Axiomatic Semantics III

The Verification Crusade

Wei Hu Memorial Homework Award

- Many turned in HW3 code like this:

```
let rec matches re s = match re with
| Star(r) -> union (singleton s)
                     (matches (Concat(r,Star(r))) s)
```

- Which is a direct translation of:

$$R[\![r^*]\!]s = \{s\} \cup R[\![rr^*]\!]s$$

or, equivalently:

$$R[\![r^*]\!]s = \{s\} \cup \{\ y \mid \exists x \in R[\![r]\!]s \wedge y \in R[\![r^*]\!]x\ \}$$

- Why doesn't this work?

Where Are We?

- Axiomatic Semantics: the meaning of a program is what is true after it executes
- Hoare Triples: {A} c {B}
- Weakest Precondition: { WP(c,B) } c {B}
- Verification Condition: $A \Rightarrow VC(c,B) \Rightarrow WP(c,b)$
  - Requires Loop Invariants
  - Backward VC works for structured programs
  - Forward VC (Symbolic Exec) works for assembly
  - Here we are today …

Today's Cunning Plan

- Symbolic Execution & Forward VCGen
- Handling Exponential Blowup
  - Invariants
  - Dropping Paths
- VCGen For Exceptions (double trouble)
- VCGen For Memory (McCarthyism)
- VCGen For Structures (have a field day)
- VCGen For "Dictator For Life"

Symex Summary

- Let $x_1, \ldots, x_n$ be all the variables and $a_1, \ldots, a_n$ fresh parameters
- Let Σ_0 be the state $[x_1 := a_1, \ldots, x_n := a_n]$
- Let $\emptyset$ be the empty Inv set

- For all functions f in your program, prove:

$$\forall a_1 \ldots a_n.\ \Sigma_0(Pre_f) \Rightarrow VC(f_{entry}, \Sigma_0, \emptyset)$$

- If you start the program by invoking any f in a state that satisfies Pre_f, then the program will execute such that
  - At all "inv e" the e holds, and
  - If the function returns then $Post_f$ holds
- Can be proved w.r.t. a real interpreter (operational semantics)
- Or via a proof technique called co-induction (or, assume-guarantee)

Forward VCGen Example

- Consider the program

```
                  Precondition: x ≤ 0
Loop: inv x ≤ 6
      if x > 5 goto End
      x := x + 1
      goto Loop
End:  return      Postconditon: x = 6
```

Forward VCGen Example (2)

$\forall x.$

$x \leq 0 \Rightarrow$

$x \leq 6 \wedge$

$\forall x'.$

$(x' \leq 6 \Rightarrow$

$x' > 5 \Rightarrow x' = 6$

$\wedge$

$x' \leq 5 \Rightarrow x' + 1 \leq 6\ )$

- VC contains both proof obligations and assumptions about the control flow

#7

VCs Can Be Large

- Consider the sequence of conditionals
  (if x < 0 then x := - x); (if x ≤ 3 then x += 3)
  - With the postcondition P(x)
- The VC is

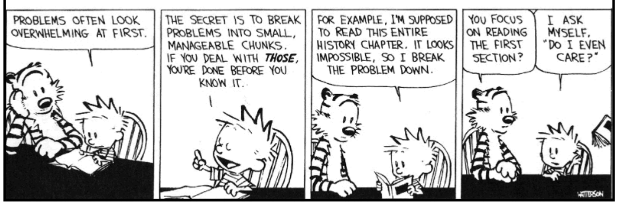
$$\begin{array}{lll} x < 0 \wedge -x \leq 3 & \Rightarrow P(-x + 3) & \wedge \\ x < 0 \wedge -x > 3 & \Rightarrow P(-x) & \wedge \\ x \geq 0 \wedge x \leq 3 & \Rightarrow P(x + 3) & \wedge \\ x \geq 0 \wedge x > 3 & \Rightarrow P(x) & \end{array}$$

- There is one conjunct for each path
  ⇒ exponential number of paths!
  - Conjuncts for infeasible paths have un-satisfiable guards!
- Try with $P(x) = x \geq 3$

#8

VCs Can Be Exponential

- VCs are exponential in the size of the source because they attempt relative completeness:
  - Perhaps the correctness of the program must be argued independently for each path
- It is unlikely that the programmer could write a program by considering an exponential number of cases
  - But possible. Any examples? Any solutions?



VCs Can Be Exponential

- VCs are exponential in the size of the source because they attempt relative completeness:
  - Perhaps the correctness of the program must be argued independently for each path
- Standard Solutions:
  - Allow invariants even in straight-line code
  - And thus do not consider all paths independently!

