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

\[
\frac{\vdash A' \Rightarrow A \quad \vdash \{A\} c \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\} c \{B'\}}
\]
Inference Rules for Hoare Logic

- One rule for each syntactic construct:

\[ \vdash \{A\} \text{skip} \{A\} \quad \vdash \{A[e/x]\} x:=e \{A\} \]

\[ \vdash \{A\} c_1 \{B\} \quad \vdash \{B\} c_2 \{C\} \]

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

\[ \vdash \{A \land b\} \ c_1 \ \{B\} \quad \vdash \{A \land \neg b\} \ c_2 \ \{B\} \]

\[ \vdash \{A\} \text{if } b \text{ then } c_1 \text{ else } c_2 \ \{B\} \]

\[ \vdash \{I \land b\} \ c \ \{I\} \]

\[ \frac{}{\vdash \{I\} \text{while } b \text{ do } c \ \{I \land \neg b\}} \]
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:
    \( \{I\} \text{while } b \text{ do } c \)
  - 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 $A \implies F(P, B)$
- 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 $B$.

$$q \models WP(c, B) \iff \exists q' \in Q. q \xrightarrow{c} q' \land q' \models B$$

$$\forall q' \in Q. q \xrightarrow{c} q' \Rightarrow q' \models B$$
The weakest (conservative) 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 $q$.
- The final state obtained after executing $c$ from $q$ satisfies $B$.

$q \models WLP(c, B)$ if and only if

\[\forall q' \in Q. \quad q \xrightarrow{c} q' \land q' \models B\]

\[\exists q' \in Q. \quad q \xrightarrow{c} q' \Rightarrow q' \models B\]
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

• $c ::= \begin{align*}
\text{assume } & b & \text{block if } b \text{ does not hold} \\
| \text{assert } & b & \text{fail if } b \text{ does not hold} \\
| \text{havoc } & x & \text{nondet. assignment} \\
| c_1 ; c_2 & \text{sequencing} \\
| c_1 \llcorner c_2 & \text{nondet. choice}
\end{align*}$
From Programs to Guarded Commands

- $\text{GC}(\text{skip}) =$
- $\text{GC}(x := e) =$
- $\text{GC}(c_1 ; c_2) =$
- $\text{GC}(\text{if } b \text{ then } c_1 \text{ else } c_2) =$
- $\text{GC}([I] \text{ while } b \text{ do } c) =$
From Programs to Guarded Commands

- \( \text{GC}(\text{skip}) = \)
  \[
  \text{assume true}
  \]

- \( \text{GC}(x := e) = \)
  \[
  \text{havoc } tmp; \text{ assume } tmp = x; \quad \text{where } tmp \text{ is fresh}
  \]
  \[
  \text{havoc } x; \text{ assume } (x = e[tmp/x])
  \]

- \( \text{GC}(c_1 ; c_2) = \)
  \[\text{GC}(c_1); \text{GC}(c_2)\]

- \( \text{GC}(\text{if } b \text{ then } c_1 \text{ else } c_2) = \)
  \[
  (\text{assume } b; \text{ GC}(c_1)) \sqcup (\text{assume } \neg b; \text{ GC}(c_2))
  \]

- \( \text{GC}([I] \text{ while } b \text{ do } c) = ? \)
Guarded Commands for Loops

• \( \text{GC}\{\text{I}\} \text{ while } b \text{ do } c \) =
  
  \[
  \begin{align*}
  &\text{assert I;} \\
  &\text{havoc } x_1; \ldots; \text{havoc } x_n; \\
  &\text{assume I;} \\
  &\text{assume } \neg b
  \end{align*}
  \]
  
  where \( x_1, \ldots, x_n \) are the variables assigned in \( c \)
Computing Weakest Preconditions

- $WP(\text{assume } b, B) =$
- $WP(\text{assert } b, B) =$
- $WP(\text{havoc } x, B) =$
- $WP(c_1; c_2, B) =$
- $WP(c_1 \parallel c_2, B) =$
Computing Weakest Preconditions

