Backwards versus Forwards Proof

✓

- Backwards proof just involves using the rules backwards

- Given the rule

$$\frac{\vdash\ S_1 \qquad \ldots \qquad \vdash\ S_n}{\vdash\ S}$$

- Forwards proof says:
  - if we have proved $\vdash\ S_1\ \ldots\ \vdash\ S_n$ we can deduce $\vdash\ S$

- Backwards proof says:
  - to prove $\vdash\ S$ it is sufficient to prove $\vdash\ S_1\ \ldots\ \vdash\ S_n$

- Having proved a theorem by backwards proof, it is simple to extract a forwards proof

Annotations ✓

- The sequencing rule introduces a new statement R

$$\frac{\vdash\ \{P\}\ C_1\ \{R\} \qquad \vdash\ \{R\}\ C_2\ \{Q\}}{\vdash\ \{P\}\ C_1;C_2\ \{Q\}}$$

- To apply this backwards, one needs to find a suitable statement R

- If C_2 is V `:=` E then sequenced assignment gives Q`[`E/V`]` for R

- If C_2 isn't an assignment then need some other way to choose R

- Similarly, to use the derived While rule, must invent an invariant

Annotate First

- **It is helpful to think up these statements before you start the proof and then annotate the program with them**
  - **the information is then available when you need it in the proof**
  - **this can help avoid you being bogged down in details**
  - **the annotation should be true whenever control reaches that point**

- **Example, the following program could be annotated at the points** $\mathtt{P}_1$ **and** $\mathtt{P}_2$ **indicated by the arrows**

```
{T}
   R:=X;
   Q:=0; {R=X $\wedge$ Q=0} $\longleftarrow$P$_1$
   WHILE Y$\leq$R DO {X = R+Y$\times$Q} $\longleftarrow$P$_2$
     (R:=R-Y; Q:=Q+1)
{X = R+Y$\times$Q $\wedge$ R<Y}
```

NEW TOPIC: Mechanizing Program Verification ✓

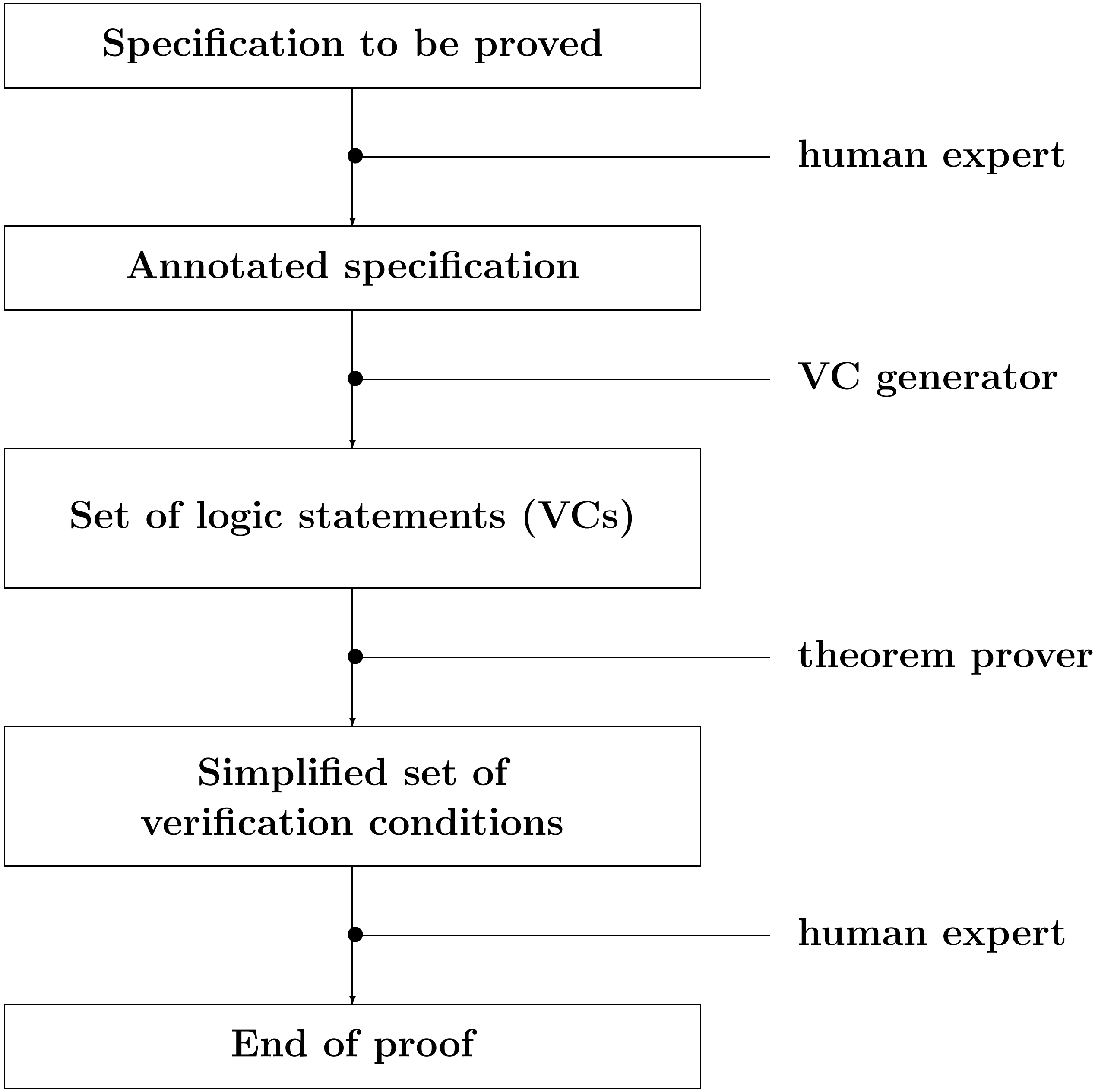
- The architecture of a simple program verifier will be described

