Recitation 4

Brandon Rozek
rozekb@rpi.edu

Rensselaer Polytechnic Institute, Troy, NY, USA

February 2022

Outline

Two things:

- Reasoning through Loops
- Dafny Program Verifier

Loops

- We briefly touched loops last recitation.
- We'll look at it in more detail today.

Technique: Unrolling the Loop

Easiest technique if the number of iterations is known, small, and the loop itself isn't complex.

```
int x = 5;
for (int i = 0; i < 3; i++) {
    x += 2;
}
// x = 11
```

```
int x = 5;

x += 2;
// i = 0 -> x = 7

x += 2;
// i = 1 -> x = 9

x += 2;
// i = 2 -> x = 11
```

Loop Invariants

- Though loops are often more complex and the number of iterations isn't necessarily known.
- To help us reason through complex loops we find a property that holds at the beginning, after each iteration, and at the end of the loop otherwise known as a *loop invariant*.

Properties

What questions are we interested in answering about loops? We often care about two properties:

- Partial Correctness (Soundness & Completeness)
- Termination

Together they form *total correctness*

Soundness

- Soundness states that our system does not produce a wrong result.
- In the context of Hoare logic and loops, that means that the postcondition holds on loop exit.
- $!Loop\_Condition \ \&\& \ Loop\_Invariant \rightarrow Post\_Condition$

Completeness

- Completeness states that every valid input has an output in the system (not necessarily in finite time...).
- In the context of Hoare logic and loops, that means for all possible inputs in our precondition, the loop can derive a result.

Partial Correctness

Combining soundness and completeness in the context of Hoare logic and loops give us partial correctness:

> *If the loop terminates, then the postcondition holds on loop exit for all possible inputs constrained by the precondition.*

Reasoning Goals for Correctness

Establish and prove loop invariant using computation induction.

- Show that the loop invariant holds for the base case
  - This is before the loop begins.
  - Often this is $i = 0$.
- Show that if the loop invariant holds for this iteration, then it will hold for the next iteration.
  - Assume $L(i_{n-1}, c_{n-1})$ holds.
  - Show $L(i_n, c_n)$ holds.

Example:

```
int mul(int a, int b) {
    int x = 0;
    int p = 0;
    while (p < b) {
        x = x + a;
        p = p + 1;
    }
    return x;
}
```

Loop Invariant:

- $x == a * p$

Base case ($p == 0$):

- $x == a * 0 == 0$✓

Example - Inductive Step

Inductive Step: Assume $x_{n-1} == a * p_{n-1}$. We know from the code:

$$x_n = x_{n-1} + a$$
$$p_n = p_{n-1} + 1$$

Substitute into our inductive hypothesis:

$$x_{n-1} == a * p_{n-1}$$
$$x_n - a == a * (p_n - 1)$$
$$x_n - a == a * p_n - a$$
$$x_n == a * p_n \checkmark$$

Termination

We often want to show that our loop actually finishes and does not run forever. The easiest way to do this is to establish a decrementing function D such that:

- It reaches the minimum value at the loop exit condition.
- D decreases at each loop iteration

Example:

What is the decrementing function for the example below?

```
int mul(int a, int b) {
    int x = 0;
    int p = 0;
    while (p < b) {
        x = x + a;
        p = p + 1;
    }
    return x;
}
```

Practice 1:

What is the loop invariant and decrementing function for the below?

```
//precondition: n >= 0
int sumn(int n) {
    int i = 0, t = 0;
    while (i < n) {
        i = i + 1;
        t = t + i;
    }
    return t;
}
// postcondition: t == n * (n + 1) / 2
```

Practice 2:

What is the loop invariant, and decrementing function, and postcondition for the following?

```
// {x > 0}
int y = 1;
int z = 1;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

What is Dafny?

Dafny is a programming language that works to verify the functional correctness of programs.

Example 1: Absolute Value

Let's write a dafny method that takes an integer x and returns its absolute value.

Example 2: Size of Array

Lets write a dafny method that takes an array *a* and produces the size of *a*.

Example 3: Array up to N

Lets write a dafny method that takes an integer $n > 0$ and creates an array with integers up to n.
For example: $n = 5 \rightarrow [0, 1, 2, 3, 4]$

Any Questions?