Inference Rules for Hoare Triples

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

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

Inference Rules for Hoare Logic

- One rule for each syntactic construct:

$$\vdash \{A\}\ \texttt{skip}\ \{A\} \qquad\qquad \vdash \{A[e/x]\}\ x{:=}e\ \{A\}$$

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

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

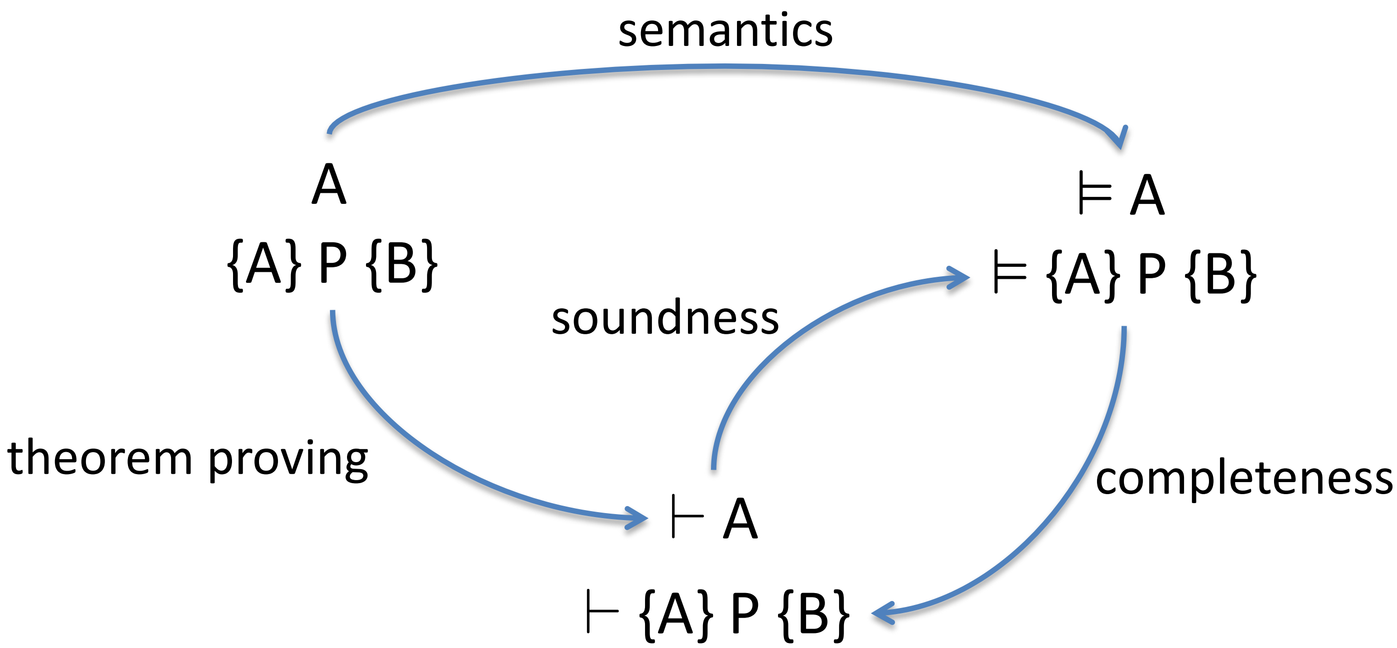
$$\frac{\vdash \{I \wedge b\}\ c\ \{I\}}{\vdash \{I\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{I \wedge \neg b\}}$$

Using Hoare Rules

- Hoare rules are mostly syntax directed
- There are three obstacles to automation of Hoare logic proofs:
  - When to apply the rule of consequence?
  - What invariant to use for `while`?
  - How do you prove the implications involved in the rule of 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?

Hoare Logic: Summary

- 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.



Verification Conditions

- **Goal**: given a Hoare triple {A} P {B}, derive a single formula *VC*(A,P,B) such that $\models$ *VC*(A,P,B) iff $\models$ {A} P {B}
- *VC*(A,P,B) is called a verification condition.
- Verification condition generation factors out the hard work
  - Finding loop invariants
  - Finding function specifications
- Assume programs are annotated with such specifications
  - We will assume that the new form of the `while` construct includes an invariant:
    `{I} while` *b* `do` c
  - The invariant formula `I` must hold every time before *b* is evaluated.

