Axiomatic Semantics

- Still need volunteers to teach
  - BDDs
  - SAT-solvers
  - SAT-based decision procedures
  - Temporal logic (and maybe other modal logics)
  - ESC/Java
- Please let me know soon

Review - Operational Semantics

- We have an imperative language with pointers and function calls
- We have defined the semantics of the language
- Operational semantics
  - Relatively simple
  - Not compositional (due to loops and recursive calls)
  - Adequate guide for an implementation

More Semantics

- There is also denotational semantics
  - Each program has a meaning in the form of a mathematical object
  - Compositional
  - More complex formalism
    - e.g. what are appropriate meanings ?
- Neither is good for arguing program correctness
  - Operational semantics requires running the code
  - Denotational semantics requires complex calculations
- We do instead: Programs → Theorems → Proofs

Programs → Theorems. Axiomatic Semantics

- Consists of:
  - A language for making assertions about programs
  - Rules for establishing when assertions hold
- Typical assertions:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with x = 0
- Partial vs. total correctness assertions
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)

Partial Correctness Assertions

- The assertions we make about programs are of the form:

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

  with the meaning that:
  - Whenever we start the execution of c in a state that satisfies A, the program either does not terminate or it terminates in a state that satisfies B
- A is called <u>precondition</u> and B is called <u>postcondition</u>
- For example:

  $\{ y \leq x \}\ z := x;\ z := z + 1\ \{ y < z \}$

  is a valid assertion
- These are called <u>Hoare triple</u> or <u>Hoare assertions</u>

Total Correctness Assertions

- {A} c {B } is a partial correctness assertion. It does not imply termination
- [A] c [B] is a total correctness assertion meaning that

  Whenever we start the execution of c in a state that satisfies A the program does terminate in a state that satisfies B

- Now let's be more formal
  - Formalize the language of assertions, A and B
  - Say when an assertion holds in a state
  - Give rules for deriving Hoare triples

Languages for Assertions

- A specification language
  - Must be easy to use and expressive (conflicting needs)
    - Most often only expression ☹
  - Syntax: how to construct assertions
  - Semantics: what assertions mean
- Typical examples
  - First-order logic
  - Temporal logic (used in protocol specification, hardware specification)
  - Special-purpose languages: Z, Larch, Java ML

State-Based Assertions

- Assertions that characterize the state of the execution
  - Recall: state = state of locals + state of memory
- Our assertions will need to be able to refer to
  - Variables
  - Contents of memory
- What are not state-based assertions
  - Variable x is live, lock L will be released
  - There is no correlation between the values of x and y

An Assertion Language

- We'll use a fragment of first-order logic first

  Formulas $P ::= A \mid \top \mid \perp \mid P_1 \wedge P_2 \mid \forall x.P \mid P_1 \Rightarrow P_2 \mid$

  Atoms $A ::= f(A_1,\ldots,A_n) \mid E_1 \leq E_2 \mid E_1 = E_2 \mid \ldots$

- We can also have an arbitrary assortment of function symbols
  - ptr(E,T) - expression E denotes a pointer to T
  - E : ptr(T) - same in a different notation
  - reachable(E_1,E_2) - list cell E_2 is reachable from E_1
  - these can be built-in or defined

Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
  - **We ignore for now references to memory**
- Notation $\rho, \sigma \models A$ to say that an assertion holds in a given state.
  - This is well-defined when ρ is defined on all variables occurring in A and σ is defined on all memory addresses referenced in A
- The $\models$ judgment is defined inductively on the structure of assertions.

Semantics of Assertions

- Formal definition (we drop σ for simplicity):

$\rho \models$ true	always
$\rho \models e_1 = e_2$	iff $\rho \vdash e_1 \Downarrow n_1$ and $\rho \vdash e_2 \Downarrow n_2$ and $n_1 = n_2$
$\rho \models e_1 \geq e_2$	iff $\rho \vdash e_1 \Downarrow n_1$ and $\rho \vdash e_2 \Downarrow n_2$ and $n_1 \geq n_2$
$\rho \models A_1 \wedge A_2$	iff $\rho \models A_1$ and $\rho \models A_2$
$\rho \models A_1 \vee A_2$	iff $\rho \models A_1$ or $\rho \models A_2$
$\rho \models A_1 \Rightarrow A_2$	iff $\rho \models A_1$ implies $\rho \models A_2$
$\rho \models \forall x.A$	iff $\forall n \in \mathbb{Z}. \rho[x:=n] \models A$
$\rho \models \exists x.A$	iff $\exists n \in \mathbb{Z}. \rho[x:=n] \models A$

Semantics of Assertions

- Now we can define formally the meaning of a partial correctness assertion
  $\models \{ A \}\ c\ \{ B \}$:
  $\forall \rho\sigma . \forall \rho'\sigma'.(\rho,\sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho',\sigma' \models B$

- … and the meaning of a total correctness assertion
  $\models [A]\ c\ [B]$ iff
  $\forall \rho\sigma.\forall \rho'\sigma'.(\rho,\sigma \models A \wedge \rho, \sigma \vdash c \Downarrow \rho', \sigma') \Rightarrow \rho',\sigma' \models B$
  $\wedge$
  $\forall \rho\sigma.\ \rho,\sigma \models A \Rightarrow \exists \rho'\sigma'.\ \rho, \sigma \vdash c \Downarrow \rho', \sigma'$

Why Isn't This Enough?

