Introduction to Axiomatic Semantics

“I think you should be more explicit here in step two.”
How’s The Homework Going?

- Remember that you can’t just define a meaning function in terms of itself - you must use some fixed point machinery.
Observations

• A key part of doing research is noticing when something is incongruous or when something changes - or otherwise spotting patterns.
Observations

- A key part of doing research is noticing when something is incongruous or when something changes - or otherwise spotting patterns.

- suffix \[\text{=== state}\]
- \[r1 \; r2 \; \text{=== c1 ; c2}\]
- \[r1^* \; \text{=== while ? do r1}\]
- \[r1 \; | \; r2 \; \text{=== if ? then r1 else r2}\]
Aujourd’hui, nous ferons ...

• Assertions
• Validity
• Derivation Rules
• Soundness
• Completeness
Assertions for IMP

• \{A\} c \{B\} is a **partial correctness assertion**.
  - Does not imply termination (= it is valid if c diverges)

• \[A\] c \[B\] is a **total correctness assertion** meaning that

  If A holds in state \(\sigma\)
  Then there exists \(\sigma'\) such that \(<c, \sigma> \Downarrow \sigma'\)
  and B holds in state \(\sigma'\)

• Now let us be more formal (you know you want it!)
  - Formalize the language of assertions, A and B
  - Say when an assertion holds in a state
  - Give rules for deriving Hoare triples
The Assertion Language

• We use first-order predicate logic on top of IMP expressions

\[ A ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \geq e_2 \]
\[ \mid A_1 \land A_2 \mid A_1 \lor A_2 \mid A_1 \Rightarrow A_2 \mid \forall x. A \mid \exists x. A \]

• Note that we are somewhat sloppy in mixing logical variables and the program variables

• All IMP variables implicitly range over integers
• All IMP boolean expressions are also assertions
Assertion Judgment ⊨

- We need to assign meanings to our assertions
- New judgment $\sigma \models A$ to say that an assertion holds in a given state (= “$A$ is true in $\sigma$”)
  - This is well-defined when $\sigma$ is defined on all variables occurring in $A$
- The $\models$ judgment is defined **inductively on the structure of assertions** (surprise!)
- It relies on the denotational semantics of arithmetic expressions from IMP
Semantics of Assertions

Formal definition

\( \sigma \models \text{true} \) always

\( \sigma \models e_1 = e_2 \) iff \([e_1] \sigma = [e_2] \sigma\)

\( \sigma \models e_1 \geq e_2 \) iff \([e_1] \sigma \geq [e_2] \sigma\)

\( \sigma \models A_1 \land A_2 \) iff \( \sigma \models A_1 \) and \( \sigma \models A_2 \)

\( \sigma \models A_1 \lor A_2 \) iff \( \sigma \models A_1 \) or \( \sigma \models A_2 \)

\( \sigma \models A_1 \Rightarrow A_2 \) iff \( \sigma \models A_1 \) implies \( \sigma \models A_2 \)

\( \sigma \models \forall x. A \) iff \( \forall n \in \mathbb{Z}. \sigma[x:=n] \models A \)

\( \sigma \models \exists x. A \) iff \( \exists n \in \mathbb{Z}. \sigma[x:=n] \models A \)
Hoare Triple Semantics

- Now we can define formally the meaning of a partial correctness assertion $\vdash \{ A \} c \{ B \}$

  $\forall \sigma \in \Sigma. \forall \sigma' \in \Sigma. (\sigma \vdash A \land <c,\sigma> \downarrow \sigma') \Rightarrow \sigma' \vdash B$

- ... and a total correctness assertion $\vdash [A] c [B]$

  $\forall \sigma \in \Sigma. \sigma \vdash A \Rightarrow \exists \sigma' \in \Sigma. <c,\sigma> \downarrow \sigma' \land \sigma' \vdash B$

- or even better yet: (explain this to me!)

