Mechanically Proving Hoare Formulae

Hoare 75 talk (revised)

Additional material
An Axiomatic Basis for Computer Programming

C. A. R. Hoare, 1969
Hoare’s Axiomatic Basis was originally both
- an axiomatic language definition method and
- a proof theory for program verification

Will focus on the verification role today
- after 40 years it is still a key idea in program correctness

However, instead of

“... accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.”

can derive axioms and rules from language semantics
- parametrizes verification technology on semantics
- semantic approach effective with current theorem provers
Range of methods for proving $\{P\} C \{Q\}$

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute
  - check states reached satisfy properties

- Full verification
  - handle unbounded loops and recursion
  - invariants, induction etc.
  - needs undecidable logics and user guided proof

- Goal: unifying framework for a spectrum of methods

decidable checking  proof of correctness
Standard backwards method of proving \([P] C [Q]\)

- A common approach is to use weakest preconditions
  - precondition \(WP \ C \ Q\) ensures \(Q\) holds after \(C\) terminates
  - \(WP \ C \ Q\) is Dijkstra’s ‘weakest liberal precondition’
    (i.e. partial correctness: \(wlp. C. Q\) from Dijkstra & Scholten)
  - easy to compute \(WP \ C \ Q\) if \(C\) has no loops

- Precondition calculation works backwards from \(Q\)
  - nice Hoare assignment calculation rule for \(WP\)
    \(WP (V := E) \ Q = Q[V ← E]\)
  - pulls postcondition \(Q\) back through program
    \(WP (C_1 ; C_2) \ Q = WP \ C_1 (WP \ C_2 \ Q)\)
  - can’t dynamically prune unreachable conditional branches
    \(WP (IF \ B \ THEN \ C_1 \ ELSE \ C_2) \ Q = (B ∧ WP \ C_1 \ Q) ∨ (¬B ∧ WP \ C_2 \ Q)\)

- \([P] C [Q] = P ⇒ WP \ C \ Q\)

- \(wlp. C. Q\) is weakest solution of \(P : ([P] C [Q])\)
  (Predicate Calculus & Program Semantics, Dijkstra & Scholten, 1990)

Happy 75 Tony!
Standard backwards method of proving \( \{P\} C \{Q\} \)

- A common approach is to use weakest preconditions
  - precondition \( WP \ C \ Q \) ensures \( Q \) holds after \( C \) terminates
  - \( WP \ C \ Q \) is Dijkstra’s ‘weakest liberal precondition’
    (i.e. partial correctness: \( wlp.\ C.\ Q \) from Dijkstra & Scholten)
  - easy to compute \( WP \ C \ Q \) ............... if \( C \) has no loops

- Precondition calculation works backwards from \( Q \)
  - nice Hoare assignment calculation rule for \( WP \)
    \( WP (V := E) Q = Q[V \leftarrow E] \)
  - pulls postcondition \( Q \) back through program
    \( WP (C_1; C_2) Q = WP C_1 (WP C_2 Q) \)
  - can’t dynamically prune unreachable conditional branches
    \( WP (IF B THEN C_1 ELSE C_2) Q = (B \land WP C_1 Q) \lor (\neg B \land WP C_2 Q) \)

- \( \{P\} C \{Q\} \equiv P \Rightarrow WP \ C \ Q \)
- \( wlp.\ C.\ Q \) is weakest solution of \( P : (\{P\} C \{Q\}) \)
  \( (Predicate\ Calculus\ &\ Program\ Semantics,\ Dijkstra\ &\ Scholten,\ 1990)\)
Proving \( \{P\} C \{Q\} \) forwards

- Less used alternative is strongest postconditions
  - \( \text{SP} \ P \ C \) holds after \( C \) terminates if started when \( P \) holds
  - \( \text{SP} \ Q \ C \) is ‘strongest postcondition’
    \( (\text{sp}.C.Q \text{ in Dijkstra & Scholten, Ch.12 – not sp}_p.C.Q) \)

- Postcondition calculation works forwards from \( P \)
  - nasty Floyd assignment rule introduces \( \exists \)-quantification
    \[
    \text{SP} \ P \ (V := E) = \exists v. \ V = E[V \leftarrow v] \land P[V \leftarrow v]
    \]
    “The problem with this rule is the accumulation of quantifiers.” [Reynolds]  “... a semantic theory based on weakest preconditions turned out to be simpler than one based on strongest postconditions.” [Dijkstra]

- compute by symbolic execution + building up constraints
  \[
  \text{SP} \ P \ (C_1; C_2) = \text{SP} \ (\text{SP} \ P \ C_1) \ C_2
  \]
  - can prune branches with symbolic state and constraints
    \[
    \text{SP} \ P \ (\text{IF} \ B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) = \\
    \text{SP} \ (P \land B) \ C_1 \lor \text{SP} \ (P \land \neg B) \ C_2
    \]

- \( \{P\} C \{Q\} \equiv \text{SP} \ P \ C \Rightarrow Q \)
- \( \text{sp}.C.P \) is strongest solution of \( Q : (\{P\} C \{Q\}) \)
Proving \( \{P\} C \{Q\} \) forwards

- Less used alternative is strongest postconditions
  - \( SP \, P \, C \) holds after \( C \) terminates if started when \( P \) holds
  - \( SP \, Q \, C \) is ‘strongest postcondition’
    \( (sp.C.Q \text{ in Dijkstra & Scholten, Ch.12 – not stp.C.Q}) \)

