Forward with Hoare!

2009: Tony Hoare is 75 and Hoare Logic is 40!

An Axiomatic Basis for
Computer Programming

C. A. R. Hoare, 1969

Overview of talk:

- Review of Hoare Logic
- Mechanical proof
- Forwards versus backwards

[Slides that follow are based on joint work with Hélène Collavizza]

Hoare's Axiomatic Basis for Computer Programming

- Originally both
  - an axiomatic language definition method and
  - a proof theory for program verification
- This talk focuses on the verification role
  - 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

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 decidable properties
- Full proof of correctness
  - add invariants to loops
  - generate verification conditions
  - prove verification conditions with a theorem prover
- Goal: unifying framework for a spectrum of methods

decidable checking *proof of correctness*

Some history: concepts related to $\{P\}\ C\ \{Q\}$

- $WP\ C\ Q$ is Dijkstra's 'weakest liberal precondition'
  (i.e. partial correctness: `wlp.C.Q` from Dijkstra & Scholten)
  - precondition $WP\ C\ Q$ ensures Q holds after C terminates
  - $wlp.C.Q$ is weakest solution of $P : (\{P\}\ C\ \{Q\})$
    (*Predicate Calculus & Program Semantics*, Dijkstra & Scholten, 1990)

- $SP\ C\ P$ is 'strongest postcondition'
  (`sp.C.Q` in Dijkstra & Scholten, Ch.12 – not `stp.C.Q`)
  - $SP\ C\ P$ holds after C terminates if started when P holds
  - $sp.C.P$ is strongest solution of $Q : (\{P\}\ C\ \{Q\})$

Defining specification notions by semantic embedding

- Semantics of commands C given by binary relation $[\![C]\!]$
  - $[\![C]\!](s, s')$ means if C run in s then it will terminate in s'
  - s is the initial state; s' is a final state
  - commands assumed deterministic – at most one final state ("Formalizing Dijkstra" by J. Harrison for non-determinism)

- $\{P\}C\{Q\} \ =_{def} \ \forall s\ s'.\ P\ s \wedge [\![C]\!](s, s') \Rightarrow Q\ s'$
- $WP\ C\ Q \ =_{def} \ \lambda s.\ \forall s'.\ [\![C\ ]\!](s, s') \Rightarrow Q\ s'$
- $\vdash \{P\}C\{Q\} \ = \ \forall s.\ P\ s \Rightarrow WP\ C\ Q\ s$
- $SP\ C\ P \ =_{def} \ \lambda s'.\ \exists s.\ P\ s \wedge [\![C]\!](s, s')$
- $\vdash \{P\}C\{Q\} \ = \ \forall s.\ SP\ C\ P\ s \Rightarrow Q\ s$

Details and notations

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

- Overload $\wedge$, $\vee$, $\Rightarrow$, $\neg$ to pointwise operations on predicates
  - $(P_1 \wedge P_2)\ s = P_1\ s \wedge P_2\ s$
  - $(P_1 \vee P_2)\ s = P_1\ s \vee P_2\ s$
  - $(P_1 \Rightarrow P_2)\ s = P_1\ s \Rightarrow P_2\ s$
  - $(\neg P)\ s = \neg(P\ s)$

- Define: $\mathrm{TAUT}(P) \;=_{def}\; \forall s.\ P\ s$ and $\mathrm{SAT}(P) \;=_{def}\; \exists s.\ P\ s$

Proving $\{P\}C\{Q\}$ by calculating $WP\ C\ Q$

- Easy consequences of definition of WP
  - $WP\ (\mathtt{SKIP})\ Q\ =\ Q$
  - $WP\ (X\ {:=}\ E)\ Q\ =\ \lambda s.\ Q(s[X \mapsto [\![E]\!]s])$
  - $WP\ (C_1 ; C_2)\ Q\ =\ WP\ C_1\ (WP\ C_2\ Q)$
  - $WP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ =$
    $([\![B]\!] \Rightarrow WP\ C_1\ Q)\ \wedge\ (\neg[\![B]\!] \Rightarrow WP\ C_2\ Q)$
  - $WP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q\ =$
    $([\![B]\!] \Rightarrow WP\ C\ (WP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q))\ \wedge\ (\neg[\![B]\!] \Rightarrow Q)$
- To prove $\{P\}C\{Q\}$ for straight line code
  - calculate $WP\ C\ Q$ .... back substitution + case splits
  - prove $\forall s.\ P\ s \Rightarrow WP\ C\ Q\ s$ ....use a theorem prover

