“The Real Deal” Axiomatic Semantics

Soundness of Axiomatic Semantics

- Formal statement of soundness:
  \[ \text{If } \vdash \{A\} \subseteq \{B\} \text{ then } \models \{A\} \subseteq \{B\} \]
  or, equivalently
  For all \(\sigma\), if \(\sigma \models A\)
  and \(\text{Op} :: <c, \sigma> \Downarrow \sigma'\)
  and \(\text{Pr} :: \vdash \{A\} \subseteq \{B\}\)
  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 \(\text{Pr} :: \vdash \{A\} \subseteq \{B\}\) was the while rule:
  \[\text{Pr}_1 :: \vdash \{A \land b\} \subseteq \{A\}\]
  \(\vdash \{A\} \text{ while } b \text{ do } c \{A \land \neg b\}\)
- Two possible rules for the root of Op (by inversion)
  \(\text{Pr}_1 :: \vdash \{A \land b\} \subseteq \{A\}\)
  \(\vdash \{A\} \text{ while } b \text{ do } c \{A \land \neg b\}\)
  \(\text{Pr}_2 :: \vdash \{A \land b\} \subseteq \{A\}\)
  \(\vdash \{A\} \text{ while } b \text{ do } c \{A \land \neg b\}\)

Assume that \(\sigma \models A\)
To show that \(\sigma'' \models A \land b\)
- By soundness of booleans and \(\text{Op}_1\), we get \(\sigma \models b\)
  - Hence \(\sigma \models A \land b\)
- By IH on \(\text{Pr}_1\) and \(\text{Op}_2\), we get \(\sigma' \models A\)
- By IH on \(\text{Pr}\) and \(\text{Op}_3\), 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 \(\text{Pr}\) did not decrease
- But \(\text{Op}_2\) was a sub-derivation of \(\text{Op}\)
- See Winskel, Chapter 6.5, for a soundness proof with denotational semantics

Completeness of Axiomatic Semantics

- If \(\vdash \{A\} \subseteq \{B\}\) can we always derive \(\vdash \{A\} \subseteq \{B\}\) ?
- 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:
  \[ \{ x < 5 \land z = 2 \} y := x + 2 \{ y < 7 \} \]
- Can we prove it?
  \[ \{ x < 5 \land z = 2 \} y := x + 2 \{ y < 7 \} \]
- Well, we **could** easily prove:
  \[ \{ x+2 < 7 \} y := x + 2 \{ y < 7 \} \]
- And we know …
  \[ x < 5 \land z = 2 \implies x+2 < 7 \]
- Shouldn’t those two proofs be enough?

Proof Idea

- Dijkstra’s idea: To verify that \( \{ A \} c \{ B \} \)
  a) Find out all predicates \( A' \) such that \( \not \vdash \{ A' \} c \{ B \} \)
      * call this set \( \text{Pre}(c, B) \)
      (Pre = “pre-conditions”)
  b) Verify for one \( A' \in \text{Pre}(c, B) \) that \( A \implies A' \)
- Assertions can be ordered:
  \[
  \begin{array}{ccc}
  \text{false} & \implies & \text{true} \\
  \text{strong} & \implies & \text{weak} \\
  \end{array}
  \]
- Thus: compute \( \text{WP}(c, B) \) and prove \( A \implies \text{WP}(c, B) \)

Proof Idea (Cont.)

- **Completeness** of axiomatic semantics:
  - If \( \vdash \{ A \} c \{ B \} \)
  then \( \vdash \{ A \} c \{ B \} \)
- Assuming that we can compute \( \text{wp}(c, B) \) with the following properties:
  1. \( \text{wp} \) is a pre-condition (according to the Hoare rules)
  \[ \vdash \{ \text{wp}(c, B) \} c \{ B \} \]
  2. \( \text{wp} \) is (truly) the weakest pre-condition
  \[ \vdash \{ A \} c \{ B \} \implies \vdash A \implies \text{wp}(c, B) \]
  \[ \vdash \text{wp}(c, B) \]
- We also need that whenever \( \not \vdash A \) then \( \vdash A' \)

