Mechanically Proving Hoare Formulae

Hoare 75 talk (revised)

Additional material

$P\{Q\}R$ ****** Happy 40th Birthday Hoare Logic! ****** $\{P\}C\{Q\}$

An Axiomatic Basis for
Computer Programming

C. A. R. Hoare, 1969

Mechanically Proving Hoare Formulae
(Joint work with Hélène Collavizza)

- 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 $\mathrm{WP}\ C\ Q$ ensures Q holds after C terminates
  - $\mathrm{WP}\ C\ Q$ is Dijkstra's 'weakest liberal precondition' (i.e. partial correctness: `wlp.C.Q` from Dijkstra & Scholten)
  - easy to compute $\mathrm{WP}\ C\ Q$ .... if C has no loops
- Precondition calculation works backwards from Q
  - nice Hoare assignment calculation rule for WP
    $\mathrm{WP}\ (V := E)\ Q\ =\ Q[V \leftarrow E]$
  - pulls postcondition Q back through program
    $\mathrm{WP}\ (C_1 ; C_2)\ Q\ =\ \mathrm{WP}\ C_1\ (\mathrm{WP}\ C_2\ Q)$
  - can't dynamically prune unreachable conditional branches
    $\mathrm{WP}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ =$
    $(B \wedge \mathrm{WP}\ C_1\ Q) \vee (\neg B \wedge \mathrm{WP}\ C_2\ Q)$
- $\{P\}C\{Q\}\ \equiv\ P \Rightarrow \mathrm{WP}\ C\ Q$
- $wlp.C.Q$ is weakest solution of $P : (\{P\}\ C\ \{Q\})$
  (*Predicate Calculus & Program Semantics*, Dijkstra & Scholten, 1990)

Standard backwards method of proving $\{P\}C\{Q\}$

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

- Precondition calculation works backwards from Q
  - nice Hoare assignment calculation rule for WP
    $\mathrm{WP}\ (V := E)\ Q \ = \ Q[V \leftarrow E]$
  - pulls postcondition Q back through program
    $\mathrm{WP}\ (C_1;C_2)\ Q \ = \ \mathrm{WP}\ C_1\ (\mathrm{WP}\ C_2\ Q)$
  - can't dynamically prune unreachable conditional branches
    $\mathrm{WP}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q \ =$
    $(B \wedge \mathrm{WP}\ C_1\ Q) \vee (\neg B \wedge \mathrm{WP}\ C_2\ Q)$

- $\{P\}C\{Q\} \ \equiv \ P \Rightarrow \mathrm{WP}\ C\ Q$

- $wlp.C.Q$ is weakest solution of $P : (\{P\}\ C\ \{Q\})$
  (*Predicate Calculus & Program Semantics*, Dijkstra & Scholten, 1990)

Happy 75 Tony!

Proving $\{P\}C\{Q\}$ forwards

- Less used alternative is strongest postconditions
  - $\mathrm{SP}\ P\ C$ holds after C terminates if started when P holds
  - $\mathrm{SP}\ Q\ C$ is 'strongest postcondition'
    (`sp.C.Q` in Dijkstra & Scholten, Ch.12 – not `stp.C.Q`)
- Postcondition calculation works forwards from P
  - nasty Floyd assignment rule introduces $\exists$-quantification
    $\mathrm{SP}\ P\ (V\ \texttt{:=}\ E)\ =\ \exists v.\ V = E[V{\leftarrow}v] \wedge P[V{\leftarrow}v]$

    "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
    $\mathrm{SP}\ P\ (C_1\texttt{;}C_2)\ =\ \mathrm{SP}\ (\mathrm{SP}\ P\ C_1)\ C_2$
  - can prune branches with symbolic state and constraints
    $\mathrm{SP}\ P\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ =$
    $\mathrm{SP}\ (P \wedge B)\ C_1\ \ \vee\ \ \mathrm{SP}\ (P \wedge \neg B)\ C_2$
- $\{P\}C\{Q\}\ \equiv\ \mathrm{SP}\ P\ C \Rightarrow Q$
- $sp.C.P$ is strongest solution of $Q : (\{P\}\ C\ \{Q\})$

