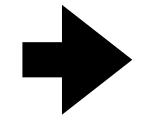
### Hoare Logic & Weakest Preconditions

Klaus v. Gleissenthall



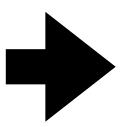
#### Logic:

- Propositional
- First order
- Theories



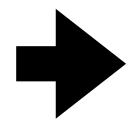
# Programs to Logic:

- Hoare Logic
- VCGen



#### **Automation**

- Horn clauses
- PredicateAbstraction



#### **Security**

- InformationFlow Control
- •Side-Channels

# Precondition Strengthening

• Is the following Hoare triple valid?

$${z = 2} y := x {y = x}$$

• Can we prove it with the assignment rule?

• Intuitively, we should be able to prove it <u>without</u> any assumptions; we should also be able to prove it <u>if we do</u> have assumptions!

# Precondition Strengthening

• We write  $P \Rightarrow P'$  to mean that formula  $P \rightarrow P'$  is valid, i.e.,  $\models P \rightarrow P'$ 

$$\vdash \{P'\} \ S \ \{Q\} \qquad P \Rightarrow P'$$

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

- To check  $P \Rightarrow P'$ , we need to call the SMT solver
- Let's now prove  $\{z = 2\}$   $y := x \{y = x\}$

# Postcondition Weakening

• We also need a dual rule for post-conditions called post-condition weakening:

$$\vdash \{P\} \text{ s } \{Q'\} \quad Q' \Rightarrow \quad Q$$

$$\vdash \{P\} \text{ s } \{Q\}$$

- If we prove some post-condition Q', we can always relax it to something weaker
- Again, we need to use an SMT solver when applying post-condition weakening

# Postcondition Weakening



- Suppose we can prove  $\{\top\}$  s  $\{x = y \land z = 2\}$
- Using post-condition weakening, which of these can we prove?
  - $\bullet \ \{\top\} \ s \ \{x = y\}$
  - $\{T\}$  S  $\{z=2\}$
  - $\{T\}$  s  $\{z>0\}$
  - $\{T\}$  s  $\{x>1\}$
  - $\{\top\}$  s  $\{\exists y. x=y\}$

# Composition

$$\vdash \{P\} s_1 \{Q\} \vdash \{Q\} s_2 \{R\}$$
 $\vdash \{P\} s_1; s_2 \{R\}$ 

• Using this proof rule, let's prove validity of the Hoare triple:

$$\{\top\} x := 2; y := x \{y = 2 \land x = 2\}$$

#### If Statements

$$\vdash \{P \land b\} S_1 \{Q\} \qquad \{P \land \neg b\} S_2 \{Q\}$$

 $\vdash \{P\}$  if b then  $s_1$  else  $s_2$   $\{Q\}$ 

- Suppose P holds before the if-statement
- When executing the then-branch, what do we know?
- What about the else-branch?

### If Statements

Example: Prove the correctness of this Hoare triple

 $\{\top\}$  if x > 0 then y := x else  $y := -x \{y \ge 0\}$ 

## While Loops

- The last rule we're missing is that for while-loops
- This is the trickiest part, and we first need to understand the concept of loop invariants
- A loop invariant *I* has the following properties:
  - I holds before the loop
  - It still holds *after* each loop iteration

# While Loops: Example



Consider the following code

```
i := 0; j := 0; n := 10; while i < n do i := i + 1; j := i + j
```

- Which of the following is a loop invariant?
  - i ≤ n
  - i < n
  - j ≥ n
- Suppose *I* is a loop invariant. Does it also hold after the loop terminates?

## While Loops

- Consider the following code while *b* do *S*
- Say *I* is an invariant for this loop. What holds after the loop terminates?

$$\vdash \{I \land b\} \text{ s } \{I\}$$

$$\vdash \{I\} \text{ while } b \text{ do } s \{I \land \neg b\}$$

- This rule says, if I is a loop invariant, then  $I \land \neg b$  must hold after the loop terminates
- How does this rule ensure that *I* is a loop invariant?

## While Loops

#### Example:

- Let's consider the statement W: while x < n do x := x + 1
- Let's prove validity of  $-\{x \le n\}$  W  $\{x \ge n\}$
- What's a good loop invariant? Let's use  $I \triangleq x \leq n$

$$\vdash \{I \land b\}$$
 s  $\{I\}$ 

 $\vdash$ {*I*} while *b* do *s* {*I*  $\land \neg b$ }

### Quiz:

- Would ⊤ also have worked as loop invariant?
- What if we changed the post-condition to x = n?