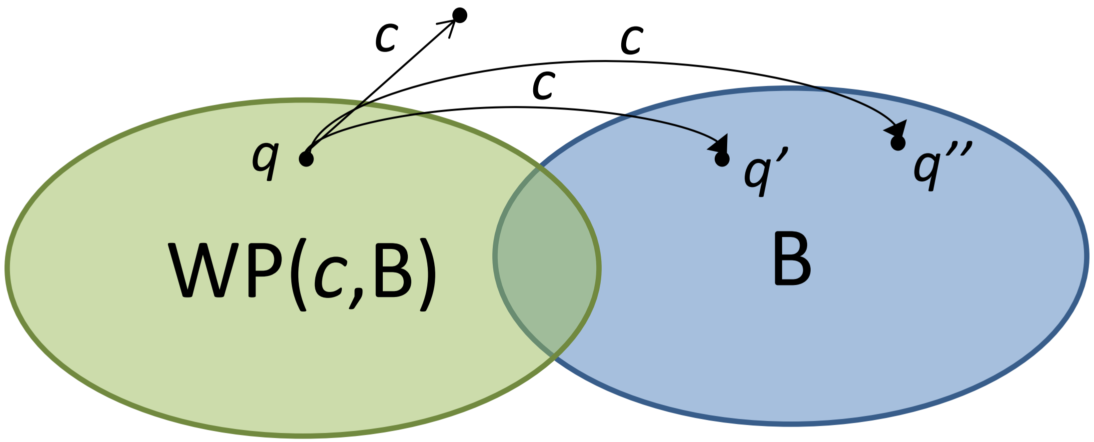
Verification Condition Generation

- Idea for VC generation: propagate the post-condition backwards through the program:
  - From {A} P {B}
  - generate $A \Rightarrow F(P, B)$
- This backwards propagation F(P, B) can be formalized in terms of weakest preconditions.

Weakest Preconditions

- The weakest (conservative) precondition WP(c,B) for a formula B and a program c is a formula that holds for any state q such that
  - c terminates normally when executed starting from b
  - the final state obtained after executing c from q satisfies B.

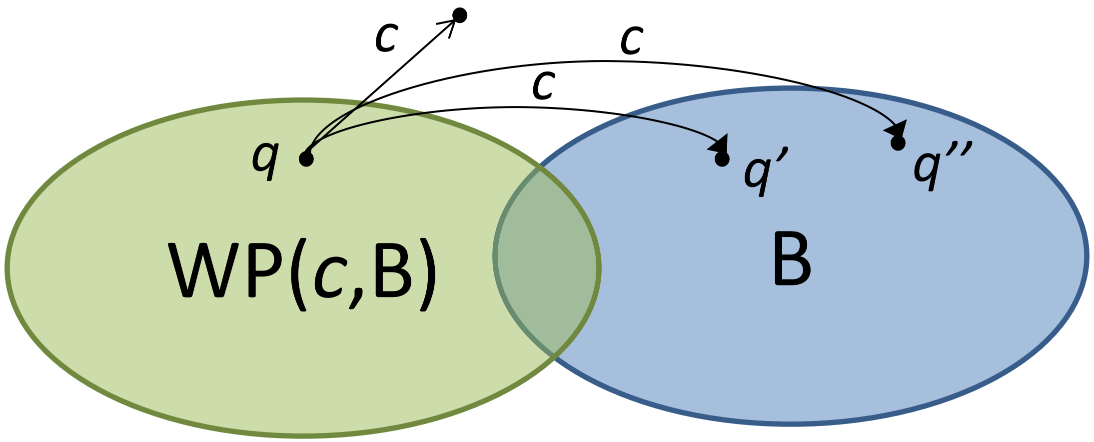
  $q \vDash \mathrm{WP}(c,\mathrm{B})$ iff $\exists q' \in Q.\ q \xrightarrow{c} q' \wedge q' \vDash \mathrm{B}$
  $\forall q' \in Q.\ q \xrightarrow{c} q' \Rightarrow q' \vDash \mathrm{B}$



Weakest Preconditions

- The weakest ~~(conservative)~~ *liberal* precondition WLP(c,B) for a formula B and a program c is a formula that holds for any state q such that
  - ~~c terminates normally when executed starting from b~~
  - the final state obtained after executing c from q satisfies B.