Proving $\{P\}C\{Q\}$ forwards

- Less used alternative is strongest postconditions
  - $\mathrm{SP}\ P\ C$ holds after C terminates if started when P holds
  - $\mathrm{SP}\ Q\ C$ is 'strongest postcondition'
    (`sp.C.Q` in Dijkstra & Scholten, Ch.12 – not `stp.C.Q`)
- Postcondition calculation works forwards from P
  - nasty Floyd assignment rule introduces $\exists$-quantification
    $\mathrm{SP}\ P\ (V\ \mathtt{:=}\ E)\ =\ \exists v.\ V = E[V \leftarrow v] \wedge P[V \leftarrow v]$

    "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
    $\mathrm{SP}\ P\ (C_1\,\mathtt{;}\,C_2)\ =\ \mathrm{SP}\ (\mathrm{SP}\ P\ C_1)\ C_2$
  - can prune branches with symbolic state and constraints
    $\mathrm{SP}\ P\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ =$
    $\mathrm{SP}\ (P \wedge B)\ C_1\ \vee\ \mathrm{SP}\ (P \wedge \neg B)\ C_2$
- $\{P\}C\{Q\}\ \equiv\ \mathrm{SP}\ P\ C \Rightarrow Q$
- $sp.C.P$ 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 $\{P\}C_1\texttt{;(IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3\texttt{);}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Forwards methods meshes better with BMC
- Example

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

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

$$\boxed{\{J \leq I\}}$$

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

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

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

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

Backwards or forwards?

- Calculating WP C Q is easy but leads to big formulae
  - can't 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; $\{J \leq I \wedge K = 0\}$

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 prune case splits 'on-the-fly'
- Calculating SP P C generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths
- Consider $\{P\}C_1\texttt{;(IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3\texttt{);}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Forwards methods meshes better with BMC
- Example

$\{J \leq I\}$

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

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

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 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

$\{I < J\}$

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 prune case splits 'on-the-fly'
- Calculating SP P C generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths
- Consider $\{P\}C_1\texttt{;(IF}\ B\ \texttt{THEN}\ C_2\ \texttt{ELSE}\ C_3\texttt{);}C_4\{Q\}$
  - going forwards P and effect of C_1 might determine B
  - if P specifies a unique state, computing SP is execution
- Forwards methods meshes better with BMC
- Example

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

Backwards or forwards?

- Calculating $\mathrm{WP}\ C\ Q$ is easy but leads to big formulae
  - can't prune case splits 'on-the-fly'
- Calculating $\mathrm{SP}\ P\ C$ generates $\exists$ at assignments
  - at branches state+constraint can reject infeasible paths
- Consider $\{P\}C_1;(\mathtt{IF}\ B\ \mathtt{THEN}\ C_2\ \mathtt{ELSE}\ C_3);C_4\{Q\}$
  - 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; $\{I < J \wedge K = 0\}$

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

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

$\{R = J - I\}$

Backwards or forwards?

- Calculating $\mathsf{WP}\ C\ Q$ is easy but leads to big formulae
  - can't prune case splits 'on-the-fly'
- Calculating $\mathsf{SP}\ P\ C$ generates $\exists$ at assignments
  - at branches state+constraint 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
- Forwards methods meshes better with BMC
- Example

$$\{I < J\}$$

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

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

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

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

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

$\{I < J\}$

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

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

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

$\{R = J - I\}$

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
  - $\mathrm{WP}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q\ =$
    $(B \wedge \mathrm{WP}\ C\ (\mathrm{WP}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ Q))\ \vee\ (\neg B \wedge Q)$
  - $\mathrm{SP}\ P\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C)\ =$
    $(\mathrm{SP}\ (\mathrm{SP}\ (P \wedge B)\ C)\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ C))\ \vee\ (P \wedge \neg B)$
- Solution: Hoare logic rule with an invariant R

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

- 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

$$\begin{aligned}\mathrm{AWP}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q &= R\\ \mathrm{ASP}\ P\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C) &= R \wedge \neg B\end{aligned}$$

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

$$\mathrm{WVC}\ C\ Q \Rightarrow \{\mathrm{AWP}\ C\ Q\}C\{Q\}$$

$$\mathrm{SVC}\ P\ C \Rightarrow \{P\}C\{\mathrm{ASP}\ P\ C\}$$

Calculating verification conditions (VCs)

- VCs to augment approximate weakest preconditions

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

$\mathrm{WVC}\,(V := E)\;Q \;=\; \mathrm{T}$

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

$\mathrm{WVC}\,(\mathtt{IF}\;B\;\mathtt{THEN}\;C_1\;\mathtt{ELSE}\;C_2)\;Q \;=$
$\quad \mathrm{WVC}\;C_1\;Q \;\wedge\; \mathrm{WVC}\;C_2\;Q$