Proving $\{P\}C\{Q\}$ by calculating $SP\ C\ P$

- Easy consequences of definition of SP
  - $SP\ \mathtt{SKIP}\ P\ =\ P$
  - $SP\ (X\ \mathtt{:=}\ E)\ P\ =\ \lambda s'.\ \exists s.\ P\ s \land (s'\ =\ s[X{\to}[\![E]\!]s])$
  - $SP\ (C_1\mathtt{;}C_2)\ P\ =\ SP\ C_2\ (SP\ P\ C_1)$
  - $SP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P\ =$
    $SP\ C_1\ (P \land [\![B]\!])\ \lor\ SP\ C_2\ (P \land \neg[\![B]\!])$
  - $SP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ P\ =$
    $SP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ (SP\ (P \land [\![B]\!])\ C)\ \lor\ (P \land \neg[\![B]\!])$
- To prove $\{P\}C\{Q\}$ for straight line code
  - calculate $SP\ P\ C$ ..... assignment generated $\exists$s a problem
  - prove $\forall s'.\ SP\ C\ P\ s' \Rightarrow Q\ s'$ ....use a theorem prover

Pruning conditional branches when going forwards

- Recall

  $$\begin{array}{l} SP\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ P\ = \\ \quad SP\ C_1\ (P \wedge [\![B]\!])\ \ \vee\ \ SP\ C_2\ (P \wedge \neg[\![B]\!]) \end{array}$$

- Because $SP\ C\ (\lambda s.\ \mathrm{F})\ =\ \lambda s'.\ \mathrm{F}$ it follows that

  $$\begin{array}{l} (P \Rightarrow [\![B]\!]) \\ \Rightarrow \\ SP\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ P\ =\ SP\ C_1\ (P \wedge [\![B]\!]) \end{array}$$

  $$\begin{array}{l} (P \Rightarrow \neg[\![B]\!]) \\ \Rightarrow \\ SP\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ P\ =\ SP\ C_2\ (P \wedge \neg[\![B]\!]) \end{array}$$

- Hence can simplify if accumulated constraints implies test

Pruning conditional branches when going backwards

- Recall

  $$WP\,(\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ Q \ = \ (\llbracket B \rrbracket \Rightarrow WP\ C_1\ Q) \ \wedge \ (\neg\llbracket B \rrbracket \Rightarrow WP\ C_2\ Q)$$

- Hence

  $$(\llbracket B \rrbracket \Rightarrow WP\ C_1\ Q) \Rightarrow WP\,(\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ Q \ = \ (\neg\llbracket B \rrbracket \Rightarrow WP\ C_2\ Q)$$

  $$(\neg\llbracket B \rrbracket \Rightarrow WP\ C_2\ Q) \Rightarrow WP\,(\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ Q \ = \ (\llbracket B \rrbracket \Rightarrow WP\ C_1\ Q)$$

- Backwards pruning conditions involve C_1 or C_2
  - forwards pruning natural – generalised execution
  - forwards pruning conditions don't involve C_1 or C_2

Backwards or forwards?

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

$$\{J \leq I\}$$
$$K \ \mathtt{:=}\ \mathtt{0;}$$
$$\mathtt{IF}\ I < J\ \mathtt{THEN}\ K\ \mathtt{:=}\ K + 1\ \mathtt{ELSE\ SKIP;}$$
$$\mathtt{IF}\ K = 1 \wedge \neg(I = J)\ \mathtt{THEN}\ R\ \mathtt{:=}\ J - I\ \mathtt{ELSE}\ R\ \mathtt{:=}\ I - J$$
$$\{R = I - J\}$$

Backwards or forwards?

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

$\{J \leq I\}$

$K := 0\texttt{;}$

$\texttt{IF}\ I < J\ \texttt{THEN}\ K := K + 1\ \texttt{ELSE SKIP;}$

$\texttt{IF}\ K = 1 \wedge \neg(I = J)\ \texttt{THEN}\ R := J - I\ \texttt{ELSE}\ R := I - J$

$\{R = I - J\}$

Backwards or forwards?

- Calculating $WP\ C\ Q$ is easy but leads to big formulae
  - can't use symbolic state to prune case splits 'on-the-fly'
- Calculating $SP\ C\ P$ generates $\exists$ at assignments
  - at branches symbolic state 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
- Example

$\{J \leq I\}$
K := 0; $\{J \leq I \land K = 0\}$
IF $I < J$ THEN K := $K + 1$ ELSE SKIP;
IF $K = 1 \land \neg(I = J)$ 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 use symbolic state to prune case splits 'on-the-fly'
- Calculating $SP\ C\ P$ generates $\exists$ at assignments
  - at branches symbolic state can reject infeasible paths
- Consider $\{P\}C_1\texttt{;}(\texttt{IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3)\texttt{;}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Example

$\{J \leq I\}$

$K := 0\texttt{;}$ $\{J \leq I \wedge K = 0\}$

$\texttt{IF}\ I < J\ \texttt{THEN}\ K := K + 1\ \texttt{ELSE SKIP;}$ $\{J \leq I \wedge K = 0\}$

$\texttt{IF}\ K = 1 \wedge \neg(I = J)\ \texttt{THEN}\ R := J - I\ \texttt{ELSE}\ R := I - J$

$\{R = I - J\}$

Backwards or forwards?