  $q \vDash \mathrm{WLP}(c,\mathrm{B})$ iff $\cancel{\exists q' \in Q.\ q \xrightarrow{c} q' \land q' \vDash \mathrm{B}}$
  $\forall q' \in Q.\ q \xrightarrow{c} q' \Rightarrow q' \vDash \mathrm{B}$



Computing Weakest Preconditions

- Idea: compute WP(*P*,B) recursively according to the structure of the program *P.*
- Problem: How to deal with loops?
- Solution:
  - introduce loop-free intermediate language
  - translate program to simplified program in the intermediate language
  - then compute WP on simplified program.

Loop-Free Guarded Commands

- Introduce loop-free guarded commands as an intermediate representation of the verification condition
- c ::= assume b — block if b does not hold
  - | assert b — fail if b does not hold
  - | havoc x — nondet. assignment
  - | $c_1 ; c_2$ — sequencing
  - | $c_1 \square c_2$ — nondet. choice

From Programs to Guarded Commands

- GC(`skip`) =

- GC(x := e) =

- GC(c_1 ; c_2) =

- GC(`if` b `then` c_1 `else` c_2) =

- GC({I} `while` b `do` c) =

From Programs to Guarded Commands

- GC(`skip`) =

  assume true

- GC(x := e) =

  havoc *tmp*; assume *tmp* = x; where *tmp* is fresh

  havoc x; assume ($x = e[tmp/x]$)

- GC(c_1 ; c_2) =

  GC(c_1) ; GC(c_2)

- GC(`if` b `then` c_1 `else` c_2) =

  (assume b; GC(c_1)) $\square$ (assume $\neg b$; GC(c_2))

- GC({I} `while` b do c) = ?

Guarded Commands for Loops

- GC({I} **while** b **do** c) =

  assert I;

  havoc x_1; …; havoc x_n;

  assume I;

  (assume b; GC(c); assert I; assume false) $\square$

  assume $\neg b$

where x_1, …, x_n are the variables assigned in c

Computing Weakest Preconditions

- WP(assume b, B) =
- WP(assert b, B) =
- WP(havoc x, B) =
- WP(c_1;c_2, B) =
- WP($c_1 \square c_2$,B) =

Computing Weakest Preconditions

- WP(assume b, B) = $b \Rightarrow$ B
- WP(assert b, B) = $b \wedge$ B
- WP(havoc x, B) = B[a/x] (*a* fresh in B)
- WP($c_1;c_2$, B) = WP(c_1, WP(c_2, B))
- WP($c_1 \square c_2$,B) = WP(c_1, B) $\wedge$ WP(c_2, B)

Computing Weakest Preconditions

- WLP(assume b, B) = $b \Rightarrow$ B
- WLP(assert b, B) = $b \Rightarrow$ B
- WLP(havoc x, B) = B[a/x] (a fresh in B)
- WLP($c_1;c_2$, B) = WP(c_1, WP(c_2, B))
- WLP($c_1 \square c_2$,B) = WP(c_1, B) $\wedge$ WP(c_2, B)

Putting Everything Together

- Given a Hoare triple $H \equiv \{A\}\ P\ \{B\}$
- Compute c_H = assume A; GC(P); assert B
- Compute $VC_H = WP(c_H, \text{true})$
- Infer $\vdash VC_H$ using a theorem prover.

Example: VC Generation

$\{n \geq 0\}$

$p := 0;$

$x := 0;$

$\{p = x * m \land x \leq n\}$

$\texttt{while}\ x < n\ \texttt{do}$

$\quad x := x + 1;$

$\quad p := p + m$

$\{p = n * m\}$

Example: VC Generation

- Computing the guarded command

```
assume n ≥ 0;
GC(   p := 0;
      x := 0;
      {p = x * m ∧ x ≤ n}
      while x < n do
          x := x + 1;
          p := p + m  );
assert p = n * m
```

Example: VC Generation

- Computing the guarded command

assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

GC($x := 0$;

$\{p = x * m \wedge x \leq n\}$

while $x < n$ do

$x := x + 1$;

$p := p + m$);

assert $p = n * m$

Example: VC Generation

- Computing the guarded command

assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

GC($\{p = x * m \wedge x \leq n\}$

while $x < n$ do

$x := x + 1$;