$\mathrm{WVC}\,(\mathtt{WHILE}\;B\;\mathtt{DO}\;\{R\}\;C)\;Q \;=$
$\quad (R \wedge B \Rightarrow \mathrm{AWP}\;C\;R) \;\wedge\; (R \wedge \neg B \Rightarrow Q) \;\wedge\; \mathrm{WVC}\;C\;R$

- VCs to augment approximate strongest postconditions

$\mathrm{SVC}\;P\;(\mathtt{SKIP}) \;=\; \mathrm{T}$

$\mathrm{SVC}\;P\;(V := E) \;=\; \mathrm{T}$

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

$\mathrm{SVC}\;P\;(\mathtt{IF}\;B\;\mathtt{THEN}\;C_1\;\mathtt{ELSE}\;C_2) \;=$
$\quad \mathrm{SVC}\;(P \wedge B)\;C_1 \;\wedge\; \mathrm{SVC}\;(P \wedge \neg B)\;C_2$

$\mathrm{SVC}\;P\;(\mathtt{WHILE}\;B\;\mathtt{DO}\;\{R\}\;C) \;=$
$\quad (P \Rightarrow R) \;\wedge\; (\mathrm{ASP}\;(R \wedge B)\;C \Rightarrow R) \;\wedge\; \mathrm{SVC}\;(R \wedge B)\;C$

Calculating verification conditions (VCs)

- VCs to augment approximate weakest preconditions

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

$$\mathrm{WVC}\ (V\ := E)\ Q \ = \ \mathrm{T}$$

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

$$\mathrm{WVC}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q \ = \ \mathrm{WVC}\ C_1\ Q \ \wedge \ \mathrm{WVC}\ C_2\ Q$$

$$\mathrm{WVC}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q \ = \ (R \wedge B \Rightarrow \mathrm{AWP}\ C\ R) \ \wedge \ (R \wedge \neg B \Rightarrow Q) \ \wedge \ \mathrm{WVC}\ C\ R$$

- VCs to augment approximate strongest postconditions

$$\mathrm{SVC}\ P\ (\mathtt{SKIP}) \ = \ \mathrm{T}$$

$$\mathrm{SVC}\ P\ (V\ := E) \ = \ \mathrm{T}$$

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

$$\mathrm{SVC}\ P\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2) \ = \ \mathrm{SVC}\ (P \wedge B)\ C_1 \ \wedge \ \mathrm{SVC}\ (P \wedge \neg B)\ C_2$$

$$\mathrm{SVC}\ P\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C) \ = \ (P \Rightarrow R) \ \wedge \ (\mathrm{ASP}\ (R \wedge B)\ C \Rightarrow R) \ \wedge \ \mathrm{SVC}\ (R \wedge B)\ C$$

Calculating verification conditions (VCs)

- VCs to augment approximate weakest preconditions

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

$$\mathrm{WVC}\ (V\ := E)\ Q\ =\ \mathrm{T}$$

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

$$\mathrm{WVC}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ =$$
$$\mathrm{WVC}\ C_1\ Q\ \wedge\ \mathrm{WVC}\ C_2\ Q$$

$$\mathrm{WVC}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q\ =$$
$$(R \wedge B \Rightarrow \mathrm{AWP}\ C\ R)\ \wedge\ (R \wedge \neg B \Rightarrow Q)\ \wedge\ \mathrm{WVC}\ C\ R$$

- VCs to augment approximate strongest postconditions

$$\mathrm{SVC}\ P\ (\mathtt{SKIP})\ =\ \mathrm{T}$$

$$\mathrm{SVC}\ P\ (V\ := E)\ =\ \mathrm{T}$$

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

$$\mathrm{SVC}\ P\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ =$$
$$\mathrm{SVC}\ (P \wedge B)\ C_1\ \wedge\ \mathrm{SVC}\ (P \wedge \neg B)\ C_2$$

$$\mathrm{SVC}\ P\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ =$$
$$(P \Rightarrow R)\ \wedge\ (\mathrm{ASP}\ (R \wedge B)\ C \Rightarrow R)\ \wedge\ \mathrm{SVC}\ (R \wedge B)\ C$$

