Partial vs. Total Correctness

• So far, we’ve dealt with “partial correctness”:
  
  *If* the assumption is true *and* the program terminates, *then* the expectation will be true.

  Partial Correctness: Assumption $\land$ Termination $\rightarrow$ Expectation

• Of even greater interest is “total correctness”:
  
  *If* the assumption is true, *then* the program *terminates and* the expectation will be true.

  Total Correctness: Assumption $\rightarrow$ Termination $\land$ Expectation
Partial vs. Total Correctness

• Total Correctness =

  Partial Correctness + Termination

• It is often easier to prove partial correctness first, then add in a proof of termination.
How to Prove Termination?

• A program terminates if it progresses inexorably to a final state.

• Identify a function \( \eta \) on states (called a \textit{variant}): 
  
  \[
  \eta : \text{States} \rightarrow \mathbb{N} \quad \text{(Natural Numbers)} \quad \text{therefore } \eta(v) \geq 0 \quad \text{always}
  \]
  
  such that, \textit{on every iteration}, \( \eta \) \textit{decreases} in value.

• Because the range of \( \eta \) is \textit{non-negative}, there is a limit to the number of iterations.
Practicalities

• Integer is a more common data type than Natural Number.
• So we can rephrase the requirement for variants:
  • \( \eta \): States \( \rightarrow \) Integers
  • \( \forall s \) \( \eta(s) \geq 0 \)
  • \( \eta(s) = K \) at start of loop body \( \rightarrow \) \( \eta(s) < K \) at end of loop body for an anchor variable \( K \) (see next slide).

• As the loop body requires \( I \land P \) to execute (where \( I \) is the invariant and \( P \) is the loop test), an implicit requirement is:
  • \( I \land P \rightarrow \eta(s) > 0 \)

• Otherwise the requirement that \( \eta \) decrease would be inconsistent with the requirement that \( \eta(s) \geq 0 \).
Anchor Variables

• By “anchor variable” I mean a variable in an assertion that does not occur in the program, but serves to “capture” the value of an expression for the purpose of a proof.

• The captured value is referred to again in some other assertion.

• Huth & Ryan refer to these as “logical variables” (as opposed to program variables).
Anchor Variable Example

Here the values of $f$ go
$1, n_0, n_0 \cdot (n_0 - 1), n_0 \cdot (n_0 - 1) \cdot (n_0 - 2), \ldots$
$= n_0! / n_0!, n_0! / (n_0 - 1)!, n_0! / (n_0 - 2)! \ldots$

Assertions with anchor variable $n_0$

$A_1: n = n_0 \land n_0 \geq 0$

A1: $f := 1$

$[n > 0] \ (n, f) := (n - 1, f \cdot n)$

$A_2: f = n_0! / n! \land n \geq 0$

$[n \leq 0]$

A2: $f = n_0!$

The anchor is necessary, because $n$ loses its original value.
Partial Correctness of the Anchor Variable Example

• The VC’s, derived using WP reasoning, are:

  • \( n = n_0 \land n_0 \geq 0 \rightarrow 1 = n_0!/n! \land n \geq 0 \)  \hspace{1cm} \text{Init}

  • \( f = n_0!/n! \land n \geq 0 \land n > 0 \)
    \rightarrow \( f*n = n_0!/(n-1)! \land n-1 \geq 0 \)  \hspace{1cm} \text{Loop}

  • \( f = n_0!/n! \land n \geq 0 \land n \leq 0 \rightarrow f = n_0! \)  \hspace{1cm} \text{Final}
Sufficient Conditions for While Loop Termination

• Here P is the loop test and I is an invariant.
• $\eta$: States $\rightarrow$ Integers
• $\eta(s) = K$ at start of loop body $\rightarrow \eta(s) < K$ at end of loop body, for an anchor variable $K$
• $I \wedge P \rightarrow \eta(s) > 0$
• The preceding are enough to guarantee termination.
• **Note**: We don’t explicitly prove $\eta(s) > 0$. In fact, can become less than 0 by the end of the iteration, but only for the last iteration.
Termination Verification Conditions (TVCs)

• We can translate the conditions described on the previous slide into VC’s:
• Here $\eta$ is the variant expression.