- Now we have the formal mechanism to decide when $\{A\}\ c\ \{B\}$
  - Start the program in all states that satisfies A
  - Run the program
  - Check that each final state satisfies B
- This is exhaustive testing
- Not enough
  - Can't try the program in all states satisfying the precondition
  - Can't find all final states for non-deterministic programs
  - And also it is impossible to effectively verify the truth of a $\forall x.A$ postcondition (by using the definition of validity)

Derivations as Proxies for Validity

- We define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas
- We write $\vdash A$ when we can derive (prove) the assertion A
  - We wish that $(\forall \rho\sigma.\ \rho,\sigma \models A)$ iff $\vdash A$
- We write $\vdash \{A\}\ c\ \{B\}$ when we can derive (prove) the partial correctness assertion
  - We wish that $\models \{A\}\ c\ \{B\}$ iff $\vdash \{A\}\ c\ \{B\}$

Derivation Rules for Assertions

- The derivation rules for $\vdash A$ are the usual ones from first-order logic with
- Natural deduction style axioms:

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B} \qquad \frac{\vdash [a/x]A \quad (a \text{ is fresh})}{\vdash \forall x.A} \qquad \frac{\vdash \forall x.A}{\vdash [E/x]A}$$

$$\frac{\begin{array}{c}\vdash A \\ \dots \\ \vdash B\end{array}}{\vdash A \Rightarrow B} \qquad \frac{\vdash A \Rightarrow B \quad \vdash A}{\vdash B} \qquad \frac{\vdash [E/x]A}{\vdash \exists x.A} \qquad \frac{\vdash \exists x.A \quad \begin{array}{c}\vdash [a/x]A \\ \dots \\ \vdash B\end{array}}{\vdash B}$$

Derivation Rules for Hoare Triples

- Similarly we write $\vdash \{A\}\ c\ \{ B\}$ when we can derive the triple using derivation rules
- There is one derivation rule for each command in the language
- Plus, the rule of consequence

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

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

$$\frac{}{\vdash \{A\}\ \text{skip}\ \{A\}}$$

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

$$\frac{\vdash \{A \wedge b\}\ c_1\ \{B\} \quad \vdash \{A \wedge \neg\, b\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if } b \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

Derivation Rules for Hoare Logic (II)

- The rule for while is not syntax directed
  - It needs a loop invariant

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

  - Exercise: try to see what is wrong if you make changes to the rule (e.g., drop "$\wedge$ b" in the premise, ...)

Hoare Rules: Assignment

- Example: { A } x := x + 2 {x >= 5 }. What is A?
  - A has to imply $x \geq 3$
- General rule:

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

- Surprising how simple the rule is !
- But try { A } *x = 5 { *x + *y = 10 }
- A is "*y = 5 or x = y"
- How come the rule does not work?

Example: Assignment

- Assume that x does not appear in e
  Prove that {true} x := e { x = e }
- We have

$$\frac{}{\vdash \{e = e\}\ x := e\ \{x = e\}}$$

because $[e/x](x = e) \equiv e = [e/x]e \equiv e = e$

- Assignment + consequence:

$$\frac{\vdash \text{true} \Rightarrow e = e \qquad \dfrac{}{\vdash \{e = e\}\ x := e\ \{x = e\}}}{\vdash \{\text{true}\}\ x := e\ \{x = e\}}$$

The Assignment Axiom (Cont.)

- Hoare said: "*Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic.*"

- Caveats are sometimes needed for languages with aliasing:
  - If x and y are aliased then
    { true } x := 5 { x + y = 10}
    is true

Multiple Hoare Rules

- For some constructs multiple rules are possible:

$$\frac{}{\vdash \{A\}\ x := e\ \{\exists x_0.[x_0/x]A \wedge x = [x_0/x]e\}}$$

(This was the "forward" axiom for assignment before Hoare)

$$\frac{\vdash A \wedge b \Rightarrow C \qquad \vdash \{C\}\ c\ \{A\} \qquad \vdash A \wedge \neg b \Rightarrow B}{\vdash \{A\}\ \text{while } b \text{ do } c\ \{B\}}$$

$$\frac{\vdash \{C\}\ c\ \{b \Rightarrow C \wedge \neg b \Rightarrow B\}}{\vdash \{b \Rightarrow C \wedge \neg b \Rightarrow B\}\ \text{while } b \text{ do } c\ \{B\}}$$

- Exercise: these rules can be derived from the previous ones using the consequence rules

Example: Conditional

$$\frac{\begin{array}{c} D_1 :: \vdash \{\text{true} \wedge y \leq 0\}\ x := 1\ \{x > 0\} \\ D_2 :: \vdash \{\text{true} \wedge y > 0\}\ x := y\ \{x > 0\} \end{array}}{\vdash \{\text{true}\}\ \text{if } y \leq 0 \text{ then } x := 1 \text{ else } x := y\ \{x > 0\}}$$

- D_1 is obtained by consequence and assignment

$$\frac{\begin{array}{c} \vdash \{1 > 0\}\ x := 1\ \{x > 0\} \\ \vdash \text{true} \wedge y \leq 0 \Rightarrow 1 > 0 \end{array}}{\vdash \{\text{true} \wedge y \leq 0\}\ x := 1\ \{x \geq 0\}}$$

- D_2 is also obtained by consequence and assignment

$$\frac{\begin{array}{c} \vdash \{y > 0\}\ x := y\ \{x > 0\} \\ \vdash \text{true} \wedge y > 0 \Rightarrow y > 0 \end{array}}{\vdash \{\text{true} \wedge y > 0\}\ x := y\ \{x > 0\}}$$

Example: Loop

- We want to derive that

  $\vdash \{x \leq 0\}$ while $x \leq 5$ do $x := x + 1 \{ x = 6\}$
- Use the rule for while with invariant $x \leq 6$

$$\dfrac{\dfrac{\vdash x \leq 6 \land x \leq 5 \Rightarrow x + 1 \leq 6 \qquad \overline{\vdash \{x + 1 \leq 6\}\ x := x + 1\ \{ x \leq 6 \}}}{\vdash \{x \leq 6 \land x \leq 5 \}\ x := x + 1\ \{x \leq 6\}}}{\vdash \{x \leq 6\} \text{ while } x \leq 5 \text{ do } x := x + 1 \{ x \leq 6 \land x > 5\}}$$

