Introduction to Program Verification

CSE 501

Spring 15
Announcements

• Project midpoint report due tonight at 11pm
  – Submit on dropbox
Course Outline

• Static analysis
• Language design
• Program Verification
  – Axiomatic semantics
  – Finding invariants
  – Verified compilers
• Dynamic analysis
• New compilers

We are here
What does verifying programs mean?

- Consider the following program:

  ```
  z = 0;
  i = x;
  while (i) {
      z = z + y;
      i = i - 1;
  }
  ```

- What is the value of z when loop exits?
  - Does the loop actually terminate?
Tools we have seen are not sufficient

- **Types**
  - Proving program termination?

- **Dataflow analysis**
  - We assumed that loops will terminate when we create merge points!

- **Abstract interpretation**
  - What is a good abstraction function?
Axiomatic Semantics

• A system for proving properties about programs

• Key idea:
  – Define the semantics of a construct by describing its effect on assertions about the program state.

• Two components
  – A language for stating assertions (“the assertion logic”)
    • First-Order Logic (FOL), separation logic, or Higher-Order Logic (HOL) etc
    • Many specialized languages developed over the years: Z, Larch, JML, Spec#
  – Deductive rules (“the program logic”) for establishing the truth of such assertions
A little history

• Early years: Unbridled optimism
  – Heavily endorsed by the likes of Hoare and Dijkstra
    If you can prove programs correct, bugs will be a thing of the past.
    – You won’t even have to test your programs!

• The middle ages
  – 1979 paper by DeMillo, Lipton and Perlis:
    – “Proofs in math only work because there is a social process in place to get people to argue them and internalize them.”
    – “Program proofs are too boring for social process to form around them.”
    – “Programs change too fast and proofs are too brittle.”

• The renaissance: new generation of automated reasoning tools
  – A handful of success stories: proving OS kernels, distributed algorithms, network protocols, etc.
  – Better appreciation of costs, benefits and limitations?
The basics

• Hoare triple
  – If the precondition holds before stmt and stmt terminates, postcondition will hold afterward.
• This is a partial correctness assertion.
• We sometimes use the notation
  \([A] \text{ stmt } [B]\)
  to denote a total correctness assertion
• which means you also have to prove termination
What do assertions mean?

• We first need to introduce a programming language

• Let’s start with the following:

\[
e := n \mid x \mid e_1 + e_2 \mid e_1 - e_2
\]
\[
c := x := e \mid c_1; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
\]
What do assertions mean?

- Language constructs defined in terms of big step operational semantics
- Expressions result in values given a state $\sigma$:
  
  $<c, \sigma> \rightarrow \sigma'$

- Examples:
  
  $<5, \sigma> \rightarrow 5$
  
  $<x := 5, \sigma> \rightarrow \sigma[x\rightarrow 5]$
What do assertions mean?

• The language of assertions:
  \[ A ::= \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \geq e_2 \mid A_1 \land A_2 \mid \neg A \mid \forall x. A \]

• Notation \( \sigma \models A \) means that the assertion holds on state \( \sigma \).
  – This is defined inductively over the structure of \( A \).
  – Ex. \( \sigma \models A \land B \) iff \( \sigma \models A \) and \( \sigma \models B \)

• Partial Correctness can then be defined in terms of operational semantics
  \[ \{A\} \ c \ \{B\} \text{ iff } \forall \sigma \forall \sigma'(\sigma \models A \land (c, \sigma) \rightarrow \sigma') \Rightarrow \sigma' \models B \]
Defining axiomatic semantics

• Establishing the truth of a Hoare triple in terms of the operational semantics is impractical

• The real power of AS is the ability to establish the validity of a Hoare triple by using deduction rules.

