Formal Verification

Announcements

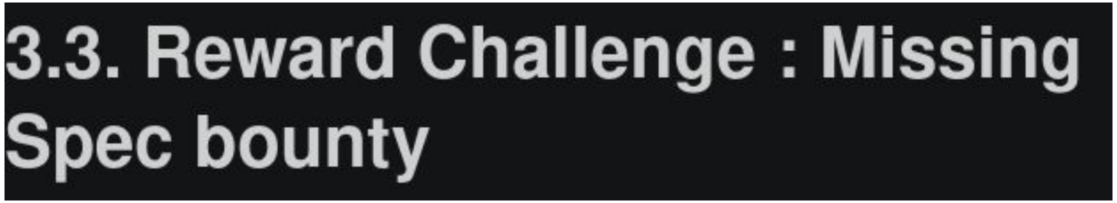
- No class 4/21 and 4/30

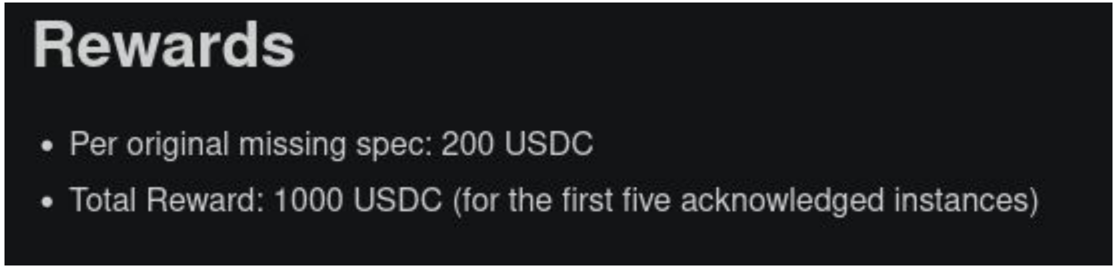
- Lab7 - due 3/30
- **HW3 released** - due 4/7
- Project part1 deadline - 3/31

Approaches to Formal Verification

1. Both models and the programs are encoded as a proof
    a. Deductive reasoning
2. Tools take as input a program in a particular language with an **annotation language.** Automatically produces a SMT formula which is fed to a verifier

What was your experience writing specs?


3.3. Reward Challenge : Missing Spec bounty


Rewards

- Per original missing spec: 200 USDC
- Total Reward: 1000 USDC (for the first five acknowledged instances)

https://docs.certora.com/projects/tutorials/en/latest/lesson3_violations/reward_challenge.html

Formal Verification Success Stories

- Paris Metro line 14
  - Driverless trains run on formally verified control software

- CompCert C Compiler
  - Used in aerospace and critical embedded systems
  - GCC is not formally verified!

- Microsoft's hypervisor
  - **Hypervisors** let you run multiple virtual machines (VMs) on one physical computer. They isolate VMs from each other and from the host system — which is crucial for cloud computing (AWS)
  - Verified using Dafny

Deductive Reasoning

Slides adapted from Isil Dillig

Hoare Logic

- Forms the basis of all deductive verification techniques
- Set of logical rules for reasoning rigorously about the correctness of programs
- Proposed in 1969 by Tony Hoare
    - 1980 turing award winner
    - Inventor of quicksort
    - “Father of formal verification”

Simple Imperative Programming Language

- To illustrate Hoare logic, we'll consider a small imperative programming language IMP
- In IMP, we distinguish three program constructs: expressions, conditionals, and statements
- Expression $E := Z \mid V \mid e_1 + e_2 \mid e_1 \times e_2$
- Conditional $C := \texttt{true} \mid \texttt{false} \mid e_1 = e_2 \mid e_1 \leq e_2$

$$
\begin{array}{lll}
\text{Statement } S := & V := E & \text{(Assignment)} \\
 & S_1; S_2 & \text{(Composition)} \\
 & \texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2 & \text{(If)} \\
 & \texttt{while } C \texttt{ do } S & \text{(While)}
\end{array}
$$

Partial Correctness Specification

- In Hoare logic, we specify partial correctness of programs using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