  • TVC1: $I \land P \land (\eta = K) \rightarrow WP[(\eta < K), Body, T]$
    where $K$ is a fresh anchor variable

  • TVC2: $I \land P \rightarrow \eta > 0$
Trivial Termination Example

n := n_0;
while( n > 0 )
{
    n := n-1;
}

Loop invariant: n ≥ 0
What is an acceptable variant η in this case?
Variant for the Trivial Loop

- $\eta = n$
- TVC1: $I \land P \land (\eta = K) \rightarrow WP[I \land (\eta < K), \text{loop-body}, T]$ is
  
  $n \geq 0 \land n > 0 \land (n = K) \rightarrow WP[n \geq 0 \land (n < K), n := n-1, T]$

  which reduces to
  
  $n > 0 \land (n = K) \rightarrow n-1 \geq 0 \land (n-1 < K) \quad [\text{valid}]$

- TVC2: $I \land P \rightarrow (\eta > 0)$ is
  
  $n \geq 0 \land n > 0 \rightarrow n > 0 \quad [\text{valid}]$
Termiation Example 2

n := 0;
while( n < c ) c is some constant
{
  ...
  n := n+1;
}

Suppose I is $n \leq c$.
What is an acceptable $\eta$ in this case?
Variant for Second Loop

- $\eta = c - n$
- TVC1: $I \land P \land (\eta = K) \rightarrow WP[I \land (\eta < K), \text{loop-body, T}]$
  is
  $n \leq c \land n < c \land (c - n = K) \rightarrow$
  $WP[n \leq c \land (c - n < K), n := n+1, T]$
  which reduces to
  $n < c \land (c - n = K) \rightarrow n+1 \leq c \land (c-(n+1) < K)$
  [valid based on integer arithmetic]

- TVC2: $I \land P \rightarrow (\eta > 0)$ is
  $n \leq c \land n < c \rightarrow c - n > 0$
  [valid]
Jape Proofs of Termination

• Jape’s Hoare Logic introduces the necessary termination conditions whenever a while is used. It cannot be avoided.

• The user does not need to set up the conditions.

• The user must specify a variant through unification.

• The VC’s are expressed in the Hoare Logic notation, which we describe subsequently.
JAPE Loop Example

• User enters theorem to be proved.
  \[
  \vdash \{ n \geq 0 \} \text{ while } n > 0 \text{ do } n := n-1 \text{ od } \{ n = 0 \}
  \]

  \[
  \ldots
  \]

  \[
  1: \{ n \geq 0 \} \text{ while } n > 0 \text{ do } n := n-1 \text{ od } \{ n = 0 \}
  \]
JAPE Loop Example

• User selects **while** rule

1: \(\{n \geq 0 \land n > 0\}(n:=n-1)\{n \geq 0\}\)

2: \(n \geq 0 \land n > 0 \rightarrow _M > 0\)

3: `integer Km` **assumption**

4: \(\{n \geq 0 \land n > 0 \land _M = Km\}(n:=n-1)\{M < Km\}\)

5: \(\{n \geq 0\} while \ n > 0 \ do \ n:=n-1 \ od\{n \geq 0 \land \neg(n > 0)\}\) while 1,2,3-4

6: \(n \geq 0 \land \neg(n > 0) \rightarrow n=0\)

7: \(\{n \geq 0\} while \ n > 0 \ do \ n:=n-1 \ od\{n=0\}\) **consequence(R) 5,6**

The VC’s are expressed as Hoare Logic triples.
Jape Loop Example Completed

1: \(n \geq 0 \land n > 0 \rightarrow n-1 \geq 0\) obviously
2: \(\{n-1 \geq 0\}(n:=n-1)\{n \geq 0\}\) variable-assignment consequence(L) 1,2
3: \(\{n \geq 0 \land n > 0\}(n:=n-1)\{n \geq 0\}\) obviously
4: \(n \geq 0 \land n > 0 \rightarrow n > 0\) assumption

5: integer \(K_m\)
6: \(n \geq 0 \land n > 0 \land n = K_m \rightarrow n-1 < K_m\) obviously
7: \(\{n-1 < K_m\}(n:=n-1)\{n < K_m\}\) variable-assignment
8: \(\{n \geq 0 \land n > 0 \land n = K_m\}(n:=n-1)\{n < K_m\}\) consequence(L) 6,7

9: \(\{n \geq 0\}\) while \(n > 0\) do \(n:=n-1\) od\(\{n \geq 0 \land \neg(n > 0)\}\) while 3,4,5–8
10: \(n \geq 0 \land \neg(n > 0) \rightarrow n = 0\) obviously
11: \(\{n \geq 0\}\) while \(n > 0\) do \(n:=n-1\) od\(\{n = 0\}\) consequence(R) 9,10

[No loop entry VC required, because no entry code.]