  $\forall \sigma \in \Sigma. \forall \sigma' \in \Sigma. (\sigma \vdash A \land <c,\sigma> \downarrow \sigma') \Rightarrow \sigma' \vdash B$

  $\land$

  $\forall \sigma \in \Sigma. \sigma \vdash A \Rightarrow \exists \sigma' \in \Sigma. <c,\sigma> \downarrow \sigma'$
Deriving Assertions

- Have a formal mechanism to decide $\models \{ A \} \subseteq \{ B \}$
  - But it is not satisfactory
  - Because $\models \{ A \} \subseteq \{ B \}$ is defined in terms of the operational semantics, we practically have to run the program to verify an assertion
  - It is impossible to effectively verify the truth of a $\forall x. A$ assertion (check every integer?)

- Plan: define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas
Derivation Rules

- We write $\vdash A$ when $A$ can be derived from basic axioms ($\vdash A$ === “we can prove $A$”)
- The derivation rules for $\vdash A$ are the usual ones from first-order logic with arithmetic:

\[
\vdash A \quad \vdash B \\
\vdash A \land B
\]

\[
\vdash [a/x]A \quad (a \text{ is fresh}) \\
\vdash \forall x. A
\]

\[
\vdash A \\
\vdots \\
\vdash B \\
\vdash A \Rightarrow B
\]

\[
\vdash \forall x. A \\
\vdash [e/x]A
\]

\[
\vdash \exists x. A \\
\vdash B
\]

\[
\vdash A \Rightarrow B \\
\vdash A \\
\vdash B
\]

\[
\vdash [a/x]A \\
\vdash [a/x]A
\]

\[
\vdash [e/x]A \\
\vdash [e/x]A
\]

\[
\vdash B \\
\vdash B
\]
Derivation Rules for Hoare Triples

- Similarly we write $\vdash \{A\} \ c \ \{B\}$ when we can derive the triple using derivation rules.
- There is one derivation rule for each command in the language.
- Plus, the *evil* rule of consequence:

\[
\begin{align*}
\vdash A' & \Rightarrow A \\
\vdash \{A\} & \ c \ \{B\} \\
\vdash B & \Rightarrow B'
\end{align*}
\]

\[\vdash \{A'\} \ c \ \{B'\}\]
Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

\[\vdash \{A\} \text{skip} \{A\}\]
\[\vdash \{[e/x]A\} x := e \{A\}\]
\[\vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\}\]
\[\vdash \{A\} c_1; c_2 \{C\}\]
\[\vdash \{A \land b\} c_1 \{B\} \quad \vdash \{A \land \neg b\} c_2 \{B\}\]
\[\vdash \{A\} \text{if } b \text{ then } c_1 \text{ else } c_2 \{B\}\]
\[\vdash \{A \land b\} c \{A\}\]
\[\vdash \{A\} \text{while } b \text{ do } c \{A \land \neg b\}\]
Alternate Hoare Rules

- For some constructs multiple rules are possible:
- (Exercise: these rules can be derived from the previous ones using the consequence rules)

\[
\vdash \{A\} \ x := \ e \ \{\exists x_0. [x_0/x]A \land x = [x_0/x]e\}
\]

(This one is called the “forward” axiom for assignment)

\[
\vdash A \land b \Rightarrow C \quad \vdash \{C\} \ c \ \{A\} \quad \vdash A \land \neg \ b \Rightarrow B
\]

\[
\vdash \{A\} \text{ while } b \text{ do } c \ \{B\}
\]

(C is the loop invariant)
Example: Assignment

• (Assuming that x does not appear in e)
  Prove that \{true\} x := e \{ x = e \}

• Assignment Rule:
  \[\vdash \{e = e\} x := e \{x = e\}\]
  because \([e/x](x = e) \rightarrow e = e\)

• Use Assignment + Consequence:
  \[\vdash true \Rightarrow e = e\]
  \[\vdash \{e = e\} x := e \{x = e\}\]
  \[\vdash \{true\} x := e \{x = e\}\]
The Assignment Axiom (Cont.)