- Justified with respect to the rules of Floyd-Hoare logic

- It is clear that
  - proofs are long and boring, even if the program being verified is quite simple
  - lots of fiddly little details to get right, many of which are trivial, e.g.

$$\vdash\ (\mathtt{R}{=}\mathtt{X}\ \wedge\ \mathtt{Q}{=}\mathtt{0})\ \Rightarrow\ (\mathtt{X} = \mathtt{R} + \mathtt{Y}\times\mathtt{Q})$$

Architecture of a Verifier



Specification to be proved

human expert

Annotated specification

VC generator

Set of logic statements (VCs)

theorem prover

Simplified set of verification conditions

human expert

End of proof

Verification conditions ✓

- **The three steps in proving $\{P\}C\{Q\}$ with a verifier**
- **$\boxed{1}$ The program C is *annotated* by inserting statements (*assertions*) expressing conditions that are meant to hold at intermediate points**
  - **tricky: needs intelligence and good understanding of how the program works**
  - **automating it is an artificial intelligence problem**
- **$\boxed{2}$ A set of logic statements called *verification conditions* (VCs) is then generated from the annotated specification**
  - **this is purely mechanical and easily done by a program**
- **$\boxed{3}$ The verification conditions are proved**
  - **needs automated theorem proving (i.e. more artificial intelligence)**
- **To improve automated verification one can try to**
  - **reduce the number and complexity of the annotations required**
  - **increase the power of the theorem prover**
  - **still a research area**

Validity of Verification Conditions ✓

- **It will be shown that**
  - **if one can prove all the verification conditions generated from** $\{P\}C\{Q\}$
  - **then** $\vdash \{P\}C\{Q\}$

- **Step** $\boxed{2}$ **converts a verification problem into a conventional mathematical problem**

- **The process will be illustrated with:**

```
{T}
   R:=X;
   Q:=0;
   WHILE Y≤R DO
     (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

Example

- **Step $\boxed{1}$ is to insert annotations P_1 and P_2**

```
{T}
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P1
   WHILE Y≤R DO {X = R+Y×Q} ⟵P2
     (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

- **The annotations P_1 and P_2 state conditions which are intended to hold** ***whenever*** **control reaches them**