#10

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after c establish Inv"
  - Same semantics as c (Inv is only for VC purposes)

  VC(after c establish Inv, P) $=_{def}$

  VC(c,Inv) $\wedge\ \forall x_i.$ Inv ⇒ P

  - where x_i are the ModifiedVars(c)
- Use when c contains many paths

  after if x < 0 then x := - x establish x ≥ 0;

  if x ≤ 3 then x += 3 { P(x) }
- VC is now:

  $(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge$

  $\forall x.\ x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3) \wedge x > 3 \Rightarrow P(x))$

#11

Dropping Paths

- In absence of annotations, we can drop some paths
- VC(if E then c_1 else c_2, P) = choose one of
  - E ⇒ VC(c_1, P) ∧ ¬E ⇒ VC(c_2, P) (drop no paths)
  - E ⇒ VC(c_1, P) (drops "else" path!)
  - ¬E ⇒ VC(c_2, P) (drops "then" path!)
- We sacrifice soundness! (we are now unsound)
  - No more guarantees
  - Possibly still a good debugging aid
- Remarks:
  - A recent trend is to sacrifice soundness to increase usability (e.g., Metal, ESP, even ESC)
  - The PREfix tool considers only 50 non-cyclic paths through a function (almost at random)

#12

VCGen for Exceptions

- We extend the source language with exceptions without arguments (cf. HW2):
  - throw — throws an exception
  - try c_1 catch c_2 — executes c_2 if c_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?

#13

VCGen for Exceptions

- We extend the source language with exceptions without arguments (cf. HW2):
  - throw — throws an exception
  - try c_1 catch c_2 — executes c_2 if c_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?
- Standard Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

#14

VCGen for Exceptions (2)

- VC(c, P, Q) is a precondition that makes c either not terminate, or terminate normally with P or throw an exception with Q
- Rules

  VC(skip, P, Q) = P

  VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)

  VC(throw, P, Q) = Q

  VC(try c_1 catch c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q))

  VC(try c_1 finally c_2, P, Q) = ?

#15

VCGen Finally

- Given these:

  VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)

  VC(try c_1 catch c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q))
- Finally is somewhat like "if":

  VC(try c_1 finally c_2, P, Q) =

  VC(c_1, VC(c_2, P, Q), true) $\wedge$

  VC(c_1, true, VC(c_2, Q, Q))
- Which reduces to:

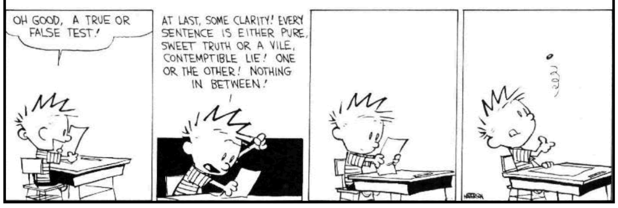
  VC(c_1, VC(c_2, P, Q), VC(c_2, Q, Q))

#16

Hoare Rules and the Heap

- When is the following Hoare triple valid?

  { A } *x := 5 { *x + *y = 10 }
- A should be "*y = 5 or x = y"
- The Hoare rule for assignment would give us:
  - [5/*x](*x + *y = 10) = 5 + *y = 10 =
  - *y = 5 (we lost one case)
- Why didn't this work?



Hoare Rules and the Heap

- When is the following Hoare triple valid?

  { A } *x := 5 { *x + *y = 10 }
- A should be "*y = 5 or x = y"
- The Hoare rule for assignment would give us:

  [5/*x](*x + *y = 10)

  = 5 + *y = 10

  = *y = 5 (we lost one case)
- Why didn't this work?

#18

Handling The Heap

- We do not yet have a way to talk about memory (the heap, pointers) in assertions
- Model the state of memory as a symbolic mapping from addresses to values:
  - If A denotes an address and M is a memory state then:
  - sel(M,A) denotes the contents of the memory cell
  - upd(M,A,V) denotes a new memory state obtained from M by writing V at address A

#19

More on Memory

- We allow variables to range over memory states
  - So we can quantify over all possible memory states
- Use the special pseudo-variable μ in assertions to refer to the current memory
- Example:

$$\forall i.\ i \geq 0 \wedge i < 5 \Rightarrow sel(\mu, A + i) > 0$$

says that entries 0..4 in array A are positive

#20

Hoare Rules: Side-Effects

- To model writes we use memory expressions
  - A memory write changes the value of memory

$$\frac{}{\{ B[upd(\mu, A, E)/\mu] \}\ {*A} := E\ \{B\}}$$

- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy Axioms, ~'67):

$$sel(upd(M, A_1, V), A_2) = \begin{cases} V & \text{if } A_1 = A_2 \\ sel(M, A_2) & \text{if } A_1 \neq A_2 \end{cases}$$

#21

Memory Aliasing

- Consider again: { A } *x := 5 { *x + *y = 10 }
- We obtain:

A = [upd(μ, x, 5)/μ] (*x + *y = 10)
  = [upd(μ, x, 5)/μ] (sel(μ, x) + sel(μ, y) = 10)
(1) = sel(upd(μ, x, 5), x) + sel(upd(μ, x, 5), y) = 10
  = 5 + sel(upd(μ, x, 5), y) = 10
  = if x = y then 5 + 5 = 10 else 5 + sel(μ, y) = 10
(2) = x = y or *y = 5