- Postcondition calculation works forwards from \( P \)
  - nasty Floyd assignment rule introduces \( \exists \)-quantification
    \[ SP \, P \, (V := E) = \exists v. \, V = E[V \leftarrow v] \land P[V \leftarrow v] \]
    “The problem with this rule is the accumulation of quantifiers.” [Reynolds]  
    “... a semantic theory based on weakest preconditions turned out to be simpler than one based on strongest postconditions.” [Dijkstra]

- compute by symbolic execution + building up constraints
  \[ SP \, P \, (C_1; C_2) = SP \, (SP \, P \, C_1) \, C_2 \]

- can prune branches with symbolic state and constraints
  \[ SP \, P \, (\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) = SP \, (P \land B) \, C_1 \lor SP \, (P \land \neg B) \, C_2 \]

- \( \{P\} C \{Q\} \equiv SP \, P \, C \Rightarrow Q \)

- \( sp.C.P \) is strongest solution of \( Q : (\{P\} \, C \, \{Q\}) \)
Backwards or forwards?

- Calculating $WP\ C\ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP\ P\ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (IF\ B\ THEN\ C_2\ ELSE\ C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example

  $\{J \leq I\}$
  $K := 0$
  $IF\ I < J\ THEN\ K := K + 1\ ELSE\ SKIP$;
  $IF\ K = 1\ AND\ not (I = J)\ THEN\ R := J - I\ ELSE\ R := I - J$
  $\{R = I - J\}$
Backwards or forwards?

- Calculating $WP \ C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP \ P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (\texttt{IF } B \texttt{ THEN } C_2 \texttt{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example
  \[
  \begin{align*}
  \{J \leq I\} \\
  K & := 0; \\
  \text{IF } I < J \text{ THEN } K & := K + 1 \text{ ELSE SKIP;} \\
  \text{IF } K = 1 \land \neg(I = J) \text{ THEN } R & := J - I \text{ ELSE } R := I - J \\
  \{R = I - J\}
  \end{align*}
  \]
Backwards or forwards?

- Calculating $WP \ C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP \ P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example

  \[
  \begin{align*}
  &\{J \leq I\} \\
  &K := 0; \quad \{J \leq I \land K = 0\} \\
  &\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP;} \\
  &\text{IF } K = 1 \land \neg (I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J \\
  &\{R = I - J\}
  \end{align*}
  \]
Backwards or forwards?

- Calculating $WP \ C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’
- Calculating $SP \ P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths
- Consider $\{P\}C_1;(\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution
- Forwards methods meshes better with BMC
- Example

\[
\begin{align*}
\{J \leq I\} \\
K := 0; & \quad \{J \leq I \wedge K = 0\} \\
\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP}; & \quad \{J \leq I \wedge K = 0\} \\
\text{IF } K = 1 \wedge \neg(= J) \text{ THEN } R := J - I \text{ ELSE } R := I - J \\
\{R = I - J\}
\end{align*}
\]
Backwards or forwards?

- Calculating $WP\ C\ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP\ P\ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1\; (\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example

\[
\begin{align*}
\{J \leq I\} \\
K := 0; &\quad \{J \leq I \land K = 0\} \\
\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP;} &\quad \{J \leq I \land K = 0\} \\
\text{IF } K = 1 \land \neg (I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J &
\end{align*}
\]
Backwards or forwards?

- Calculating $\text{WP } C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $\text{SP } P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $\text{SP}$ is execution

- Forwards methods meshes better with BMC

- Example

  $\{I < J\}$
  
  $K := 0;$
  
  $\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP;}$
  
  $\text{IF } K = 1 \land \neg(I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J$
  
  $\{R = J - I\}$
Backwards or forwards?

- Calculating $WP$ $C$ $Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’
- Calculating $SP$ $P$ $C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths
- Consider $\{P\}C_1;(\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3);C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution
- Forwards methods meshes better with BMC
- Example

\[
\{I < J\}
\]

\[
K := 0;
\]

\[
\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP;}
\]

\[
\text{IF } K = 1 \land \neg (I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J
\]

\[
\{R = J - I\}
\]
Backwards or forwards?

- Calculating $WP \ C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP \ P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1;(\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example

\[
\{I < J\} \\
K := 0; \quad \{I < J \land K = 0\} \\
\text{IF } I < J \text{ THEN } K := K + 1 \text{ ELSE SKIP;} \\
\text{IF } K = 1 \land \neg (I = J) \text{ THEN } R := J - I \text{ ELSE } R := I - J \\
\{R = J - I\}
\]
Backwards or forwards?

- Calculating $WP\ C\ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $SP\ P\ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $SP$ is execution

- Forwards methods meshes better with BMC

- Example

\[
\begin{align*}
\{I < J\} \\
K &:= 0; \quad \{I < J \land K = 0\} \\
\text{IF } I < J \text{ THEN } K &:= K + 1 \text{ ELSE SKIP;} \quad \{I < J \land K = 1\} \\
\text{IF } K = 1 \land \neg (I = J) \text{ THEN } R &:= J - I \text{ ELSE } R := I - J \\
\{R = J - I\}
\end{align*}
\]
Backwards or forwards?