Example Continued ✓

```
{T}
   R:=X;
   Q:=0; {R=X ∧ Q=0} ⟵P₁
   WHILE Y≤R DO {X = R+Y×Q} ⟵P₂
     (R:=R-Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

- **Control only reaches the point at which P_1 is placed once**
- **It reaches P_2 each time the `WHILE` body is executed**
  - **whenever this happens `X=R+Y×Q` holds, even though the values of `R` and `Q` vary**
  - **P_2 is an *invariant* of the `WHILE`-command**

Generating and Proving Verification Conditions ✓

- Step $\boxed{2}$ will generate the following four verification conditions

(i) $T \Rightarrow (X=X \wedge 0=0)$

(ii) $(R=X \wedge Q=0) \Rightarrow (X = R+(Y\times Q))$

(iii) $(X = R+(Y\times Q)) \wedge Y\leq R) \Rightarrow (X = (R-Y)+(Y\times (Q+1)))$

(iv) $(X = R+(Y\times Q)) \wedge \neg(Y\leq R) \Rightarrow (X = R+(Y\times Q) \wedge R<Y)$

- Notice that these are statements of arithmetic
  - the constructs of our programming language have been ‘compiled away’

- Step $\boxed{3}$ consists in proving the four verification conditions
  - easy with modern automatic theorem provers

Annotation of Commands ✓

- An annotated command is a command with statements (*assertions*) embedded within it

- A command is *properly annotated* if statements have been inserted at the following places

  (i) before C_2 in $C_1;C_2$ if C_2 is *not* an assignment command
  (ii) after the word `DO` in `WHILE` commands

- The inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs

- Can reduce number of annotations using weakest preconditions (see later)

Annotation of Specifications

✓

- A properly annotated specification is a specification $\{P\}C\{Q\}$ where C is a properly annotated command

- Example: To be properly annotated, assertions should be at points ① and ② of the specification below