### Recap: While Loops

• We saw the proof rule for while loops:

$$\vdash \{I \land b\} \text{ s } \{I\}$$

$$\vdash \{I\} \text{ while } b \text{ do } s \{I \land \neg b\}$$

- The rule requires us to provide loop invariant *I* with the following properties:
  - I holds before the loop
  - It still holds *after* each loop iteration

#### Quiz:

• Consider the following program:

$$\vdash \{I\} \text{ while } b \text{ do } s \{I \land \neg b\}$$

 $\vdash \{I \land b\}$  s  $\{I\}$ 

$$i := 1; j := 1;$$
 while  $i < n do \{j := j + i; i := i + 1\}$ 

- Let us pick  $I = j \ge 1$ . Is I a loop invariant?
- Can we prove the Hoare triple  $\{j \ge 1 \land i < n\}$  j := j + i; i := i + 1  $\{j \ge l\}$ ?
- What about the strengthened invariant  $I \triangleq j \geq 1 \land i \geq 1$ ?

• Not all invariants can be used in the proof rule for while!

 $\vdash \{I \land b\}$  s  $\{I\}$ 

 $\vdash \{I\}$  while  $b \operatorname{do} s \{I \land \neg b\}$ 

- Invariants like  $I \triangleq j \geq 1 \land i \geq 1$  are called inductive invariants
- Only inductive invariants can be used to prove program correctness
- The Hoare proof rule requires us to supply an inductive invariant to prove correctness
- A key challenge in verification is finding those inductive invariants

#### Example:

• Consider the following statement W:

while 
$$x < n do x := y; y := x+1$$

 $\vdash \{I \land b\}$  s  $\{I\}$ 

 $\vdash$ {*I*} while *b* do *s* {*I*  $\land \neg b$ }

We want to prove the following Hoare triple:

$$\{x=0 \land y=1\} \ \mathbf{W} \ \{x \ge 0\}$$

• What is an inductive invariant *I* that allows us to prove the triple?

#### Example:

• Let's try  $x \ge 0$ 

 $\vdash \{I \land b\} \text{ s } \{I\}$   $\vdash \{I\} \text{ while } b \text{ do } s \{I \land \neg b\}$ 

- We have to prove  $\vdash \{x \ge 0 \land x \le n\}$  x:=y; y:=x+1  $\{x \ge 0\}$
- Can we prove this triple? No, *I* is <u>not inductive</u>. What information is missing?
- Let us instead try invariant  $I \triangleq x \geq 0 \land y = x + 1$
- We have to prove  $\vdash \{x \ge 0 \land y = x + 1\}$  x:=y; y:=x+1  $\{x \ge 0 \land y = x + 1\}$
- This invariant is inductive and concludes the proof

• What if we add arrays to Nano?

$$a[e_1] := e_2$$

- What proof rule should we add for this statement?
- Let's try treating arrays like assignments

$$\vdash \{Q[e_2/a[e_1]]\} \ a[e_1] := e_2 \{Q\}$$

• Is this rule correct?

• No! Let's look at a counterexample!

```
\{i = 1\} a[i] := 3; a[1] := 2 \{a[i] = 3\}
```

```
\vdash \{Q[e_2/a[e_1]]\} \ a[e_1] := e_2 \{Q\}
```

- What's the value of a[i] after the program?
- But we're able to prove post-condition a[i] = 3 using to proof rule
- Clearly this rule is unsound!

• Here is the correct proof rule:

$$+\{Q[a\langle e_1 \triangleleft e_2 \rangle/a]\} \ a[e_1] := e_2 \{Q\}$$

• Substitute a by the array where position e<sub>1</sub> is set to e<sub>2</sub>

Reasoning about this requires the theory of arrays

#### Example:

• Let's look again at our example

$$\{i = 1\}$$
 a[i] := 3; a[1] := 2  $\{a[i] = 3\}$ 

 $+\{Q[a\langle e_1 \triangleleft e_2 \rangle/a]\} \ a[e_1] := e_2 \{Q\}$ 

 $+\{Q[a\langle e_1 \triangleleft e_2 \rangle/a]\} \ a[e_1] := e_2 \{Q\}$ 

#### Example:

• Let's consider the following loop

$$+\{Q[a\langle e_1 \triangleleft e_2 \rangle/a]\} \ a[e_1] := e_2 \{Q\}$$

while 
$$i < n do \{a[i] := 0; i := i + 1\}$$

- Suppose our precondition is  $i = o \land n > o$  and postcondition is  $\forall j. o \le j < n \rightarrow a[j] = o$
- What's an inductive invariant that shows correctness?