- Calculating $\text{WP } C \ Q$ is easy but leads to big formulae
  - can’t prune case splits ‘on-the-fly’

- Calculating $\text{SP } P \ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths

- Consider $\{P\}C_1; (\text{IF } B \text{ THEN } C_2 \text{ ELSE } C_3); C_4\{Q\}$
  - going forwards $P$ and effect of $C_1$ might determine $B$
  - if $P$ specifies a unique state, computing $\text{SP}$ is execution

- Forwards methods meshes better with BMC

- Example

\[
\begin{align*}
\{I < J\} \\
K & := 0; \quad \{I < J \land K = 0\} \\
\text{IF } I < J \text{ THEN } K & := K + 1 \text{ ELSE SKIP;} \quad \{I < J \land K = 1\} \\
\text{IF } K = 1 \land \neg(I = J) \text{ THEN } R & := J - I \text{ ELSE } R := I - J \\
\{R = J - I\}
\end{align*}
\]
Can’t compute finite **WP** or **SP** for loops

- **Loop-free:** symbolic evaluation is just calculating **SP**

- **Loops:** no finite formula for **WP** or **SP** in general
  
  - \[
  WP \left( \text{WHILE } B \text{ DO } C \right) Q = (B \land WP C (WP \left( \text{WHILE } B \text{ DO } C \right) Q)) \lor (\neg B \land Q)
  \]
  
  - \[
  SP P \left( \text{WHILE } B \text{ DO } C \right) = (SP (SP (P \land B) C) (WHILE B DO C)) \lor (P \land \neg B)
  \]

- **Solution:** Hoare logic rule with an invariant \( R \)
  
  \[
  \frac{P \Rightarrow R \quad \{ R \land B \} C \{ R \} \quad R \land \neg B \Rightarrow Q}{\{ P \} \text{WHILE } B \text{ DO } C \{ Q \}}
  \]

- **Use approximate **WP** or **SP** plus verification conditions**
Method of verification conditions (VCs)

- Define **AWP** and **ASP** (“A” for “approximate”)
  - like **WP**, **SP** for skip, assignment, sequencing, conditional
  - for while-loops assume invariant $R$ magically supplied

\[
\text{AWP (WHILE } B\text{ DO } \{ R \} \ C) \ Q = R
\]
\[
\text{ASP } P (\text{WHILE } B \text{ DO } \{ R \} \ C) = R \wedge \neg B
\]

- Define **WVC** $C \ Q$ and **SVC** $P \ C$ to generate VCs
  (more details on next slide)

- Prove $\{ P \} C\{ Q \}$ using theorems

\[
\text{WVC } C \ Q \Rightarrow \{ \text{AWP } C \ Q \} C\{ Q \}
\]
\[
\text{SVC } P \ C \Rightarrow \{ P \} C\{ \text{ASP } P \ C \}
\]
Calculating verification conditions (VCs)

VCs to augment approximate weakest preconditions

\[ \text{WVC}(\text{SKIP}) \quad Q = T \]
\[ \text{WVC}(V := E) \quad Q = T \]
\[ \text{WVC}(C_1; C_2) \quad Q = \text{WVC}(C_1 \text{ (AWP } C_2 \text{ } Q) \land \text{WVC } C_2 \text{ } Q) \]
\[ \text{WVC}(\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) \quad Q = \text{WVC } C_1 \text{ } Q \land \text{WVC } C_2 \text{ } Q \]
\[ \text{WVC}(\text{WHILE } B \text{ DO } \{R\} \ C) \quad Q = \text{AWP } C \text{ } R) \land (R \land \neg B \Rightarrow Q) \land \text{WVC } C \text{ } R \]

VCs to augment approximate strongest postconditions

\[ \text{SVC} \text{ } P \text{ (SKIP)} = T \]
\[ \text{SVC} \text{ } P \text{ (V := E)} = T \]
\[ \text{SVC} \text{ } P \text{ (C_1; C_2)} = \text{SVC } C_1 \land \text{SVC } (\text{ASP } P \text{ } C_1) \text{ } C_2 \]
\[ \text{SVC} \text{ } P \text{ (IF } B \text{ THEN } C_1 \text{ ELSE } C_2) = \text{SVC } (P \land B) \text{ } C_1 \land \text{SVC } (P \land \neg B) \text{ } C_2 \]
\[ \text{SVC} \text{ } P \text{ (WHILE } B \text{ DO } \{R\} \ C) = (P \Rightarrow R) \land (\text{ASP } (R \land B) \ C \Rightarrow R) \land \text{SVC } (R \land B) \ C \]
Calculating verification conditions (VCs)

▷ VCs to augment approximate weakest preconditions

\[
WVC(\text{SKIP}) \ Q = T \\
WVC(V := E) \ Q = T \\
WVC(C_1; C_2) \ Q = WVC \ C_1 (\text{AWP} \ C_2 \ Q) \land WVC \ C_2 \ Q \\
WVC(\text{IF} B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) \ Q = \\
WVC \ C_1 \ Q \land WVC \ C_2 \ Q \\
WVC(\text{WHILE} \ B \ \text{DO} \ \{R\} \ C) \ Q = \\
(R \land B \Rightarrow \text{AWP} \ C \ R) \land (R \land \neg B \Rightarrow Q) \land WVC \ C \ R
\]

▷ VCs to augment approximate strongest postconditions

\[
SVC \ P(\text{SKIP}) = T \\
SVC \ P(V := E) = T \\
SVC \ P(C_1; C_2) = SVC \ P \ C_1 \land SVC(\text{ASP} P \ C_1) \ C_2 \\
SVC \ P(\text{IF} B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) = \\
SVC \ (P \land B) \ C_1 \land SVC \ (P \land \neg B) \ C_2 \\
SVC \ P(\text{WHILE} \ B \ \text{DO} \ \{R\} \ C) = \\
(P \Rightarrow R) \land (\text{ASP} \ (R \land B) \ C \Rightarrow R) \land SVC \ (R \land B) \ C
\]

Happy 75 Tony!
Calculating verification conditions (VCs)