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

$\{J \leq I\}$

K := 0; $\{J \leq I \wedge K = 0\}$

IF $I < J$ THEN K := $K + 1$ ELSE SKIP; $\{J \leq I \wedge K = 0\}$

IF $K = 1 \wedge \neg(I = J)$ 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 use symbolic state to prune case splits 'on-the-fly'
- Calculating $SP\ C\ P$ generates $\exists$ at assignments
  - at branches symbolic state can reject infeasible paths
- Consider $\{P\}C_1\texttt{;(IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3\texttt{);}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Example

$$\{I < J\}$$
$$K\ \texttt{:=}\ 0\texttt{;}$$
$$\texttt{IF}\ I < J\ \texttt{THEN}\ K\ \texttt{:=}\ K + 1\ \texttt{ELSE SKIP;}$$
$$\texttt{IF}\ K = 1 \wedge \neg(I = J)\ \texttt{THEN}\ R\ \texttt{:=}\ J - I\ \texttt{ELSE}\ R\ \texttt{:=}\ I - J$$
$$\{R = J - I\}$$

Backwards or forwards?

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

$$\{I < J\}$$

$$K\ \texttt{:=}\ 0\texttt{;}$$

$$\texttt{IF}\ I < J\ \texttt{THEN}\ K\ \texttt{:=}\ K+1\ \texttt{ELSE SKIP;}$$

$$\texttt{IF}\ K = 1 \wedge \neg(I = J)\ \texttt{THEN}\ R\ \texttt{:=}\ J - I\ \texttt{ELSE}\ R\ \texttt{:=}\ I - J$$

$$\{R = J - I\}$$

Backwards or forwards?

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

$\{I < J\}$

K := 0; $\{I < J \wedge K = 0\}$

IF $I < J$ THEN K := $K + 1$ ELSE SKIP;

IF $K = 1 \wedge \neg(I = J)$ THEN R := $J - I$ ELSE R := $I - J$

$\{R = J - I\}$

Backwards or forwards?

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

$\{I < J\}$

$K := 0\texttt{;}$ $\{I < J \wedge K = 0\}$

$\texttt{IF}\ I < J\ \texttt{THEN}\ K := K + 1\ \texttt{ELSE SKIP;}$ $\{I < J \wedge K = 1\}$

$\texttt{IF}\ K = 1 \wedge \neg(I = J)\ \texttt{THEN}\ R := J - I\ \texttt{ELSE}\ R := I - J$

$\{R = J - I\}$

Backwards or forwards?

- Calculating $\mathit{WP}\ C\ Q$ is easy but leads to big formulae
  - can't use symbolic state to prune case splits 'on-the-fly'
- Calculating $\mathit{SP}\ C\ P$ generates $\exists$ at assignments
  - at branches symbolic state can reject infeasible paths
- Consider $\{P\}C_1\texttt{;}(\texttt{IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3)\texttt{;}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Example

$\{I < J\}$

$K := 0\texttt{;}$ $\quad \{I < J \land K = 0\}$

$\texttt{IF}\ I < J\ \texttt{THEN}\ K := K + 1\ \texttt{ELSE SKIP;}$ $\quad \{I < J \land K = 1\}$

$\texttt{IF}\ K = 1 \land \neg(I = J)\ \texttt{THEN}\ R := J - I\ \texttt{ELSE}\ R := I - J$

$\{R = J - I\}$

Summary so far

- Define $\{P\}C\{Q\}$, $WP\ C\ Q$ and $SP\ C\ P$ from semantics
- Prove rules for calculating $WP\ C\ Q$ and $SP\ C\ P$
  - one-off proofs
- For particular P, C, Q, to prove $\{P\}C\{Q\}$:
  - calculate $WP\ C\ Q$ by backwards substitution
  - prove $\forall s.\ P\ s \Rightarrow WP\ C\ Q\ s$ using theorem prover

  or

  - calculate $SP\ C\ P$ by symbolic execution
  - prove $\forall s'.\ SP\ C\ P\ s' \Rightarrow Q\ s'$ using theorem prover
- Next: what about loops?