Symbolic execution as postcondition calculation

- Recall $\mathrm{SP}\ P\ (V\ \texttt{:=}\ E)\ =\ \exists v.\ V = E[V{\leftarrow}v] \wedge P[V{\leftarrow}v]$
- Suppose P has form

  $$\exists x_1 \cdots x_n.\ \underbrace{S}_{\text{constraint}}\ \wedge\ \underbrace{X_1 = e_1 \wedge \ldots \wedge X_n = e_n}_{\text{symbolic state}}$$

  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: $[\overline{X}{\leftarrow}\overline{e}]$ for $[X_1{\leftarrow}e_1, \ldots, X_n{\leftarrow}e_n]$
- It follows that $\mathrm{SP}\ P\ (X_i\ \texttt{:=}\ E_i)$ is then

  $$\exists x_1 \cdots x_n.\ S \wedge X_1 = e_1 \wedge \ldots \wedge X_i = E_i[\overline{X}{\leftarrow}\overline{e}] \wedge \ldots \wedge X_n = e_n$$

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

Symbolic execution of conditional branches

- Recall

  $\mathrm{SP}\ P\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ =$
  $\mathrm{SP}\ (P \wedge B)\ C_1\ \vee\ \mathrm{SP}\ (P \wedge \neg B)\ C_2$

- Hence

  $$\begin{array}{l}\mathrm{SP}\ (\exists x_1 \cdots x_n.\ S \wedge\ X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n)\\ \quad (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\\ =\ \mathrm{SP}\ (\exists x_1 \cdots x_n.\ (S \wedge B[\overline{X}{\leftarrow}\overline{e}]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n)\ C_1\\ \quad \vee\\ \quad \mathrm{SP}\ (\exists x_1 \cdots x_n.\ (S \wedge \neg B[\overline{X}{\leftarrow}\overline{e}]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n)\ C_2\end{array}$$

- Prune paths by checking $S \wedge B[\overline{X}{\leftarrow}\overline{e}]$ with a solver
  - $\mathrm{F} \vee P\ =\ P \vee \mathrm{F}\ =\ P$

Happy 75 Tony!

Approximate symbolic execution of while-loops

- Symbolically execute straight line code as before
- For while-loops, recall from previous slide
  $\text{ASP}\ P\ (\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C) \ = \ R \ \wedge \ \neg B$
- Hence execute while-loops as follows

$$\text{ASP}\ (\exists x_1 \cdots x_n.\ S \wedge\ X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n)$$
$$(\texttt{WHILE}\ B\ \texttt{DO}\ \{R\}\ C)$$

$$=\ (\exists x_1 \cdots x_n.\ (R \wedge \neg B[\overline{X}{\leftarrow}\overline{x}]) \wedge X_1{=}x_1 \wedge \ldots \wedge X_n{=}x_n)$$

  - 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

$$((\exists x_1 \cdots x_n.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n) \Rightarrow R)$$
$$\wedge$$
$$(\text{ASP}\,(\exists x_1 \cdots x_n.\,(R \wedge B[\overline{X}{\leftarrow}\overline{x}]) \wedge X_1{=}x_1 \wedge \ldots \wedge X_n{=}x_n)\,C \Rightarrow R)$$

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

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)
  - $s[x{\rightarrow}v]$ is the state mapping x to v and like s elsewhere
  - $[x_1{\rightarrow}v_1; \cdots, x_n{\rightarrow}v_n]$ has domain $\{x_1, \cdots, x_n\}$; maps x_i to v_i
  - $[\![C]\!] : state \times state \rightarrow bool$
  - $[\![B]\!] : state \rightarrow bool$ ($[\![B]\!]$ short for $\mathrm{BVAL}\ B$)
  - $[\![E]\!] : state \rightarrow value$ ($[\![E]\!]$ short for $\mathrm{NVAL}\ B$)
  - $\mathrm{WP}\ C\ Q : state \rightarrow bool$
  - $\mathrm{SP}\ P\ C : state \rightarrow bool$

- 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: $\models P \ =_{def} \ \forall s.\ P\ s$

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

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

- To prove $\{P\}C\{Q\}$ for straight line code
  - calculate $\mathrm{WP}\ C\ Q$ .... back substitution + case splits
  - prove $\models P \Rightarrow \mathrm{WP}\ C\ Q$ ....use a theorem prover

