3 liters and one of size 5 liters. The good guys are asked to somehow place a jug with exactly 4 liters on a scale. If the weight is off even by a little, a bomb will go off. Since they cannot rely on "eyeballing it", they must use the two capacities together to produce exactly 4 liters of water. They soon figure it out (with just 5 seconds left on the bomb timer, of course!). They fill the small jug and use it to pour 3 liters into the large one. They then fill it again and pour two more liters into the large one, until it is full. This leaves 1 liter in the small one. They can then empty the large one, pour the 1 liter in the large one, and then pour another 3 liters in for an exact 4 liters!

In this exercise, you are asked to write down the state machine for such a problem. You have two jugs, `A` and `B`, each with its own capacity, `capA` and `capB` (unlike the movie, you don't know specific values for these capacities). Both jugs initially start off empty. The only actions allowed by the state machine are:

- Fill one of the jugs entirely
- Empty one of the jugs entirely
- Pour the contents of one jug into the other, possibly leaving some water behind if the "destination" jug has been filled.

Your predicates should include the appropriate checks to ensure that trivial transitions are not possible (e.g. emptying an already empty jug or filling an already full one). For full marks your state machine should use update expressions where applicable.

You have 2.5 blank pages. You don't need to fill all of that. Some people have large and messy hand-writing though, so we are giving you enough space to write comfortably.

Question 3 Takeaways

- Key components of transition State Machine: Constants, Variables, Init, Next
- Separating your transitions into enabling conditions and update step (using update expression) leads to cleaner and easier to reason about code
- When implementing transitions, be extra-careful to check that your formal description agrees with your informal ideas/intuition: try examples and corner cases to perform sanity checks

Question 4

4 Loop invariants (25 minutes, 22 points)

You are asked to write the specification, implementation and loop invariants for the `countFrequencies` method below. The method takes in a sequence of integers and returns a map `f`, which is a mapping from each integer in the sequence to the number of times that integer appears in the sequence. There is no need to write additional proof annotations (e.g. assertions) other than the loop invariants; just assume Dafny is able to infer all such assertions.

In your specification, you may find the following syntax useful. Given a multiset `m`, the notation `m[i]` represents the number of times the element `i` appears in `m`. Note, however, that, since many languages do not have a multiset in their standard library, **you cannot use multiset in the non-ghost part of your implementation**.

```
method countFrequencies(a:seq<int>) returns (f:map<int,nat>)
    // There are no pre-conditions. Add post-conditions below




{
    f := map[];
    var i:nat := 0;
    while(i < |a|)
        // Add loop invariants here




    {  // Add loop body here
```

Question 4: Key Takeaways

- Multisets are very useful for verification but sometimes neglected
- As mentioned for Q2, make sure your specification is exact
- In a lot of cases, a good idea for an invariant is just your post-conditions parametrized by “i” and something telling you what i is at the end (all of you seemed to understand this very well!), but this is not always the case
- Can check if invariant is correct/figure out what’s missing by performing an “inductive” proof on the loop body
- Loops and recursive functions have parallels, so you can reason about one in the language of the other if you prefer

Question 5

```
datatype Direction = Up | Down | Left | Right
datatype Status = Normal | Disoriented | Recovered
datatype Variables = Variables(x:int, y:int, direction:Direction, status:Statu

ghost predicate Init(v: Variables) {
    && v.x == 0
    && v.y == 5
    && v.direction == Up
    && v.status == Normal
}

ghost predicate StraightenAndMoveUp(v: Variables, v': Variables) {
    && v.status != Disoriented
    && v' == v.(y := v.y + 1, x:= 0, direction := Up, status := Normal)
}

ghost predicate MoveDown(v: Variables, v': Variables) {
    && v.direction == Down
    && v.status == Disoriented
    && v' == v.(y := v.y - 1, status := Recovered)
}

ghost predicate MoveLeft(v: Variables, v': Variables) {
    && v.direction == Left
    && v.status == Disoriented
    && v' == v.(x := v.x - 1, status := Recovered)
}

ghost predicate MoveRight(v: Variables, v': Variables) {
    && v.direction == Right
    && v.status == Disoriented
    && v' == v.(x := v.x + 1, status := Recovered)
}

ghost predicate Warp(v: Variables, v': Variables) {
    && v.x*v.x > 1
    && v' == v.(x := 0, y := 0)
}

ghost predicate Disorient(v: Variables, v': Variables) {
    && v.status == Normal
    && v' == v.(direction := v'.direction, status := Disoriented)
}
```

Inductive invariant

Just like the original crawler, we want it avoid the dark hole of radius 3 in the middle of the Cartesian space. The safety property is given below. Your task is to fill in the inductive invariant that allows this proof to succeed (this is the standard inductive proof we've seen in class).

Note: You **cannot change anything else** about the protocol, safety property and proof. You can only fill in the `Inv()` predicate.

```
// The Safety property effectively means:
// "the crawler stays outside a circle of radius 3, centered at (0,0)"
predicate Safety(v: Variables) {
    && v.x*v.x + v.y*v.y > 3*3
}
```

```
predicate Inv(v: Variables) {
    // Fill in your inductive invariant here
```

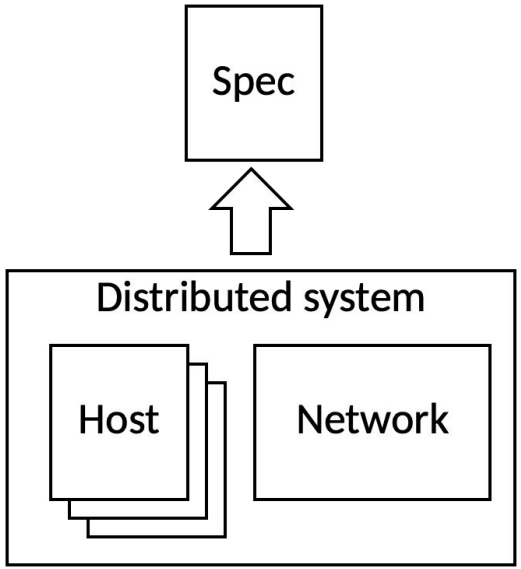
Question 5 Takeaways

- When coming up with an invariant, you may find the following approaches helpful (this is just a heuristic, and you can apply both simultaneously) :
  - Build up the reachable states (recall for a “correct” protocol the set of reachable states is an inductive invariant implying safety)
  - Trim out the dangerous states (i.e those that can eventually transition to unsafe states)
- Tracing out a few initial examples will give you a better intuition for what the invariant might me
- Check your invariant by breaking it down to each step; if not inductive carefully consider what transition breaks your invariant and adjust. **This iterative approach is key for assignments/projects in this class.**

Questions?

Refinement

- Goal : Showing that a complex/distributed system behaves exactly like a simpler state machine
- We typically show that a distributed system behaves like a sequential program (this is not *always* the goal)
- Refinement can be also multilevel



Event Based Refinement

- We can reason about the correctness of our system only through the world-visible events
- Our spec can be defined in a way that doesn’t require implementation details of the protocol

Parallel Reduction

- Protocol:
  1. N servers each receive the a portion of a large array
  2. Each computes their own portion of the sum independently
  3. After Computing sum, send to a designated leader (server 0 in our case)
  4. Server 0 sums the value and replies with it