Can't compute finite *WP* or *SP* for loops

- Loop-free: can calculate finite formulae for *WP* and *SP*
- Loops: no simple finite formula for *WP* or *SP* in general
  - $WP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q\ =$
    $(\llbracket B \rrbracket \wedge WP\ C\ (WP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q))\ \vee\ (\neg\llbracket B \rrbracket \wedge Q)$
  - $SP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ P\ =$
    $(SP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ (SP\ C\ (P \wedge \llbracket B \rrbracket)))\ \ \vee\ \ (P \wedge \neg\llbracket B \rrbracket)$
- Solution inspired by Hoare logic rule (*R* is an invariant)

$$\frac{\vdash P \Rightarrow R \qquad \vdash \{R \wedge B\}C\{R\} \qquad \vdash R \wedge \neg B \Rightarrow Q}{\vdash \{P\}\mathtt{WHILE}\ B\ \mathtt{DO}\ C\{Q\}}$$

- 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

$$\begin{aligned} AWP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ Q &= R \\ ASP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P &= R \wedge \neg [\![B]\!] \end{aligned}$$

- Define $WVC\ C\ Q$ and $SVC\ C\ P$ to generate VCs (details later)
- Prove $\{P\}C\{Q\}$ using theorems

$$\begin{aligned} WVC\ C\ Q &\Rightarrow \{AWP\ C\ Q\}C\{Q\} \\ SVC\ C\ P &\Rightarrow \{P\}C\{ASP\ C\ P\} \end{aligned}$$

- If C is loop-free (i.e. straight line code) then this becomes

$$\begin{aligned} \mathrm{T} &\Rightarrow \{WP\ C\ Q\}C\{Q\} \\ \mathrm{T} &\Rightarrow \{P\}C\{SP\ C\ P\} \end{aligned}$$

A problem

- Have $SP\ C\ (\lambda s.\ \mathrm{F}) \ = \ (\lambda s'.\ \mathrm{F})$ so can reduce
  $SP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P$
  to
  $SP\ C_1\ (P \wedge [\![B]\!])$ or $SP\ C_2\ (P \wedge \neg[\![B]\!])$
  if P determines value of $[\![B]\!]$
- But $ASP\ C\ (\lambda s.\ \mathrm{F})$ is not necessarily $(\lambda s'.\ \mathrm{F})$
  $ASP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P \ = \ R \ \wedge \ \neg[\![B]\!]$
  so cannot reduce $ASP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P$
- A solution is to define
  $ASP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P \ =$
  $\lambda s'.\ \mathrm{SAT}(P) \wedge R\ s' \wedge \neg([\![B]\!]\ s')$
- Can then show $ASP\ C\ (\lambda s.\ \mathrm{F}) \ = \ (\lambda s'.\ \mathrm{F})$
- A dual argument suggests defining
  $AWP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q \ = \ \lambda s.\ \mathrm{SAT}(\neg Q) \ \Rightarrow \ R\ s$
  (note: $\mathrm{SAT}(\neg Q) = \neg(\mathrm{TAUT}(Q))$)

A problem

- Have $SP\ C\ (\lambda s.\ \mathrm{F}) = (\lambda s'.\ \mathrm{F})$ so can reduce

  $SP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P$

  to

  $SP\ C_1\ (P \wedge [\![B]\!])$ or $SP\ C_2\ (P \wedge \neg[\![B]\!])$

  if P determines value of $[\![B]\!]$

- But $ASP\ C\ (\lambda s.\ \mathrm{F})$ is not necessarily $(\lambda s'.\ \mathrm{F})$

  $ASP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P \ = \ R \ \wedge \ \neg[\![B]\!]$

  so cannot reduce $ASP\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P$

- A solution is to define

  $ASP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P \ =$
  $\lambda s'.\ \mathrm{SAT}(P) \wedge R\ s' \wedge \neg([\![B]\!]\ s')$

- Can then show $ASP\ C\ (\lambda s.\ \mathrm{F}) = (\lambda s'.\ \mathrm{F})$

- A dual argument suggests defining

  $AWP\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q \ = \ \lambda s.\ \mathrm{SAT}(\neg Q) \Rightarrow R\ s$

  (note: $\mathrm{SAT}(\neg Q) = \neg(\mathrm{TAUT}(Q))$)