- VCs to augment approximate weakest preconditions
  \[
  \text{WVC (SKIP)} Q = T \\
  \text{WVC (V := E)} Q = T \\
  \text{WVC (C₁ ; C₂)} Q = \text{WVC C₁ (AWP C₂ Q)} \land \text{WVC C₂ Q} \\
  \text{WVC (IF B THEN C₁ ELSE C₂)} Q = \\
  \text{WVC C₁ Q} \land \text{WVC C₂ Q} \\
  \text{WVC (WHILE B DO \{R\} C)} Q = \\
  (R \land B \Rightarrow \text{AWP C R}) \land (R \land \neg B \Rightarrow Q) \land \text{WVC C R}
  \]

- VCs to augment approximate strongest postconditions
  \[
  \text{SVC P (SKIP)} = T \\
  \text{SVC P (V := E)} = T \\
  \text{SVC P (C₁ ; C₂)} = \text{SVC P C₁} \land \text{SVC (ASP P C₁)} C₂ \\
  \text{SVC P (IF B THEN C₁ ELSE C₂)} = \\
  \text{SVC (P \land B)} C₁ \land \text{SVC (P \land \neg B)} C₂ \\
  \text{SVC P (WHILE B DO \{R\} C)} = \\
  (P \Rightarrow R) \land (\text{ASP (R \land B) C} \Rightarrow R) \land \text{SVC (R \land B) C}
  \]
Symbolic execution as postcondition calculation

- **Recall**  
  \[ \text{SP } P \ (V := E) = \exists v. \ V = E[V \leftarrow v] \land P[V \leftarrow v] \]

- **Suppose**  
  \( P \) has form  
  \[ \exists x_1 \ldots x_n. \ S \land X_1 = e_1 \land \ldots \land X_n = e_n \]

  where
  - \( X_1, \ldots, X_n \) are program variables (e.g. string constants)
  - \( x_1, \ldots, x_n \) are logic variables (i.e. symbolic values)
  - \( S, e_1, \ldots, e_n \) only contain variables \( x_1, \ldots, x_n \) and constants

- **Abbreviating notation**:  
  \[ [\overline{X} \leftarrow \overline{e}] \text{ for } [X_1 \leftarrow e_1, \ldots, X_n \leftarrow e_n] \]

- **It follows that**  
  \[ \text{SP } P \ (X_i := E_i) \text{ is then} \]
  \[ \exists x_1 \ldots x_n. \ S \land X_1 = e_1 \land \ldots \land X_i = E_i[\overline{X} \leftarrow \overline{e}] \land \ldots \land X_n = e_n \]

- **Computing **  
  \( \text{SP} \) is now symbolic execution  
  - no new existential quantifiers generated by assignments!
  - \( \text{SP } P \ (\text{SKIP}) = P \)
  - \( \text{SP } P \ (C_1 ; C_2) = \text{SP } (\text{SP } P \ C_1) \ C_2 \)

Happy 75 Tony!
Symbolic execution of conditional branches

Recall

\[ \text{SP} \ P \ (\text{IF} \ B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) = \]
\[ \text{SP} \ (P \land B) \ C_1 \lor \text{SP} \ (P \land \neg B) \ C_2 \]

Hence

\[ \text{SP} \ (\exists x_1 \cdots x_n. \ S \land X_1 = e_1 \land \cdots \land X_n = e_n) \]
\[ (\text{IF} \ B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) \]
\[ = \text{SP} \ (\exists x_1 \cdots x_n. \ (S \land B[\overline{X} \leftarrow \overline{e}]) \land X_1 = e_1 \land \cdots \land X_n = e_n) \ C_1 \]
\[ \lor \]
\[ \text{SP} \ (\exists x_1 \cdots x_n. \ (S \land \neg B[\overline{X} \leftarrow \overline{e}]) \land X_1 = e_1 \land \cdots \land X_n = e_n) \ C_2 \]

Prune paths by checking \( S \land B[\overline{X} \leftarrow \overline{e}] \) with a solver

\[ F \lor P = P \lor F = P \]
Approximate symbolic execution of while-loops

- Symbolically execute straight line code as before
- For while-loops, recall from previous slide
  \[ \text{ASP } P (\text{WHILE } B \text{ DO } \{ R \} \ C) = R \land \neg B \]
- Hence execute while-loops as follows
  \[
  \text{ASP } (\exists x_1 \cdots x_n. S \land X_1 = e_1 \land \ldots \land X_n = e_n) \\
  (\text{WHILE } B \text{ DO } \{ R \} \ C) \\
  = (\exists x_1 \cdots x_n. (R \land \neg B[X \leftarrow \bar{x}]) \land X_1 = x_1 \land \ldots \land X_n = x_n)
  \]
  - constraint \( S \) computed up to loop is discarded
  - create new state satisfying invariant and loop exit condition
  - link between pre and post loop states provided by VCs
    \[
    ((\exists x_1 \cdots x_n. S \land X_1 = e_1 \land \ldots \land X_n = e_n) \Rightarrow R) \\
    \land \\
    (\text{ASP } (\exists x_1 \cdots x_n. (R \land B[X \leftarrow \bar{x}]) \land X_1 = x_1 \land \ldots \land X_n = x_n) \ C \Rightarrow R)
    \]
Combining BMC and full verification

- BMC unrolls programs and symbolically executes them
  - paths dynamically pruned via accumulated properties
- Traditional full verification generates WP + VCs for loops
  - working backwards precludes BMC-style forwards pruning
- Computing postconditions unifies BMC and full verification
  - symbolic execution is SP calculation
  - add forward VCs for verification of loops
Overview of the implementation

- Everything is programmed deduction in a theorem prover
  - semantic embedding plus custom theorem proving tools
  - for efficiency external oracles used to prune paths
  - oracle provenance tracking via theorem tags

- HOL4 used for implementation of theorem proving
  - provides higher order logic for representing semantics
  - LCF-style proof tools (deriving Hoare logic, solving VCs)
  - ML for proof scripting and general programming

- YICES used as oracle
  - SMT solver from SRI International
  - used to quickly check conditional branch feasibility
  - ‘blow away’ easy VCs (hard ones by HOL4 interactive proof)
Happy 75th for Hoare!
Happy 75th for Hoare! ............ Happy 40th for Hoare Logic!
Happy 75th for Hoare! .......... Happy 40th for Hoare Logic!