• “Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic.” - Tony Hoare

• Caveats are sometimes needed for languages with aliasing (the strong update problem):
  - If x and y are aliased then
    \[
    \{ \text{true} \} x := 5 \{ x + y = 10 \}
    \]
    is true
Example: Conditional

\[
D_1 :: \vdash \{\text{true} \land y \leq 0\} \ x := 1 \ \{x > 0\}
\]

\[
D_2 :: \vdash \{\text{true} \land y > 0\} \ x := y \ \{x > 0\}
\]

\[
\vdash \{\text{true}\} \text{ if } y \leq 0 \text{ then } x := 1 \text{ else } x := y \ \{x > 0\}
\]

- \(D_1\) and \(D_2\) were obtained by consequence and assignment. \(D_1\) details:

\[
\vdash \{1 > 0\} \ x := 1 \ \{x > 0\} \quad \vdash \text{true} \land y \leq 0 \Rightarrow 1 > 0
\]

\[
\vdash D_1 :: \{\text{true} \land y \leq 0\} \ x := 1 \ \{x > 0\}
\]
Example: Loop

- We want to derive that
  \[ \vdash \{ x \leq 0 \} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x = 6 \} \]
- Use the rule for while with invariant \( x \leq 6 \)

\[
\begin{align*}
\vdash x \leq 6 \land x \leq 5 & \Rightarrow x + 1 \leq 6 & \vdash \{ x + 1 \leq 6 \} \ x := x + 1 \{ x \leq 6 \} \\
\vdash \{ x \leq 6 \land x \leq 5 \} \ x := x + 1 \{ x \leq 6 \}
\end{align*}
\]

\[ \vdash \{ x \leq 6 \} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x \leq 6 \land x > 5 \} \]

- Then finish-off with consequence

\[
\begin{align*}
\vdash x \leq 0 & \Rightarrow x \leq 6 \\
\vdash x \leq 6 \land x > 5 & \Rightarrow x = 6 & \vdash \{ x \leq 6 \} \text{ while } ... \{ x \leq 6 \land x > 5 \}
\end{align*}
\]

\[ \vdash \{ x \leq 0 \} \text{ while } ... \{ x = 6 \} \]
Using Hoare Rules

• Hoare rules are mostly syntax directed
• There are three wrinkles:
  - What invariant to use for while? (fix points, widening)
  - When to apply consequence? (theorem proving)
  - How do you prove the implications involved in consequence? (theorem proving)
• This is how theorem proving gets in the picture
  - This turns out to be doable!
  - The loop invariants turn out to be the hardest problem!
    (Should the programmer give them? See Dijkstra, ESC.)
Where Do We Stand?

- We have a **language for asserting properties of programs**
- We know when such **an assertion is true**
- We also have a symbolic **method for deriving assertions**

\[ \{A\} \subset \{B\} \]

\[ \sigma \models A \]

\[ \models \{A\} \subset \{B\} \]
Soundness and Completeness
Soundness of Axiomatic Semantics

- Formal statement of **soundness**: if \( \vdash \{ A \} \preceq \{ B \} \) then \( \models \{ A \} \preceq \{ B \} \)

  or, equivalently

  For all \( \sigma \), if \( \sigma \models A \)

  and \( \text{Op} ::= \langle c, \sigma \rangle \Downarrow \sigma' \)

  and \( \text{Pr} ::= \vdash \{ A \} \preceq \{ B \} \)

  then \( \sigma' \models B \)

- “Op” === “Opsem Derivation”
- “Pr” === “Axiomatic Proof”

How shall we prove this, oh class?
Not Easily!

• By induction on the structure of $c$?
  - No, problems with while and rule of consequence

• By induction on the structure of $Op$?
  - No, problems with while

• By induction on the structure of $Pr$?
  - No, problems with consequence

• By simultaneous induction on the structure of $Op$ and $Pr$?
  - Yes! New Technique!
Simultaneous Induction

• Consider two structures Op and Pr
  - Assume that \( x < y \) iff \( x \) is a substructure of \( y \)

• Define the ordering
  \[ (o, p) \prec (o', p') \text{ iff } \]
  \[ o < o' \text{ or } o = o' \text{ and } p < p' \]
  - Called lexicographic (dictionary) ordering

• This \( \prec \) is a well-founded order and leads to simultaneous induction

• If \( o < o' \) then \( h \) can actually be larger than \( h' \)!