- \( WP(\text{assume } b, B) = b \Rightarrow B \)
- \( WP(\text{assert } b, B) = b \land B \)
- \( WP(\text{havoc } x, B) = B[a/x] \quad (a \text{ fresh in } B) \)
- \( WP(c_1; c_2, B) = WP(c_1, WP(c_2, B)) \)
- \( WP(c_1 ◊ c_2, B) = WP(c_1, B) \land WP(c_2, B) \)
Computing Weakest Preconditions

- \( \text{WLP}(\text{assume } b, B) = b \Rightarrow B \)
- \( \text{WLP}(\text{assert } b, B) = b \Rightarrow B \)
- \( \text{WLP}(\text{havoc } x, B) = B[a/x] \) (\( a \) fresh in \( B \))
- \( \text{WLP}(c_1 ; c_2, B) = \text{WP}(c_1, \text{WP}(c_2, B)) \)
- \( \text{WLP}(c_1 \sqcap c_2, B) = \text{WP}(c_1, B) \land \text{WP}(c_2, B) \)
Putting Everything Together

• Given a Hoare triple \( H \equiv \{A\} P \{B\} \)

• Compute \( c_H = \text{assume } A; \text{ GC}(P); \text{ assert } B \)

• Compute \( VC_H = WP(c_H, \text{ true}) \)

• Infer \( \vdash VC_H \) using a theorem prover.
Example: VC Generation

\{n \geq 0\}

\begin{align*}
p &:= 0; \\
x &:= 0; \\
\{p = x \cdot m \land x \leq n\}
\end{align*}

\textbf{while } x < n \textbf{ do}

\hspace{1cm} x := x + 1;

\hspace{1cm} p := p + m

\hspace{1cm} \{ p = n \cdot m \}
Example: VC Generation

• Computing the guarded command

```plaintext
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