```
{X=n}
   Y:=1; ⟵①
   WHILE X≠0 DO ⟵②
     (Y:=Y×X; X:=X-1)
{X=0 ∧ Y=n!}
```

- Suitable statements would be

at ①: `{Y = 1 ∧ X = n}`
at ②: `{Y×X! = n!}`

Verification Condition Generation ✓

- The VCs generated from an annotated specification $\{P\}C\{Q\}$ are obtained by considering the various possibilities for C

- We will describe it command by command using rules of the form:

- The VCs for $C(C_1, C_2)$ are
  - vc_1, ... , vc_n
  - together with the VCs for C_1 and those for C_2

- Each VC rule corresponds to either a primitive or derived rule

Justification of VCs ✓

- **This process will be justified by showing that $\vdash \{P\}C\{Q\}$ if all the verification conditions can be proved**

- **We will prove that for any `C`**
  - **assuming the VCs of $\{P\}C\{Q\}$ are provable**
  - **then $\vdash \{P\}C\{Q\}$ is a theorem of the logic**

Justification of Verification Conditions ✓

- The argument that the verification conditions are sufficient will be by *induction* on the structure of C

- Such inductive arguments have two parts

  - show the result holds for atomic commands, i.e. assignments

  - show that when C is not an atomic command, then if the result holds for the constituent commands of C (this is called the *induction hypothesis*), then it holds also for C

- The first of these parts is called the *basis* of the induction

- The second is called the *step*

- The basis and step entail that the result holds for all commands

VC for Assignments ✓

Assignment commands

The single verification condition generated by

$$\{P\}\ V\texttt{:=}E\ \{Q\}$$

is

$$P \ \Rightarrow\ Q\texttt{[}E/V\texttt{]}$$

- **Example:** The verification condition for

  $\{\texttt{X=0}\}\ \texttt{X:=X+1}\ \{\texttt{X=1}\}$

  is

  $\texttt{X=0} \ \Rightarrow\ \texttt{(X+1)=1}$

  (which is clearly true)

- **Note:** $Q\texttt{[}E/V\texttt{]} \ = \ \texttt{wlp}(\texttt{"}V\texttt{:=}E\texttt{"}, Q)$

Justification of Assignment VC ✓

- We must show that if the VCs of $\{P\}\ V := E\ \{Q\}$ are provable then $\vdash\ \{P\}\ V := E\ \{Q\}$

- Proof:
  - Assume $\vdash\ P \Rightarrow Q[E/V]$ as it is the VC
  - From derived assignment rule it follows that $\vdash\ \{P\}\ V := E\ \{Q\}$

VCs for Conditionals

✓

Conditionals

The verification conditions generated from

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$$

are

(i) the verification conditions generated by

$$\{P\ \wedge\ S\}\ C_1\ \{Q\}$$

(ii) the verifications generated by

$$\{P\ \wedge\ \neg S\}\ C_2\ \{Q\}$$

- **Example: The verification conditions for**

  $\{$`T`$\}$ `IF X`$\geq$`Y THEN MAX:=X ELSE MAX:=Y` $\{$`MAX=max(X,Y)`$\}$

  are

  (i) the VCs for $\{$`T` $\wedge$ `X`$\geq$`Y`$\}$ `MAX:=X` $\{$`MAX=max(X,Y)`$\}$

  (ii) the VCs for $\{$`T` $\wedge$ $\neg$`(X`$\geq$`Y)`$\}$ `MAX:=Y` $\{$`MAX=max(X,Y)`$\}$

Justification for the Conditional VCs (1) ✓

- **Must show that if VCs of**

  $\{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$

  are provable, then

  $\vdash\ \{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$

- **Proof:**
  - **Assume the VCs** $\{P\ \wedge\ S\}\ C_1\ \{Q\}$ **and** $\{P\ \wedge\ \neg S\}\ C_2\ \{Q\}$
  - **The inductive hypotheses tell us that if these VCs are provable then the corresponding Hoare Logic theorems are provable**
  - **i.e. by induction** $\vdash\ \{P\ \wedge\ S\}\ C_1\ \{Q\}$ **and** $\vdash\ \{P\ \wedge\ \neg S\}\ C_2\ \{Q\}$
  - **Hence by the conditional rule** $\vdash\ \{P\}$ `IF` S `THEN` C_1 `ELSE` C_2 $\{Q\}$

Review of Annotated Sequences ✓

- **If $C_1;C_2$ is properly annotated, then either**

  $\boxed{\textbf{Case 1:}}$ **it is of the form $C_1;\{R\}C_2$ and C_2 is not an assignment**

  $\boxed{\textbf{Case 2:}}$ **it is of the form $C;V := E$**

- **And C, C_1 and C_2 are properly annotated**

VCs for Sequences ✓

Sequences

1. **The verification conditions generated by**

$$\{P\}\ C_1\ \{R\}\ C_2\ \{Q\}$$

(where C_2 is not an assignment) are the union of:

   (a) **the verification conditions generated by** $\{P\}\ C_1\ \{R\}$

   (b) **the verifications generated by** $\{R\}\ C_2\ \{Q\}$