Summary: definitions of *ASP* and *AWP*

$$\mathit{ASP}\ \texttt{SKIP}\ P \ = \ P$$

$$\mathit{ASP}\ (X\ \texttt{:=}\ E)\ P \ = \ \lambda s'.\ \exists s.\ P\ s \wedge (s' = s[X \mapsto [\![E]\!]s])$$

$$\mathit{ASP}\ (C_1\texttt{;}C_2)\ P \ = \ \mathit{ASP}\ C_2\ (\mathit{ASP}\ C_1\ P)$$

$$\mathit{ASP}\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ P \ = \ \mathit{SP}\ C_1\ (P \wedge [\![B]\!]) \ \vee \ \mathit{SP}\ C_2\ (P \wedge \neg[\![B]\!])$$

$$\mathit{ASP}\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P \ = \ \lambda s'.\ \mathrm{SAT}(P) \wedge R\ s' \wedge \neg([\![B]\!]\,s')$$

$$\mathit{AWP}\ \texttt{SKIP}\ Q \ = \ Q$$

$$\mathit{AWP}\ (X\ \texttt{:=}\ E)\ Q \ = \ \lambda s.\ Q(s[X \mapsto [\![E]\!]\,s])$$

$$\mathit{AWP}\ (C_1\texttt{;}C_2)\ Q \ = \ \mathit{AWP}\ C_1\ (\mathit{AWP}\ C_2\ Q)$$

$$\mathit{AWP}\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ Q \ = \ ([\![B]\!] \Rightarrow \mathit{WP}\ C_1\ Q) \ \wedge \ (\neg[\![B]\!] \Rightarrow \mathit{WP}\ C_2\ Q)$$

$$\mathit{AWP}\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ Q \ = \ \lambda s.\ \mathrm{SAT}(\neg Q) \ \Rightarrow \ R\ s$$

Calculating verification conditions

- $\mathit{SVC}\ P\ C$ is a 'forwards' calculation

$$\mathit{SVC}\ \mathtt{SKIP}\ P\ =\ \mathtt{T}$$

$$\mathit{SVC}\ (X := E)\ P\ =\ \mathtt{T}$$

$$\mathit{SVC}\ (C_1 ; C_2)\ P\ =\ \mathit{SVC}\ C_1\ P \wedge \mathit{SVC}\ C_2\ (\mathit{ASP}\ C_1\ P)$$

$$\begin{aligned}&\mathit{SVC}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P\ =\\ &\quad \mathrm{SAT}(P \wedge [\![B]\!]) \Rightarrow \mathit{SVC}\ C_1\ (P \wedge [\![B]\!])\ \wedge\\ &\quad \mathrm{SAT}(P \wedge \neg[\![B]\!]) \Rightarrow \mathit{SVC}\ C_2\ (P \wedge \neg[\![B]\!])\end{aligned}$$

$$\begin{aligned}&\mathit{SVC}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P\ =\\ &\quad \mathrm{TAUT}(P \Rightarrow R)\ \wedge\ \mathrm{TAUT}(\mathit{ASP}\ C\ (R \wedge [\![B]\!]) \Rightarrow R)\ \wedge\ \mathit{SVC}\ C\ (R \wedge [\![B]\!])\end{aligned}$$

- $\mathit{WVC}\ C\ Q$ is a standard 'backwards' calculation

$$\mathit{WVC}\ (\mathtt{SKIP})\ Q\ =\ \mathtt{T}$$

$$\mathit{WVC}\ (X := E)\ Q\ =\ \mathtt{T}$$

$$\mathit{WVC}\ (C_1 ; C_2)\ Q\ =\ \mathit{WVC}\ C_1\ (\mathit{AWP}\ C_2\ Q) \wedge \mathit{WVC}\ C_2\ Q$$

$$\begin{aligned}&\mathit{WVC}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ =\\ &\quad \mathrm{TAUT}(Q)\ \vee\ (\mathit{WVC}\ C_1\ Q \wedge \mathit{WVC}\ C_2\ Q)\end{aligned}$$

$$\begin{aligned}&\mathit{WVC}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q\ =\\ &\quad \mathrm{TAUT}(R \wedge [\![B]\!] \Rightarrow \mathit{AWP}\ C\ R) \wedge \mathrm{TAUT}(R \wedge \neg[\![B]\!] \Rightarrow Q) \wedge \mathit{WVC}\ C\ R\end{aligned}$$

Calculating verification conditions

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

$$SVC\ \mathtt{SKIP}\ P\ =\ \mathtt{T}$$

$$SVC\ (X\ \mathtt{:=}\ E)\ P\ =\ \mathtt{T}$$

$$SVC\ (C_1\mathtt{;}C_2)\ P\ =\ SVC\ C_1\ P \wedge SVC\ C_2\ (ASP\ C_1\ P)$$

$$\begin{array}{l} SVC\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P\ = \\ \quad \mathrm{SAT}(P \wedge [\![B]\!]) \ \Rightarrow SVC\ C_1\ (P \wedge [\![B]\!])\ \wedge \\ \quad \mathrm{SAT}(P \wedge \neg[\![B]\!]) \Rightarrow SVC\ C_2\ (P \wedge \neg[\![B]\!]) \end{array}$$

$$\begin{array}{l} SVC\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P\ = \\ \quad \mathrm{TAUT}(P \Rightarrow R)\ \wedge\ \mathrm{TAUT}(ASP\ C\ (R \wedge [\![B]\!]) \Rightarrow R)\ \wedge\ SVC\ C\ (R \wedge [\![B]\!]) \end{array}$$