• It can even be unrelated to \( h' \)!
"The Real Deal"
Axiomatic Semantics

"INTERNATIONAL OBFUSCATED C++ CODE CONTEST FINALS"

"N O B O D Y U N D E R S T A N D S M E . "
Soundness of Axiomatic Semantics

• Formal statement of **soundness**:  
  \[
  \text{If } \vdash \{ A \} \subseteq \{ B \} \text{ then } \models \{ A \} \subseteq \{ B \}
  \]
  or, equivalently
  \[
  \text{For all } \sigma, \text{ if } \sigma \models A \\
  \text{and Op :: } <c, \sigma> \Downarrow \sigma' \\
  \text{and Pr :: } \vdash \{ A \} \subseteq \{ B \} \\
  \text{then } \sigma' \models B
  \]

• “Op” = “Opsem Derivation”
• “Pr” = “Axiomatic Proof”
Simultaneous Induction

- Consider two structures Op and Pr
  - Assume that $x < y$ iff $x$ is a substructure of $y$

- Define the ordering
  \[(o, p) < (o', p')\] iff
  \[o < o' \text{ or } o = o' \text{ and } p < p'\]
  - Called lexicographic (dictionary) ordering

- This $<$ is a well founded order and leads to simultaneous induction

- If $o < o'$ then $p$ can actually be larger than $p'$!

- It can even be unrelated to $p'$!
Soundness of the While Rule
(Indiana Proof and the Slide of Doom)

• Case: last rule used in Pr : ⊢ {A} c {B} was the while rule:

\[
\begin{align*}
\text{Pr}_1 &:: \vdash \{A \land b\} c \{A\} \\
\vdash \{A\} \text{ while } b \text{ do } c \{A \land \neg b\}
\end{align*}
\]

• Two possible rules for the root of Op (by inversion)
  - We’ll only do the complicated case:

\[
\begin{align*}
\text{Op}_1 &:: <b, \sigma> \downarrow \text{true} \quad \text{Op}_2 :: <c,\sigma> \downarrow \sigma' \quad \text{Op}_3 :: <\text{while } b \text{ do } c, \sigma'> \downarrow \\
\hline
<\text{while } b \text{ do } c, \sigma> &\downarrow \sigma''
\end{align*}
\]

Assume that \( \sigma \models A \)

To show that \( \sigma'' \models A \land \neg b \)

• By soundness of booleans and Op\(_1\) we get \( \sigma \models b \)
  - Hence \( \sigma \models A \land b \)

• By IH on Pr\(_1\) and Op\(_2\) we get \( \sigma' \models A \)

• By IH on Pr and Op\(_3\) we get \( \sigma'' \models A \land \neg b \), q.e.d.
  - This is the tricky bit!
Soundness of the While Rule

• Note that in the last use of IH the derivation $Pr$ did not decrease
• But $Op_3$ was a sub-derivation of $Op$
• See Winskel, Chapter 6.5, for a soundness proof with denotational semantics
Completeness of Axiomatic Semantics

• If $\models \{A\} \preceq \{B\}$ can we always derive $\vdash \{A\} \preceq \{B\}$?
• If so, axiomatic semantics is **complete**
• If not then there are valid properties of programs that we cannot verify with Hoare rules :-(

• Good news: for our language the Hoare triples are complete

• Bad news: only if the underlying logic is complete (whenever $\models A$ we also have $\vdash A$)
  - this is called **relative completeness**
Examples, General Plan

• OK, so:
  \[ \vdash \{ x < 5 \land z = 2 \} \ y := x + 2 \{ y < 7 \} \]

• Can we prove it?
  \[ ?\vdash ? \{ x < 5 \land z = 2 \} \ y := x + 2 \{ y < 7 \} \]

• Well, we could easily prove:
  \[ \vdash \{ x + 2 < 7 \} \ y := x + 2 \{ y < 7 \} \]

• And we know ...
  \[ \vdash x < 5 \land z = 2 \Rightarrow x + 2 < 7 \]