The VC’s are expressed using Hoare Logic.
The Second Loop Example in Jape

Km is the anchor variable.

Expresses that n and x are not aliases.
What’s Bad about Aliases?

• Consider $x := 2$
  with post-condition $\{y = 3\}$

• Then $WP[y = 3, x := 2, T]$ is still $y = 3$.

• But if $x$ and $y$ are aliases, then $y$ will also be 2, not 3 in the post-condition, following execution.

• Aliasing is **assumed not to occur** unless provided for otherwise.
Hoare Logic (HL)

• Tony Hoare discovered a way to put VC’s into an elegant natural-deduction formalism.

• Most of chapter 4 of Huth&Ryan is expressed using HL.
Hoare Logic

• C.A.R. ("Tony") Hoare was the first to express program construction along with proofs of correctness as a single unified logic.
• “An axiomatic basis for computer programming”, 1969.

Sir Tony Hoare (FRS)
Microsoft Research Laboratory, Cambridge
Formulas of Hoare Logic

• A formula in HL consists of a **triple**
  \{Pre\} Fragment \{Post\}
  • Pre is a predicate logic formula: the **Pre-Assertion**
  • Fragment is a program fragment
  • Post is a predicate logic formula: the **Post-Assertion**

• The idea is that the triple forms a VC for the entire fragment.
HL Rules

• Hoare gave rules of inference for these triples.

• These rules follow along the lines of WP reasoning expressed earlier.

• The WP formulation was due to Dijkstra, and came after Hoare’s work, and after Hoare had expressed the elegant Assignment Rule.
One of the Rules from Hoare’s Paper

D2  Rule of Composition

If \( \vdash P \{ Q_1 \} R_1 \) and \( \vdash R_1 \{ Q_2 \} R \) then \( \vdash P \{ (Q_1 ; Q_2) \} R \)

Most people now put the braces around the formulas rather than the code

\{Pre\} Fragment \{Post\}

rather than

Pre \{Fragment\} Post

as Hoare did.
Assignment Rule

The assignment rule has no antecedent.

\[
\begin{align*}
{P[t/v]} & \quad v := t \quad \{P\} \\
\hline
\text{Assignment}
\end{align*}
\]

where \( t \) is a term and \( v := t \) is an assignment statement.

\( P \) is a post-assertion, and \( P[t/v] \) is seen to be the same as \( WP[P, v := t, T] \) discussed earlier.
Assignment Rule Example

\[
\{ P[t/v] \} \ v := t \ { P } \quad \text{Assignment}
\]

\[
\{ 2*x + y < y \} \ x := 2*x + y \ \{ x < y \}
\]
Conditional Rule

- The conditional rule has two antecedents.

- \( \{ R \land P \} \text{ Fragment}_1 \{ S \} \quad \{ R \land \neg P \} \text{ Fragment}_2 \{ S \} \)
  
  \{ R \} \text{ if } P \text{ then Fragment}_1 \text{ else Fragment}_2 \{ S \}

- We can see how to derive a suitable R, given S:
  
  \[
  R = \text{wp}[S, \text{ if } P \text{ then Fragment}_1 \text{ else Fragment}_2, T] \\
  = \quad P \rightarrow \text{ wp}[S, \text{ Fragment}_1, T] \\
  \land \neg P \rightarrow \text{ wp}[S, \text{ Fragment}_2, T]
  \]
Conditional Rule Example

\[
\begin{align*}
\{ R \land P \} & \text{ Fragment}_1 \{ S \} \\
\{ R \land \neg P \} & \text{ Fragment}_2 \{ S \} \\
\{ R \} & \text{ if } P \text{ then } \text{ Fragment}_1 \text{ else } \text{ Fragment}_2 \{ S \}
\end{align*}
\]

\[
\begin{align*}
\{ T \land x > y \} & \ z := x \ { z = \text{max}(x, y)} \\
\{ T \land x \leq y \} & \ z := y \ { z = \text{max}(x, y)}
\end{align*}
\]

\[
\begin{align*}
\{ T \} & \text{ if } x > y \text{ then } z := x \text{ else } z := y \ { z = \text{max}(x, y)}
\end{align*}
\]
While Rule