- Then finish-off with consequence

$$\dfrac{\begin{array}{l}\vdash x \leq 0 \Rightarrow x \leq 6 \\ \vdash x \leq 6 \land x > 5 \Rightarrow x = 6\end{array} \qquad \vdash \{x \leq 6\} \text{ while } \ldots \{ x \leq 6 \land x > 5\}}{\vdash \{x \leq 0\} \text{ while } \ldots \{x = 6\}}$$

Another Example

- Verify that

  $\vdash \{A \}$ while true do c $\{ B\}$

  holds for any A, B and c
- We must construct a derivation tree

$$\dfrac{\begin{array}{l}\vdash A \Rightarrow \text{true} \\ \vdash \text{true} \land \text{false} \Rightarrow B\end{array} \qquad \dfrac{\vdash \{\text{true} \land \text{true}\}\ c\ \{ \text{true} \}}{\{\text{true}\} \text{ while true do } c\ \{\text{true} \land \text{false}\}}}{\vdash \{A\} \text{ while true do } c\ \{ B\}}$$

- We need an additional lemma:

  $\forall c. \vdash \{ \text{true} \}\ c\ \{\text{true}\}$
  - How do you prove this one?

GCD Example

- Let c be the program:

```
while (x ≠ y) do
      if(x ≤ y)
        then y := y − x
        else x := x − y
```

- We'll derive that

  $\vdash \{ x = m \land y = n \}\ c\ \{ x = \text{gcd}(m, n) \}$

GCD Example (2)

- Crucial to select good loop invariant
  - Let the precondition Pre be

    $x = m \land y = n$
  - Let the postcondition $Post$ be

    $x = \text{gcd}(m, n)$

  We use the loop invariant

  $I \stackrel{def}{=} \text{gcd}(x, y) = \text{gcd}(m, n)$

GCD Example (3)

We first use the rule of consequence to obtain the subgoal

$\vdash \{ I \}\ c\ \{ I \land \neg(x \neq y) \}$ (1)

But we also need to prove

$\vdash Pre \Rightarrow I$ (2)

$\vdash I \land \neg(x \neq y) \Rightarrow Post$ (3)

Subgoal 2 reduces to

$x = m \land y = n \Rightarrow \text{gcd}(x, y) = \text{gcd}(m, n)$

Subgoal 3 reduces to

$\text{gcd}(x, y) = \text{gcd}(m, n) \land x = y \Rightarrow x = \text{gcd}(m, n)$

GCD Example (4)

Now we still have to derive subgoal 1:

$\vdash \{ I \}\ c\ \{ I \land \neg(x \neq y) \}$

We can apply the rule for `while` and we get the subgoal

$\vdash \{ I \land x \neq y \}\ d\ \{ I \}$ (4)

where d is the body of the loop:

```
if(x ≤ y)
  then y := y − x
  else x := x − y
```

GCD Example (5)

We can derive subgoal 4 using the rule for conditionals and we get two subgoals

$\vdash \{ I \wedge x \neq y \wedge x \leq y \}\, y := y - x\, \{ I \}$ (5)

$\vdash \{ I \wedge x \neq y \wedge x > y \}\, x := x - y\, \{ I \}$ (6)

Each of the subgoals 5 and 6 can be derived using the rule of consequence followed by assignment:

$\vdash I \wedge x \neq y \wedge x \leq y \Rightarrow \mathsf{gcd}(m, n) = \mathsf{gcd}(x, y - x)$ (7)

$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \mathsf{gcd}(m, n) = \mathsf{gcd}(x - y, y)$ (8)

GCD Example (6)

$\vdash I \wedge x \neq y \wedge x > y \Rightarrow \mathsf{gcd}(m, n) = \mathsf{gcd}(x - y, y)$

- The above can be proved by realizing that
  gcd(x,y) = gcd(x-y,y)

- Q.e.d.
- This completes the proof
- We used a lot of arithmetic
- We had to invent the loop invariants

- What about the proof for total correctness?

Hoare Rule for Function Call

- If no recursion we can inline the function call

$$\frac{f(x_1,\ldots,x_n) = C_f \in \text{Program} \quad \{ A \}\ C_f\ \{B[f/x]\}}{\vdash \{A[e_1/x_1,\ldots,e_n/x_n]\}\ x := f(e_1, \ldots, e_n)\ \{B\}}$$

- In general,
  1. each function f has a Pre_f and Post_f

$$\frac{}{\vdash \{\text{Pre}_f[e_1/x_1,\ldots,e_n/x_n]\}\ x := f(e_1, \ldots, e_n)\ \{\text{Post}_f\,[x/f]\}}$$

  2. For each function we check $\{ \text{Pre}_f \}\ C_f\ \{\text{Post}_f \}$

Axiomatic Semantics in Presence of Side-Effects

Naïve Handling of Program State

- We allow memory read in assertions: *x + *y = 5
- We try:
  { A } *x = 5 { *x + *y = 10 }
- A ought to be "*y = 5 or x = y"

- The Hoare rule would give us:
  (*x + *y = 10)[5/*x]
  = 5 + *y = 10
  = *y = 5 (we lost one case)

- How come the rule does not work?

Handling Program State

- We cannot have side-effects in assertions
  - While creating the theorem we must remove side-effects !
  - But how to do that when lacking precise aliasing information ?