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

Calculating verification conditions

- $SVC\ P\ C$ is a 'forwards' calculation

$$SVC\ \mathtt{SKIP}\ P\ =\ \mathtt{T}$$

$$SVC\ (X\ \mathtt{:=}\ E)\ P\ =\ \mathtt{T}$$

$$SVC\ (C_1\mathtt{;}C_2)\ P\ =\ SVC\ C_1\ P \wedge SVC\ C_2\ (ASP\ C_1\ P)$$

$$\begin{array}{l} SVC\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P\ = \\ \quad \mathrm{SAT}(P \wedge [\![B]\!]) \ \ \Rightarrow SVC\ C_1\ (P \wedge [\![B]\!])\ \wedge \\ \quad \mathrm{SAT}(P \wedge \neg[\![B]\!]) \Rightarrow SVC\ C_2\ (P \wedge \neg[\![B]\!]) \end{array}$$

$$\begin{array}{l} SVC\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P\ = \\ \quad \mathrm{TAUT}(P \Rightarrow R)\ \wedge\ \mathrm{TAUT}(ASP\ C\ (R \wedge [\![B]\!]) \Rightarrow R)\ \wedge\ SVC\ C\ (R \wedge [\![B]\!]) \end{array}$$

- $WVC\ C\ Q$ is a standard 'backwards' calculation

$$WVC\ (\mathtt{SKIP})\ Q\ =\ \mathtt{T}$$

$$WVC\ (X\ \mathtt{:=}\ E)\ Q\ =\ \mathtt{T}$$

$$WVC\ (C_1\mathtt{;}C_2)\ Q\ =\ WVC\ C_1\ (AWP\ C_2\ Q) \wedge WVC\ C_2\ Q$$

$$\begin{array}{l} WVC\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ = \\ \quad \mathrm{TAUT}(Q)\ \vee\ (WVC\ C_1\ Q \wedge WVC\ C_2\ Q) \end{array}$$

$$\begin{array}{l} WVC\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q\ = \\ \quad \mathrm{TAUT}(R \wedge [\![B]\!] \Rightarrow AWP\ C\ R)\ \wedge\ \mathrm{TAUT}(R \wedge \neg[\![B]\!] \Rightarrow Q)\ \wedge\ WVC\ C\ R \end{array}$$

Calculating verification conditions

- $SVC\ P\ C$ is a 'forwards' calculation

$$
\begin{array}{l}
SVC\ \mathtt{SKIP}\ P\ =\ \mathrm{T} \\
SVC\ (X\ \mathtt{:=}\ E)\ P\ =\ \mathrm{T} \\
SVC\ (C_1\mathtt{;}C_2)\ P\ =\ SVC\ C_1\ P \wedge SVC\ C_2\ (ASP\ C_1\ P) \\
SVC\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ P\ = \\
\quad \mathrm{SAT}(P \wedge [\![B]\!]) \ \Rightarrow SVC\ C_1\ (P \wedge [\![B]\!])\ \wedge \\
\quad \mathrm{SAT}(P \wedge \neg[\![B]\!]) \Rightarrow SVC\ C_2\ (P \wedge \neg[\![B]\!]) \\
SVC\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ P\ = \\
\quad \mathrm{TAUT}(P \Rightarrow R)\ \wedge\ \mathrm{TAUT}(ASP\ C\ (R \wedge [\![B]\!]) \Rightarrow R)\ \wedge\ SVC\ C\ (R \wedge [\![B]\!])
\end{array}
$$

- $WVC\ C\ Q$ is a standard 'backwards' calculation

$$
\begin{array}{l}
WVC\ (\mathtt{SKIP})\ Q\ =\ \mathrm{T} \\
WVC\ (X\ \mathtt{:=}\ E)\ Q\ =\ \mathrm{T} \\
WVC\ (C_1\mathtt{;}C_2)\ Q\ =\ WVC\ C_1\ (AWP\ C_2\ Q) \wedge WVC\ C_2\ Q \\
WVC\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ = \\
\quad \mathrm{TAUT}(Q)\ \vee\ (WVC\ C_1\ Q \wedge WVC\ C_2\ Q) \\
WVC\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q\ = \\
\quad \mathrm{TAUT}(R \wedge [\![B]\!] \Rightarrow AWP\ C\ R)\ \wedge\ \mathrm{TAUT}(R \wedge \neg[\![B]\!] \Rightarrow Q)\ \wedge\ WVC\ C\ R
\end{array}
$$