- Here, S is a statement in programming language IMP
- P and Q are SMT formulas
- P is called precondition and Q is called post-condition

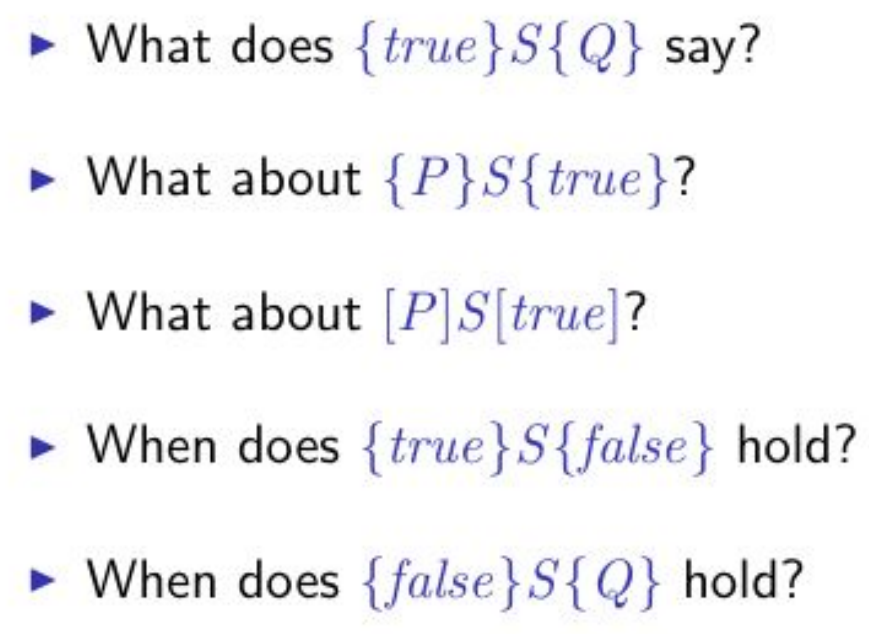
Hoare Triples

- Meaning of Hoare triple $\{P\}S\{Q\}$:
  - If S is executed in state satisfying P
  - and if execution of S terminates
  - then the program state after S terminates satisfies Q
- Is $\{x = 0\}\ \mathtt{x := x + 1}\ \{x = 1\}$ valid Hoare triple?
- What about $\{x = 0 \land y = 1\}\ \mathtt{x := x + 1}\ \{x = 1 \land y = 2\}$?
- What about $\{x = 0\}\ \mathtt{x := x + 1}\ \{x = 1 \lor y = 2\}$?
- What about $\{x = 0\}\ \mathtt{while\ true\ do\ x := 0}\{x = 1\}$?

Partial vs Total Correctness

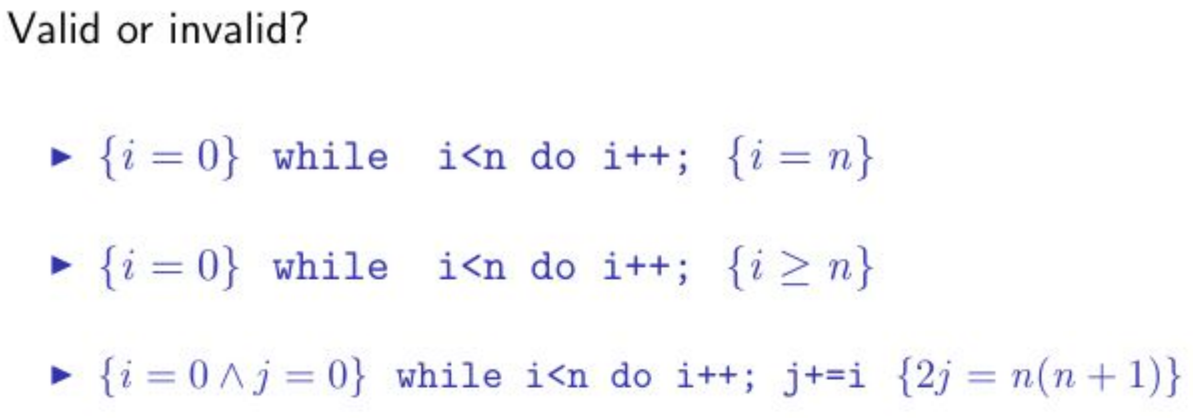
- The specification $\{P\}S\{Q\}$ called partial correctness spec. b/c doesn't require S to terminate
- There is also a stronger requirement called total correctness
- Total correctness specification written $[P]S[Q]$
- Meaning of $[P]S[Q]$:
  - If S is executed in state satisfying P
  - then the execution of S terminates
  - and program state after S terminates satisfies Q
- Is $[x = 0]$ `while true do x` $:= 0[x = 1]$ valid?

