WHILE-rule for Total Correctness (i)

- **WHILE**-commands are the only commands in our little language that can cause non-termination
  - they are thus the only kind of command with a non-trivial termination rule

- The idea behind the **WHILE**-rule for total correctness is
  - to prove **WHILE** $S$ **DO** $C$ terminates
  - show that some non-negative quantity decreases on each iteration of $C$
  - this decreasing quantity is called a **variant**
In the rule below, the variant is $E$, and the fact that it decreases is specified with an auxiliary variable $n$.

The hypothesis $\vdash P \land S \Rightarrow E \geq 0$ ensures the variant is non-negative.

\[
\begin{align*}
\vdash [P \land S \land (E = n)] C [P \land (E < n)], & \quad \vdash P \land S \Rightarrow E \geq 0 \\
\vdash [P] \text{WHILE } S \text{ DO } C [P \land \neg S]
\end{align*}
\]

where $E$ is an integer-valued expression
and $n$ is an identifier not occurring in $P$, $C$, $S$ or $E$. 

\textbf{WHILE-Rule for Total Correctness (ii)}
Example

- We show

\[ \vdash [Y > 0] \text{WHILE } Y \leq R \text{ DO } (R := R - Y; \ Q := Q + 1) \ [T] \]

- Take

\[
\begin{align*}
P & = Y > 0 \\
S & = Y \leq R \\
E & = R \\
C & = (R := R - Y; \ Q := Q + 1)
\end{align*}
\]

- We want to show \( \vdash [P] \text{WHILE } S \text{ DO } C \ [T] \)

- By the WHILE-rule for total correctness it is sufficient to show

(i) \( \vdash [P \land S \land (E = n)] \ C \ [P \land (E < n)] \)

(ii) \( \vdash P \land S \Rightarrow E \geq 0 \)

- Then use postcondition weakening and \( \vdash P \land \neg S \Rightarrow T \)
Example Continued (1)

- From previous slide:

\[
\begin{align*}
P &= Y > 0 \\
S &= Y \leq R \\
E &= R \\
C &= (R := R - Y; \ Q := Q + 1)
\end{align*}
\]

- We want to show

(i) \( \vdash [P \land S \land (E = n)] \ C [P \land (E < n)] \)

(ii) \( \vdash P \land S \Rightarrow E \geq 0 \)

- The first of these, (i), can be proved by establishing

\( \vdash \{ P \land S \land (E = n) \} \ C \{ P \land (E < n) \} \)

- Then using the total correctness rule for non-looping commands
Example Continued (2)

- From previous slide:
  \[
  P = Y > 0 \\
  S = Y \leq R \\
  E = R \\
  C = R := R - Y; \ Q := Q + 1)
  \]

- The verification condition for \{P \land S \land (E = n)\} \ C \{P \land (E < n)\} is:
  \[
  Y > 0 \land Y \leq R \land R = n \Rightarrow \(Y > 0 \land R < n\)[Q+1/Q] [R−Y/R]
  \]
  i.e. \(Y > 0 \land Y \leq R \land R = n \Rightarrow Y > 0 \land R−Y < n\)
  which follows from the laws of arithmetic

- The second subgoal, (ii), is just \(\vdash Y > 0 \land Y \leq R \Rightarrow R \geq 0\)
The relation between partial and total correctness is informally given by the equation

\[ \text{Total correctness} = \text{Termination} + \text{Partial correctness} \]

This informal equation can be represented by the following two rules of inferences

\[
\frac{\vdash \{P\} \ C \ \{Q\}}{\vdash [P] \ C \ [Q]} \quad \frac{\vdash [P] \ C \ [T]}{\vdash [P] \ C \ [Q]} \quad \frac{\vdash [P] \ C \ [Q]}{\vdash \{P\} \ C \ \{Q\}} \quad \frac{\vdash \{P\} \ C \ \{Q\}}{\vdash [P] \ C \ [T]}
\]
Derived Rules

- Multiple step rules for total correctness can be derived in the same way as for partial correctness
  - the rules are the same up to the brackets used
  - same derivations with total correctness rules replacing partial correctness ones
  - only significant change is the derived WHILE-rule