Weakest Preconditions

- Define \( \text{wp}(c, B) \) inductively on \( c \), following the Hoare rules:
  \[ \begin{align*}
  \text{wp}(c_1; c_2, B) &= \{ A \} c_1 \{ C \} c_2 \{ B \} \\
  \text{wp}(x := e, B) &= [e/x]B \\
  \text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) &= \begin{cases} 
  \{ A \} c_1 \{ B \} & \text{if } E \\
  \{ A \} c_2 \{ B \} & \text{if } \neg E 
  \end{cases}
  \end{align*}\]
- \( \text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \implies \text{wp}(c_1, B) \land \neg E \implies \text{wp}(c_2, B) \)

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?
  \[ A \sqsubseteq A' \iff \vdash A' \implies A \]
- Is this partial order complete?
  - Take a chain \( A_1 \sqsubseteq A_2 \sqsubseteq \ldots \)
  - Let \( \land A_i \) be the infinite conjunction of \( A_i \)
    \[ \sigma \vdash \land A_i \iff \text{for all } i \text{ we have that } \sigma \vdash A_i \]
  - 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
  \[ wp(w, B) = \bigwedge F(true) \]
- 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
  \[ wp_k(\text{while } e \text{ do } c, B) = \text{weakest precondition on which the loop terminates in } B \text{ if it terminates in } k \text{ or fewer iterations} \]

- \( wp_0 = \neg E \Rightarrow B \)
- \( wp_1 = E \Rightarrow wp(c, wp_0) \land \neg E \Rightarrow B \)

- \( wp(\text{while } e \text{ do } c, B) = \bigwedge_{k \geq 0} wp_k = \text{lub} \{ wp_k | k \geq 0 \} \)
- 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:
  \[
  \begin{array}{ccc}
  \text{false} & \Rightarrow & \text{true} \\
  \text{strong} & \text{precondition: } WP(c, B) & \text{weak} \\
  \text{verification condition: } VC(c, B) \\
  \end{array}
  \]

- 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 \( A: A \Rightarrow VC(c, B) \Rightarrow WP(c, B) \)

Verification Condition Generation

- Mostly follows the definition of the \( wp \) function:
  \[
  \begin{align*}
  VC(\text{skip}, B) &= B \\
  VC(c_1; c_2, B) &= VC(c_1, VC(c_2, B)) \\
  VC(\text{if } b \text{ then } c_1 \text{ else } c_2, B) &= \left\{ \begin{array}{ll}
  b & \Rightarrow VC(c_1, B) \land \neg b \Rightarrow VC(c_2, B) \\
  \end{array} \right. \\
  VC(x := e, B) &= \left[ e/x \right] B \\
  VC(\text{let } x = e \text{ in } c, B) &= \left[ e/x \right] VC(c, B) \\
  VC(\text{while}_\text{inv} b \text{ do } c, B) &= ?
  \end{align*}
  \]

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 \( inv \) must hold every time before \( b \) is evaluated
  - A process for computing \( VC(\text{annotated}_\text{command}, \text{post}_\text{condition}) \) is called \( VCGen \)

VCGen for WHILE

- \( VC(\text{while}_\text{inv} b \text{ do } c, B) = \)
  \[ \text{Inv} \land (\forall x_1, \ldots, x_n. \text{Inv} \Rightarrow (e \Rightarrow VC(c, \text{Inv}) \land \neg e \Rightarrow B)) \]
  \( inv \) holds on entry
  \( inv \) is preserved in an arbitrary iteration
  \( B \) holds when the loop terminates in an arbitrary iteration

- \( inv \) 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 $w$):

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

- $VCGen(w, x \neq 0) = x+y=2$ and $\forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow x \neq 0)$

- $VCGen(y:=y-1; x:=x+1, x+y=2) = (x+1)+(y-1)=2$ and $\forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow x \neq 0)$