$p := p + m$);

assert $p = n * m$

Example: VC Generation

- Computing the guarded command

assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$;

havoc x; havoc p; assume $p = x * m \wedge x \leq n$;

(assume $x < n$;
GC($x := x + 1$;
$p := p + m$);
assert $p = x * m \wedge x \leq n$; assume false)

□ assume $x \geq n$;

assert $p = n * m$

Example: VC Generation

- Computing the guarded command

assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \land x \leq n$;

havoc x; havoc p; assume $p = x * m \land x \leq n$;

(assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;
assume $p_1 = p$; havoc p; assume $p = p_1 + m$;
assert $p = x * m \land x \leq n$; assume false)

$\square$ assume $x \geq n$;

assert $p = n * m$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;
assume $p_0 = p$; havoc p; assume $p = 0$;
assume $x_0 = x$; havoc x; assume $x = 0$;
assert $p = x * m \wedge x \leq n$;
havoc x; havoc p; assume $p = x * m \wedge x \leq n$;
(assume $x < n$;
assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;
assume $p_1 = p$; havoc p; assume $p = p_1 + m$;
assert $p = x * m \wedge x \leq n$; assert false)
$\square$ assume $x \geq n$;
assert $p = n * m$, true)

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$;

(assume x < n;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \wedge x \leq n$; assume false)

$\square$ assume $x \geq n$, $p = n * m$)

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;
assume $p_1 = p$; havoc p; assume $p = p_1 + m$;
assert $p = x * m \wedge x \leq n$; assume false)) $\Rightarrow$
$p = n * m)$

$\wedge$ $(x \geq n \Rightarrow p = n * m)))$

Example: VC Generation

- Computing the weakest precondition

WP(assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume x < n;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \wedge x \leq n$), false $\Rightarrow p = n * m$)

$\wedge$ (x $\geq$ n $\Rightarrow p = n * m$)))

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume x < n;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p; assume $p = p_1 + m$;

assert $p = x * m \wedge x \leq n$), true)

$\wedge$ (x $\geq$ n $\Rightarrow p = n * m$)))

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume x < n;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$;

assume $p_1 = p$; havoc p),

$p = p_1 + m \Rightarrow p = x * m \wedge x \leq n)$

$\wedge$ ($x \geq n \Rightarrow p = n * m$)))

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume $x < n$;

assume $x_1 = x$; havoc x; assume $x = x_1 + 1$),

$p_1 = p \wedge pa_1 = p_1 + m \Rightarrow pa_1 = x * m \wedge x \leq n$)

$\wedge (x \geq n \Rightarrow p = n * m)))$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

(WP((assume x < n),

$(x_1 = x \wedge xa_1 = x_1 + 1 \wedge$
$p_1 = p \wedge pa_1 = p_1 + m) \Rightarrow pa_1 = x * m \wedge x \leq n)$

$\wedge$ (x $\geq$ n $\Rightarrow p = n * m)))$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \wedge x \leq n$,

WP(havoc x; havoc p; assume $p = x * m \wedge x \leq n$,

$(x < \mathrm{n} \wedge$

$x_1 = x \wedge xa_1 = x_1 + 1 \wedge$

$p_1 = p \wedge pa_1 = p_1 + m) \Rightarrow pa_1 = x * m \wedge x \leq n)$

$\wedge\ (\mathrm{x} \geq \mathrm{n} \Rightarrow p = n * m)))$

Example: VC Generation

- Computing the weakest precondition

WP (assume $n \geq 0$;

assume $p_0 = p$; havoc p; assume $p = 0$;

assume $x_0 = x$; havoc x; assume $x = 0$;

assert $p = x * m \land x \leq n$,

$(pa_2 = xa_2 * m \land xa_2 \leq n \land$

$xa_2 < \mathrm{n} \land$

$x_1 = xa_2 \land xa_1 = x_1 + 1 \land$

$p_1 = pa_2 \land pa_1 = p_1 + m) \Rightarrow pa_1 = xa_2 * m \land xa_2 \leq n)$