• Shouldn’t those two proofs be enough?
Proof Idea

• Dijkstra’s idea: To verify that \( \{ A \} \subseteq \{ B \} \)

  a) Find out all predicates \( A' \) such that \( \models \{ A' \} \subseteq \{ B \} \)
      • call this set \( \text{Pre}(c, B) \) (Pre = “pre-conditions”)
  b) Verify for one \( A' \in \text{Pre}(c, B) \) that \( A \Rightarrow A' \)

• Assertions can be ordered:

  false \( \Rightarrow \) \( \text{Pre}(c, B) \) \( \Rightarrow \) true

  strong \( \uparrow \) \( \Rightarrow \) \( \text{weakest} \) weak

  A
  precondition: \( \text{WP}(c, B) \)

• Thus: compute \( \text{WP}(c, B) \) and prove \( A \Rightarrow \text{WP}(c, B) \)
Proof Idea (Cont.)

- **Completeness** of axiomatic semantics:
  
  \[ \text{If } \vdash \{ A \} \subseteq \{ B \} \text{ then } \vdash \{ A \} \subseteq \{ B \} \]

- Assuming that we can compute \( \text{wp}(c, B) \) with the following properties:
  - \( \text{wp} \) is a precondition (according to the Hoare rules)
    
    \[ \vdash \{ \text{wp}(c, B) \} \subseteq \{ B \} \]
  
  - \( \text{wp} \) is (truly) the weakest precondition
    
    \[ \text{If } \vdash \{ A \} \subseteq \{ B \} \text{ then } \vdash A \Rightarrow \text{wp}(c, B) \]
    
    \[ \vdash A \Rightarrow \text{wp}(c, B) \quad \vdash \{ \text{wp}(c, B) \} \subseteq \{ B \} \]
    
    \[ \vdash \{ A \} \subseteq \{ B \} \]
  
  - We also need that whenever \( \vdash A \) then \( \vdash A \)!
Weakest Preconditions

- Define $\text{wp}(c, B)$ inductively on $c$, following the Hoare rules:

  - $\text{wp}(c_1; c_2, B) = \text{wp}(c_1, \text{wp}(c_2, B))$

  - $\text{wp}(x := e, B) = [e/x]B$

  - $\text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = \{ E \Rightarrow A_1 \land \neg E \Rightarrow A_2 \} \text{ if } E \text{ then } c_1 \text{ else } c_2 \{ B \}$

- $\text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \Rightarrow \text{wp}(c_1, B) \land \neg E \Rightarrow \text{wp}(c_2, B)$
Weakest Preconditions for Loops

• We start from the unwinding equivalence
  \[
  \text{while } b \text{ do } c = \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip}
  \]
• Let \( w = \text{while } b \text{ do } c \) and \( W = wp(w, B) \)
• We have that
  \[
  W = b \Rightarrow wp(c, W) \land \neg b \Rightarrow B
  \]
• But this is a recursive equation!
  - We know how to solve these using domain theory
• But we need a domain for assertions
A Partial Order for Assertions

• Which assertion contains the least information?
  - “true” - does not say anything about the state

• What is an appropriate information ordering?
  \[ A \sqsubseteq A' \text{ iff } A' \models A' \Rightarrow A \]

• Is this partial order complete?
  - Take a chain \( A_1 \sqsubseteq A_2 \sqsubseteq \ldots \)
  - Let \( \land A_i \) be the infinite conjunction of \( A_i \)
    \[ \sigma \models \land A_i \text{ iff for all } i \text{ we have that } \sigma \models A_i \]
  - I assert that \( \land A_i \) is the least upper bound

• Can \( \land A_i \) be expressed in our language of assertions?
  - In many cases: yes (see Winskel), we’ll assume yes for now
Weakest Precondition for WHILE

• Use the fixed-point theorem
  \[ F(A) = b \Rightarrow wp(c, A) \land \neg b \Rightarrow B \]
  - (Where did this come from? Two slides back!)
  - I assert that F is both monotonic and continuous

• The least-fixed point (= the weakest fixed point) is
  \[ wp(w, B) = \bigwedge F^i(true) \]

• Notice that unlike for denotational semantics of IMP we are not working on a flat domain!
Weakest Preconditions (Cont.)