 $+\{Q[a\langle e_1 \triangleleft e_2 \rangle/a]\} \ a[e_1] := e_2 \{Q\}$ 

# Soundness & Completeness

- One can show that the proof rules for Hoare logic are sound:
  - If  $\vdash \{P\}$  s  $\{Q\}$  then  $\models \{P\}$  s  $\{Q\}$
- That is, if we can prove triple  $\{P\}$  s  $\{Q\}$ , then it is indeed valid
- Soundness ensures that we cannot prove invalid Hoare triples
- But, are there valid Hoare triples we cannot prove?
- Completeness: If  $\models \{P\}$  s  $\{Q\}$  then  $\vdash \{P\}$  s  $\{Q\}$
- Completeness only holds with an important caveat

## Relative Completeness

- Completeness: If  $\models \{P\}$  s  $\{Q\}$  then  $\vdash \{P\}$  s  $\{Q\}$
- Completeness only holds with an important <u>caveat</u>
- Rules for precondition strengthening and postcondition weakening require checking  $A \Rightarrow B$
- Thus, we need to be able to prove such assertions, but for undecidable logics (Peano arithmetic, arrays), this might not always be possible
- Thus, Hoare's proof rules guarantee relative completeness
- Given an oracle for deciding  $A \Rightarrow B$ , any valid Hoare triple can be proven using our rules
- In particular, if we stick to decidable logics, our rules are complete

## Relative Completeness

- Completeness: If  $\models \{P\}$  s  $\{Q\}$  then  $\vdash \{P\}$  s  $\{Q\}$
- You can find a full proof of relative completeness here: <a href="https://web.eecs.umich.edu/~weimerw/2006-615/">https://web.eecs.umich.edu/~weimerw/2006-615/</a> reading/Necula-Axiomatic-Complete.pdf
- For now, we move on to the question of how to <u>automate</u> proofs using Hoare logic

# Automating Proofs in Hoare Logic

- Writing out proofs for programs is tedious, so we want to automate it
- Idea: we're given loop invariants, but automate the remaining reasoning steps
- For now, we assume that loop invariants are given by the user (of the verification tool)
- We will show how to automate finding them later

# Automating Proofs in Hoare Logic

- Automating Hoare logic is based on generating verification conditions (VC)
- A verification condition is a formula  $\varphi$  such that program is correct iff  $\varphi$  is valid

- Deductive verification has two components:
  - Generate VC's from source code
  - Use SMT solver to check validity of formulas

#### Weakest Preconditions

- Idea: Suppose we want to verify Hoare triple  $\{P\}$  s  $\{Q\}$
- We'll start with Q and, going backwards, compute a formula wp(s, Q) called weakest precondition of Q w.r.t. to s
- wp(s, Q) has the property that it is the <u>weakest condition</u> that guarantees Q will hold after s in any execution
- Thus, the triple  $\{P\}$  s  $\{Q\}$  is valid, iff  $P \Rightarrow \mathbf{wp}(s, Q)$
- This check can be automated by the SMT solver

#### Weakest Preconditions

• What's the weakest precondition for  $\{?\}$   $x := e \{Q\}$ 

$$\mathbf{wp}(x := e, Q) \triangleq Q[e/x]$$

• For composition  $s_1$ ;  $s_2$  we can compute the weakest precondition as

$$\mathbf{wp}(\mathbf{S_1}; \mathbf{S_2}, Q) \triangleq \mathbf{wp}(\mathbf{S_1}, \mathbf{wp}(\mathbf{S_2}, Q))$$

• For an if-statement  $\{?\}$  if b then  $s_1$  else  $s_2$   $\{Q\}$  we get

wp(if b then 
$$s_1$$
 else  $s_2$ ,  $Q$ )  $\triangleq (b \rightarrow wp(s_1, Q))$   $\land (\neg b \rightarrow wp(s_2, Q))$ 

#### Weakest Preconditions



• Consider the statement s:

```
x := y + 1; if x > 0 then z := 1 else z := -1
```

- What is wp(s, z>0)?
- What is  $wp(s, z \le 0)$ ?
- Can we prove  $\{-1 \le y\}$  s  $\{z > 0\}$ ?
- What about  $\{y > -1\}$  s  $\{z > 0\}$ ?