```plaintext
assume n \geq 0;
assume p_0 = p; havoc p; assume p = 0;
GC(
  x := 0;
  \{ p = x \times m \land x \leq n \}
 while x < n do
  x := x + 1;
  p := p + m 
); assert p = n \times m
```
Example: VC Generation

• Computing the guarded command

\[
\text{assume } n \geq 0; \\
\text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{GC(} \{ p = x \ast m \land x \leq n \} \\
\text{ \textbf{while} } x < n \text{ \textbf{do} } \\
\quad x := x + 1; \\
\quad p := p + m ); \\
\text{assert } p = n \ast m
\]
Example: VC Generation

• Computing the guarded command

\[
\text{assume } n \geq 0; \\
\text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{assert } p = x \times m \land x \leq n; \\
\text{havoc } x; \text{ havoc } p; \text{ assume } p = x \times m \land x \leq n; \\
\text{(assume } x < n; \\
\text{GC( } x := x + 1; \\
\text{ } p := p + m); \\
\text{assert } p = x \times m \land x \leq n; \text{ assume false }) \\
\text{}\text{ assume } x \geq n; \\
\text{assert } p = n \times m
\]
Example: VC Generation

• Computing the guarded command

\[
\text{assume } n \geq 0; \\
\text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{assert } p = x \cdot m \land x \leq n; \\
\text{havoc } x; \text{ havoc } p; \text{ assume } p = x \cdot m \land x \leq n; \\
(\text{assume } x < n; \\
\text{assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1; \\
\text{assume } p_1 = p; \text{ havoc } p; \text{ assume } p = p_1 + m; \\
\text{assert } p = x \cdot m \land x \leq n; \text{ assume false}) \\
\] \\
\text{assume } x \geq n; \\
\text{assert } p = n \cdot m
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 \( p = x \cdot m \land x \leq n \); 
havoc \( x \); havoc \( p \); assume \( p = x \cdot m \land x \leq n \); 
  (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 \( p = x \cdot m \land x \leq n \); assert false 
)
\( \neg \) assume \( x \geq n \); 
assert \( p = n \cdot m \), true)
Example: VC Generation

• Computing the weakest precondition

\[
\text{WP ( assume } n \geq 0; \\
\text{ assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{ assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{ assert } p = x \cdot m \land x \leq n, \\
\text{ WP(havoc } x; \text{ havoc } p; \text{ assume } p = x \cdot m \land x \leq n; \\
(\text{assume } x < n; \\
\text{ assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1; \\
\text{ assume } p_1 = p; \text{ havoc } p; \text{ assume } p = p_1 + m; \\
\text{ assert } p = x \cdot m \land x \leq n; \text{ assume false }) \\
\text{ assume } x \geq n, p = n \cdot m) \\
\]

• Computing the weakest precondition
Example: VC Generation

• Computing the weakest precondition

\[
WP ( \begin{align*}
& \text{assume } n \geq 0; \\
& \text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
& \text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
& \text{assert } p = x \cdot m \land x \leq n, \\
\end{align*} ) \Rightarrow
( WP ( \begin{align*}
& \text{assume } x < n; \\
& \text{assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1; \\
& \text{assume } p_1 = p; \text{ havoc } p; \text{ assume } p = p_1 + m; \\
& \text{assert } p = x \cdot m \land x \leq n; \text{ assume false } ) ) \Rightarrow \\
& \quad p = n \cdot m \\
& \land ( x \geq n \Rightarrow p = n \cdot m ) \\
\end{align*} )
\]
Example: VC Generation

- Computing the weakest precondition

\[
\text{WP} \left( \begin{array}{l}
\text{assume } n \geq 0; \\
\text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{assert } p = x \times m \land x \leq n,
\end{array} \right)
\]

\[
\text{WP} \left( \begin{array}{l}
\text{havoc } x; \text{ havoc } p; \text{ assume } p = x \times m \land x \leq n,
\end{array} \right)
\]

\[
(WP \left( \begin{array}{l}
\text{assume } x < n;
\end{array} \right)
\]

\[
\begin{array}{l}
\text{assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1;
\end{array}
\]

\[
\text{assume } p_1 = p; \text{ havoc } p; \text{ assume } p = p_1 + m;
\]

\[
\text{assert } p = x \times m \land x \leq n), \text{ false } \Rightarrow p = n \times m
\]

\[
\wedge (x \geq n \Rightarrow p = n \times m))
\]
Example: VC Generation

- Computing the weakest precondition

\[
\text{WP} \left( \begin{align*}
\text{assume } n & \geq 0; \\