Symbolic execution of loops

$$\boxed{ASP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ =\ \lambda s'.\ \textsf{SAT}(P) \wedge R\ s' \wedge \neg(\llbracket B \rrbracket\, s')}$$

- New state satisfying invariant R and loop-exit condition
- Pre and post loop states linked by verification conditions
  $SVC\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ =$
  $\textsf{TAUT}(P \Rightarrow R)\ \wedge\ \textsf{TAUT}(ASP\ C\ (R \wedge \llbracket B \rrbracket) \Rightarrow R)\ \wedge\ SVC\ C\ (R \wedge \llbracket B \rrbracket)$
- Various approaches to symbolic execution:
  - generate fresh set of state variables
    (need some metatheoretic proof of correctness)
  - manage variable scopes inside logic using $\exists$
    (correct-by-construct, but inefficient)
- Question (Plotkin)
  - is there a semantics characterisation of AWP and ASP ?

Symbolic execution of loops

$$\boxed{ASP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ =\ \lambda s'.\ \textsc{sat}(P) \wedge R\ s' \wedge \neg(\llbracket B \rrbracket\, s')}$$

- New state satisfying invariant R and loop-exit condition
- Pre and post loop states linked by verification conditions
  $SVC\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ =$
  $\mathrm{TAUT}(P \Rightarrow R)\ \wedge\ \mathrm{TAUT}(ASP\ C\ (R \wedge \llbracket B \rrbracket) \Rightarrow R)\ \wedge\ SVC\ C\ (R \wedge \llbracket B \rrbracket)$
- Various approaches to symbolic execution:
  - generate fresh set of state variables
    (need some metatheoretic proof of correctness)
  - manage variable scopes inside logic using $\exists$
    (correct-by-construct, but inefficient)
- Question (Plotkin)
  - is there a semantics characterisation of *AWP* and *ASP* ?

Shallow embedding of symbolic execution in logic

- $\vdash SP\,(X\ := E)\ P\ =\ \lambda s'.\ \exists s.\ P\ s \wedge (s'\ =\ s[X \to [\![E]\!]\,s])$
- Consider P of form

  $\lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X} \to \overline{e}\,])$

  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
  - $[\overline{X} \to \overline{e}\,]$ abbreviates $[X_1 \to e_1; \ldots; X_n \to e_n]$

- It follows that

  $$\vdash SP\,(X_i\ := E_i)\,(\lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X} \to \overline{e}\,]\,))$$
  $$=\ \lambda s. \exists x_1 \cdots x_n. S \wedge (s = [\overline{X} \to \overline{e}\,][X_i \to ([\![E_i]\!]\,[\overline{X} \to \overline{e}\,])]\,)$$

  where

  - $$[\overline{X} \to \overline{e}\,][X_i \to ([\![E_i]\!]\,[\overline{X} \to \overline{e}\,])]$$
    $$= [X_1 \to e_1, \ldots, X_i \to ([\![E_i]\!]\,[\overline{X} \to \overline{e}\,]), \ldots, X_n \to e_n]$$

Symbolic state notation for predicates

- Abbreviate

  $\lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X} \rightarrow \overline{e}\,])$

  as

  $\langle \exists \overline{x}.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle$

  then it follows that

  $SP\ (X_i\ \texttt{:=}\ E_i)\ \langle \exists \overline{x}.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle$
  $=\ \langle \exists \overline{x}.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_i{=}[\![E_i]\!]\,[\overline{X} \rightarrow \overline{e}\,] \wedge \ldots \wedge X_n{=}e_n \rangle$

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

- Simpler symbolic state represention OK for loop-free code

Symbolic state notation for predicates

- Abbreviate

  $\lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X} \to \overline{e}\,])$

  as

  $\langle \exists \overline{x}.\ S \wedge X_1 {=} e_1 \wedge \ldots \wedge X_n {=} e_n \rangle$

  then it follows that

  $SP\ (X_i\ \texttt{:=}\ E_i)\ \langle \exists \overline{x}.\ S \wedge X_1 {=} e_1 \wedge \ldots \wedge X_n {=} e_n \rangle$
  $=\ \langle \exists \overline{x}.\ S \wedge X_1 {=} e_1 \wedge \ldots \wedge X_i {=} [\![E_i]\!]\,[\overline{X} \to \overline{e}\,] \wedge \ldots \wedge X_n {=} e_n \rangle$

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

- Simpler symbolic state represention OK for loop-free code

Symbolic execution of conditional branches

- Recall

  $SP\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ P$
  $= SP\ C_1\ (P \wedge [\![B]\!])\ \vee\ SP\ C_2\ (P \wedge \neg[\![B]\!])$