- Define a family of wp’s
  - \( wp_k(\text{while } e \text{ do } c, B) = \) weakest precondition on which the loop terminates in \( B \) if it terminates in \( k \) or fewer iterations
    \[
    wp_0 = \neg E \Rightarrow B \\
    wp_1 = E \Rightarrow wp(c, wp_0) \land \neg E \Rightarrow B \\
    ...
    \]
  - \( wp(\text{while } e \text{ do } c, B) = \bigwedge_{k \geq 0} wp_k = \text{lub } \{ wp_k \mid k \geq 0 \} \)

- See Necula document on the web page for the proof of completeness with weakest preconditions

- Weakest preconditions are
  - Impossible to compute (in general)
  - Can we find something easier to compute yet sufficient?
Not Quite Weakest Preconditions

• Recall what we are trying to do:

\[ \text{false} \quad \Rightarrow \quad \text{true} \]

**Pre(s, B)**

• Construct a **verification condition**: \( VC(c, B) \)
  - Our loops will be annotated with loop invariants!
  - \( VC \) is guaranteed to be stronger than \( WP \)
  - But still weaker than \( A \): \( A \implies VC(c, B) \implies WP(c, B) \)
Groundwork

• Factor out the hard work
  - Loop invariants  
  - Function specifications (pre- and post-conditions)

• Assume programs are annotated with such specs
  - Good software engineering practice anyway
  - Requiring annotations = Kiss of Death?

• New form of while that includes a loop invariant:

  \[ \text{while}_{\text{Inv}} b \text{ do } c \]

  - Invariant formula $\text{Inv}$ must hold every time before $b$ is evaluated

• A process for computing VC(annotated_command, post_condition) is called \texttt{VCGen}
Verification Condition Generation

• Mostly follows the definition of the wp function:

\[
\begin{align*}
VC(\text{skip}, B) &= B \\
VC(c_1; c_2, B) &= VC(c_1, VC(c_2, B)) \\
VC(\text{if } b \text{ then } c_1 \text{ else } c_2, B) &= b \Rightarrow VC(c_1, B) \land \neg b \Rightarrow VC(c_2, B) \\
VC(x := e, B) &= [e/x] B \\
VC(\text{let } x = e \text{ in } c, B) &= [e/x] VC(c, B)
\end{align*}
\]
VCGen for WHILE

\[
VC(\text{while}_{\text{Inv}} e \text{ do } c, B) =
\]

\[
\text{Inv} \land (\forall x_1 \ldots x_n. \text{Inv} \Rightarrow (e \Rightarrow VC(c, \text{Inv}) \land \neg e \Rightarrow B))
\]

- \text{Inv} is the loop invariant (provided externally)
- \(x_1, \ldots, x_n\) are all the variables modified in \(c\)
- The \(\forall\) is similar to the \(\forall\) in mathematical induction:
  \[
P(0) \land \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)
  \]
Example VCGen Problem

• Let’s compute the VC of this program with respect to post-condition $x \neq 0$

```plaintext
x = 0;
y = 2;
while $x + y = 2$ and $y > 0$ do
  y := y - 1;
x := x + 1
```

First, what do we expect? What precondition do we need to ensure $x \neq 0$ after this?
Example of VC

- By the sequencing rule, first we do the while loop (call it $w$):

  ```
  while $x + y = 2$ and $y > 0$ do
    $y := y - 1$;
    $x := x + 1$
  ```

- VCGen($w$, $x \neq 0$) = $x + y = 2 \land$
  `\forall x, y. x + y = 2 \Rightarrow (y > 0 \Rightarrow VC(c, x + y = 2) \land y \leq 0 \Rightarrow x \neq 0)`

- VCGen($y := y - 1$; $x := x + 1$, $x + y = 2$) =
  `(x + 1) + (y - 1) = 2`

- $w$ Result: $x + y = 2 \land$
  `\forall x, y. x + y = 2 \Rightarrow (y > 0 \Rightarrow (x + 1) + (y - 1) = 2 \land y \leq 0 \Rightarrow x \neq 0)`
Example of VC (2)