\text{assume } p_0 &= p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 &= x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{assert } p &= x \times m \land x \leq n, \\
\text{WP}(\text{havoc } x; \text{ havoc } p; \text{ assume } p = x \times m \land x \leq n), \\
(\text{WP}( \text{assume } x < n; \\
\text{assume } x_1 &= x; \text{ havoc } x; \text{ assume } x = x_1 + 1; \\
\text{assume } p_1 &= p; \text{ havoc } p; \text{ assume } p = p_1 + m; \\
\text{assert } p &= x \times m \land x \leq n), \text{ true}) \\
\land (x \geq n \Rightarrow p = n \times m)) \end{align*} \right)
\]
Example: VC Generation

• Computing the weakest precondition

\[
\begin{align*}
\text{WP (} & \text{ assume } n \geq 0; \\
& \text{ assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
& \text{ assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
& \text{ assert } p = x \times m \land x \leq n, \\
\text{WP(} & \text{ havoc } x; \text{ havoc } p; \text{ assume } p = x \times m \land x \leq n, \\
\text{ (WP(} & \text{ assume } x < n; \\
& \text{ assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1; \\
& \text{ assume } p_1 = p; \text{ havoc } p), \\
& p = p_1 + m \Rightarrow p = x \times m \land x \leq n) \\
& \land (x \geq n \Rightarrow p = n \times m))))
\end{align*}
\]
Example: VC Generation

- Computing the weakest precondition

\[
\text{WP (} \quad \text{assume } n \geq 0; \nonumber \\
\quad \text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \nonumber \\
\quad \text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \nonumber \\
\quad \text{assert } p = x \cdot m \land x \leq n, \nonumber \\
\text{WP (} \quad \text{havoc } x; \text{ havoc } p; \text{ assume } p = x \cdot m \land x \leq n, \nonumber \\
\quad \text{WP (} \quad \text{assume } x < n; \nonumber \\
\quad \quad \text{assume } x_1 = x; \text{ havoc } x; \text{ assume } x = x_1 + 1), \nonumber \\
\quad \quad p_1 = p \land p a_1 = p_1 + m \Rightarrow p a_1 = x \cdot m \land x \leq n) \nonumber \\
\quad \land (x \geq n \Rightarrow p = n \cdot m)) \nonumber 
\]
Example: VC Generation

• Computing the weakest precondition

\[ WP ( \text{assume } n \geq 0; \text{assume } p_0 = p; \text{havoc } p; \text{assume } p = 0; \text{assume } x_0 = x; \text{havoc } x; \text{assume } x = 0; \text{assert } p = x \times m \land x \leq n, \text{WP(havoc } x; \text{havoc } p; \text{assume } p = x \times m \land x \leq n, \text{WP( (assume } x < n), (WP( (\text{assume } x < n), (x_1 = x \land xa_1 = x_1 + 1 \land p_1 = p \land pa_1 = p_1 + m) \Rightarrow pa_1 = x \times m \land x \leq n) \land (x \geq n \Rightarrow p = n \times m)))) \]
Example: VC Generation

- Computing the weakest precondition

\[\text{WP (} \begin{align*} &\text{assume } n \geq 0; \\
&\text{assume } p_0 = p; \text{ havoc } p; \text{ assume } p = 0; \\
&\text{assume } x_0 = x; \text{ havoc } x; \text{ assume } x = 0; \\
&\text{assert } p = x \times m \land x \leq n, \\
\end{align*} \]

\[\text{WP(} \begin{align*} &\text{havoc } x; \text{ havoc } p; \text{ assume } p = x \times m \land x \leq n, \\
&\begin{align*} &\text{(} x < n \land \\
&x_1 = x \land x_{a_1} = x_1 + 1 \land \\
p_1 = p \land p_{a_1} = p_1 + m) \Rightarrow p_{a_1} = x \times m \land x \leq n \end{align*} \]

\[\land (x \geq n \Rightarrow p = n \times m)) \]
Example: VC Generation

- Computing the weakest precondition

\[ \text{WP} ( \begin{align*}
\text{assume } n &\geq 0; \\
\text{assume } p_0 &= p; \text{ havoc } p; \text{ assume } p = 0; \\
\text{assume } x_0 &= x; \text{ havoc } x; \text{ assume } x = 0; \\
\text{assert } p &= x \cdot m \land x \leq n, \\
(p a_2 &= x a_2 \cdot m \land x a_2 \leq n \land \\
x a_2 &< n \land \\
x_1 &= x a_2 \land x a_1 = x_1 + 1 \land \\
p_1 &= p a_2 \land p a_1 = p_1 + m) \Rightarrow p a_1 = x a_2 \cdot m \land x a_2 \leq n \\
\land (x &\geq n \Rightarrow p = n \cdot m))) \]
Example: VC Generation

- Computing the weakest precondition

\[ n \geq 0 \land p_0 = p \land pa_3 = 0 \land x_0 = x \land xa_3 = 0 \Rightarrow \\
\quad pa_3 = xa_3 \cdot m \land xa_3 \leq n \land \\
\quad (pa_2 = xa_2 \cdot m \land xa_2 \leq n \land \\
\quad xa_2 < n \land \\
\quad x_1 = xa_2 \land xa_1 = x_1 + 1 \land \\
\quad p_1 = pa_2 \land pa_1 = p_1 + m) \Rightarrow pa_1 = xa_2 \cdot m \land xa_2 \leq n) \\
\quad \land (x \geq n \Rightarrow p = n \cdot m) ]}
Example: VC Generation