Tony has many years ahead
Happy 75th for Hoare! ........... Happy 40th for Hoare Logic!

Tony has many years ahead ........ and so does Hoare Logic!
Happy 75th for Hoare! ............ Happy 40th for Hoare Logic!

Tony has many years ahead ........ and so does Hoare Logic!

THE END
Happy 75th for Hoare! ........... Happy 40th for Hoare Logic!

Tony has many years ahead ........ and so does Hoare Logic!

THE END

only of the main talk
Happy 75th for Hoare! ............ Happy 40th for Hoare Logic!

Tony has many years ahead .......... and so does Hoare Logic!

THE END

only of the main talk ... actually there are lots more slides!
Mechanically Proving Hoare Formulae

Hoare 75 talk (revised)

Additional material
Semantic embedding

- Semantics of commands $C$ given by $\text{SEM } C \ s \ s'$
  - $\text{SEM } C \ s \ s'$ is an inductively defined relation
  - if $C$ run in state $s$ then it will terminate in state $s'$
  - commands assumed deterministic – at most one final state
    (“Formalizing Dijkstra” by J. Harrison for non-determinism)

- Notation: abbreviate $\text{SEM } C \ s \ s'$ to $\llbracket C \rrbracket(s, s')$

- $\{P\} C \{Q\} = \text{def } \forall s \ s'. \ P \ s \land \llbracket C \rrbracket(s, s') \Rightarrow Q \ s'$

- $\text{WP } C \ Q = \text{def } \lambda s. \forall s'. \llbracket C \rrbracket(s, s') \Rightarrow Q \ s'$

- $\vdash \{P\} C \{Q\} = \forall s. \ P \ s \Rightarrow \text{WP } C \ Q \ s$

- $\text{SP } P \ C = \text{def } \lambda s'. \exists s. \ P \ s \land \llbracket C \rrbracket(s, s')$

- $\vdash \{P\} C \{Q\} = \forall s. \text{SP } P \ C \ s \Rightarrow Q \ s$
Details and notations

- \( \{ P \} C \{ Q \} =_{\text{def}} \forall s s'. P s \land [C](s, s') \Rightarrow Q s' \)
  - \( P, Q : \text{state} \rightarrow \text{bool} \)
  - \( \text{state} = \text{string} \mapsto \text{value} \) (finite map)
  - \( s[x\mapsto v] \) is the state mapping \( x \) to \( v \) and like \( s \) elsewhere
  - \( [x_1\mapsto v_1; \cdots; x_n\mapsto v_n] \) has domain \( \{ x_1, \cdots, x_n \} \); maps \( x_i \) to \( v_i \)
  - \([C] : \text{state} \times \text{state} \rightarrow \text{bool}\)
  - \([B] : \text{state} \rightarrow \text{bool} \) (\([B]\) short for \( \text{BVAL} B \))
  - \([E] : \text{state} \rightarrow \text{value} \) (\([E]\) short for \( \text{NVAL} B \))
  - \( \text{WP} C Q : \text{state} \rightarrow \text{bool} \)
  - \( \text{SP} P C : \text{state} \rightarrow \text{bool} \)

- Overload \( \land, \lor, \Rightarrow, \neg \) to pointwise operations on predicates
  - \( (P_1 \land P_2) s = P_1 s \land P_2 s \)
  - \( (P_1 \lor P_2) s = P_1 s \lor P_2 s \)
  - \( (P_1 \Rightarrow P_2) s = P_1 s \Rightarrow P_2 s \)
  - \( (\neg P) s = \neg (P s) \)

- Define: \( \models P =_{\text{def}} \forall s. P s \)
Proving \( \{P\} C \{Q\} \) by calculating \( WP \ C \ Q \)

- Easy consequences of definition of \( WP \)
  - \( WP (\text{SKIP}) \ Q = Q \)
  - \( WP (V := E) \ Q = \lambda s. \ Q(s[V\rightarrow[E]s]) \)
  - \( WP (C_1; C_2) \ Q = WP C_1 (WP C_2 Q) \)
  - \( WP (\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) \ Q = ([B] \Rightarrow WP C_1 Q) \land (\neg[B] \Rightarrow WP C_2 Q) \)
  - \( WP (\text{WHILE } B \text{ DO } C) \ Q = ([B] \Rightarrow WP C (WP (\text{WHILE } B \text{ DO } C) Q)) \land (\neg[B] \Rightarrow Q) \)

- To prove \( \{P\} C \{Q\} \) for straight line code
  - calculate \( WP C Q \) .......... back substitution + case splits
  - prove \( \models P \Rightarrow WP C Q \) .................use a theorem prover
Proving \( \{P\} C \{Q\} \) by calculating \( \text{SP} \ P \ C \)