## Weakest Preconditions: Loops

- What's the weakest precondition for a loop:  $W \triangleq \text{while } b \text{ do } s$
- From our semantics, we know that we can unwind the loop as follows

if *b* then s else skip; while *b* do *s* 

• Then, we can derive

• 
$$\operatorname{wp}(W, Q) = (b \to \operatorname{wp}(W, Q)) \land (\neg b \to Q)$$

- But that's a recursive equation, so we're not really any further
- Idea: our Hoare logic proofs used invariants. Let's compute wp wrt. a given invariant

### Weakest Preconditions: Loops

- What's the weakest precondition for a loop?
- Say all our loops are annotated with a loop invariant *I*

 $W \triangleq \text{while } [I] \ b \ \text{do } s$ 



• Is it sound to let  $wp(W, Q) \triangleq I$ ?

## Weakest Preconditions: Loops

- What's the weakest precondition for a loop?
- Say all our loops are annotated with a loop invariant *I*

 $W \triangleq \text{while } [I] \ b \ \text{do } s$ 



- Is it sound to let  $wp(W, Q) \triangleq I$ ?
- No! We need to check that *I* implies post-condition *Q*
- We need to check that *I* is actually a loop invariant!
- Define function vc(s) that encodes these additional conditions

#### Verification Conditions

- How should we define vc(while [I] b do s, Q)?
- We need to ensure that *Q* holds *after* the loop, that is  $(I \land \neg b) \Rightarrow Q$
- I needs to be a loop invariant, that is, needs to be preserved under s, i.e.,

$${I \land b} s {I}$$

- We can prove this by showing  $I \wedge b \Rightarrow wp(s, I) \wedge vc(s, I)$
- This means, we can define

vc(while 
$$I b \text{ do } s, Q) \triangleq (I \land b \Rightarrow \text{wp}(s, I)) \land \text{vc}(s, I) \land (I \land \neg b) \Rightarrow Q$$

#### Verification Conditions



```
vc(while [I] b \operatorname{do} s, Q) \triangleq (I \land b \Rightarrow \operatorname{wp}(s, I)) \land \operatorname{vc}(s, I) \land (I \land \neg b) \Rightarrow Q
```

- Let W \( \text{\Lefth} \) while  $[x \le 6] \ x \le 5 \ \text{do} \ x : = x + 1; \text{ we want to prove } \{x \le 0\} \ \text{W} \ \{x = 6\}$
- What do we get for vc(W, x=6)?

- To show validity of a Hoare triple  $\{P\}$  s  $\{Q\}$ , we thus need to
  - Compute wp(s, Q)
  - Compute vc(s, Q)
- Then  $\{P\}$  s  $\{Q\}$  is valid, if the following formula is valid

$$\operatorname{vc}(s, Q) \wedge (P \to \operatorname{wp}(s, Q)) (*)$$

• Thus, if we prove (\*), we have shown that the program conforms to its specification



• Is our method complete, that is if  $\{P\}$  s  $\{Q\}$ , then the following formula is valid

$$\operatorname{vc}(s, Q) \wedge (P \to \operatorname{wp}(s, Q)) (*)$$



```
i := 1; sum := 0;

while i \le n do [sum \geq 0]

{ j := 1;

while j \le i do [sum \geq 0 \lambda j \geq 0]

sum := sum + j;

j := j + 1

i := i + 1

}
```

- Show the VC's generated for this program for post-condition sum ≥ o can it be verified?
- What is the post-condition we need to show for inner loop? sum  $\geq$  o?

vc(while [I]  $b \operatorname{do} s, Q$ )  $\triangleq (I \land b \Rightarrow \operatorname{wp}(s, I)) \land \operatorname{vc}(s, I) \land (I \land \neg b) \Rightarrow Q$ 

• For the assignment, you will implement a function

```
vcgen:: Nano.Stmt -> Logic.Base -> (Logic.Base, [Logic.Base])
```

wp List of extra VCs for invariants

- Takes as inputs Nano programs annotated with invariants
- Check that VCs hold

### Extensions & Plan

- Nano is missing many features of real programming languages
- In the next lecture, we will look at two extensions:
  - Functions
  - Pointers
- After that, we'll look at techniques to discover loop invariants (semi-) automatically