- Derived WHILE-rule needs to handle the variant

<table>
<thead>
<tr>
<th>Derived WHILE-rule for total correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢  P ⇒ R</td>
</tr>
<tr>
<td>⊢  R ∧ S ⇒ E ≥ 0</td>
</tr>
<tr>
<td>⊢  R ∧ ¬S ⇒ Q</td>
</tr>
<tr>
<td>⊢  [R ∧ S ∧ (E = n)] C [R ∧ (E &lt; n)]</td>
</tr>
<tr>
<td>⊢  [P] WHILE S DO C [Q]</td>
</tr>
</tbody>
</table>
VCs for Termination

- Verification conditions are easily extended to total correctness

- To generate total correctness verification conditions for WHILE-commands, it is necessary to add a variant as an annotation in addition to an invariant

- Variant added directly after the invariant, in square brackets

- No other extra annotations are needed for total correctness

- VCs generation algorithm same as for partial correctness
A correctly annotated total correctness specification of a **WHILE**-command thus has the form

\[ [P] \text{WHILE } S \text{ DO } \{R\}[E] C \{Q\] \]

where \( R \) is the invariant and \( E \) the variant.

- Note that the variant is intended to be a **non-negative** expression that **decreases** each time around the **WHILE** loop.

- The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them (as before).
A correctly annotated specification of a `WHILE`-command has the form

\[ [P] \text{WHILE } S \text{ DO } \{R\}[E] \text{ C } [Q] \]

The verification conditions generated from

\[ [P] \text{WHILE } S \text{ DO } \{R\}[E] \text{ C } [Q] \]

are

(i) \( P \Rightarrow R \)
(ii) \( R \land \neg S \Rightarrow Q \)
(iii) \( R \land S \Rightarrow E \geq 0 \)
(iv) the verification conditions generated by

\[ [R \land S \land (E = n)] C[R \land (E < n)] \]

where \( n \) is a variable not occurring in \( P, R, E, C, S \) or \( Q \).
The verification conditions for

\[
[R=X \land Q=0] \\
\text{WHILE } Y \leq R \text{ DO } \{X=R+Y \times Q\}[R] \\
\quad (R:=R-Y; \; Q:=Q+1) \\
[X = R+(Y \times Q) \land R<Y]
\]

are:

(i) \(R=X \land Q=0 \Rightarrow (X = R+(Y \times Q))\)

(ii) \(X = R+Y \times Q \land \neg(Y \leq R) \Rightarrow (X = R+(Y \times Q) \land R<Y)\)

(iii) \(X = R+Y \times Q \land Y \leq R \Rightarrow R\geq0\)

together with the verification condition for

\[
[X = R+(Y \times Q) \land (Y \leq R) \land R=n] \\
(R:=R-Y; \; Q:=Q+1) \\
[X=R+(Y\times Q) \land R<n]
\]
Example Continued

- The single verification condition for

\[
X = R + (Y \times Q) \land (Y \leq R) \land R = n
\]

\[
(R := R - Y; \ Q := Q + 1)
\]

\[
X = R + (Y \times Q) \land R < n
\]

is

\[
(iv) \quad X = R + (Y \times Q) \land (Y \leq R) \land R = n \Rightarrow \\
X = (R - Y) + (Y \times (Q + 1)) \land (R - Y) < n
\]

- But this isn’t true
  - take \( Y = 0 \)

- To prove \( R - Y < n \) we need to know \( Y > 0 \)

- Exercise: Explain why one would not expect to be able to prove the verification conditions of this last example

- Hint: Consider the original specification
Summary

- We have given rules for total correctness
- They are similar to those for partial correctness
- The main difference is in the WHILE-rule
  - because WHILE commands are the only ones that can fail to terminate
- Must prove a non-negative expression is decreased by the loop body
- Derived rules and VC generation rules for partial correctness easily extended to total correctness
- Interesting stuff on the web
  - http://research.microsoft.com/TERMINATOR
Overview

- All the axioms and rules given so far were quite straightforward
  - may have given a false sense of simplicity

- Hard to give rules for anything other than very simple constructs
  - an incentive for using simple languages

- We already saw with the assignment axiom that intuition over how to formulate a rule might be wrong
  - the assignment axiom can seem ‘backwards’

- We now add some new commands to our little language
  - array assignments
  - blocks
  - FOR-commands
Array assignments

- Syntax: $V(E_1) := E_2$

- Semantics: the state is changed by assigning the value of the term $E_2$ to the $E_1$-th component of the array variable $V$

- Example: $A(X+1) := A(X) + 2$
  
  - if the value of $X$ is $x$
  - and the value of the $x$-th component of $A$ is $n$
  - then the value stored in the $(x+1)$-th component of $A$ becomes $n+2$
Naive Array Assignment Axiom Fails