- Important technique: Postpone alias analysis

- Model the state of memory as a symbolic mapping from addresses to values:
  - If E denotes an address and M a memory state then:
  - sel(M,E) denotes the contents of memory cell
  - upd(M,E,V) denotes a new memory state obtained from M by writing V at address E

More on Memory

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

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

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

Semantics of Memory Expressions

- We need a new kind of values (memory values)

Values v ::= n | a | σ

$$\frac{}{\rho, \sigma \vdash \mu \Downarrow \sigma}$$

$$\frac{\rho, \sigma \vdash E_m \Downarrow \sigma' \qquad \rho, \sigma \vdash E_2 \Downarrow a}{\rho, \sigma \vdash sel(E_m, E_2) \Downarrow \sigma'(a)}$$

$$\frac{\rho, \sigma \vdash E_m \Downarrow \sigma' \qquad \rho, \sigma \vdash E_a \Downarrow a \qquad \rho, \sigma \vdash E_v \Downarrow v}{\rho, \sigma \vdash upd(E_m, E_a, E_v) \Downarrow \sigma'[a := v]}$$

Hoare Rules: Side-Effects

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

$$\frac{}{\{\ B[upd(\mu, E_1, E_2)/\mu]\ \}\ {*E_1} := E_2\ \{B\}}$$

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

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = E_3 \\ sel(M, E_3) & \text{if } E_1 \neq E_3 \end{cases}$$

Memory Aliasing

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

$A = (*x + *y = 10)[upd(\mu, x, 5)/\mu]$
$= (sel(\mu, x) + sel(\mu, y) = 10)\ [upd(\mu, x, 5)/\mu]$
$= sel(upd(\mu, x, 5), x) + sel(upd(\mu, x, 5), y) = 10$ (*)
$= 5 + sel(upd(\mu, x, 5), y) = 10$
$=$ if $x = y$ then $5 + 5 = 10$ else $5 + sel(\mu, y) = 10$
$= x = y$ or $*y = 5$ (**)

- To (*) is theorem generation
- From (*) to (**) is theorem proving

Alternative Handling for Memory

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

$$sel(upd(M, E_1, E_2), E_3) = \begin{cases} E_2 & \text{if } E_1 = \text{(obviously) } E_3 \\ sel(M, E_3) & \text{if } E_1 \neq \text{(obviously) } E_3 \\ p & \text{otherwise (p is a fresh new parameter)} \end{cases}$$

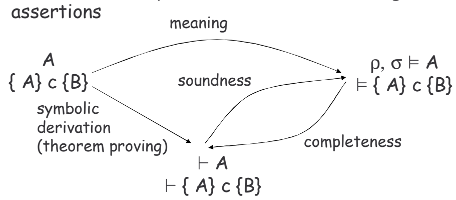
- The meaning of "obvious" varies:
  - The addresses of two distinct globals are $\neq$
  - The address of a global and one of a local are $\neq$
- PREfix and GCC use such schemes

Using Hoare Rules. Notes

- Hoare rules are mostly syntax directed
- There are three wrinkles:
  - When to apply the rule of consequence ?
  - What invariant to use for while ?
  - How do you prove the implications involved in consequence ?
- The last one is how theorem proving gets in the picture
  - This turns out to be doable !
  - The loop invariants turn out to be the hardest problem ! (Should the programmer give them? See Dijkstra.)

Where Do We Stand?

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We also have a symbolic method for deriving assertions



Soundness of Axiomatic Semantics

- Formal statement

  If $\vdash$ { A } c { B} then $\models$ { A} c { B}

  or, equivalently

  For all ρ, σ, if ρ, σ $\models$ A and D :: ρ, σ $\vdash$ c $\Downarrow$ ρ', σ' and H :: $\vdash$ { A } c { B} then ρ', σ' $\models$ B

Completeness of Axiomatic Semantics
Weakest Preconditions

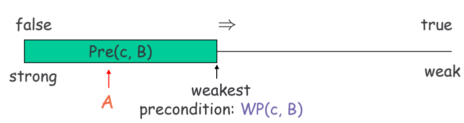
Completeness of Axiomatic Semantics

- Is it true that whenever $\models$ {A} c {B} we can also derive $\vdash$ {A} c {B} ?
- If it isn't then it means that 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

Proof Idea

- Dijkstra's idea: To verify that { A } c { B}
  a) Find out all predicates A' such that $\models$ { A'} c { B}
    - call this set Pre(c, B)
  b) Verify for one A' ∈ Pre(c, B) that A ⇒ A'
- Assertions can be ordered:



- Thus: compute WP(c, B) and <u>prove</u> A ⇒ WP(c, B)

Proof Idea (Cont.)

- Completeness of axiomatic semantics:

  If $\models$ { A } c { B } then $\vdash$ { A } c { B}
- Assuming that we can compute wp(c, B) with the following properties:
  1. wp is a precondition (according to the Hoare rules)

     $\vdash$ { wp(c, B) } c { B}
  2. wp is the weakest precondition

     If $\models$ { A } c { B} then $\models$ A ⇒ wp(c, B)

$$\frac{\vdash A \Rightarrow wp(c, B) \qquad \vdash \{wp(c, B)\}\ c\ \{B\}}{\vdash \{A\}\ c\ \{B\}}$$

- We also need that whenever $\models$ A then $\vdash$ A !

Weakest Preconditions

- Define wp(c, B) inductively on c, following Hoare rules:

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

$wp(c_1;\ c_2, B) = wp(c_1, wp(c_2, B))$

$$\frac{}{\{\ [e/x]B\ \}\ x := E\ \{B\}}$$

$wp(x := e, B) = [e/x]B$

$$\frac{\{A_1\}\ c_1\ \{B\} \qquad \{A_2\}\ c_2\ \{B\}}{\{\ E \Rightarrow A_1 \wedge \neg E \Rightarrow A_2\}\ \text{if } E \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

$wp(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \Rightarrow wp(c_1, B) \wedge \neg E \Rightarrow wp(c_2, B)$

Weakest Preconditions for Loops

- We start from the equivalence
  while b do c = if b then c; while b do c else skip
- Let w = while b do c and W = wp(w, B)
- We have that
  $W = b \Rightarrow wp(c, W) \wedge \neg b \Rightarrow B$
- But this is a recursive equation !
  - We know how to solve these using domain theory
- We need a domain for assertions