• This is the most cryptic rule. It involves the loop invariant $I$.

\[
\begin{array}{c}
{I \land P} \text{ Body } {I} \\
{I} \text{ while } P \text{ do } \text{ Body } {I \land \neg P}
\end{array}
\]

• There is no closed-form WP rule for while. We have to discover the invariant and incorporate it.
While Rule Example

\[
\{I \land P\} \text{ Body } \{I\} \\
\{I\} \text{ while } P \text{ do } \text{ Body } \{I \land \neg P\}
\]

\[
\{i \leq n+1 \land i < n\} \text{ } i := i+1 \{i \leq n+1\}
\]

\[
\{i \leq n+1\} \text{ while } i < n \text{ do } i := i+1 \{i \leq n+1 \land \neg (i < n)\}
\]
How to Remember the While Rule

• The statement form is **while** P **do** Body.
• The test P will always be **true** before Body.
• The invariant will be true before and after everything.
  \[ \{I \land P\} \text{Body} \{I\} \]
• (The invariant will never by the same as P.)
• The test P will be **false** after the **while**.
• \[ \{I\} \textbf{while} P \textbf{do} \text{Body} \{I \land \neg P\} \]
Sequence Rule

• This rule connects two Fragments sequentially, as indicated by the semicolon (alluding to C family of languages)

• \{(Q) \text{Fragment}_1 \{R\}} \{R\} \text{Fragment}_2 \{S\}
\{(Q) \text{Fragment}_1; \text{Fragment}_2 \{S\}\}
Consequent Rule
(called “Implied” Rule in H&R)

• This rule is based on implications.

• $Q' \rightarrow Q \quad \{Q\} \text{ Fragment } \{R\} \quad R \rightarrow R' \\
  \{Q'\} \text{ Fragment } \{R'\}$
An Application of Consequent Rule

- Sometimes in a sequential composition, assertions don’t quite match up.
- We want

\[
\begin{align*}
\{Q\} \text{ Fragment}_1 \{R\} & \quad \{R\} \text{ Fragment}_2 \{S\} \\
\{Q\} \text{ Fragment}_1; \text{ Fragment}_2 \{S\}
\end{align*}
\]

but we only have

\[
\begin{align*}
\{Q\} \text{ Fragment}_1 \{R\} & \quad \{R'\} \text{ Fragment}_2 \{S\} \\
\{Q\} \text{ Fragment}_1; \text{ Fragment}_2 \{S\}
\end{align*}
\]

If we can derive \(R \rightarrow R'\) then the fragments can still be composed, since by the consequent rule, we also have

\[
\begin{align*}
R \rightarrow R' & \quad \{R'\} \text{ Fragment}_2 \{S\} \\
\{R\} \text{ Fragment}_2 \{S\}
\end{align*}
\]
Tree Presentation of Proofs

• Complete proofs can be presented as trees, as in natural deduction.

• Figure 4.2 p274 of H&R is an example.

• They could also be presented in a numbered linear form, as in JAPE.

• There is a third possibility, in-line assertions.
Tree Presentation of the Simple while Example

_assignment

\[
x \leq n \land x < n \rightarrow x+1 \leq n \quad \{x+1 \leq n\} \quad x := x+1 \quad \{x \leq n\}
\]

---

**Consequent**

\[
\{x \leq n \land x < n\} \quad x := x+1 \quad \{x \leq n\}
\]

---

**While**

\[
\{x \leq n\} \quad \textbf{while}(x < n) \quad \textbf{do} \quad x := x+1 \quad \{x \leq n \land \neg(x < n)\} \quad \text{\quad} x \leq n \land \neg(x < n) \rightarrow x = n
\]

---

Consequent

\[
\{x \leq n\} \quad \textbf{while}(x < n) \quad \textbf{do} \quad x := x+1 \quad \{x = n\}
\]
In-Line Assertions (H&R’s Tableaux) vs. Trees

- Presenting a verified program using a tree-like derivation is cumbersome.

- Instead we may embed the assertions into the code.

- In the case of consequent rules, we don’t show the implications explicitly.