⊢ \{A\} c \{B\}

means we can deduce the triple from a set of basic axioms
Derivation Rules

• Derivation rules for each language construct

\[ \vdash \{ A[ x \rightarrow e]\} \ x := e \ \{ A \} \]

\[ \vdash \{ A \wedge b \} \ c \ \{ A \} \]

\[ \vdash \{ A \wedge b \} \ c \ \{ A \} \]

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

\[ \vdash \{ A \wedge b \} \ c_1 \ \{ B \} \quad \vdash \{ A \wedge \neg b \} \ c_2 \ \{ B \} \]

\[ \vdash \{ A \} \ \text{if } b \ \text{then } c_1 \ \text{else } c_2 \ \{ B \} \]

\[ \vdash \{ A \} \ c_1 \ \{ C \} \quad \vdash \{ C \} \ c_2 \ \{ B \} \]

\[ \vdash \{ A \} \ c_1 ; \ c_2 \ \{ B \} \]

• Can be combined with the rule of consequence

\[ \vdash A' \rightarrow A \quad \vdash \{ A \} \ c \ \{ B \} \quad \vdash \{ B \rightarrow B' \} \]

\[ \vdash \{ A' \} \ c \ \{ B' \} \]
Soundness and Completeness

• What does it mean for our deduction rules to be sound?
  – You will never be able to prove anything that is not true
  – truth is defined in terms of our original definition of \( \{A\} \sqsubseteq \{B\} \)
    \[
    \forall \sigma \forall \sigma' (\sigma \sqsubseteq A \land \langle c, \sigma \rangle \rightarrow \sigma') \Rightarrow \sigma' \sqsubseteq B
    \]
  – we can prove this, but it’s tricky

• What does it mean for them to be complete?
  – If a statement is true, we should be able to prove it via deduction

• So are they complete?
  – yes and no
  • They are complete relative to the logic
  • but there are no complete and consistent logics for elementary arithmetic (Gödel)
Example

\[ x = x_0 \text{ and } y = y_0 \]

if \( x > y \) {
    \[ t = x - y; \]
    while \( t > 0 \) {
        \[ x = x - 1; \]
        \[ y = y + 1; \]
        \[ t = t - 1; \]
    }
}

\[ x_0 > y_0 \Rightarrow y = x_0 \text{ and } x = y_0 \]
From partial to total correctness

• Total correctness:
  \[ \vdash [A] \ c \ [B] \]

  — Same as before, but must also prove termination

\[
\vdash [A \land b] c_1 \ [B] \quad \vdash [A \land not \ b] c_2 \ [B] \\
\vdash [A if \ b \ then \ c_1 \ else \ c_2] \ [B] \quad \vdash [A[x \rightarrow e]] x := e \ [A]
\]

\[
\vdash [A] c_1 \ [C] \quad \vdash [C] c_2 \ [B] \\
\vdash [A] c_1; c_2 \ [B]
\]

But what about loops??
Rank function

- Function F of the state that
  a) Maps state to an integer
  b) Decreases with every iteration of the loop
  c) Is guaranteed to stay greater than zero

– Also called variant function

\[
\vdash [A \land b \land F = z]c [A \land F < z] \quad \vdash A \land b \Rightarrow F \geq 0
\]

\[
\vdash [A] \text{while } b \text{ do } c [A \land \neg b]
\]
Example

• Can we prove this?

\[ x=x_0 \text{ and } y=y_0 \]

\[
\text{if}(x > y)\{
    t = x - y;
    \text{while}(t > 0)\{
        x = x - 1;
        y = y + 1;
        t = t - 1;
    \}
\}
\]

\[
[ x_0 > y_0 \Rightarrow y=x_0 \text{ and } x=y_0 ]
\]
Soundness

• We gave a semantic soundness condition for \{A\} c \{B\}:
  \[ \forall \sigma, \sigma'. (A(\sigma) \land (\sigma, c) \rightarrow \sigma') \Rightarrow B(\sigma') \]

• Then what does it mean for [A] c [B]?
  1. \[ \forall \sigma, \sigma'. (A(\sigma) \land (\sigma, c) \rightarrow \sigma') \Rightarrow B(\sigma') \]
  2. \[ \forall \sigma. A(\sigma) \Rightarrow \exists \sigma'. (\sigma, c) \rightarrow \sigma' \]
  (i.e., c terminates whenever A is true)
Verification Pragmatics

• Constructing Hoare logic proofs manually is tedious. We should be able to automate most of it.