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

- Easy consequences of definition of SP
  - $\text{SP}\ P\ (\texttt{SKIP})\ =\ P$
  - $\text{SP}\ P\ (V\ := E)\ =\ \lambda s'.\ \exists s.\ P\ s \wedge (s'\ =\ s[V \rightarrow [\![E]\!]\,s])$
  - $\text{SP}\ P\ (C_1\,;C_2)\ =\ \text{SP}\ (\text{SP}\ P\ C_1)\ C_2$
  - $\text{SP}\ P\ (\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2)\ =$
    $\text{SP}\ (P \wedge [\![B]\!])\ C_1\ \vee\ \text{SP}\ (P \wedge \neg[\![B]\!])\ C_2$
  - $\text{SP}\ P\ (\texttt{WHILE}\ B\ \texttt{DO}\ C)\ =$
    $\text{SP}\ (\text{SP}\ (P \wedge [\![B]\!])\ C)\ (\texttt{WHILE}\ B\ \texttt{DO}\ C)\ \vee\ (P \wedge \neg[\![B]\!])$
- To prove $\{P\}C\{Q\}$ for straight line code
  - calculate SP P C .... calculating with $\exists$ a problem
  - prove $\models \text{WP}\ P\ C \Rightarrow Q$ ....use a theorem prover

Computing assignment postconditions

- $\vdash \mathrm{SP}\ P\ (V\ := E\ )\ =\ \lambda s'.\ \exists s.\ P\ s \wedge (s'\ =\ s[V {\to} [\![E]\!]\,s])$
- 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

$$\begin{array}{l} \vdash \mathrm{SP}\ (\lambda s.\ \exists x_1 \cdots x_n.\ S\ \ \wedge (s = \boxed{[\overline{X} {\to} \overline{e}\,]}\,))\\ \qquad (X_i\ := E_i\,)\\ \quad = \lambda s. \exists x_1 \cdots x_n. S \wedge (s = \boxed{[\overline{X} {\to} \overline{e}\,][X_i \to ([\![E_i]\!]\,[\overline{X} {\to} \overline{e}\,])]}\,) \end{array}$$

  where
  - $$\begin{array}{l} [\overline{X} {\to} \overline{e}\,][X_i \to ([\![E_i]\!]\,[\overline{X} {\to} \overline{e}\,])]\\ \quad = [X_1 {\to} e_1, \ldots, X_i \to ([\![E_i]\!]\,[\overline{X} {\to} \overline{e}\,]), \ldots, X_n {\to} e_n] \end{array}$$

Symbolic state notation for predicates

- Abbreviate

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

  as

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

  then it follows that

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

- 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!
  - $\mathrm{SP}\ P\ (\texttt{SKIP})\ =\ P$
  - $\mathrm{SP}\ P\ (C_1\texttt{;}C_2)\ =\ \mathrm{SP}\ (\mathrm{SP}\ P\ C_1)\ C_2$

- Simpler symbolic state represention OK for loop-free code

Symbolic state notation for predicates

- Abbreviate

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

  as

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

  then it follows that

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

- 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!
  - $\mathrm{SP}\ P\ (\texttt{SKIP})\ =\ P$
  - $\mathrm{SP}\ P\ (C_1\texttt{;}C_2)\ =\ \mathrm{SP}\ (\mathrm{SP}\ P\ C_1)\ C_2$

- Simpler symbolic state represention OK for loop-free code

Symbolic execution of conditional branches

- Recall

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

- Now

  $$\begin{array}{l}\langle \exists \overline{x}.\ S \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n\rangle\ \wedge\ [\![B]\!]\\ \quad = \ (\lambda s.\ \exists x_1 \cdots x_n.\ S\ s\ \wedge (s = [\overline{X}{\to}\overline{e}\,]))\ \wedge\ \mathrm{BVAL}\ B\\ \quad = \ \lambda s.\ (\exists x_1 \cdots x_n.\ S\ s\ \wedge (s = [\overline{X}{\to}\overline{e}\,]))\ \wedge\ \mathrm{BVAL}\ B\ s\\ \quad = \ \lambda s.\ \exists x_1 \cdots x_n.\ S\ s\ \wedge (s = [\overline{X}{\to}\overline{e}\,])\ \wedge\ \mathrm{BVAL}\ B\ s\\ \quad = \ \lambda s.\ \exists x_1 \cdots x_n.\ (S\ s \wedge \mathrm{BVAL}\ B\ s) \wedge (s = [\overline{X}{\to}\overline{e}\,])\\ \quad = \ \lambda s.\ \exists x_1 \cdots x_n.\ (S \wedge \mathrm{BVAL}\ B\ [\overline{X}{\to}\overline{e}\,])\ s \wedge (s = [\overline{X}{\to}\overline{e}\,])\\ \quad = \ \langle \exists \overline{x}.\ (S \wedge [\![B]\!]\ [\overline{X}{\to}\overline{e}\,]) \wedge X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n\rangle\end{array}$$