- $\text{VC}(w, x \neq 0) = x+y=2 \land$
  $\forall x, y. x+y=2 \Rightarrow$
  $(y>0 \Rightarrow (x+1)+(y-1)=2 \land y\leq 0 \Rightarrow x \neq 0)$

- $\text{VC}(x := 0; y := 2 ; w, x \neq 0) = 0+2=2 \land$
  $\forall x, y. x+y=2 \Rightarrow$
  $(y>0 \Rightarrow (x+1)+(y-1)=2 \land y\leq 0 \Rightarrow x \neq 0)$

- So now we ask an automated theorem prover to prove it.
$ ./Simplify

> (AND (EQ (+ 0 2) 2)
   (FORALL (x y) (IMPLIES (EQ (+ x y) 2)
    (AND (IMPLIES (> y 0)
      (EQ (+ (+ x 1)(- y 1)) 2))
    (IMPLIES (<= y 0) (NEQ x 0))))))

1: Valid.

• Huzzah!
• Simplify is a non-trivial five megabytes
Can We Mess Up VCGen?

• The invariant is from the user (= the adversary, the untrusted code base)

• Let’s use a loop invariant that is too weak, like “true”.

• \( VC = true \land \forall x, y. true \Rightarrow (y > 0 \Rightarrow true \land y \leq 0 \Rightarrow x \neq 0) \)

• Let’s use a loop invariant that is false, like “\(x \neq 0\)”. 

• \( VC = 0 \neq 0 \land \forall x, y. x \neq 0 \Rightarrow (y > 0 \Rightarrow x + 1 \neq 0 \land y \leq 0 \Rightarrow x \neq 0) \)
$ ./Simplify
> (AND TRUE
   (FORALL ( x y ) (IMPLIES TRUE
      (AND (IMPLIES (> y 0) TRUE)
      (IMPLIES (<= y 0) (NEQ x 0))))))

Counterexample: context:
   (AND
      (EQ x 0)
      (<= y 0)
   )

1: Invalid.

• OK, so we won’t be fooled.
Soundness of VCGen

• Simple form

\[ \vdash \{ \text{VC}(c,B) \} \ c \ \{ \ B \} \]

• Or equivalently that

\[ \vdash \text{VC}(c, B) \Rightarrow \text{wp}(c, B) \]

• Proof is by induction on the structure of \( c \)
  - Try it!

• Soundness holds for any choice of invariant!

• Next: properties and extensions of VCs
VC and Invariants

• Consider the Hoare triple:
  \[ \{ x \leq 0 \} \text{ while}_{I(x)} x \leq 5 \text{ do } x := x + 1 \{ x = 6 \} \]

• The VC for this is:
  \[ x \leq 0 \Rightarrow I(x) \land \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \land x \leq 5 \Rightarrow I(x+1)) \]

• Requirements on the invariant:
  - Holds on entry \[ \forall x. x \leq 0 \Rightarrow I(x) \]
  - Preserved by the body \[ \forall x. I(x) \land x \leq 5 \Rightarrow I(x+1) \]
  - Useful \[ \forall x. I(x) \land x > 5 \Rightarrow x = 6 \]

• Check that \( I(x) = x \leq 6 \) satisfies all constraints
Forward VCGen

- Traditionally the VC is computed backwards
  - That’s how we’ve been doing it in class
  - It works well for structured code
- But it can also be computed forward
  - Works even for un-structured languages (e.g., assembly language)
  - Uses symbolic execution, a technique that has broad applications in program analysis
    - e.g., the PREfix tool (Intrinsa, Microsoft) does this
Forward VC Gen Intuition

• Consider the sequence of assignments

\[ x_1 := e_1; \ x_2 := e_2 \]

• The \( \text{VC}(c, B) = [e_1/x_1]([e_2/x_2]B) \)
\[ = [e_1/x_1, \ e_2[e_1/x_1]/x_2] \ B \]