$\land (\mathrm{x} \geq \mathrm{n} \Rightarrow p = n * m)))$

Example: VC Generation

- Computing the weakest precondition

$$
\begin{aligned}
&\mathrm{n} \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \Rightarrow \\
&\qquad pa_3 = xa_3 * m \wedge xa_3 \leq n \wedge \\
&\qquad (pa_2 = xa_2 * m \wedge xa_2 \leq n \wedge \\
&\qquad xa_2 < \mathrm{n} \wedge \\
&\qquad x_1 = xa_2 \wedge xa_1 = x_1 + 1 \wedge \\
&\qquad p_1 = pa_2 \wedge pa_1 = p_1 + m) \Rightarrow pa_1 = xa_2 * m \wedge xa_2 \leq n) \\
&\qquad \wedge (\mathrm{x} \geq \mathrm{n} \Rightarrow p = n * m)))
\end{aligned}
$$

Example: VC Generation

- The resulting VC is equivalent to the conjunction of the following implications

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \Rightarrow$$
$$pa_3 = xa_3 * m \wedge xa_3 \leq n$$

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \wedge pa_2 = xa_2 * m \wedge xa_2 \leq n \Rightarrow$$
$$xa_2 \geq n \Rightarrow pa_2 = n * m$$

$$n \geq 0 \wedge p_0 = p \wedge pa_3 = 0 \wedge x_0 = x \wedge xa_3 = 0 \wedge pa_2 = xa_2 * m \wedge xa_2 < n \wedge x_1 = xa_2 \wedge xa_1 = x_1 + 1 \wedge p_1 = pa_2 \wedge pa_1 = p_1 + m \Rightarrow$$
$$pa_1 = xa_1 * m \wedge xa_1 \leq n$$

Example: VC Generation

- simplifying the constraints yields

$$\mathrm{n} \geq 0 \Rightarrow 0 = 0 * m \wedge 0 \leq n$$

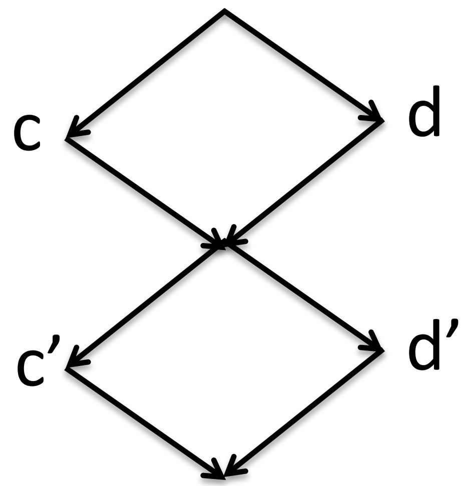
$$xa_2 \leq n \wedge \mathrm{xa}_2 \geq \mathrm{n} \Rightarrow xa_2 * m = n * m$$

$$xa_2 < \mathrm{n} \Rightarrow xa_2 * m + m = (xa_2 + 1) * m \wedge xa_2 + 1 \leq n$$

- all of these implications are valid, which proves that the original Hoare triple was valid, too.

The Diamond Problem

assume A;
c □ *d*;
c′ □ *d′*;
assert B



$A \Rightarrow$ WP (c, WP(c’, B) $\wedge$ WP(d’, B)) $\wedge$
WP (d, WP(c’, B) $\wedge$ WP(d’, B))

- Number of paths through the program can be exponential in the size of the program.
- Size of weakest precondition can be exponential in the size of the program.

Avoiding the Exponential Explosion

Ideas?

Avoiding the Exponential Explosion

Ideas?

1. Introduce propositional variables that stand for repeated subformulas
   – yields formulas that are linear in the program size
   – burden has now shifted to the theorem prover (often still exponential behavior)
2. Remove redundancies from the VCs entirely
   – yields formula that are quadratic in the program size
   – usually more efficient once theorem prover is factored in

Removing Redundancy from VCs

- The following equivalence holds for arbitrary programs c and formulas B:

$$\mathrm{WP}(c, B) \equiv \mathrm{WP}(c, \mathit{true}) \wedge \mathrm{WLP}(c, B)$$

- We got rid of B below WP. Can we also get rid of B below WLP?

Passive Guarded Commands

• c ::= assume b	block if b does not hold
| assert b	fail if b does not hold
~~| havoc x~~	~~nondet. assignment~~
| c_1 ; c_2	sequencing
| $c_1 \Box c_2$	nondet. choice

Passive programs are also often said to be in static single assignment (SSA) form. For loop-free programs, the SSA form can be obtained using a simple program transformation.