2. **The verification conditions generated by**

$$\{P\}\ C\,\texttt{;}V\,\texttt{:=}E\ \{Q\}$$

are the verification conditions generated by $\{P\}\ C\ \{Q\texttt{[}E/V\texttt{]}\}$

Justification of VCs for Sequences (1) ✓

- $\boxed{\text{Case 1:}}$ If the verification conditions for

  $\{P\}\ C_1\ ;\ \{R\}\ C_2\ \{Q\}$

  are provable

- Then the verification conditions for

  $\{P\}\ C_1\ \{R\}$

  and

  $\{R\}\ C_2\ \{Q\}$

  must both be provable

- Hence by induction

  $\vdash\ \{P\}\ C_1\ \{R\}$ and $\vdash\ \{R\}\ C_2\ \{Q\}$

- Hence by the sequencing rule

  $\vdash\ \{P\}\ C_1;C_2\ \{Q\}$

Justification of VCs for Sequences (2)

✓

- $\boxed{\textbf{Case 2:}}$ **If the verification conditions for**

  $$\{P\}\ C\,;V := E\ \{Q\}$$

  are provable, then the verification conditions for

  $$\{P\}\ C\ \{Q\texttt{[}E/V\}$$

  are also provable

- **Hence by induction**

  $$\vdash\ \{P\}\ C\ \{Q\texttt{[}E/V\texttt{]}\}$$

- **Hence by the derived sequenced assignment rule**

  $$\vdash\ \{P\}\ C\,;V := E\ \{Q\}$$

VCs for `WHILE`-Commands ✓

- **A correctly annotated specification of a `WHILE`-command has the form**

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

- **The annotation R is called an invariant**

`WHILE`-commands

The verification conditions generated from

$$\{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}\ C\ \{Q\}$$

are

(i) $P \ \Rightarrow\ R$

(ii) $R\ \wedge\ \neg S\ \Rightarrow\ Q$

(iii) the verification conditions generated by $\{R\ \wedge\ S\}\ C\{R\}$

Justification of `WHILE` VCs ✓

- **If the verification conditions for**

  $\{P\}$ `WHILE` S `DO` $\{R\}\ C\ \{Q\}$

  are provable, then

  $\vdash\ P \Rightarrow R$

  $\vdash\ (R\ \wedge\ \neg S)\ \Rightarrow\ Q$

  and the verification conditions for

  $\{R\ \wedge\ S\}\ C\ \{R\}$

  are provable

- **By induction**

  $\vdash\ \{R\ \wedge\ S\}\ C\ \{R\}$

- **Hence by the derived `WHILE`-rule**

  $\vdash\ \{P\}$ `WHILE` S `DO` $C\ \{Q\}$

Summary ✓

- Have outlined the design of an automated program verifier

- Annotated specifications compiled to mathematical statements
  - if the statements (VCs) can be proved, the program is verified

- Human help is required to give the annotations and prove the VCs

- The algorithm was justified by an inductive proof
  - it appeals to the derived rules

- All the techniques introduced earlier are used
  - backwards proof
  - derived rules
  - annotation

Dijkstra's weakest preconditions ✓

- **Weakest preconditions is a theory of refinement**
  - **idea is to calculate a program to achieve a postcondition**
  - **not a theory of post hoc verification**
- **Non-determinism a key idea in Dijksta's presentation**
  - **start with a non-deterministic high level pseudo-code**
  - **refine to deterministic and efficient code**
- **Weakest preconditions (wp) are for total correctness**
- **Weakest *liberal* preconditions (wlp) for partial correctness**
- **If C is a command and Q a predicate, then informally:**
  - $\texttt{wlp}(C, Q) \ = \ $ '*The weakest predicate P such that $\{P\}\ C\ \{Q\}$*'
  - $\texttt{wp}(C, Q) \ = \ $ '*The weakest predicate P such that $[P]\ C\ [Q]$*'
- **If P and Q are predicates then $Q \Rightarrow P$ means P is 'weaker' than Q**

Rules for weakest preconditions ✓

- **Relation with Hoare specifications:**

$$\begin{array}{lcl} \{P\}\ C\ \{Q\} & \Leftrightarrow & P \Rightarrow \mathtt{wlp}(C,Q) \\ [P]\ C\ [Q] & \Leftrightarrow & P \Rightarrow \mathtt{wp}(C,Q) \end{array}$$

- **Dijkstra gives rules for computing weakest preconditions:**

$$\begin{array}{lcl} \mathtt{wp}(V\,\mathtt{:=}E, Q) & = & Q[E/V] \\ \mathtt{wp}(C_1\mathtt{;}C_2,\ Q) & = & \mathtt{wp}(C_1, \mathtt{wp}(C_2,\ Q)) \\ \mathtt{wp}(\mathtt{IF}\ S\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2,\ Q) & = & (S \Rightarrow \mathtt{wp}(C_1,Q))\ \wedge\ (\neg S \Rightarrow \mathtt{wp}(C_2,Q)) \end{array}$$

  for deterministic loop-free code the same equations hold for `wlp`

- **Rule for** `WHILE`**-commands doesn't give a first order result**

