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 ⟵ We are here
  - Finding invariants
  - Verified compilers
- Dynamic analysis
- New compilers

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

{A} stmt {B}

Precondition — Postcondition

- 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] 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 | x | e1 + e2 | e1 - e2
c := x := e |
     c1; c2 |
     if e then c1 else c2 |
     while e do c
```

What do assertions mean?

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

  $\langle 5, \sigma \rangle \rightarrow 5$

  $\langle x := 5, \sigma \rangle \rightarrow \sigma[x \rightarrow 5]$

What do assertions mean?

- The language of assertions:
  $A := \text{true} \mid \text{false} \mid e1 = e2 \mid$
  $e1 \geq e2 \mid A1 \wedge A2 \mid \neg A \mid \forall x.\ A$
- Notation $\sigma \models A$ means that the assertion holds on state σ.
  - This is defined inductively over the structure of A.
  - Ex. $\sigma \models A \wedge B$ iff $\sigma \models A$ and $\sigma \models B$
- Partial Correctness can then be defined in terms of operational semantics

$\{A\}$ c $\{B\}$ iff

$$\forall \sigma \forall \sigma' (\sigma \models A \wedge \langle c, \sigma \rangle \to \sigma') \Rightarrow \sigma' \models B$$

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.

  $\vdash \{A\}\ c\ \{B\}$

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

Derivation Rules

- Derivation rules for each language construct

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

$$\frac{\vdash \{A \wedge b\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg b\}\ c_2\ \{B\}}{\vdash \{A\}\ \texttt{if b then}\ c_1\ \texttt{else}\ c_2\ \{B\}}$$

$$\frac{\vdash \{A \wedge b\}\ c\ \{A\}}{\vdash \{A\}\ \texttt{while b do c}\ \{A \wedge \neg b\}}$$

$$\frac{\vdash \{A\}\ c_1\ \{C\} \qquad \vdash \{C\}\ c_2\ \{B\}}{\vdash \{A\}\ c_1\ ;\ c_2\ \{B\}}$$

- Can be combined with the rule of consequence

$$\frac{\vdash\ A' \rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash \{B \rightarrow B'\}}{\vdash \{A'\}\ c\ \{B'\}}$$

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} c {B}

$$\forall\sigma\forall\sigma'(\sigma \vDash A \ \wedge \ \langle c,\sigma\rangle \rightarrow \sigma') \Rightarrow \sigma' \vDash B$$

  - 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

$$\frac{}{\vdash \{A[x \to e]\} x := e \ \{A\}}$$

$$\frac{\vdash \{A \wedge b\} c_1 \ \{B\} \qquad \vdash \{A \wedge not\ b\} c_2 \ \{B\}}{\vdash \{A\} if\ b\ then\ c_1 else\ c_2 \ \{B\}}$$

$$\frac{\vdash A' \Rightarrow A \vdash \{A\} c \ \{B\} \vdash B \Rightarrow B'}{\vdash \{A'\} c \ \{B'\}}$$

$$\frac{\vdash \{A \wedge b\} c \ \{A\}}{\vdash \{A\} while\ b\ do\ c\ \{A \wedge not\ b\}}$$

$$\frac{\vdash \{A\} c_1 \ \{C\} \qquad \vdash \{C\} c_2 \ \{B\}}{\vdash \{A\} c_1 ; c_2 \ \{B\}}$$

{ $x=x_0$ and $y=y_0$ }

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

{ $x_0 > y_0 \Rightarrow y=x_0$ and $x=y_0$ }

From partial to total correctness

- Total correctness:
  - $\vdash$ [A] c [B]
  - – Same as before, but must also prove termination

$$\frac{\vdash [A \wedge b]c_1\ [B] \qquad \vdash [A \wedge not\ b]c_2\ [B]}{\vdash [A] if\ b\ then\ c_1 else\ c_2\ [B]}$$

$$\frac{}{\vdash [A[x \to e]]x := e\ [A]}$$

$$\frac{\vdash [A]c_1\ [C] \qquad \vdash [C]c_2\ [B]}{\vdash [A]c_1; c_2\ [B]}$$

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

$$\frac{\vdash [A \wedge b \wedge F = z]c\ [A \wedge F < z] \qquad \vdash A \wedge b \Rightarrow F \geq 0}{\vdash [A] while\ b\ do\ c\ [A \wedge not\ b]}$$

Example

- Can we prove this?

[$x=x_0$ and $y=y_0$]

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

[$x_0 > y_0 \Rightarrow y=x_0$ and $x=y_0$]

Soundness

- We gave a semantic soundness condition for $\{A\}$ c $\{B\}$:

  $\forall \sigma, \sigma'. (A(\sigma) \wedge (\sigma, c) \rightarrow \sigma') \Rightarrow B(\sigma')$

- Then what does it mean for $[A]$ c $[B]$?

  (1) $\forall \sigma, \sigma'. (A(\sigma) \wedge (\sigma, c) \rightarrow \sigma') \Rightarrow B(\sigma')$

  (2) $\forall \sigma . A(\sigma) \Rightarrow \exists \sigma' . (\sigma, c) \rightarrow \sigma')$

  (i.e., c terminates whenever A is true)

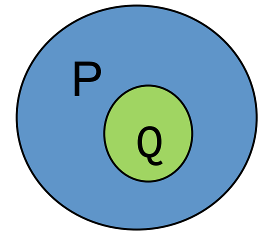
Verification Pragmatics

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

- (At least that is the hope)

Weakest Preconditions

$$P = wp(\texttt{c}, A)$$



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

Weakest Preconditions

- `while` is tricky!

- Let $W = wp($`while b do c`$, A)$
  Then: $W \leftrightarrow (b \Rightarrow wp(c, W) \wedge \neg b \Rightarrow A)$
- 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 | c;c | if b then c else c |`
  `{I} while b do c`

Weakest Preconditions

- $wp(\texttt{x := e}, A) = A[e \rightarrow x]$
- $wp(c_1 ; c_2, A) = wp(c_1, wp(c_2, A))$
- $wp(\texttt{if b then } c_1 \texttt{ else } c_2, A) =$ $(b \wedge wp(c_1, A)) \vee (\neg b \wedge wp(c_2, A))$
- $wp(\{I\}\ \texttt{while b do c}, A) = I$ $\wedge\ written(c) = \{x_1, \ldots, x_n\}$ $\wedge (\forall x_1, \ldots, x_n.\ I \wedge b \Rightarrow wp(c, I))$ $\wedge (\forall x_1, \ldots, x_n.\ I \wedge \neg b \Rightarrow A)$

Is this really the weakest?

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

- **if** $\vdash \{A\}$ c $\{B\}$,
  then $A \Rightarrow wp(c', B)$

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 σ, then
  $\forall \sigma'.$ `<while b do c,` σ`>` $\rightarrow \sigma' \Rightarrow B(\sigma')$

Weakest Preconditions

- $wp(\{I\}$ `while b do c`$, A) = I$
  $\wedge$ $written(c) = \{x_1, \ldots, x_n\}$
  $\wedge (\forall x_1, \ldots, x_n.\ I \wedge b \Rightarrow wp(c, I))$
  $\wedge (\forall x_1, \ldots, x_n.\ I \wedge \neg b \Rightarrow A)$

- 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}
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

→

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

→

Now what?

Can we use the standard rule for assignment?

$wp(x := e, A) = A[e \rightarrow x]$

Problem with arrays

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

→

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

→

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

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
  - $a\{i \rightarrow e\}[k] = a[k]$ if $k \neq i$, or e otherwise

Theory of arrays

- We can then reason about TOA expressions assuming Zero is the zeroed out array
  - Example:
  - $Zero\{i \rightarrow 5\}\{j \rightarrow 7\}[k] = 5 \Leftrightarrow i = k \wedge i \neq j$

Assignment rule with TOA

⊢ {P[a → a{i→e}]} `a[i] = e` {P}

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

{k ≠ j}
{a{k→1}{j→2}[k] + a{k→1}{j→2}[j]=3}
```
a[k]=1;
```
{a{j→2}[k] + a{j→2}[j]=3}
```
a[j]=2;
```
{a[k]+a[j]=3}
```
x=a[k]+a[j];
```
{x=3}

Arrays and loops

$\{ 0 \leq i < n \}$