Example Specifications



- What does $\{true\}S\{Q\}$ say?
- What about $\{P\}S\{true\}$?
- What about $[P]S[true]$?
- When does $\{true\}S\{false\}$ hold?
- When does $\{false\}S\{Q\}$ hold?

More Example Specifications

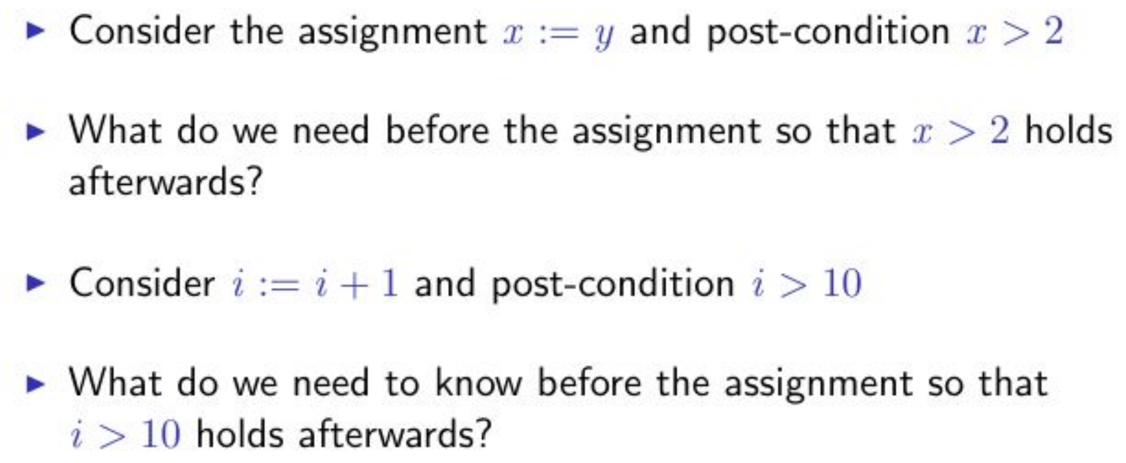


Valid or invalid?

- $\{i = 0\}$ `while i<n do i++;` $\{i = n\}$
- $\{i = 0\}$ `while i<n do i++;` $\{i \geq n\}$
- $\{i = 0 \wedge j = 0\}$ `while i<n do i++; j+=i` $\{2j = n(n+1)\}$