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

\( n \geq 0 \land p_0 = p \land pa_3 = 0 \land x_0 = x \land xa_3 = 0 \Rightarrow \)

\( pa_3 = xa_3 \cdot m \land xa_3 \leq n \)

\( n \geq 0 \land p_0 = p \land pa_3 = 0 \land x_0 = x \land xa_3 = 0 \land pa_2 = xa_2 \cdot m \land xa_2 \leq n \Rightarrow \)

\( xa_2 \geq n \Rightarrow pa_2 = n \cdot m \)

\( n \geq 0 \land p_0 = p \land pa_3 = 0 \land x_0 = x \land xa_3 = 0 \land pa_2 = xa_2 \cdot m \land xa_2 < n \land x_1 = xa_2 \land xa_1 = x_1 + 1 \land p_1 = pa_2 \land pa_1 = p_1 + m \Rightarrow \)

\( pa_1 = xa_1 \cdot m \land xa_1 \leq n \)
Example: VC Generation

- simplifying the constraints yields

\[ n \geq 0 \Rightarrow 0 = 0 \cdot m \land 0 \leq n \]

\[ xa_2 \leq n \land xa_2 \geq n \Rightarrow xa_2 \cdot m = n \cdot m \]

\[ xa_2 < n \Rightarrow xa_2 \cdot m + m = (xa_2 + 1) \cdot m \land xa_2 + 1 \leq n \]

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

assume A;
c \sqsubseteq d;
c' \sqsubseteq d';
assert B

A \Rightarrow WP(c, WP(c', B) \land WP(d', B)) \land WP(d, WP(c', B) \land 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?

• 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)

• Eliminate redundancies from the VCs entirely – yields formula that are quadratic in the program size – usually more efficient once theorem prover is factored in
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 \( B \):

\[
WP(c, B) \equiv WP(c, \text{true}) \land WLP(c, B)
\]

• We got rid of \( B \) below \( WP \). Can we also get rid of \( B \) below \( WLP \)?
Passive Guarded Commands

- $c ::= \text{assume } b$ block if $b$ does not hold
  - $\text{assert } b$ fail if $b$ does not hold
  - $\text{havoc } x$ nondet. assignment
  - $c_1 ; c_2$ sequencing
  - $c_1 \parallel c_2$ 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 $B$:

$$ WP(c, B) \equiv WP(c, \text{true}) \land WLP(c, B) $$

- For passive programs $c$ we also have:

$$ WLP(c, B) \equiv WLP(c, \text{false}) \lor B $$
Removing Redundancy from VCs

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

\[ WP(c, B) \equiv WP(c, \text{true}) \land (WLP(c, \text{false}) \lor B) \]

• \( WP(c, B) \) is now quadratic in the size of \( c \)
• There is no duplication of \( B \) for each path in \( c \)
Translating Method Calls to GCs

\begin{verbatim}
method m (p_1: T_1, ..., p_k: T_k) returns (r: T)
  requires P
  modifies x_1, ..., x_n
  ensures Q

A method call
  y := y_0.m(y_1, ..., y_k);

is desugared into the guarded command
  assert P[y_0/this, y_1/p_1, ..., y_k/p_k];
  havoc x_1; ..., havoc x_n; havoc y;
  assume Q[y_0/this, y_1/p_1, ..., y_k/p_k, y/r]
\end{verbatim}
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 ∨ 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 \( \text{sel}(r,f) \) and \( r.f := e \) is \( r := \text{upd}(r, f, e) \)

• Model 2 (Burstall-Bornat)
  – One “heap” for each field
  – The object address is the index
  – \( r.f \) is \( \text{sel}(f,r) \) and \( r.f := e \) is \( f := \text{upd}(f,r,e) \)
Hoare Rule for Field Assignments

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

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

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

\[
\text{sel}(\text{upd}(h, e_1, e_2), e_3) = \begin{cases} 
  e_2 & \text{if } e_1 = e_3 \\
  \text{sel}(h, e_3) & \text{if } e_1 \neq e_3
\end{cases}
\]
Example: Hoare Rule for Field Writes

• Consider again: \[ \{ A \} x.f := 5 \{ x.f + y.f = 10 \} \]

• We obtain:
  \[
  A \equiv (x.f + y.f = 10)[\text{upd}(f, x, 5)/f]
  \equiv (\text{sel}(f, x) + \text{sel}(f, y) = 10)[\text{upd}(f, x, 5)/f]
  \equiv \text{sel(}\text{upd}(f \times 5, x)) + \text{sel(}\text{upd}(f \times 5, y)) = 10
  \equiv 5 + \text{sel(}\text{upd}(f, x, 5), y) = 10
  \equiv \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(f, y) = 10
  \equiv x = y \lor y.f = 5
  \]

• Theorem generation.

• Theorem proving.
Modeling new Statements

• Introduce a set-valued variable $\text{Alloc}$ that denotes the set of all objects that are currently allocated

• Translate $x := \text{new } \text{T.Init}()$ to
  
  havoc $x$;
  assume $x \notin \text{Alloc}$;
  assume $\text{Type}(x) = \text{T}$;
  assume $x \neq \text{null}$;
  $\text{Alloc} := \text{Alloc} \cup \{x\}$;

**Translation of call to constructor T.Init()**