- Hence

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

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

Summary so far

- All one needs
  - semantics of commands ($[\![C]\!]$)
  - suitable theorem prover
- Define $\{P\}C\{Q\}$ and $\mathsf{SP}\ P\ C$ from semantics
- Prove rules for calculating $\mathsf{SP}\ P\ C$ (one-off proof)
- For particular P, C, Q prove $\{P\}C\{Q\}$ by
  - calculating $\mathsf{SP}\ P\ C$ using rules and a theorem prover
  - prove $\models \mathsf{SP}\ P\ C \Rightarrow Q$ 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

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

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

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

$$\mathrm{WVC}\ C\ Q \Rightarrow \{\mathrm{AWP}\ C\ Q\}C\{Q\}$$
$$\mathrm{SVC}\ P\ C \Rightarrow \{P\}C\{\mathrm{ASP}\ P\ C\}$$

Calculating verification conditions

- WVC C Q is a standard 'backwards' calculation

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

- SVC P C is a 'forwards' calculation

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

Calculating verification conditions

- WVC C Q is a standard 'backwards' calculation

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

- SVC P C is a 'forwards' calculation

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

Calculating verification conditions

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

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

$$\mathrm{WVC}\ (V\ := E)\ Q\ =\ \mathrm{T}$$

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

$$\mathrm{WVC}\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ Q\ =\ \mathrm{WVC}\ C_1\ Q \wedge \mathrm{WVC}\ C_2\ Q$$

$$\mathrm{WVC}\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ Q\ =$$
$$(\models R \wedge [\![B]\!] \Rightarrow \mathrm{AWP}\ C\ R)\ \wedge\ (\models R \wedge \neg[\![B]\!] \Rightarrow Q)\ \wedge\ \mathrm{WVC}\ C\ R$$

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

$$\mathrm{SVC}\ P\ (\mathtt{SKIP})\ =\ \mathrm{T}$$

$$\mathrm{SVC}\ P\ (V\ := E)\ =\ \mathrm{T}$$

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

$$\mathrm{SVC}\ P\ (\mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2)\ =$$
$$\mathrm{SVC}\ (P \wedge [\![B]\!])\ C_1\ \wedge\ \mathrm{SVC}\ (P \wedge \neg[\![B]\!])\ C_2$$

$$\mathrm{SVC}\ P\ (\mathtt{WHILE}\ B\ \mathtt{DO}\ \{R\}\ C)\ =$$
$$(\models P \Rightarrow R)\ \wedge\ (\models \mathrm{ASP}\ (R \wedge [\![B]\!])\ C \Rightarrow R)\ \wedge\ \mathrm{SVC}\ (R \wedge [\![B]\!])\ C$$

Approximate symbolic execution of while-loops

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

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

- Hence execute while-loops as follows

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

  - 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

    $$\models \langle S \wedge\ X_1{=}e_1 \wedge \ldots \wedge X_n{=}e_n \rangle \Rightarrow R$$
    $$\wedge$$
    $$\models \text{ASP}\ \langle (R \wedge [\![B]\!]) \wedge X_1{=}x_1 \wedge \ldots \wedge X_n{=}x_n \rangle\ C \Rightarrow R$$

Pretty slides hide messy HOL details!

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

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

  is also for a given X_1,...,X_n
- HOL theorem generating specific assignment rule is:

```
|- ∀xl f P v e.
     ALL_DISTINCT xl ⇒
     (∀l. (MAP FST l = xl) ⇒ (MAP FST (f l) = xl)) ⇒
     (LP
       xl
       (λs. ∃l. (MAP FST l = xl) ∧ P l ∧ (s = FEMPTY |++ f l))
       (v ::= e) =
      (λs.
         ∃l.
           (MAP FST l = xl) ∧ P l ∧
           (s = FEMPTY |++ (ASSIGN_FUN v e o f) l)))
```

- Won't rexplain this here beyond:
  - LP represents SP
  - `∃l` instantiated to $\exists x_1 \ldots x_n$ for a specific program

THE END

THE END

Really!