- The axiom
  \[ \vdash \{P[E_2/A(E_1)]\} \ A(E_1) := E_2 \ \{P\} \]
  doesn’t work

- Take \( P \equiv 'X=Y \land A(Y)=0' \), \( E_1 \equiv 'X' \), \( E_2 \equiv '1' \)
  - since \( A(X) \) does not occur in \( P \)
  - it follows that \( P[1/A(X)] = P \)
  - hence the axiom yields: \( \vdash \{X=Y \land A(Y)=0\} \ A(X):=1 \ \{X=Y \land A(Y)=0\} \)

- Must take into account possibility that changes to \( A(X) \) may change \( A(Y), A(Z) \) etc
  - since \( X \) might equal \( Y, Z \) etc (i.e. aliasing)

- Related to the Frame Problem in AI
Reasoning About Arrays

• The naive array assignment axiom

\[ \vdash \{ P[E_2/A(E_1)] \} \ A(E_1) := E_2 \ \{ P \} \]

does not work: changes to \( A(X) \) may also change \( A(Y), A(Z), \ldots \).

• The solution, due to Hoare, is to treat an array assignment

\[ A(E_1) := E_2 \]

as an ordinary assignment

\[ A := A\{ E_1 \leftarrow E_2 \} \]

where the term \( A\{ E_1 \leftarrow E_2 \} \) denotes an array identical to \( A \), except that the \( E_1 \)-th component is changed to have the value \( E_2 \).
Array Assignment axiom

- Array assignment is a special case of ordinary assignment

\[ A := A \{ E_1 \leftarrow E_2 \} \]

- Array assignment axiom just ordinary assignment axiom

\[ \vdash \{ P[A\{E_1 \leftarrow E_2\}/A] \} \; A := A\{E_1 \leftarrow E_2\} \; \{ P \} \]

- Thus:

The array assignment axiom

\[ \vdash \{ P[A\{E_1 \leftarrow E_2\}/A] \} \; A(E_1) := E_2 \; \{ P \} \]

Where \( A \) is an array variable, \( E_1 \) is an integer valued expression, \( P \) is any statement and the notation \( A\{E_1 \leftarrow E_2\} \) denotes the array identical to \( A \), except that \( A(E_1) = E_2 \).
Array Axioms

- In order to reason about arrays, the following axioms, which define the meaning of the notation \( A\{E_1 \leftarrow E_2 \} \), are needed.

\[
\begin{align*}
\text{The array axioms} & \\
\vdash A\{E_1 \leftarrow E_2 \}(E_1) &= E_2 \\
\vdash E_1 \neq E_3 \Rightarrow A\{E_1 \leftarrow E_2 \}(E_3) &= A(E_3)
\end{align*}
\]

- Second of these is a \textit{Frame Axiom}.
For more rigour recall first order theory \textsc{array}

- $\mathcal{L}_{\text{ARRAY}} = \{\text{isarray}, \text{lookup}, \text{update}\}$
  - \text{isarray} has arity 1, \text{lookup} has arity 2, \text{update} has arity 3

- $\mathcal{I}_{\text{ARRAY}}$
  - domain is $\mathbb{V} \cup \{\phi \mid \phi : \mathbb{N} \rightarrow \mathbb{V}\}$ for some set of values $\mathbb{V}$
  - $\mathcal{I}_{\text{ARRAY}}[\text{isarray}](a)$ is true iff $a$ is a function $\phi$
  - $\mathcal{I}_{\text{ARRAY}}[\text{lookup}](a, i) = \text{if } a \text{ is a function } \phi \text{ then } \phi(i) \text{ else } 0$
  - $\mathcal{I}_{\text{ARRAY}}[\text{update}](a, i, v) = \text{if } a \text{ is a function } \phi \text{ then } \phi[v/i] \text{ else } a$

- \textsc{array} contains the following axioms
  - $\forall a \ i \ v. \ \text{isarray}(a) \implies (\text{lookup}(\text{update}(a, i, v), i) = v)$
  - $\forall a \ i \ j \ v. \ \text{isarray}(a) \land \neg(i = j) \implies (\text{lookup}(\text{update}(a, i, v), j) = \text{lookup}(a, j))$
  - $\forall a_1 \ a_2. \ \text{isarray}(a_1) \land \text{isarray}(a_2) \land (\forall i. \ \text{lookup}(a_1, i) = \text{lookup}(a_2, i)) \implies (a_1 = a_2)$