```
j = i+1;
while j<n do
   a[i] = a[i] + a[j];
   j = j+1;
```

$\{ a[i] = \Sigma_{i \leq k < n}\, a_0[k] \}$

Reasonable I:
$\{ a[i] = \Sigma_{i \leq k < j}\, a_0[k] \}$

Proving with loop invariant

- Recall $wp(\{I\}$ `while b do c`, $A) = I$
  $\wedge$ $written(c) = \{x_1, \ldots, x_n\}$
  $\wedge (\forall x_1, \ldots, x_n.\ I \wedge b \Rightarrow wp(c, I))$
  $\wedge (\forall x_1, \ldots, x_n.\ I \wedge \neg b \Rightarrow A)$

- Let's check $I \wedge b \Rightarrow wp(c, I)$

$\{ a\{i \rightarrow a[i]+a[j]\}[i] = \Sigma_{i \leq k < j+1}\, a_0[k] \}$

```
a[i] = a[i] + a[j];
j = j+1;
```

$\{ a[i] = \Sigma_{i \leq k < j}\, a_0[k] \}$

Do we know $a[j] = a_0[j]$?

Make it part of I!

Proving with improved invariant

Improved I:

$$\{ a[i] = \Sigma_{i \le k < j}\, a_0[k] \wedge \forall j \le k < n \,.\, a[k] = a_0[k] \}$$

$$\downarrow$$

$$\{ a\{i \rightarrow a[i]+a[j]\}[i] = \Sigma_{i \le k < j+1}\, a_0[k] \wedge$$
$$\forall j+1 \le k < n \,.\, a\{i \rightarrow a[i]+a[j]\}[k] = a_0[k] \}$$

```
a[i] = a[i] + a[j]
```

$$\{ a[i] = \Sigma_{i \le k < j+1}\, a_0[k] \wedge \forall j+1 \le k < n \,.\, a[k] = a_0[k] \}$$

```
j = j + 1
```

$$\{ a[i] = \Sigma_{i \le k < j}\, a_0[k] \wedge \forall j \le k < n \,.\, a[k] = a_0[k] \}$$

Proving with improved invariant

- Still need to check

$\{ 0 \leq i < n \}$

$\{ a[i+1] = \Sigma_{i \leq k < i+1} a_0[k] \wedge$
$\forall i+1 \leq k < n \,.\, a[k] = a_0[k] \}$

```
j = i + 1
```

$\{ a[i] = \Sigma_{i \leq k < j} a_0[k] \wedge \forall j \leq k < n \,.\, a[k] = a_0[k] \}$

```
while j < n 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:

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

Bottom line:

Coming up with good invariants is hard!

We will see how to deal with that next time