- H&R call this type of presentation a “tableau”, but it is unlike the tableau proof method for pure logic.
In-Line Assertions Presentation

\{x \leq n\} \\
\textbf{while}( x < n ) \\
\{x \leq n \land x < n\} \\
\{x + 1 \leq n\} \\
\textbf{x} := \textbf{x} + 1 \\
\{x \leq n\} \\
x \leq n \land \neg(x < n) \\
\{x = n\}

\textbf{Implicit Logic} for Consequent Rules:
\[
x \leq n \land x < n \rightarrow x + 1 \leq n \\
x \leq n \land \neg(x < n) \rightarrow x = n
\]
Numbered Triples Presentation
This is JAPE’s Hoare Logic.
[It includes a termination proof (in lines 4-8) as well.]

1: $x \leq n \land x < n \rightarrow x + 1 \leq n$
2: $\{x + 1 \leq n\}(x := x + 1)\{x \leq n\}$
3: $\{x \leq n \land x < n\}(x := x + 1)\{x \leq n\}$
4: $x \leq n \land x < n \rightarrow n - x > 0$
5: integer $K_m$
6: $x \leq n \land x < n \land n - x = K_m \rightarrow n - (x + 1) < K_m$
7: $\{n - (x + 1) < K_m\}(x := x + 1)\{n - x < K_m\}$
8: $\{x \leq n \land x < n \land n - x = K_m\}(x := x + 1)\{n - x < K_m\}$
9: $\{x \leq n\}$while $x < n$ do $x := x + 1$ od$\{x \leq n \land \neg(x < n)\}$ while $3, 4, 5-8$
10: $x \leq n \land \neg(x < n) \rightarrow x = n$
11: $\{x \leq n\}$while $x < n$ do $x := x + 1$ od$\{x = n\}$

Provided:
DISTINCT n, x
Total Correctness Example: gcd  
(due to Euclid, 300 BCE)

\{m = m_0 \land n = n_0 \land m_0 > 0 \land n_0 > 0\}  \quad // \text{assumption using anchors}

while( \neg (m = n) )

\{
  \quad \text{if}( \ m < n \ ) \ n := n - m; \ else \ m := m - n;
\}

\{m = \gcd(m_0, n_0)\}  \quad // \text{expectation}
Total Correctness of the gcd program

• What is the loop invariant?

• What is an appropriate variant?
Lemmas for gcd

- $\gcd(A, A) = A$
- $\gcd(A, B) = \gcd(B, A)$
- $A > B \rightarrow \gcd(A-B, B) = \gcd(A, B)$
Informal Proof of $A > B \rightarrow \gcd(A-B, B) = \gcd(A, B)$

- Show that pairs $\{A, B\}$ and $\{A-B, B\}$ have the same divisors. Therefore they have the same gcd.

- If $d$ divides both $A$ and $B$, then there are $A'$ and $B'$ such that $A=dA'$ and $B=dB'$.

- But then $A-B = dA' - dB' = d(A' - B')$, so $d$ divides $A-B$ as well.

- Conversely, if $d$ divides both $A-B$ and $B$, then $d$ divides $(A-B)+B$, which is $A$. 
Proposed GCD Loop Invariant

• \( \text{gcd}(m, n) = \text{gcd}(m_0, n_0) \)
Possible GCD Variants?
Choice of Variant

- It may be necessary to **strengthen the invariant**, to add to it conditions that make the termination VC’s provable.

- For example, for termination we may need $m > 0 \land n > 0$, which was not required for partial correctness.
Using Quantifiers in Assertions

• Quantifiers may be needed in more complex programs.

• For example, in a prime-testing program, it may be necessary to assert that a variable is not divisible by a range of values.
  \[ \forall m \ [1 < m \land m < n \rightarrow \neg \text{divides}(m, n)] \]

• Quantifiers are also useful in programs dealing with arrays, e.g. in asserting an array is sorted:
  \[ \forall i \ (0 < i \land i < n \rightarrow A[i-1] < A[i]) \]
Handling Array Assignment

• The standard way to handle array assignment is to treat the entire array as a variable.

• Then an assignment is treated as installing a new array similar to the old one, but with the assigned index modified.

• Think of an array as a function from indices to values.

• Assignment
  
  \[ A[i] := v \]

  is treated as assignment
  
  \[ A := \text{new\_array}\{j = i \rightarrow v \land j \neq i \rightarrow A[j]\} \]

  where \text{new\_array} establishes array values from a function.
Invariants in Object-Oriented Programming

• Often want methods to preserve an invariant when called.
  • Example: Sorted List
    • insert and remove maintain the invariant that the list is sorted.
  • Example: Priority Queue using a Heap
    • insert and remove maintain the heap invariant.
We have just scratched the surface.