- Easy consequences of definition of SP
  - \( \text{SP} \ P \ (\text{SKIP}) = P \)
  - \( \text{SP} \ P \ (V := E) = \lambda s'. \exists s. \ P \ s \land (s' = s[V\rightarrow[E]s]) \)
  - \( \text{SP} \ P \ (C_1 ; C_2) = \text{SP} \ (\text{SP} \ P \ C_1) \ C_2 \)
  - \( \text{SP} \ P \ (\text{IF} \ B \ \text{THEN} \ C_1 \ \text{ELSE} \ C_2) = \text{SP} \ (P \land [B]) \ C_1 \lor \text{SP} \ (P \land \neg[B]) \ C_2 \)
  - \( \text{SP} \ P \ (\text{WHILE} \ B \ \text{DO} \ C) = \text{SP} \ (\text{SP} \ (P \land [B]) \ C) \ (\text{WHILE} \ B \ \text{DO} \ C) \lor (P \land \neg[B]) \)

- To prove \( \{P\} C \{Q\} \) for straight line code
  - calculate \( \text{SP} \ P \ C \) ................ calculating with \( \exists \) a problem
  - prove \( \vdash \text{WP} \ P \ C \Rightarrow Q \) ................ use a theorem prover
Computing assignment postconditions

- \( \vdash \text{SP } P \ (V := E) = \lambda s'. \exists s. \ P \ s \land (s' = s[V \rightarrow [E] s]) \)

Consider \( P \) of form
\[
\lambda s. \exists x_1 \cdots x_n. \ S \land (s = [X \rightarrow \overline{e}])
\]
where

- \( X_1, \ldots, X_n \) are distinct program variables (string constants)
- \( x_1, \ldots, x_n \) are logic variables (i.e. symbolic values)
- \( S, e_1, \ldots, e_n \) only contain variables \( x_1, \ldots, x_n \) and constants
- \([X \rightarrow \overline{e}]\) abbreviates \([X_1 \rightarrow e_1, \ldots, X_n \rightarrow e_n]\)

It follows that
\[
\vdash \text{SP } (\lambda s. \exists x_1 \cdots x_n. \ S \land (s = [X \rightarrow \overline{e}])) \quad (X_i := E_i)
= \lambda s. \exists x_1 \cdots x_n. \ S \land (s = [X \rightarrow \overline{e}][X_i \rightarrow ([E_i][X \rightarrow \overline{e}])])
\]
where

- \([X \rightarrow \overline{e}][X_i \rightarrow ([E_i][X \rightarrow \overline{e}])] = [X_1 \rightarrow e_1, \ldots, X_i \rightarrow ([E_i][X \rightarrow \overline{e}]), \ldots, X_n \rightarrow e_n]\)
Symbolic state notation for predicates

- Abbreviate
  \[ \lambda s. \exists x_1 \cdots x_n. S \, s \land (s = [\overline{X} \rightarrow \overline{e}]) \]
as
  \[ \langle \exists \overline{x}. \, S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \]
then it follows that

\[
\begin{align*}
\text{SP} \langle \exists \overline{x}. \, S \land X_1 &= e_1 \land \ldots \land X_n = e_n \rangle (X_i := E_i) \\
&= \langle \exists \overline{x}. \, S \land X_1 = e_1 \land \ldots \land X_i = [E_i][\overline{X} \rightarrow \overline{e}] \land \ldots \land X_n = e_n \rangle
\end{align*}
\]

- Computing \text{SP} is now symbolic execution
  - symbolic state term: \[ \langle \exists \overline{x}. \, S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \]
  - no new existential quantifiers generated by assignments!
  - \[ \text{SP} \, P \left( \text{SKIP} \right) = P \]
  - \[ \text{SP} \, P \left( C_1; C_2 \right) = \text{SP} \left( \text{SP} \, P \, C_1 \right) \, C_2 \]

- Simpler symbolic state representation OK for loop-free code
Symbolic state notation for predicates

- Abbreviate
  \[ \lambda s. \exists x_1 \cdots x_n. S \ s \land (s = [X \to e]) \]
  as
  \[ \langle \exists \vec{x}. S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \]
  then it follows that
  \[
  \text{SP} \langle \exists \vec{x}. S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle (X_i := E_i) \\
  = \langle \exists \vec{x}. S \land X_1 = e_1 \land \ldots \land X_i = [E_i] [X \to e] \land \ldots \land X_n = e_n \rangle
  \]

- Computing \text{SP} is now symbolic execution
  - symbolic state term: \[ \langle \exists \vec{x}. S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \]
  - no new existential quantifiers generated by assignments!
  - \[
  \text{SP} \ P \ (\text{SKIP}) = P \\
  \text{SP} \ P \ (C_1 ; C_2) = \text{SP} \ (\text{SP} \ P \ C_1) \ C_2
  \]

- Simpler symbolic state representation OK for loop-free code
Symbolic execution of conditional branches

- **Recall**

\[
\text{SP } P (\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2) = \text{SP } (P \land \llbracket B \rrbracket) \ C_1 \lor \text{SP } (P \land \neg \llbracket B \rrbracket) \ C_2
\]