- “$a(i)$” means “$\text{lookup}(a, i)$” and “$a\{i \leftarrow v\}$” means “$\text{update}(a, i, v)$”

- Assuming $a$ is an array ($\text{isarray}(a)$ is true) then from array axioms:
  - $a\{i \leftarrow v\}(i) = v$
  - $\neg(i = j) \implies (a\{i \leftarrow v\}(j) = a(j))$
Example

• We show

\[
\begin{align*}
\Gamma & \vdash \{ A(X) = x \land A(Y) = y \} \\
R & := A(X); \\
A(X) & := A(Y); \\
A(Y) & := R \\
\{ A(X) = y \land A(Y) = x \}
\end{align*}
\]

• Working backwards using the array assignment axiom

\[
\begin{align*}
\Gamma & \vdash \{ (A\{ Y \leftarrow R \})(X) = y \land (A\{ Y \leftarrow R \})(Y) = x \} \\
A(Y) & := R \\
\{ A(X) = y \land A(Y) = x \}
\end{align*}
\]

• Array assignments are variable assignments of array values, so:

\[
\begin{align*}
\Gamma & \vdash \{ (A\{ Y \leftarrow R \})(X) = y \land (A\{ Y \leftarrow R \})(Y) = x \} \\
A & := A\{ Y \leftarrow R \} \\
\{ A(X) = y \land A(Y) = x \}
\end{align*}
\]
• Using

\[ \vdash A\{Y \leftarrow R\}(Y) = R \]

• It follows that

\[ \vdash \{(A\{Y \leftarrow R\})(X) = y \land R = x\} \]

\[ A(Y) := R \]

\[ \{A(X) = y \land A(Y) = x\} \]

• Continuing backwards

\[ \vdash \{(A\{Y \leftarrow R\})\{Y \leftarrow R\}(X) = y \land R = x\} \]

\[ A(X) := A(Y) \]

\[ \{(A\{Y \leftarrow R\})(X) = y \land R = x\} \]

• Maybe more intuitive if the assignment is rewritten

\[ \vdash \{(A\{X \leftarrow A(Y)\})\{Y \leftarrow R\}(X) = y \land R = x\} \]

\[ A := A\{X \leftarrow A(Y)\} \]

\[ \{(A\{Y \leftarrow R\})(X) = y \land R = x\} \]
Continuing backwards

\[
\frac{}{\{ ((A\{X \leftarrow A(Y)\}) \{Y \leftarrow A(X)\}) (X) = y \land A(X) = x \} \\
R := A(X) \\
\{ ((A\{X \leftarrow A(Y)\}) \{Y \leftarrow R\}) (X) = y \land R = x \}
\]

Hence by the derived sequencing rule

\[
\frac{}{\{ ((A\{X \leftarrow A(Y)\}) \{Y \leftarrow A(X)\}) (X) = y \land A(X) = x \} \\
R := A(X); A(X) := A(Y); A(Y) := R \\
\{A(X) = y \land A(Y) = x \}
\]

By the array axioms (considering the cases \(X = Y\) and \(X \neq Y\) separately):

\[
\frac{}{((A\{X \leftarrow A(Y)\}) \{Y \leftarrow A(X)\}) (X) = A(Y)}
\]

Hence (as desired)

\[
\frac{}{\{A(Y) = y \land A(X) = x \} \\
R := A(X); A(X) := A(Y); A(Y) := R \\
\{A(X) = y \land A(Y) = x \}}
\]
Blocks *(not in handout)*

- **Syntax:** `BEGIN VAR V_1; ⋯ VAR V_n; C END`

- **Semantics:** command $C$ is executed, then the values of $V_1, \cdots, V_n$ are restored to the values they had before the block was entered
  - the initial values of $V_1, \cdots, V_n$ inside the block are unspecified

- **Example:** `BEGIN VAR R; R:=X; X:=Y; Y:=R END`
  - the values of $X$ and $Y$ are swapped using $R$ as a temporary variable
  - this command does *not* have a side effect on the variable $R`
The Block Rule

- The block rule takes care of local variables

\[
\begin{align*}
\vdash & \{P\} C \{Q\} \\
\vdash & \{P\} \text{ BEGIN VAR } V_1; \ldots; \text{ VAR } V_n; \text{ END } \{Q\}
\end{align*}
\]

where none of the variables \(V_1, \ldots, V_n\) occur in \(P\) or \(Q\).