A Partial-Order for Assertions

- What is the assertion that contains least information?
  - true – does not say anything about the state
- What is an appropriate information ordering ?
  $A \sqsubseteq A'$ iff $\models A' \Rightarrow A$
- Is this partial order complete?
  - Take a chain $A_1 \sqsubseteq A_2 \sqsubseteq \ldots$
  - Let $\bigwedge A_i$ be the infinite conjunction of A_i
    $\sigma \models \bigwedge A_i$ iff for all i we have that $\sigma \models A_i$
  - Verify that $\bigwedge A_i$ is the least upper bound
- Can $\bigwedge A_i$ be expressed in our language of assertions?
  - In many cases yes (see Winskel), we'll assume yes

Weakest Precondition for WHILE

- Use the fixed-point theorem
  $F(A) = b \Rightarrow wp(c, A) \wedge \neg b \Rightarrow B$
  - Verify that F is both monotonic and continuous
- The least-fixed point (i.e. the weakest fixed point) is
  $wp(w, B) = \bigwedge F^i(\text{true})$

Weakest Preconditions (Cont.)

- Define a family of wp's
  - wp_k(while e do c, B) = weakest precondition on which the loop if it terminates in k or fewer iterations, it terminates in B
  - $wp_0 = \neg E \Rightarrow B$
  - $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$
  - …
- $wp(\text{while e do c}, B) = \bigwedge_{k \geq 0} wp_k = \text{lub}\ \{wp_k \mid k \geq 0\}$
- Is it the case that $wp_k \Rightarrow wp_{k-1}$? The opposite?
- Weakest preconditions are
  - Impossible to compute (in general)
  - Can we find something easier to compute yet sufficient ?

Weakest Precondition. Example 1

- Consider the code:
  while $x \leq 5$ do $x := x + 1$
  with postcondition $P = x \geq 7$
- What is the weakest precondition ?
- $wp(x := x + 1, A) = A[x+1/x]$
- $WP_0 = \neg (x \leq 5) \Rightarrow x \geq 7 \qquad = x \notin \{ 6 \}$
- $WP_1 = x \leq 5 \Rightarrow x + 1 \notin 6 \ \wedge WP_0 \qquad = x \notin \{ 5, 6\}$
- $WP_2 = x \leq 5 \Rightarrow x + 1 \notin \{ 5, 6 \} \wedge WP_0 = x \notin \{ 4, 5, 6 \}$
- …
- $WP = x \geq 7$

Weakest Precondition. Example 2

- Consider the code:
  while $x \geq 5$ do $x := x + 1$
  with postcondition $P = x \geq 7$
- What is the weakest precondition ?
- $wp(x := x + 1, A) = A[x+1/x]$
- $WP_0 = \neg (x \geq 5) \Rightarrow x \geq 7 \quad = x \geq 5$
- $WP_1 = x \geq 5 \Rightarrow x + 1 \geq 5 \wedge WP_0 \quad = x \geq 5$
- …
- $WP = x \geq 5$

Theorem Proving and Program Analysis (again)

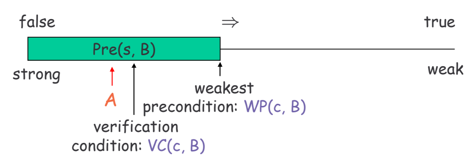
- Predicates form a lattice:
  $WP(s, B) = lub_{\Rightarrow}(Pre(s, B))$
- This is not obvious at all:
  - $lub \{P_1, P_2\} = P_1 \vee P_2$
  - $lub\ PS = \vee_{P \in PS} P$
  - But can we always write this with a finite number of $\vee$?
- Even checking implication can be quite hard
- Compare with program analysis in which lattices are of finite height and quite simple

Program Verification is Program Analysis on the lattice of first order formulas

Verification Conditions

Not Quite Weakest Preconditions

- Recall what we are trying to do:



- We shall construct a verification condition: $VC(c, B)$
  - The loops are annotated with loop invariants !
  - VC is guaranteed stronger than WP
  - But hopefully still weaker than A: $A \Rightarrow VC(c, B) \Rightarrow WP(c, B)$

Verification Conditions

- Factor out the hard work
  - Loop invariants
  - Function specifications
- Assume programs are annotated with such specs.
  - Good software engineering practice anyway
- We will assume that the new form of the while construct includes an invariant:
  $while_I$ b do c
  - The invariant formula must hold every time before b is evaluated