Result:

```
$ ./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

Example of VC (2)

- $VCGen(w, x \neq 0) = x+y=2$ and $\forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow x \neq 0)$

- $VCGen(x := 0; y := 2 ; w, x \neq 0) = 0+2=2$ and $\forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow x \neq 0)$

So now we ask an automated theorem prover to prove it.

Thoreau, Thoreau, Thoreau

```
$ ./Simplify
> (AND TRUE
  (FORALL ( x y ) (IMPLIES TRUE
    (AND (IMPLIES (> y 0)
      (IMPLIES (<= y 0) (NEQ x 0)))))))
Counterexample: context: (AND
  (EQ x 0)
  (<= y 0))
1: Invalid.
```

OK, so we won’t be fooled.

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 “$x \neq 0$”.
  - $VC = 0 \neq 0 \land \forall x,y. x \neq 0 \Rightarrow (y>0 \Rightarrow x+1 \neq 0 \land y\leq 0 \Rightarrow x \neq 0)$

Emerson, Emerson, Emerson

```
$ ./Simplify
> (AND TRUE
  (FORALL ( x y ) (IMPLIES TRUE
    (AND (IMPLIES (> y 0)
      (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:
  \[ \{ x \leq 0 \} \text{while}_{\text{if}} \ x \leq 5 \text{do} \ x := x + 1 [x = 6] \]
- The VC for this is:
  \[ x \leq 0 \Rightarrow l(x) \land \forall x. (l(x)) \Rightarrow (x > 5 \Rightarrow x = 6 \land x \leq 5 \Rightarrow l(x+1)) \]
- Requirements on the invariant:
  - Holds on entry
    \[ \forall x. x \leq 0 \Rightarrow l(x) \]
  - Preserved by the body
    \[ \forall x. l(x) \land x \leq 5 \Rightarrow l(x+1) \]
  - Useful
    \[ \forall x. l(x) \land x > 5 \Rightarrow x = 6 \]
- Check that \( l(x) = x \leq 6 \) 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 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 \( x_2 := e_2, \Sigma_2 = \{ x_1 \rightarrow e_1, x_2 \rightarrow e_2 \} \)
    - 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:
  \[ I ::= x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L: \mid \text{return} \mid \text{inv } e \]
- 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:
  \[ \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \]
  \[ \Sigma(x) = \text{the symbolic value of } x \text{ in state } \Sigma \]
  \[ \Sigma[x:=e] = \text{a new state in which } x \text{'s value is } e \]
- 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 \( \text{inv} \) instruction only twice:
  - The first time it is encountered
  - Once more time around an arbitrary iteration

Symex Invariants (2a)

Two cases when seeing an invariant instruction:
1. We see the invariant for the first time
   - \( I_k = \text{inv } e \)
   - \( k \notin \text{Inv} \) (= “not in the set of invariants we’ve seen”)
   - Let \( \{y_1, ..., y_m\} \) = the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, ..., a_m \) be fresh new symbolic parameters
   \[
   \text{VC}(k, \Sigma, \text{Inv}) = \\
   \Sigma(e) \land \forall a_1 ... a_m. \Sigma'(e) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv} \cup \{k\})
   \]
   with \( \Sigma' = \Sigma[y_1 := a_1, ..., y_m := a_m] \)

Symex Invariants (2b)

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

Homework

- Homework 3 Due Today
- Homework 4 Out Today
- Read Winskel 7.4-7.6 (on VC’s)
- Read Dijkstra article
- Bonus Lecture Shortly

Old Questions Answered

- Denotational Semantics class question:
  - “What’s up with the continuity requirement?”
  - A function \( F : S^m \rightarrow S^n \) is **continuous** if for every chain \( W \subseteq S^n \)
    - \( F(W) \) has a LUB = \( \sqcup F(W) \)
    - and \( F(\sqcup W) = \sqcup F(W) \)
  - See the Ed Lee paper on the lectures page.