- **Weakest preconditions closely related to verification conditions**

- **VCs for $\{P\}\ C\ \{Q\}$ are related to $P \Rightarrow \mathtt{wlp}(C,Q)$**
  - **VCs use annotations to ensure first order formulas can be generated**

Using `wlp` to improve verification condition method ✓

- **If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $P \Rightarrow \texttt{wlp}(C,Q)$**
  - **no annotations needed in sequences!**

- **Cannot in general compute a finite formula for $\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)$**

- **The following holds**

$$\texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)\ =\ \textit{if}\ S\ \textit{then}\ \texttt{wlp}(C,\ \texttt{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q))\ \textit{else}\ Q$$

- **Above doesn't define $\texttt{wlp}(C,Q)$ as a finite statement**

- **Could use a hybrid VC and `wlp` method**

Strongest postconditions ✓

- **Define $\texttt{sp}(C, P)$ to be 'strongest' Q such that $\{P\}\ C\ \{Q\}$**
  - **partial correctness:** $\{P\}\ C\ \{\texttt{sp}(C, P)\}$
  - **strongest means if $\{P\}\ C\ \{Q\}$ then $\texttt{sp}(C, P) \Rightarrow Q$**
- **Note that `wlp` goes 'backwards', but `sp` goes 'forwards'**
  - **verification condition for $\{P\}\ C\ \{Q\}$ is: $\texttt{sp}(C, P) \Rightarrow Q$**
- **By 'strongest' and Hoare logic postcondition weakening**
  - $\{P\}\ C\ \{Q\}$ **if and only if** $\texttt{sp}(C, P) \Rightarrow Q$

Strongest postconditions for loop-free code ✓

- **Only consider loop-free code**
- $\texttt{sp}(V\ \texttt{:=}\ E,\ P)\ =\ \exists v.\ V = E\texttt{[}v/V\texttt{]} \wedge P\texttt{[}v/V\texttt{]}$
- $\texttt{sp}(C_1\ \texttt{;}\ C_2,\ P)\ =\ \texttt{sp}(C_2,\ \texttt{sp}(C_1,\ P))$
- $\texttt{sp}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ P)\ =\ \texttt{sp}(C_1,\ P \wedge S)\ \vee\ \texttt{sp}(C_2,\ P \wedge \neg S)$

- $\texttt{sp}(V\texttt{:=}E,\ P)$ **corresponds to Floyd assignment axiom**
- **Can** *dynamically prune* **conditionals because** $\texttt{sp}(C, \texttt{F}) = \texttt{F}$
- **Computer strongest postconditions is** *symbolic execution*

Computing `sp` versus `wlp` ✓

- **Computing `sp` is like execution**
  - **can simplify as one goes along with the 'current state'**
  - **may be able to resolve branches, so can avoid executing them**
  - **Floyd assignment rule complicated in general**
  - **`sp` used for symbolically exploring 'reachable states'**
    (related to *model checking*)

- **Computing `wlp` is like backwards proof**
  - **don't have 'current state', so can't simplify using it**
  - **can't determine conditional tests, so get big `if-then-else` trees**
  - **Hoare assignment rule simpler for arbitrary formulae**
  - **`wlp` used for improved verification conditions**

Using `sp` to generate verification conditions ✓

- **If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $\texttt{sp}(C,\ P) \Rightarrow Q$**

- **Cannot in general compute a finite formula for $\texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P)$**

- **The following holds**

  $$\texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P)\ = \texttt{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ \texttt{sp}(C,\ (P \wedge S)))\ \vee\ (P \wedge \neg S)$$

- **Above doesn't define $\texttt{sp}(C, P)$ to be a finite statement**

- **As with `wlp`, can use a hybrid VC and `sp` method**

Summary ✓

- **Annotate then generate VCs is the classical method**
  - practical tools: Gypsy (1970s), SPARK (current)
  - weakest preconditions are alternative explanation of VCs
  - `wlp` needs fewer annotations than VC method described earlier
  - `wlp` also used for refinement

- **VCs and `wlp` go backwards, `sp` goes forward**
  - `sp` provides verification method based on symbolic simulation
  - widely used for loop-free code
  - current research potential for forwards full proof of correctness
  - probably need mixture of forwards and backwards methods (Hoare's view)

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

- Research goal: unifying framework for a spectrum of methods

decidable checking *proof of correctness*