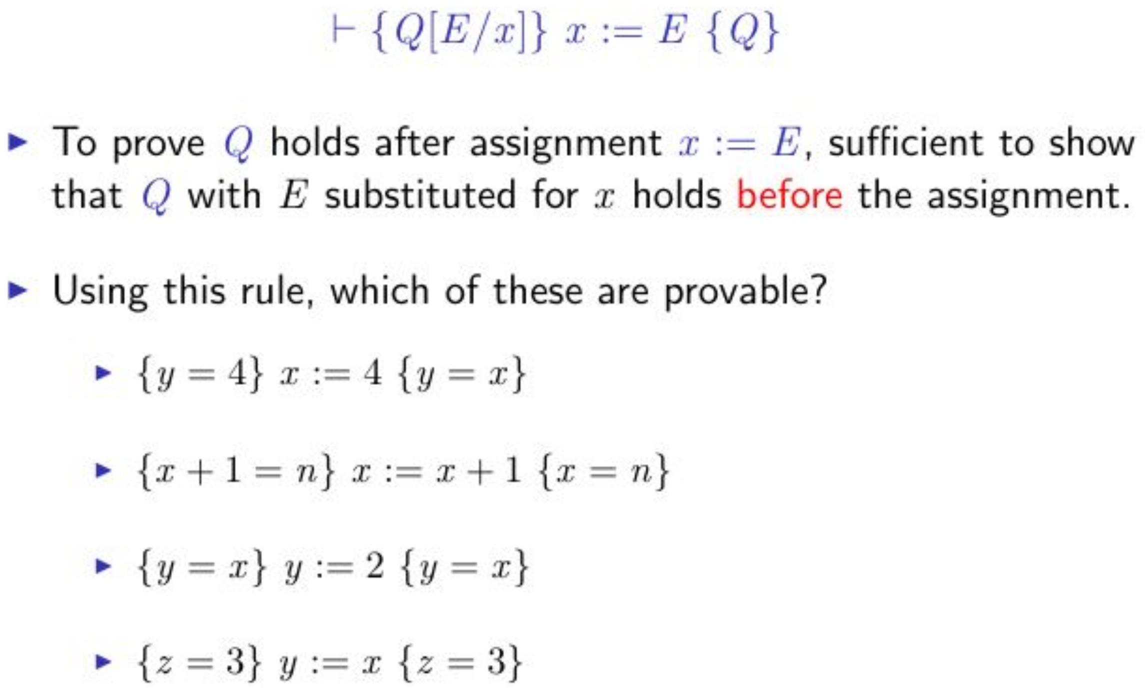
Proving Partial Correctness

- Inference rules for each statement in the IMP language



- Consider the assignment $x := y$ and post-condition $x > 2$
- What do we need before the assignment so that $x > 2$ holds afterwards?
- Consider $i := i + 1$ and post-condition $i > 10$
- What do we need to know before the assignment so that $i > 10$ holds afterwards?

Proof Rule for Assignment



$$\vdash \{Q[E/x]\}\ x := E\ \{Q\}$$

- To prove Q holds after assignment $x := E$, sufficient to show that Q with E substituted for x holds before the assignment.
- Using this rule, which of these are provable?
  - $\{y = 4\}\ x := 4\ \{y = x\}$
  - $\{x + 1 = n\}\ x := x + 1\ \{x = n\}$
  - $\{y = x\}\ y := 2\ \{y = x\}$
  - $\{z = 3\}\ y := x\ \{z = 3\}$

Proof Rule for Composition

$$\frac{\vdash \{P\}S_1\{Q\} \quad \vdash \{Q\}S_2\{R\}}{\vdash \{P\}S_1; S_2\{R\}}$$

Horizontal line -> “This is the conclusion we can draw”

If S1 results in Q and S2 results in R then S1;S2 results in R.