• (At least that is the hope)
Weakest Preconditions

$P = wp(c, A)$

- $P$ is the weakest predicate such that $\{P\} c \{A\}$
- $P$ is weaker than $Q$ iff $Q \Rightarrow P$
- $wp(x := e, A) = A[e\rightarrow x]$
- $wp(c_1; c_2, A) = wp(c_1, wp(c_2, A))$
- $wp(if b then c_1 else c_2, A) = (b \land wp(c_1, A)) \lor (\neg b \land wp(c_2, A))$
Weakest Preconditions

• while is tricky!

• Let $W = \text{wp}(\text{while } b \text{ do } c, A)$
  Then: $W \leftrightarrow (b \Rightarrow \text{wp}(c, W) \land \neg b \Rightarrow A)$

• This is a recursive equation, where it isn't obvious a solution exists!

• Pragmatic solution: ask programmers to annotate loops with loop invariants.

• $c := x := e \mid c;c \mid \text{if } b \text{ then } c \text{ else } c \mid \{I\} \text{ while } b \text{ do } c$
Weakest Preconditions

• \( \text{wp}(x := e, A) = A[e \rightarrow x] \)
• \( \text{wp}(c_1; c_2, A) = \text{wp}(c_1, \text{wp}(c_2, A)) \)
• \( \text{wp}(\text{if } b \text{ then } c_1 \text{ else } c_2, A) = (b \land \text{wp}(c_1, A)) \lor (\neg b \land \text{wp}(c_2, A)) \)
• \( \text{wp}({\{I}\} \text{ while } b \text{ do } c, A) = I \land \text{written}(c) = \{x_1, \ldots, x_n\} \land (\forall x_1, \ldots, x_n. I \land b \Rightarrow \text{wp}(c, I)) \land (\forall x_1, \ldots, x_n. I \land \neg b \Rightarrow A) \)
Is this really the weakest?

• **Theorem** (Completeness of wpc): 
  For any command $c$ and postcondition $\mathcal{B}$, there exists a command $c'$ annotated with proper loop invariants, such that for any candidate precondition $\mathcal{A}$,

• if $\vdash \{\mathcal{A}\} \ c \ \{\mathcal{B}\}$,
  then $\mathcal{A} \Rightarrow \wp(c', \mathcal{B})$
Is this really the weakest?

• if \( \vdash \{ A \} \ c \ \{ B \} \)
  then \( A \Rightarrow \wp(c', B) \)

• **Proof:** By structural induction on c.

• *Trickiest case:* “while” (unsurprisingly)
Need to pick a good loop invariant for arbitrary
while b do c and B.

• This one works:
Given a program state \( \sigma \), then
\( \forall \sigma'. <\text{while } b \text{ do } c, \sigma> \rightarrow \sigma' \Rightarrow B(\sigma') \)
Weakest Preconditions

• \(\wp(\{I\} \text{ while } b \text{ do } c, A) = I \land \text{written}(c) = \{x_1, \ldots, x_n\} \land (\forall x_1, \ldots, x_n. I \land b \Rightarrow \wp(c, I)) \land (\forall x_1, \ldots, x_n. I \land \neg b \Rightarrow A)\)

• But who comes up with \(I\)?
  – See next lecture for details
Language with arrays

• e := n | x | e1 + e2 | e1 - e2 | a[e]
• c := x := e |
  c1; c2 |
if e then c1 else c2 |
while e do c
Problem with arrays

\{true\}
\begin{align*}
a[k] &= 1; \\
a[j] &= 2; \\
x &= a[k] + a[j]; \\
\{x = 3\}
\end{align*}

\{true\}
\begin{align*}
a[k] &= 1; \\
a[j] &= 2; \\
\{a[k] + a[j] = 3\} \\
x &= a[k] + a[j]; \\
\{x = 3\}
\end{align*}