Invariants Are Not Easy

- Consider the following code from QuickSort

```
int partition(int *a, int L0, int H0, int pivot) {
    int L = L0, H = H0;
    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 ?

Verification Condition Generation (1)

- Mostly follows the definition of the wp function

VC(skip, B) = B
VC(c_1; c_2, B) = VC(c_1, VC(c_2, B))
VC(if b then c_1 else c_2, B) = b $\Rightarrow$ VC(c_1, B) $\neg$b $\Rightarrow$ VC(c_2, B)
VC(x := e, B) = [e/x]B
VC(let x = e in c, B) = [e/x] VC(c, B)
VC(*e_1 := e_2, B) = [upd(μ, e_1, e_2)/μ]B
VC(while b do c, B) = ?

Verification Condition Generation for WHILE

VC(while_I e do c, B) =

$$\underbrace{I}_{I \text{ holds on entry}} \wedge \underbrace{(\forall x_1 \ldots x_n.\ I \Rightarrow (e \Rightarrow VC(c, I)}_{I \text{ is preserved in an arbitrary iteration}} \wedge \underbrace{\neg e \Rightarrow B)}_{B \text{ holds when the loop terminates in an arbitrary iteration}})$$

- I 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:
  $P(0) \wedge \forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)$

Verification Conditions for Function Calls

- VC(x := f(e_1,…,e_n), P) =
  $Pre_f[E_i/x_i] \wedge \forall\mu\forall f\ Post_f \Rightarrow P[f/x]$
- We check the precondition
  - with actuals substituted for formals
- We assume that the memory and "f" are changed
- We check that function postcondition is stronger than P

Soundness and Completeness of VCGen

- We must prove that, for all c and B
  $\models$ { VC(c, B) } c { B}
- Or, equivalently
  VC(c, B) $\Rightarrow$ WP(c, B)
- Try it (the case for loops is interesting)
- Completeness is trickier
- One formulation: it is possible to choose the loop invariants such that
  WP(c, B) $\Rightarrow$ VC(c, B)
- Try it !!

Forward Verification Condition Generation

- Traditionally VC is computed backwards
  - Works well for structured code
- But it can also be computed in a forward direction
  - Works even for un-structured languages (e.g., assembly language)
  - Uses symbolic evaluation, a technique that has broad applications in program analysis
    - e.g. the PREfix tool (Intrinsa, Microsoft) works this way

Symbolic Evaluation

- Consider the language of instructions:
  x := e | f() | if e goto L | goto L | L: | return | inv e
- The "inv e" instruction is an annotation
  - Says that boolean expression e holds at that point
- Programs are sequences of instructions
- Notation: I_k is the instruction at address k

Symbolic Evaluation. Basic Intuition

- VC generation is traditionally backwards due to assignments

  $VC(x_1 := e_1; \ldots, x_n := e_n, P) =$

  $(P[e_n/x_n])\ [e_{n-1}/x_{n-1}] \ldots [e_1/x_1]$
- We can use the following rule

  $(P[e_2/x_2])[e_1/x_1] = P[e_2[e_1/x_1]/x_2, e_1/x_1]$

- Symbolic evaluation computes the substitution in a forward direction, and applies it when it reaches the postcondition

Symbolic Evaluation. The State.

- We set up a symbolic evaluation state:

$\Sigma : Var \rightarrow SymbolicExpressions$

$\Sigma(x)$ = the symbolic value of x in state Σ

$\Sigma[x{:=}e]$ = a new state in which x's value is e

We shall use states also as substitutions:

$\Sigma(e)$ - obtained from e by replacing x with $\Sigma(x)$

So far this is pretty much like the operational semantics

Symbolic Evaluation. The Invariants.

- The symbolic evaluator keeps track of the encountered invariants

- A new element of execution state: $Inv \subseteq \{1\ldots n\}$

- If $k \in Inv$ then
  - I_k is an invariant instruction that we have already executed

- Basic idea: execute an inv instruction only twice:
  - The first time it is encountered
  - And one more time around an arbitrary iteration

Symbolic Evaluation. Rules.

- Define a VC function as an interpreter:

  $VC : 1..n \times SymbolicState \times InvariantState \rightarrow Assertion$

$VC(k, \Sigma, Inv) =$

$VC(L, \Sigma, Inv)$	if I_k = goto L
$e \Rightarrow VC(L, \Sigma, Inv) \quad \wedge$ $\neg e \Rightarrow VC(k+1, \Sigma, Inv)$	if I_k = if e goto L
$VC(k+1, \Sigma[x{:=}\Sigma(e)], Inv)$	if $I_k = x := e$
$\Sigma(Post_{current\text{-}function})$	if I_k = return
$\Sigma(Pre_f) \quad \wedge$ $\forall a_1..a_m.\Sigma'(Post_f) \Rightarrow VC(k+1, \Sigma', Inv)$ (where $y_1, \ldots, y_m$ are modified by f) and $a_1, \ldots, a_m$ are fresh parameters and $\Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m]$	if $I_k = f()$

Symbolic Evaluation. Invariants.

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time
   - I_k = inv e.
   - $k \notin Inv$
   - Let $\{y_1, \ldots, y_m\}$ = the variables that could be modified on a path from the invariant back to itself
   - Let $a_1, \ldots, a_m$ be fresh new symbolic parameters

$VC(k, \Sigma, Inv) =$

$\Sigma(e) \wedge \forall a_1 \ldots a_m.\ \Sigma'(e) \Rightarrow VC(k+1, \Sigma', Inv \cup \{k\})$

with $\Sigma' = \Sigma[y_1 := a_1, \ldots, y_m := a_m]$

(like a function call)

Symbolic Evaluation. Invariants.

2. We see the invariant for the second time
   - I_k = inv E
   - $k \in Inv$

$VC(k, \Sigma, Inv) = \Sigma(e)$

(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 (Compaq SRC)
  - Sacrifice completeness for usability

Symbolic Evaluation. Putting it all together

- Let
  - $x_1, \ldots, x_n$ be all the variables and $a_1, \ldots, a_n$ fresh parameters
  - Σ_0 be the state $[x_1 := a_1, \ldots, x_n := a_n]$
  - $\emptyset$ be the empty Inv set
- For all functions f in your program, compute

  $\forall a_1 \ldots a_n.\ \Sigma_0(\mathrm{Pre}_f) \Rightarrow VC(f_{entry}, \Sigma_0, \emptyset)$
- If all of these predicates are valid then:
  - If you start the program by invoking any f in a state that satisfies Pre_f 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)
  - Proof technique called co-induction (or, assume-guarantee)

VC Generation Example

- Consider the program

  Precondition: $x \leq 0$

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

Postconditon: $x = 6$

VC Generation Example (cont.)

$$
\begin{array}{l}
\forall x. \\
\quad x \leq 0 \Rightarrow \\
\quad\quad x \leq 6 \wedge \\
\quad\quad\quad \forall x'. \\
\quad\quad\quad\quad (x' \leq 6 \Rightarrow \\
\quad\quad\quad\quad\quad x' > 5 \Rightarrow x' = 6 \\
\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad x' \leq 5 \Rightarrow x' + 1 \leq 6 \;)
\end{array}
$$

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

VC Generation Example (with memory)

- Consider the program

Precondition: $B : \mathrm{bool} \wedge A : \mathrm{array}(\mathrm{bool}, L)$