Removing Redundancy from VCs

- The following equivalence holds for arbitrary programs c and formulas B:

$$\mathrm{WP}(c, B) \equiv \mathrm{WP}(c, \mathrm{true}) \wedge \mathrm{WLP}(c, B)$$

- For passive programs c we also have:

$$\mathrm{WLP}(c, B) \equiv \mathrm{WLP}(c, \mathrm{false}) \vee B$$

Removing Redundancy from VCs

- Using the equations from the previous slides, we can compute WP for passive programs recursively according to the following equation:

$$\mathrm{WP}(c, B) \equiv \mathrm{WP}(c, \mathrm{true}) \wedge (\mathrm{WLP}(c, \mathrm{false}) \vee B)$$

- WP(c, B) is now quadratic in the size of c
- There is no duplication of B for each path in c

Translating Method Calls to GCs

```
method m (p₁: T_1, ..., pₖ: Tₖ) returns (r: T)
    requires P
    modifies x₁, ..., xₙ
    ensures Q
```

A method call

```
  y := y₀.m(y₁, ..., yₖ);
```

is desugared into the guarded command

assert $P[y_0/\mathtt{this}, y_1/p_1, \ldots, y_k/p_k]$;

havoc x_1; …, havoc x_n; havoc y;

assume $Q[y_0/\mathtt{this}, y_1/p_1, \ldots, y_k/p_k, y/r]$

Handling More Complex Program State

- When is the following Hoare triple valid?

  $\{A\}\ x.f := 5\ \{x.f + y.f = 10\}$

- A ought to imply "$y.f = 5 \vee x = y$"
- The IMP Hoare rule for assignment would give us:

  $(x.f + y.f = 10)\ [5/x.f]$

  $\equiv 5 + y.f = 10$

  $\equiv y.f = 5$ (we lost one case)

- How come the rule does not work?

Modeling the Heap

- We cannot have side-effects in assertions
  - While generating the VC we must remove side-effects!
  - But how to do that when lacking precise aliasing information?
- Simple solution: postpone alias analysis to the theorem prover
- Model the state of the heap as a symbolic mapping from addresses to values:
  - If e denotes an address and h a heap state then:
  - sel(h,e) denotes the contents of the memory cell
  - upd(h,e,v) denotes a new heap state obtained from h by writing v at address e

Heap Models

- We allow variables to range over heap states
  - So we can quantify over all possible heap states.
- Model 1
  - One "heap" for each object
  - One index constant for each field (we postulate $f_1 \neq f_2$).
  - r.f is sel(r,f) and r.f := e is r := upd(r, f, e)
- Model 2 (Burstall-Bornat)
  - One "heap" for each field
  - The object address is the index
  - r.f is sel(f,r) and r.f := e is f := upd(f,r,e)

Hoare Rule for Field Assignments

- To model assignments correctly, we use heap expressions
  - A field assignment changes the heap of that field

$$\{\ B[upd(f, e_1, e_2)/f]\ \}\ e_1.f := e_2\ \{B\}$$

- Simple technique:
  - model heap as a semantic object
  - defer reasoning about heap expressions to the theorem prover with inference rules such as (McCarthy):

$$sel(upd(h, e_1, e_2), e_3) = \begin{cases} e_2 & \text{if } e_1 = e_3 \\ sel(h, e_3) & \text{if } e_1 \neq e_3 \end{cases}$$

Example: Hoare Rule for Field Writes

- Consider again: { A } x.f := 5 { x.f + y.f = 10 }
- We obtain:
  A $\equiv$ (x.f + y.f = 10)[upd(f, x, 5)/f]
  $\equiv$ (sel(f, x) + sel(f, y) = 10)[upd(f, x, 5)/f]
  $\equiv$ sel(upd(f x 5) x) + sel(upd(f x 5) y) = 10
  $\equiv$ 5 + sel(upd(f, x, 5), y) = 10
  $\equiv$ if x = y then 5 + 5 = 10 else 5 + sel(f, y) = 10
  $\equiv$ x = y $\vee$ y.f = 5
- Theorem generation.
- Theorem proving.

Modeling new Statements

- Introduce a set-valued variable Alloc that denotes the set of all objects that are currently allocated

- Translate x := `new T.Init()` to

  havoc x;
  assume $x \notin Alloc$;
  assume Type(x) = T;
  assume $x \neq null$;
  Alloc := Alloc $\cup$ {x};
  Translation of call to constructor T.Init()