- Now

$$
\begin{aligned}
&\langle \exists \overline{x}.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle\ \wedge\ [\![B]\!] \\
&= (\lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X}{\rightarrow}\overline{e}\,]))\ \wedge\ [\![B]\!] \\
&= \lambda s.\ (\exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X}{\rightarrow}\overline{e}\,]))\ \wedge\ [\![B]\!]\,s \\
&= \lambda s.\ \exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X}{\rightarrow}\overline{e}\,])\ \wedge\ [\![B]\!]\,s \\
&= \lambda s.\ (\exists x_1 \cdots x_n.\ S\ \wedge (s = [\overline{X}{\rightarrow}\overline{e}\,])\ \wedge\ [\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,] \\
&= \lambda s.\ \exists x_1 \cdots x_n.\ (S \wedge [\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,])\ \wedge (s = [\overline{X}{\rightarrow}\overline{e}\,]) \\
&= \langle \exists \overline{x}.\ (S \wedge [\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle
\end{aligned}
$$

- Hence

$$
\begin{aligned}
&SP\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ \langle \exists \overline{x}.\ S \wedge\ X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle \\
&= SP\ C_1\ \langle \exists \overline{x}.\ (S \wedge [\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle \\
&\quad \vee \\
&\quad SP\ C_2\ \langle \exists \overline{x}.\ (S \wedge \neg[\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle
\end{aligned}
$$

- Prune paths by checking $S \wedge [\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,]$ and $S \wedge \neg[\![B]\!]\,[\overline{X}{\rightarrow}\overline{e}\,]$

Approximate symbolic execution of while-loops

- Symbolically execute straight line code as before
- For while-loops, recall from previous slide

$$ASP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ =\ \lambda s'.\ \mathrm{SAT}(P) \wedge R\ s' \wedge \neg([\![B]\!]\,s')$$

- Hence execute while-loops as follows

$$\begin{array}{l} ASP\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ \langle \exists \overline{x}.\ S \wedge\ X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n\rangle \\ \quad = \langle \exists \overline{x}.\ ((\exists \overline{x}.\ S\ \overline{x}) \wedge R[\overline{X}{\to}\overline{x}\,] \wedge \neg[\![B]\!]\,[\overline{X}{\to}\overline{x}\,]) \\ \qquad\qquad \wedge \\ \qquad\qquad X_1{=}x_1 \wedge \ldots \wedge X_n{=}x_n\rangle \end{array}$$

  - 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

$$\begin{array}{l} SVC\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)\ P\ = \\ \quad \mathrm{TAUT}(P{\Rightarrow}R)\ \wedge\ \mathrm{TAUT}(ASP\ C\ (R{\wedge}[\![B]\!]){\Rightarrow}R)\ \wedge\ SVC\ C\ (R{\wedge}[\![B]\!]) \end{array}$$

Two cultures have evolved from Floyd-Hoare ideas

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute forwards
  - essentially $SP\ C\ P \Rightarrow Q$
  - automatically check states reached satisfy properties
- Full proof of correctness
  - generate verification conditions
  - usually backwards by computing weakest preconditions
  - essentially $P \Rightarrow WP\ C\ Q$
  - interactively prove resulting subgoal formulae
- Computing postconditions unifies BMC and full verification
  - symbolic execution is ASP calculation
  - add forward VCs for verification of loops
- Other application of Floyd-Hoare ideas
  - *refinement*:
    synthesize code to achive a postcondition (WP)
  - *reverse engineering*:
    execute symbolically to find out what code does (SP)

Two cultures have evolved from Floyd-Hoare ideas

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute forwards
  - essentially $SP\ C\ P \Rightarrow Q$
  - automatically check states reached satisfy properties
- Full proof of correctness
  - generate verification conditions
  - usually backwards by computing weakest preconditions
  - essentially $P \Rightarrow WP\ C\ Q$
  - interactively prove resulting subgoal formulae
- Computing postconditions unifies BMC and full verification
  - symbolic execution is *ASP* calculation
  - add forward VCs for verification of loops
- Other application of Floyd-Hoare ideas
  - *refinement*:
    synthesize code to achive a postcondition (*WP*)
  - *reverse engineering*:
    execute symbolically to find out what code does (*SP*)

Overview of 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 (future: Z3)
  - SMT solver from SRI International
  - used to quickly check conditional branch feasibility
  - ‘blow away’ easy VCs (hard ones by HOL4 interactive proof)
- Experiments needed to compare forwards vs backwards!

THE END

Slides at: `http://www.cl.cam.ac.uk/~mjcg/Hoare75/`

THE END

Slides at: `http://www.cl.cam.ac.uk/~mjcg/Hoare75/`