```
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
```

Postconditon: $R : \mathrm{bool}$

VC Generation Example (cont.)

$$
\begin{array}{l}
\forall A.\ \forall B.\ \forall L.\ \forall \mu \\
\quad B : \mathrm{bool} \wedge A : \mathrm{array}(\mathrm{bool}, L) \Rightarrow \\
\quad\quad 0 \geq 0 \wedge B : \mathrm{bool} \wedge \\
\quad\quad\quad \forall I.\ \forall R. \\
\quad\quad\quad\quad I \geq 0 \wedge R : \mathrm{bool} \Rightarrow \\
\quad\quad\quad\quad\quad I \geq L \Rightarrow R : \mathrm{bool} \\
\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad I < L \Rightarrow \mathrm{saferd}(A + I) \;\wedge \\
\quad\quad\quad\quad\quad\quad\quad I + 1 \geq 0 \wedge \\
\quad\quad\quad\quad\quad\quad\quad \mathrm{sel}(\mu, A + I) : \mathrm{bool}
\end{array}
$$

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

VC and Invariants

- Consider the Hoare triple:

  $\{x \leq 0\}\ \mathrm{while}_I\ x \leq 5\ \mathrm{do}\ x := x + 1\ \{x = 6\}$
- The VC for this is:

  $x \leq 0 \Rightarrow I(x) \;\wedge\; \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \wedge x \leq 5 \Rightarrow I(x+1)))$
- Requirements on the invariant:
  - Holds on entry $\forall x.\ x \leq 0 \Rightarrow I(x)$
  - Preserved by the body $\forall x.\ I(x) \wedge x \leq 5 \Rightarrow I(x+1)$
  - Useful $\forall x.\ I(x) \wedge x > 5 \Rightarrow x = 6$
- Check that $I(x) = x \leq 6$ satisfies all constraints

VC 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
  $x < 0 \wedge -x \leq 3 \Rightarrow P(-x + 3) \wedge$
  $x < 0 \wedge -x > 3 \Rightarrow P(-x) \wedge$
  $x \geq 0 \wedge x \leq 3 \Rightarrow P(x + 3) \wedge$
  $x \geq 0 \wedge x > 3 \Rightarrow P(x)$
- There is one conjunct for each path
  => exponential number of paths !
  - Conjuncts for non-feasible paths have un-satisfiable guard !
- Try with $P(x) = x \geq 3$

VC Can Be Large (2)

- VCs are exponential in the size of the source because they attempt relative completeness:
  - To handle the case then the correctness of the program must be argued independently for each path
- Remark:
  - It is unlikely that the programmer could write a program by considering an exponential number of cases
  - But possible. Any examples?
- Solutions:
  - Allow invariants even in straight-line code
  - Thus do not consider all paths independently !

Invariants in Straight-Line Code

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

  $VC(\text{after } c \text{ establish } I, P) =_{def} VC(c,I) \wedge \forall x_i.\ I \Rightarrow P$
    - where x_i 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 now is (for $P(x) = x \geq 3$)
  $(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge$
  $\forall x.\ x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3)) \wedge x > 3 \Rightarrow P(x))$

Dropping Paths

- In absence of annotations drop some paths
- VC(if E then c_1 else c_2, P) = choose one of
  - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$
  - $E \Rightarrow VC(c_1, P)$
  - $\neg E \Rightarrow VC(c_2, P)$
- We sacrifice soundness !
  - No more guarantees but possibly still a good debugging aid
- Remarks:
  - A recent trend is to sacrifice soundness to increase usability
  - 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:
  - throw throws an exception
  - try c_1 handle c_2 executes c_2 if c_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?
- Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

VCGen for Exceptions (2)

- Define: 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 handle c_2, P, Q) = VC(c_1, P, VC(c_2, Q, Q))
  VC(try c_1 finally c_2, P, Q) = ?

VCGen for Exceptions (3)

- What if we have exceptions with arguments
- Introduce global variable ex for the exception argument
- The exceptional postcondition can now refer to ex
- Remember that we must add an exceptional postcondition to functions also
  - Like the THROWS clause in Java or Modula-3

Additional Language Extensions

- wrong
  - Semantics: terminate execution with an error
  - VC(wrong, P) = false
- fail
  - Semantics: terminate execution with a benign error
  - Used to denote error conditions that are allowed (e.g. index out of bounds)
  - VC(fail, P) = true
- assert E = if E then skip else wrong
  - Used to specify verification goals
- assume E = if E then skip else fail
  - Used to specify assumptions (e.g. about inputs)

Additional Language Extensions (2)

- Undefined variables
  - $VC(\text{let } x \text{ in } c, P) = \forall x.\ VC(c, P)$

Call-by-Reference

- Let $f(\text{VAR } x, y)\ \{\ x \leftarrow 5;\ f \leftarrow x + y\ \}$
- with
  - $Pre_f = y \geq 10$
  - $Post_f = f \geq 15$
- VC of the body is $\forall xy.\ y \geq 10 \Rightarrow 5 + y \geq 15$
  - Which is provable !
- Let $\{x = 10\}\ y \leftarrow f(x,x)\ \{y \geq 15\}$
- Its VC is $x = 10 \Rightarrow (x \geq 10 \wedge (\forall y.\ y \geq 15 \Rightarrow y \geq 15)$
  - Also provable
  - But if we run the code, y is 10 at the end !
  - What's going on?

Call-by-Reference

- Axiomatic semantics can model only call-by-value
- Not a big problem, just have to model call-by-reference with call-by-value
  - Pass pointers instead
  - Use the sel/upd axioms instead of assignment axioms

Arrays

- Arrays can be modeled like we did pointers
- In a safe language (no & and no pointer arithmetic)
  - We can have a store value for each array
  - Instead of a unique store μ
- $A[E]$ is considered as $sel(A,E)$