- **Now**

\[
\langle \exists \overline{x}. \ S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \land \llbracket B \rrbracket = \langle \lambda s. \exists x_1 \cdots x_n. \ S \ s \land (s = \llbracket \overline{X} \rightarrow \overline{e} \rrbracket) \rangle \land \text{BVAL } B
\]

\[
= \langle \lambda s. \exists x_1 \cdots x_n. \ S \ s \land (s = \llbracket \overline{X} \rightarrow \overline{e} \rrbracket) \rangle \land \text{BVAL } B \ s
\]

\[
= \langle \lambda s. \exists x_1 \cdots x_n. \ S \ s \land (s = \llbracket \overline{X} \rightarrow \overline{e} \rrbracket) \rangle \land \text{BVAL } B \ s
\]

\[
= \langle \lambda s. \exists x_1 \cdots x_n. \ (S \ s \land \text{BVAL } B \ s) \land (s = \llbracket \overline{X} \rightarrow \overline{e} \rrbracket) \rangle
\]

\[
= \langle \lambda s. \exists x_1 \cdots x_n. \ (S \ s \land \text{BVAL } B \ [\overline{X} \rightarrow \overline{e}]) \ s \land (s = \llbracket \overline{X} \rightarrow \overline{e} \rrbracket) \rangle
\]

\[
= \langle \exists \overline{x}. \ (S \land \llbracket B \rrbracket [\overline{X} \rightarrow \overline{e}]) \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle
\]

- **Hence**

\[
\text{SP } \langle \exists \overline{x}. \ S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle (\text{IF } B \text{ THEN } C_1 \text{ ELSE } C_2)
\]

\[
= \text{SP } \langle \exists \overline{x}. \ (S \land \llbracket B \rrbracket [\overline{X} \rightarrow \overline{e}]) \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \ C_1
\]

\[
\lor
\]

\[
\text{SP } \langle \exists \overline{x}. \ (S \land \neg \llbracket B \rrbracket [\overline{X} \rightarrow \overline{e}]) \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \ C_2
\]

- **Prune paths by checking** \( S \land \llbracket B \rrbracket [\overline{X} \rightarrow \overline{e}] \) and \( S \land \neg \llbracket B \rrbracket [\overline{X} \rightarrow \overline{e}] \)
Summary so far

► All one needs
  ► semantics of commands \([C]\)
  ► suitable theorem prover

► Define \(\{P\} C\{Q\}\) and \(SP\ P\ C\) from semantics

► Prove rules for calculating \(SP\ P\ C\) (one-off proof)

► For particular \(P, C, Q\) prove \(\{P\} C\{Q\}\) by
  ► calculating \(SP\ P\ C\) using rules and a theorem prover
  ► prove \(\models SP\ P\ C \Rightarrow Q\) using theorem prover

► Next: what about loops?
Method of verification conditions (VCs)

- Define **AWP** and **ASP** (“A” for “approximate”)
  - like **WP**, **SP** for skip, assignment, sequencing, conditional
  - for while-loops assume invariant $R$ magically supplied

$$\text{AWP (WHILE } B \text{ DO } \{ R \} \ C \} \ Q = R$$
$$\text{ASP } P (\text{WHILE } B \text{ DO } \{ R \} \ C \} = R \land \neg [B]$$

- Define **WVC C Q** and **SVC P C** to generate VCs
  (more details on next slide)

- Prove $\{P\} C \{Q\}$ using theorems

$$\text{WVC C Q } \Rightarrow \{\text{AWP C Q}\} C \{Q\}$$
$$\text{SVC P C } \Rightarrow \{P\} C \{\text{ASP P C}\}$$
Calculating verification conditions

- **WVC** $C \; Q$ is a standard ‘backwards’ calculation

  - $\text{WVC}(\text{SKIP}) \; Q = T$
  - $\text{WVC}(V := E) \; Q = T$
  - $\text{WVC}(C_1; C_2) \; Q = \text{WVC}(C_1) \; (\text{AWP} \; C_2 \; Q) \land \text{WVC}(C_2) \; Q$
  - $\text{WVC}(\text{IF } B \text{ THEN } C_1 \; \text{ELSE } C_2) \; Q = \text{WVC}(C_1) \; Q \land \text{WVC}(C_2) \; Q$
  - $\text{WVC}(\text{WHILE } B \text{ DO } \{R\} \; C) \; Q =$
    $\left(\models R \land \llbracket B \rrbracket \Rightarrow \text{AWP} \; C \; R\right) \land \left(\models R \land \neg \llbracket B \rrbracket \Rightarrow Q\right) \land \text{WVC}(C \; R)$

- **SVC** $P \; C$ is a ‘forwards’ calculation

  - $\text{SVC}(P) \; (\text{SKIP}) = T$
  - $\text{SVC}(P) \; (V := E) = T$
  - $\text{SVC}(P) \; (C_1; C_2) = \text{SVC}(P \; C_1) \land \text{SVC}(\text{ASP} \; P \; C_1) \; C_2$
  - $\text{SVC}(P \; (\text{IF } B \text{ THEN } C_1 \; \text{ELSE } C_2)) =$
    $\text{SVC}(P \land \llbracket B \rrbracket) \; C_1 \land \text{SVC}(P \land \neg \llbracket B \rrbracket) \; C_2$
  - $\text{SVC}(P \; (\text{WHILE } B \text{ DO } \{R\} \; C)) =$
    $\left(\models P \Rightarrow R\right) \land \left(\models \text{ASP} \; (R \land \llbracket B \rrbracket) \; C \Rightarrow R\right) \land \text{SVC}(R \land \llbracket B \rrbracket) \; C$
Calculating verification conditions

- **WVC C Q** is a standard ‘backwards’ calculation
  
  \[
  \text{WVC (SKIP)} Q = T \\
  \text{WVC (V := E)} Q = T \\
  \text{WVC (C₁; C₂)} Q = \text{WVC C₁ (AWP C₂ Q)} \land \text{WVC C₂ Q} \\
  \text{WVC (IF B THEN C₁ ELSE C₂)} Q = \text{WVC C₁ Q} \land \text{WVC C₂ Q} \\
  \text{WVC (WHILE B DO \{R\} C)} Q = \\
  (\models R \land [B] \Rightarrow \text{AWP C R}) \land (\models R \land \neg [B] \Rightarrow Q) \land \text{WVC C R}
  \]