Now what?
Can we use the standard rule for assignment?

$$wp(x := e, A) = A[e \rightarrow x]$$
What if $k = j$??
Theory of arrays

• Extend the language of assertions with array expressions

• Let \(a\) be an array

• \(a\{i \rightarrow e\}\) is a new array whose \(i^{th}\) entry has value \(e\)
  
  \[a\{i \rightarrow e\}[k] = a[k] \text{ if } k \neq i, \text{ or } e \text{ otherwise}\]
Theory of arrays

• We can then reason about TOA expressions assuming Zero is the zeroed out array

  – Example:
  – Zero\{i\rightarrow 5\}\{j\rightarrow 7\}[k] = 5 \iff i = k \land i \neq j
Assignment rule with TOA

\[ \vdash \{P[a \rightarrow a[i \rightarrow e]]\} \ a[i] = e \ \{P\} \]

\{true\}
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}\]

\{k \neq j\}
a[k \rightarrow 1][j \rightarrow 2][k] + a[k \rightarrow 1][j \rightarrow 2][j]=3
a[k]=1;
a[j]=2;
a[k]+a[j]=3
x=a[k]+a[j];
{x=3}\]
Arrays and loops

\[
\begin{align*}
\{0 \leq i < n\} \\
j = i+1; \\
\text{while } j < n \text{ do} \\
\quad a[i] = a[i] + a[j]; \\
\quad j = j+1; \\
\{ a[i] = \sum_{i \leq k < j} a_0[k] \}
\end{align*}
\]

Reasonable $I$:
\[
\{ a[i] = \sum_{i \leq k < j} a_0[k] \}
\]
Proving with loop invariant

• Recall \( \text{wp}({I} \text{ while } b \text{ do } c, A) = I \land \text{written}(c) = \{x_1, ..., x_n\} \land (\forall x_1, ..., x_n. I \land b \Rightarrow \text{wp}(c, I)) \land (\forall x_1, ..., x_n. I \land \neg b \Rightarrow A) \)

• Let’s check \( I \land b \Rightarrow \text{wp}(c, I) \)

\[
\begin{align*}
\{ a[i \rightarrow a[i] + a[j]] \}[i] &= \sum_{i \leq k < j+1} a_0[k] \\
a[i] &= a[i] + a[j] \\
j &= j + 1
\end{align*}
\]

Do we know \( a[j] = a_0[j] \)?

Make it part of \( I \)!
Proving with improved invariant

Improved I:

\[
\{ a[i] = \sum_{i \leq k < j} a_0[k] \land \forall j \leq k < n . a[k] = a_0[k] \} \\
\downarrow \\
\{ a[i \mapsto a[i] + a[j]][i] = \sum_{i \leq k < j+1} a_0[k] \land \\
\forall j+1 \leq k < n . a[i \mapsto a[i] + a[j]][k] = a_0[k] \} \\
\]

\[a[i] = a[i] + a[j]\]

\[
\{ a[i] = \sum_{i \leq k < j+1} a_0[k] \land \forall j+1 \leq k < n . a[k] = a_0[k] \} \\
j = j + 1 \\
\{ a[i] = \sum_{i \leq k < j} a_0[k] \land \forall j \leq k < n . a[k] = a_0[k] \} \\
\]
Proving with improved invariant

• Still need to check

\{ 0 \leq i < n \} \\
\{ a[i+1] = \Sigma_{i \leq k < i+1} a_0[k] \land \\
\forall i+1 \leq k < n . a[k] = a_0[k] \} \\
\text{while } j < n \text{ do } a[i] = a[i] + a[j]; j = j+1; \\
\{ a[i] = \Sigma_{i \leq k < n} a_0[k] \}
An even better invariant

Check this:

$$\{ a[i] = \sum_{i \leq k < j} a_0[k] \land \forall j \leq k < n . a[k] = a_0[k] \land i < j \}$$

Bottom line:
Coming up with good invariants is hard!

We will see how to deal with that next time