- $A[E_1] \leftarrow E_2$ is considered as $A \leftarrow upd(A, E_1, E_2)$
- What is the advantage over $sel(\mu, A+E)$?

Mutable Records - Two Models

- Let r : RECORD f1 : T1; f2 : T2 END
- Records are reference types
- Method 1
  - One "memory" for each record
  - One index constant for each field. We postulate f1 ≠ f2
  - 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)

VC Generation for Assembly Language

- Issues when extending this to real assembly language
- A fixed set of variables
  - Machine registers
- Handling of stack slots
  - Option 1: as memory locations (too inefficient)
  - Option 2: as additional registers (reverse spilling)
    - Careful with renaming at function calls and returns
    - Does not work in presence of address-of operator
- Two's complement arithmetic
  - Careful to distinguish between + and ⊕
- See Necula-Ph.D. thesis for x86 and Alpha example

VC as a "Semantic CheckSum"

- Weakest preconditions are an expression of the program semantics
  - Two equivalent programs have (logically) equivalent WP
- VC are almost the same
  - Except for the loop invariants and function specifications
- In fact, VC abstract over some syntactic details
  - Order of unrelated expressions
  - Names of variables
  - Common subexpressions

VC for Type Checking Low Level Code

- Type checking low level code is hard because
  - We cannot have variable declarations
    - Each instance of a register use can have different types
  - The type of a register depends on the data flow
    - Varies depending on how a use was reached
- Consider the program and its VC

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

4 == 5 : bool ∧ ¬ (4 == 5)

- No live logical variables
- Not confused by reuse of x

Invariance of VC Through Optimizations

- In fact, VC is so good at abstracting syntactic details that it is invariant through many common optimiz.
  - Preserves syntactic form, not just semantic meaning !
- We'll take a look at some optimizations and see how VC changes/doesn't change
- This can be used to verify correctness of compiler optimizations (Translation Validation)

Dead-Code Elimination

- VC is insensitive to dead-code
- The symbolic evaluator won't even look at it !

Before		After	
Code	VC	Code	VC
1 $i = e_1$ 2 **go to** L 3 $i = e_2$ 4 L: ...P...	$[^{e_1}/_{i}]P$	$i = e_1$ L: ...P...	$[^{e_1}/_{i}]P$

Common-Subexpression Elimination

Before		After	
Code	VC	Code	VC
$i = e_1$ $j = e_1$ $...P...$	$[^{e_1}/_i, ^{e_1}/_j]P$	$i = e_1$ $j = i$ $...P...$	$[^{e_1}/_i, ^{e_1}/_j]P$

- CSE does not change the VC

Copy Propagation

Before		After	
Code	VC	Code	VC
$i = e_1$ $j = i$ $...P...$	$[^{e_1}/_i, ^{e_1}/_j]P$	$i = e_1$ $...[^{i}/_j]P...$	$[^{e_1}/_i, ^{e_1}/_j]P$

- Does not change the VC
- Just like CSE

Instruction Scheduling

Before		After	
Code	VC	Code	VC
$i = e_1$ $j = e_2$ $...P...$	$[^{e_1}/_i, ^{e_2}/_j]P$	$j = e_2$ $i = e_1$ $...P...$	$[^{e_1}/_i, ^{e_2}/_j]P$

- Instruction scheduling does not change VC

Register Allocation

Before		After	
Code	VC	Code	VC
$i = e$ $j = e'$ $...P...$	$\left[^{[^{e}/_i]e'}/_j\right] P$	$\mathbf{r}_j = [^{\mathbf{r}_i}/_i]e$ $\mathbf{r}_j = [^{\mathbf{r}_j}/_i]e'$ $...[^{\mathbf{r}_j}/_j]P...$	$[^{\mathbf{r}_i}/_i](\left[^{[^{e}/_i]e'}/_j\right] P)$

- Does not change VC
  - Final logical variables are quantified over
- Even spilling can be accomodated
  - By giving register names to spill slots on the stack

Loop Invariant Hoisting

Before		After	
Code	VC	Code	VC
L_1: **if** C **go to** L_2 $i = e_1$ $...P \wedge I...$ L_2: $...P'...$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(\neg C \supset [^{e_1}/_i]I) \wedge$ $(\neg C \supset [^{e_1}/_i]P))$	$j = e_1$ L_1: **if** C **go to** L_2 $i = j$ $...P \wedge I...$ L_2: $...P'...$	$I \wedge$ $\forall i. I \supset ((C \supset P') \wedge$ $(\neg C \supset [^{e_1}/_i]I) \wedge$ $(\neg C \supset [^{e_1}/_i]P)$

- VC is not changed
- For same reasons as with CSE

Redundant Conditional Elimination

Before		After	
Code	VC	Code	VC
if $\neg C_1$ **go to** L_2 **if** $\neg C_2$ **go to** L_2 ... **if** $\neg C_n$ **go to** L_2 **if** C **go to** L_1 $...P''...$ L_1: $...P'...$ L_2: $...P...$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset ...$ $C_n \supset (C \supset P') \wedge$ $(\neg C \supset P'')$	**if** $\neg C_1$ **go to** L_2 **if** $\neg C_2$ **go to** L_2 ... **if** $\neg C_n$ **go to** L_2 **inv** $C, \{\}$ $...P'...$ L_2: $...P...$	$\neg C_1 \supset P \wedge$ $C_1 \supset (\neg C_2 \supset P) \wedge$ $(C_2 \supset ...$ $C_n \supset C \wedge$ $C \supset P',$

- VC changes syntactically
- But not semantically (since $C_1 \wedge C_2 \wedge ... \wedge C_n \Rightarrow C$)
  - We do have to prove something here
- Same for array-bounds checking elimination

VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
  - As it goes along it performs checks (e.g. saferd, validString)
  - 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)

Conclusion

- Verification conditions
  - Are an expression of semantics and specifications
  - Completely independent of a language
  - Can be computed backward/forward on structured/unstructured code
  - Can be computed on high-level/low-level code
- Using symbolic evaluation we can hope to check correctness of compiler optimizations
  - See "Translation Validation for an Optimizing Compiler" off the class web page
- Next time: We start proving VC predicates