Wei Hu Memorial Homework Award

- Many turned in HW3 code like this:

```haskell
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:

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

or, equivalently:

\[
R[r^*]s = \{s\} \cup \{ y | \exists x \in R[r]s \land y \in R[r^*]x \}
\]

- 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”

Forward VCGen Example

- Consider the program

  **Precondition**: \( x \leq 0 \)

  **Loop**: \( \text{inv } x \leq 6 \)
  \[
  \text{if } x > 5 \text{ goto End}
  \]
  \[
  x := x + 1
  \]
  \[
  \text{goto Loop}
  \]
  **End**: \( \text{return} \)

  **Postcondition**: \( x = 6 \)
Forward VCGen Example (2)

\[ \forall x. \]
\[ x \leq 0 \Rightarrow \]
\[ x \leq 6 \land \]
\[ \forall x'. \]
\[ (x' \leq 6 \Rightarrow \]
\[ x' > 5 \Rightarrow x' = 6 \]
\[ \land \]
\[ x' \leq 5 \Rightarrow x' + 1 \leq 6 \]

\[ \text{• VC contains both proof obligations and assumptions about the control flow} \]

VCs Can Be Large

- Consider the sequence of conditionals
  
  (if \( x < 0 \) then \( x := -x \); (if \( x \leq 3 \) then \( x += 3 \))

- With the postcondition \( P(x) \)

- The VC is

\[ x < 0 \land x > 3 \Rightarrow P(x) \]
\[ x > 0 \land x - 3 \Rightarrow P(x + 3) \]
\[ x = 0 \land x > 3 \Rightarrow P(x) \]

- There is one conjunct for each path

\[ \Rightarrow \text{exponential number of paths!} \]

- Conjects for infeasible paths have un-satisfiable guards!

- Try with \( P(x) = x \geq 3 \)

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?

• Standard Solutions:

  - Allow invariants even in straight-line code
  - And thus do not consider all paths independently!

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)

  \[ \text{VC}(c, \text{Inv}) = \text{def} \]

  \[ \text{VC}(c, \text{Inv}) \land \forall x. \text{Inv} \Rightarrow P \]

  - where \( x \) are the ModifiedVars(c)

- Use when c contains many paths

  after if \( x < 0 \) then \( x := -x \) establish \( x \geq 0 \);

  if \( x \leq 3 \) then \( x += 3 \) \( P(x) \}

- VC is now:

\[ (x < 0 \Rightarrow x = 0) \land (x \geq 0 \Rightarrow x = 0) \land \]
\[ \forall x. x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x + 3)) \land x > 3 \Rightarrow P(x) \]

Dropping Paths

- In absence of annotations, we can drop some paths

  - \( \text{VC}(\text{if } E \text{ then } c_1 \text{ else } c_2, P) = \text{choose one of} \)

  - \( E \Rightarrow \text{VC}(c_1, P) \land \neg E \Rightarrow \text{VC}(c_2, P) \) (drop no paths)

  - \( E \Rightarrow \text{VC}(c_1, P) \) (drops “else” path!)

  - \( \neg E \Rightarrow \text{VC}(c_2, P) \) (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)
VCGen for Exceptions
• We extend the source language with exceptions without arguments (cf. HW2):
  - throw throws an exception
  - try c₁ catch c₂ executes c₂ if c₁ throws
• Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?

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(c₁; c₂, P, Q) = VC(c₁, VC(c₂, P, Q), Q)
  VC(try c₁ catch c₂, P, Q) = VC(c₁, P, VC(c₂, P, Q))
  VC(try c₁ finally c₂, P, Q) = ?

VCGen Finally
• Given these:
  VC(c₁; c₂, P, Q) = VC(c₁, VC(c₂, P, Q), Q)
  VC(try c₁ catch c₂, P, Q) = VC(c₁, P, VC(c₂, P, Q))
• Finally is somewhat like “if”:
  VC(try c₁ finally c₂, P, Q) =
    VC(c₁, VC(c₂, P, Q), true) ∧
    VC(c₁, true, VC(c₂, Q, Q))
• Which reduces to:
  VC(c₁, VC(c₂, P, Q), VC(c₂, Q, Q))

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?
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:
  - \( \text{sel}(M,A) \) denotes the contents of the memory cell
  - \( \text{upd}(M,A,V) \) denotes a new memory state obtained from \( M \) by writing \( V \) at address \( A \)

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 \( \mu \) in assertions to refer to the current memory state
- Example:
  \[
  \forall i. i \geq 0 \land i < 5 \Rightarrow \text{sel}(\mu, A + i) > 0
  \]
says that entries 0..4 in array \( A \) are positive

Hoare Rules: Side-Effects

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

\[
[ B[\text{upd}(\mu, A, E)/\mu] ] \ast A := E \{ B \}
\]
- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy Axioms, ~'67):

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

Memory Aliasing

- Consider again:
  \[
  \{ A \} \ast x := 5 \{ \ast x + \ast y = 10 \}
  \]
- We obtain:
  \[
  A = \begin{cases} 
  \text{upd}(\mu, x, 5)/\mu & (\ast x + \ast y = 10) \\
  \text{upd}(\mu, x, 5)/\mu & (\text{sel}(\mu, x) + \text{sel}(\mu, y) = 10)
\end{cases}
\]

(1) = \text{sel}(\text{upd}(\mu, x, 5), x) + \text{sel}(\text{upd}(\mu, x, 5), y) = 10
  = 5 + \text{sel}(\text{upd}(\mu, x, 5), y) = 10
  = \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, y) = 10

(2) = \ast x = \ast y \lor \ast y = 5
- To (1) is theorem generation
- From (1) to (2) is theorem proving

Alternative Handling for Memory

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

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

The meaning of “obvious” varies:
- The addresses of two distinct globals are \( \neq \)
- The address of a global and one of a local are \( \neq \)
- “PREfix” and GCC use such schemes

VCGen Overarching Example

- Consider the program
  - Precondition: \( B : \text{bool} \land A : \text{array} (\text{bool}, L) \)
  1: I := 0
  2: R := B
  3: inv I \geq 0 \land R : \text{bool}
  \text{if } I \geq L \text{ goto 9}
  \text{assert saferd}(A + I)
  T := x \ast (A + I)
  4: I := I + 1
  5: R := T
  6: goto 3
  7: return R
- Postcondition: \( R : \text{bool} \)
VCGen Overarching Example

\[ \forall A. \forall B. \forall L. \forall \mu. B : \text{bool} \land A : \text{array(boolean, L)} \Rightarrow \\
0 \geq 0 \land B : \text{bool} \land \\
\forall l. \forall r, \\
l \geq 0 \land R : \text{bool} \Rightarrow \\
l \geq L \Rightarrow R : \text{bool} \land \\
l < L \Rightarrow \text{saferd}(A + l) \land \\
l + 1 \geq 0 \land \\
sel(\mu, A + l) : \text{bool} \]

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

Mutable Records - Two Models

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

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

VC as a “Semantic Checksum” (2)

- Consider the “assembly language” program to the right:
  \[
  x := 4 \\
x := x == 5 \\
\text{assert } x : \text{bool} \\
x := \text{not } x \\
\text{assert } x
  \]
- High-level type checking is not appropriate here
- The VC is: \( 4 == 5 : \text{bool} \land \text{not}(4 == 5) \)
- No confusion from reuse of \( x \) with different types

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)

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 \( \Rightarrow \) interpreter never fails
  - We enforce same level of “correctness”
  - But better (static + more powerful checks)
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
  ```c
  int partition(int *a, int L, int H, int pivot) {
    int L = L, H = H;
    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