- **SVC P C** is a ‘forwards’ calculation
  
  \[
  \text{SVC P (SKIP)} = T \\
  \text{SVC P (V := E)} = T \\
  \text{SVC P (C₁; C₂)} = \text{SVC P C₁} \land \text{SVC (ASP P C₁)} C₂ \\
  \text{SVC P (IF B THEN C₁ ELSE C₂)} = \\
  \text{SVC (P \land [B]) C₁} \land \text{SVC (P \land \neg [B]) C₂} \\
  \text{SVC P (WHILE B DO \{R\} C)} = \\
  (\models P \Rightarrow R) \land (\models \text{ASP (R \land [B]) C \Rightarrow R}) \land \text{SVC (R \land [B]) C}
  \]
Calculating verification conditions

- **WVC C Q** is a standard ‘backwards’ calculation
  
  \[
  \text{WVC (\text{SKIP}) } Q = T \\
  \text{WVC (V := E) } Q = T \\
  \text{WVC (C}_1; C_2) \ Q = \text{WVC C}_1 (\text{AWP C}_2 \ Q) \land \text{WVC C}_2 \ Q \\
  \text{WVC (IF B THEN C}_1 \ \text{ELSE C}_2) \ Q = \text{WVC C}_1 \ Q \land \text{WVC C}_2 \ Q \\
  \text{WVC (WHILE B DO \{R\} C) } Q = \\
  (\models R \land \lbrack B \rbrack \Rightarrow \text{AWP C} \ R) \land (\models R \land \neg \lbrack B \rbrack \Rightarrow Q) \land \text{WVC C} \ R
  \]

- **SVC P C** is a ‘forwards’ calculation
  
  \[
  \text{SVC P (\text{SKIP}) } = T \\
  \text{SVC P (V := E) } = T \\
  \text{SVC P (C}_1; C_2) = \text{SVC P C}_1 \land \text{SVC (ASP P C}_1) \ C_2 \\
  \text{SVC P (IF B THEN C}_1 \ \text{ELSE C}_2) = \\
  \text{SVC (P \land \lbrack B \rbrack) C}_1 \land \text{SVC (P \land \neg \lbrack B \rbrack) C}_2 \\
  \text{SVC P (WHILE B DO \{R\} C) } = \\
  (\models P \Rightarrow R) \land (\models \text{ASP (R \land \lbrack B \rbrack) C} \Rightarrow R) \land \text{SVC (R \land \lbrack B \rbrack) C}
  \]
Approximate symbolic execution of while-loops

- Symbolically execute straight line code as before

- For while-loops, recall from previous slide

\[ \text{ASP } P \left( \texttt{WHILE } B \texttt{ DO } \{ R \} \texttt{ C} \right) = R \land \neg \llbracket B \rrbracket \]

- Hence execute while-loops as follows

\[ \text{ASP } \langle \exists \overline{x}. \ S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \left( \texttt{WHILE } B \texttt{ DO } \{ R \} \texttt{ C} \right) = \langle \exists \overline{x}. \ (R \land \neg \llbracket B \rrbracket [\overline{X} \mapsto \overline{x}]) \land X_1 = x_1 \land \ldots \land X_n = x_n \rangle \]

  - constraint \( S \) computed up to loop is discarded
  - create new state satisfying invariant and loop exit condition
  - link between pre and post loop states provided by VCs

\[ \models \langle S \land X_1 = e_1 \land \ldots \land X_n = e_n \rangle \Rightarrow R \]
\[ \land \]
\[ \models \text{ASP} \langle (R \land \llbracket B \rrbracket) \land X_1 = x_1 \land \ldots \land X_n = x_n \rangle \texttt{ C} \Rightarrow R \]
Pretty slides hide messy HOL details!

- Term \( \lambda s. \exists x_1 \cdots x_n. S s \wedge (s = [X \rightarrow \overline{e}]) \) is for a given \( X \)

- The rule

  \[
  \text{SP } \langle \exists \overline{x}. S \wedge x_1 = e_1 \land \ldots \land x_n = e_n \rangle \ (X_i := E_i) \\
  = \langle \exists \overline{x}. S \wedge x_1 = e_1 \land \ldots \land x_i = [E_i] \ [X \rightarrow \overline{e}] \wedge \ldots \wedge x_n = e_n \rangle
  \]

  is also for a given \( X_1, \ldots, X_n \)

- HOL theorem generating specific assignment rule is:

  \[
  |- \ \forall x_1 f P v e. \\
  \text{ALL\_DISTINCT } x_1 \Rightarrow \\
  (\forall l. \ (\text{MAP FST } l = x_1) \Rightarrow (\text{MAP FST } (f \ l) = x_1)) \Rightarrow \\
  (\text{LP} \\
  \begin{array}{l}
  x_1 \\
  (\lambda s. \exists l. (\text{MAP FST } l = x_1) \land P l \land (s = \text{FEMPTY } |++ f l)) \\
  (v ::= e) = \\
  (\lambda s. \exists l. \\
  (\text{MAP FST } l = x_1) \land P l \land \\
  (s = \text{FEMPTY } |++ (\text{ASSIGN\_FUN } v e o f) \ l)))
  \end{array}
  \]

- Won’t rexplain this here beyond:
  - \( \text{LP} \) represents \( \text{SP} \)
  - \( \exists l \) instantiated to \( \exists x_1 \ldots x_n \) for a specific program
THE END
THE END

Really!