- To (1) is theorem generation
- From (1) to (2) is theorem proving

#22

Alternative Handling for Memory

- Reasoning about aliasing can be expensive (it is NP-hard)
- Sometimes completeness is sacrificed with the following (approximate) rule:

$$sel(upd(M, A_1, V), A_2) = \begin{cases} V & \text{if } A_1 = \text{(obviously) } A_2 \\ sel(M, A_2) & \text{if } A_1 \neq \text{(obviously) } A_2 \\ P & \text{otherwise (p is a fresh new parameter)} \end{cases}$$

- The meaning of "obvious" varies:
  - The addresses of two distinct globals are ≠
  - The address of a global and one of a local are ≠
- "PREfix" and GCC use such schemes

#23

VCGen Overarching Example

- Consider the program
  - Precondition: B : bool ∧ A : array(bool, L)

```
1: I := 0
   R := B
3: inv I ≥ 0 ∧ R : bool
   if I ≥  L goto 9
   assert saferd(A + I)
   T := *(A + I)
   I := I + 1
   R := T
   goto 3
9: return R
```

  - Postcondition: R : bool

#24

VCGen Overarching Example

∀A. ∀B. ∀L. ∀μ
B : bool ∧ A : array(bool, L) ⇒
0 ≥ 0 ∧ B : bool ∧
∀I. ∀R.
I ≥ 0 ∧ R : bool ⇒
I ≥ L ⇒ R : bool
∧
I < L ⇒ saferd(A + I) ∧
I + 1 ≥ 0 ∧
sel(μ, A + I) : bool

- VC contains both proof obligations and assumptions about the control flow

#25

Mutable Records - Two Models

- Let r : RECORD { f1 : T1; f2 : T2 } END
- For us, records are reference types
- Method 1: one "memory" for each record
  - One index constant for each field
  - r.f1 is sel(r,f1) and r.f1 := E is r := upd(r,f1,E)
- Method 2: one "memory" for each field
  - The record address is the index
  - r.f1 is sel(f1,r) and r.f1 := E is f1 := upd(f1,r,E)
- Only works in strongly-typed languages like Java
  - Fails in C where &r.f2 = &r + sizeof(T1) + sizeof(T2)

#26

VC as a "Semantic Checksum"

- Weakest preconditions are an expression of the program's semantics:
  - Two equivalent programs have logically equivalent WPs
  - No matter how different their syntax is!
- VC are almost as powerful

#27

VC as a "Semantic Checksum" (2)

- Consider the "assembly language" program to the right

```
x := 4
x := x == 5
  assert x : bool
x := not x
  assert x
```

- High-level type checking is not appropriate here
- The VC is: 4 == 5 : bool ∧ not (4 == 5)
- No confusion from reuse of x with different types

#28

Invariance of VC Across Optimizations

- VC is so good at abstracting syntactic details that it is syntactically preserved by many common optimizations
  - Register allocation, instruction scheduling
  - CSE, constant and copy propagation
  - Dead code elimination
- We have *identical* VCs whether or not an optimization has been performed
  - Preserves syntactic form, not just semantic meaning!
- This can be used to verify correctness of compiler optimizations (Translation Validation)

#29

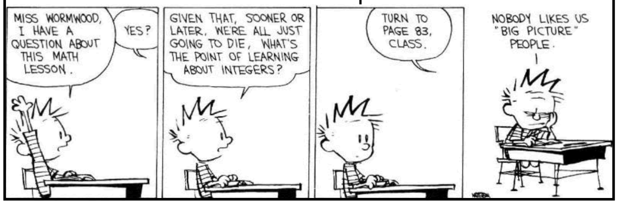
VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
  - As it goes along it performs checks (e.g. "safe to read from this memory addr", "this is a null-terminated string", "I have not already acquired this lock")
  - Some of these would actually be hard to implement
- The VC describes all of the checks to be performed
  - Along with their context (assumptions from conditionals)
  - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid ⇒ interpreter never fails
  - We enforce same level of "correctness"
  - But better (static + more powerful checks)

#30

VC Big Picture

- Verification conditions
  - Capture the semantics of code + specifications
  - Language independent
  - Can be computed backward/forward on structured/unstructured code
  - Make Axiomatic Semantics practical



Invariants Are Not Easy

- Consider the following code from QuickSort

int partition(int *a, int L_0, int H_0, int pivot) {
    int L = L_0, H = H_0;
    while(L < H) {
        while(a[L] < pivot) L ++;
        while(a[H] > pivot) H --;
        if(L < H) { swap a[L] and a[H] }
    }
    return L
}

- Consider verifying only memory safety
- What is the loop invariant for the outer loop ?

Homework

- Homework 4 Due Thursday
- Read Cousot & Cousot article
- Read Abramski article
- Project Proposal Due In One Week