- Note that the block rule is regarded as including the case when there are no local variables (the \(n = 0\) case)
The Side Condition

- The syntactic condition that none of the variables $V_1, \ldots, V_n$ occur in $P$ or $Q$ is an example of a side condition.

- From
  \[ \frac{}{\{ X=x \land Y=y \} \quad R:=X; \quad X:=Y; \quad Y:=R \quad \{ Y=x \land X=y \}} \]
  it follows by the block rule that
  \[ \frac{}{\{ X=x \land Y=y \} \quad \text{BEGIN VAR } R; \quad R:=X; \quad X:=Y; \quad Y:=R \quad \text{END} \quad \{ Y=x \land X=y \}} \]
  since $R$ does not occur in $X=x \land Y=y$ or $Y=x \land Y=y$.

- However from
  \[ \frac{}{\{ X=x \land Y=y \} \quad R:=X; \quad X:=Y \quad \{ R=x \land X=y \}} \]
  one cannot deduce
  \[ \frac{}{\{ X=x \land Y=y \} \quad \text{BEGIN VAR } R; \quad R:=X; \quad X:=Y \quad \text{END} \quad \{ R=x \land X=y \}} \]
  since $R$ occurs in $R=x \land X=y$. 
Exercises

• Consider the specification

\{X=x\} \text{ BEGIN } \text{VAR } X; X:=1 \text{ END } \{X=x\}

Can this be deduced from the rules given so far?
(i) if so, give a proof of it
(ii) if not, explain why not and suggest additional rules
    and/or axioms to enable it to be deduced

• Is the following true?

\vdash \{X=x \land Y=y\} \ X:=X+Y; \ Y:=X-Y; \ X:=X-Y \ \{Y=x \land X=y\}

• Show

\vdash \{X=R+(Y\times Q)\} \text{ BEGIN } R:=R-Y; \ Q:=Q+1 \text{ END } \{X=R+(Y\times Q)\}
FOR-commands

- **Syntax:** FOR $V := E_1$ UNTIL $E_2$ DO $C$
  
  - **restriction:** $V$ must not occur in $E_1$ or $E_2$,
    or be the left hand side of an assignment in $C$
    (explained later)

- **Semantics:**
  
  - if the values of terms $E_1$ and $E_2$ are positive numbers $e_1$ and $e_2$
  - and if $e_1 \leq e_2$
  - then $C$ is executed $(e_2 - e_1) + 1$ times with the variable $V$ taking on the sequence
    of values $e_1, e_1+1, \ldots, e_2$ in succession
  - for any other values, the FOR-command has no effect

- **Example:** FOR $N := 1$ UNTIL $M$ DO $X := X + N$
  
  - if the value of the variable $M$ is $m$ and $m \geq 1$, then the command $X := X + N$ is
    repeatedly executed with $N$ taking the sequence of values $1, \ldots, m$
  - if $m < 1$ then the FOR-command does nothing
Subtleties of *FOR*-commands

- There are many subtly different versions of *FOR*-commands
- For example
  - the expressions $E_1$ and $E_2$ could be evaluated at each iteration
  - and the controlled variable $V$ could be treated as global rather than local
- Early languages like Algol 60 failed to notice such subtleties
- Note that with the semantics presented here, *FOR*-commands cannot *generate* non termination
More on the semantics of FOR-commands

- The semantics of

  \[
  \text{FOR } V := E_1 \text{ UNTIL } E_2 \text{ DO } C
  \]

  is as follows

(i) \( E_1 \) and \( E_2 \) are evaluated once to get values \( e_1 \) and \( e_2 \), respectively.

(ii) If either \( e_1 \) or \( e_2 \) is not a number, or if \( e_1 > e_2 \), then nothing is done.

(iii) If \( e_1 \leq e_2 \) the FOR-command is equivalent to:

\[
\text{BEGIN VAR } V; V := e_1; C; V := e_1 + 1; C; \ldots; V := e_2; C \text{ END}
\]

i.e. \( C \) is executed \((e_2 - e_1) + 1\) times with \( V \) taking on the sequence of values \( e_1, e_1 + 1, \ldots, e_2 \)

- If \( C \) doesn’t modify \( V \) then FOR-command equivalent to:

\[
\text{BEGIN VAR } V; V := e_1; \ldots \underbrace{C}_{\text{repeated}}; V := V + 1; \ldots; V := e_2; C \text{ END}
\]