• We can compute the substitution in a forward way using **symbolic execution** (aka **symbolic evaluation**)
  - Keep a symbolic state that maps variables to expressions
  - Initially, \( \Sigma_0 = \{ \} \)
  - After \( x_1 := e_1, \Sigma_1 = \{ x_1 \rightarrow e_1 \} \)
  - After \( x_2 := e_2, \Sigma_2 = \{x_1 \rightarrow e_1, \ x_2 \rightarrow e_2[e_1/x_1]\} \)
  - Note that we have applied \( \Sigma_1 \) as a substitution to right-hand side of assignment \( x_2 := e_2 \)
Simple Assembly Language

- Consider the language of instructions:
  \[ I ::= \ x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid \text{L: } \mid \text{return } \mid \text{inv } e \]

- The “\text{inv } e” instruction is an annotation
  - Says that boolean expression \( e \) holds at that point

- Each function \( f() \) comes with \( \text{Pre}_f \) and \( \text{Post}_f \) annotations (\text{pre-} and \text{post-conditions})

- New Notation (yay!): \( I_k \) is the instruction at address \( k \)
Symex States

• We set up a symbolic execution state:
  \( \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \)
  \( \Sigma(x) \) = the symbolic value of \( x \) in state \( \Sigma \)
  \( \Sigma[x:=e] \) = a new state in which \( x \)'s value is \( e \)

• We use states as substitutions:
  \( \Sigma(e) \) - obtained from \( e \) by replacing \( x \) with \( \Sigma(x) \)

• Much like the opsem so far ...
Symex Invariants

• The symbolic executor tracks invariants passed

• A new part of symex state: \( \text{Inv} \subseteq \{1...n\} \)

• If \( k \in \text{Inv} \) then \( I_k \) is an invariant instruction that we have already executed

• Basic idea: execute an \textit{inv} instruction only \textbf{twice}:
  - The \textbf{first time} it is encountered
  - Once more time around an \textbf{arbitrary} iteration
Symex Rules

• Define a VC function as an interpreter:

\[ VC : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion} \]

\[
\begin{align*}
\text{VC}(k, \Sigma, \text{Inv}) &= \text{VC}(L, \Sigma, \text{Inv}) \quad \text{if } l_k = \text{goto } L \\
&= e \Rightarrow \text{VC}(L, \Sigma, \text{Inv}) \land \neg e \Rightarrow \text{VC}(k+1, \Sigma, \text{Inv}) \quad \text{if } l_k = \text{if } e \text{ goto } L \\
&= \text{VC}(k+1, \Sigma[x:=\Sigma(e)], \text{Inv}) \quad \text{if } l_k = x := e \\
&= \Sigma(\text{Post}_{\text{current-function}}) \quad \text{if } l_k = \text{return} \\
&= \Sigma(\text{Pre}_f) \land \forall a_1 \ldots a_m. \Sigma'(\text{Post}_f) \Rightarrow \\text{VC}(k+1, \Sigma', \text{Inv}) \quad \text{if } l_k = f() \\
\end{align*}
\]

(where \(y_1, \ldots, y_m\) are modified by \(f\))

and \(a_1, \ldots, a_m\) are fresh parameters

and \(\Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m]\)
Symex Invariants (2a)

Two cases when seeing an invariant instruction:

2. We see the invariant for the first time
   - \( I_k = \text{inv e} \)
   - \( k \not\in \text{Inv} \) (= “not in the set of invariants we’ve seen”)
   - Let \( \{y_1, \ldots, y_m\} = \) the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, \ldots, a_m \) be fresh new symbolic parameters

\[
\text{VC}(k, \Sigma, \text{Inv}) = \Sigma(\text{e}) \land \forall a_1 \ldots a_m. \Sigma'(\text{e}) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv} \cup \{k\})
\]

with \( \Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m] \)

(like a function call)
Symex Invariants (2b)

1. We see the invariant for the second time
   - \( I_k = \text{inv } E \)
   - \( k \in \text{Inv} \)
   \[ \text{VC}(k, \Sigma, \text{Inv}) = \Sigma(e) \]
   (like a function return)

• Some tools take a more simplistic approach
  - Do not require invariants
  - Iterate through the loop a fixed number of times
  - PREfix, versions of ESC (DEC/Compaq/HP SRC)
  - Sacrifice completeness for usability