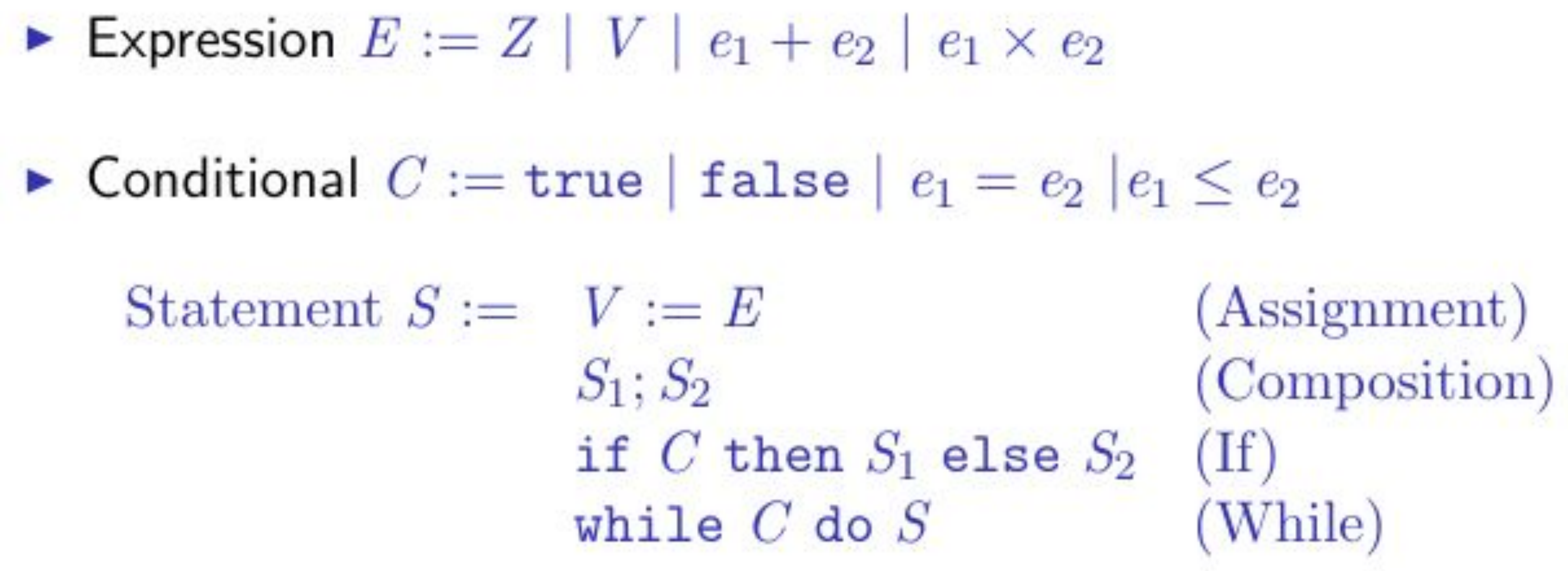
Proof Rule for Composition

$$\frac{\vdash \{P\}S_1\{Q\} \quad \vdash \{Q\}S_2\{R\}}{\vdash \{P\}S_1; S_2\{R\}}$$

Horizontal line -> “This is the conclusion we can draw”

If S1 results in Q and S2 results in R then S1;S2 results in R.

Reminder of our IMP language



- Expression $E := Z \mid V \mid e_1 + e_2 \mid e_1 \times e_2$
- Conditional $C := \texttt{true} \mid \texttt{false} \mid e_1 = e_2 \mid e_1 \leq e_2$

$$
\begin{array}{lll}
\text{Statement } S := & V := E & \text{(Assignment)} \\
 & S_1; S_2 & \text{(Composition)} \\
 & \texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2 & \text{(If)} \\
 & \texttt{while } C \texttt{ do } S & \text{(While)}
\end{array}
$$

Proof Rule for Composition

$$\frac{\vdash \{P\}S_1\{Q\} \quad \vdash \{Q\}S_2\{R\}}{\vdash \{P\}S_1; S_2\{R\}}$$

Horizontal line -> “This is the conclusion we can draw”

If S1 results in Q and S2 results in R then S1;S2 results in R.

Proof Rule for If-Statement

$$\frac{\begin{array}{lccc} \vdash & \{P \wedge C\} & S_1 & \{Q\} \\ \vdash & \{P \wedge \neg C\} & S_2 & \{Q\} \end{array}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}$$

- Suppose P holds before the if-statement and we want to show Q holds afterwards
- Consider both branches

Example

Prove the correctness of this Hoare triple:

$$\{true\}\ \texttt{if x} > 0\ \texttt{then y} := \texttt{x else y} := -\texttt{x}\ \{y \geq 0\}$$

Proof Rule for While Loop

Proofs for loops require *loop invariants*

A loop invariant has the following properties:

1. Holds initially before the loop
2. Holds after each iteration of the loop

Examples

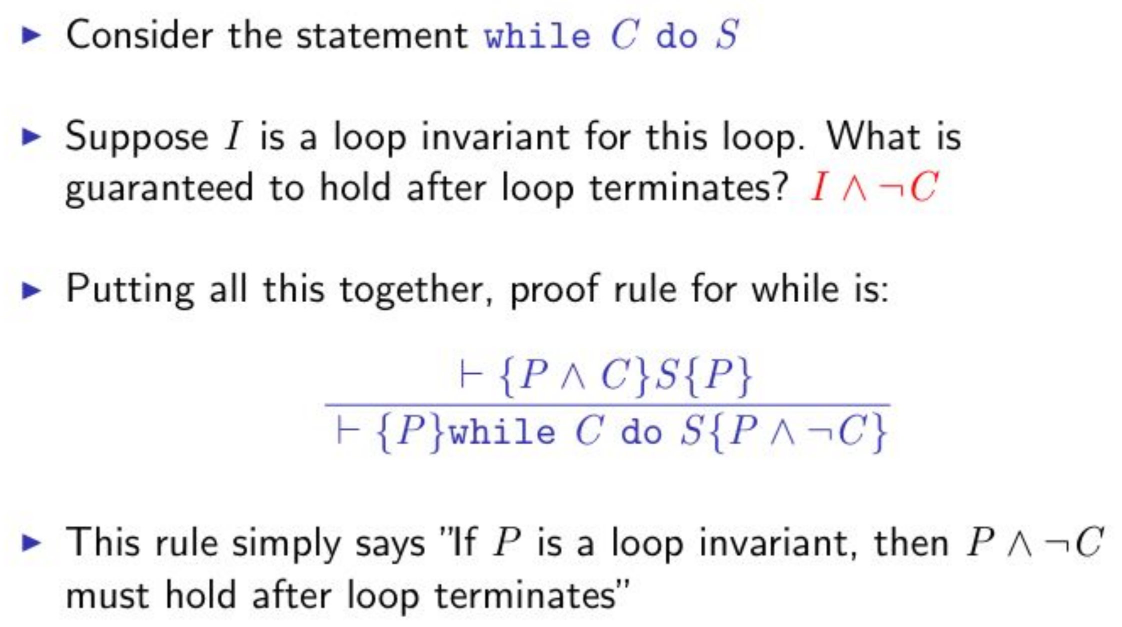
Consider the following code

```
i := 0; j := 0; n := 10;

while i < n do

      i := i + 1;

      j := i + j;
```

Which of the following are loop invariants?

- i <= n
- i < n
- j >= 0

Proof Rule for While Loop



- Consider the statement `while` C `do` S

- Suppose I is a loop invariant for this loop. What is guaranteed to hold after loop terminates? $I \land \neg C$

- Putting all this together, proof rule for while is:

$$\frac{\vdash \{P \land C\}S\{P\}}{\vdash \{P\}\texttt{while } C \texttt{ do } S\{P \land \neg C\}}$$

- This rule simply says "If P is a loop invariant, then $P \land \neg C$ must hold after loop terminates"

Proof Rules for IMP language

$$\vdash \{Q[E/x]\}\ x = E\ \{Q\} \qquad \text{(Assignment)}$$

$$\frac{\vdash \{P\}C_1\{Q\} \quad \vdash \{Q\}C_2\{R\}}{\vdash \{P\}C_1; C_2\{R\}} \qquad \text{(Composition)}$$

$$\frac{\begin{array}{lccc} \vdash & \{P \wedge C\} & S_1 & \{Q\} \\ \vdash & \{P \wedge \neg C\} & S_2 & \{Q\} \end{array}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}} \qquad \text{(If)}$$

$$\frac{\vdash \{P \wedge C\}S\{P\}}{\vdash \{P\}\texttt{while}\ C\ \texttt{do}\ S\{P \wedge \neg C\}} \qquad \text{(While)}$$

Proof Assistant Activity

Dafny

- Install dafny-lang vs code extension

- Dafny is a *verification-aware programming language*
    - Developed at MSFT

- As you type in your program, Dafny's verifier constantly looks over your shoulder, flags any errors, shows you counterexamples, and congratulates you when your code matches your specifications. When you're done, Dafny can **compile your code to C#, Go, Python, Java, or JavaScript**

Summary

- Formal verification
    - More guarantees than testing
    - Does not execute the program

- FV either requires
    - Annotations
    - program as part of the proof