Proofs for Programs

• For many reasons, it is desirable to accompany programs with a proof that the program meets a certain specification.

• One way to do this is to derive the proof along with deriving the program.

Related text material

• Huth & Ryan
  Chapter 4, Program verification

• Note: Their "tableau proofs" should not be confused with the tableaux we have discussed so far.

• Also, they use funny braces that are a combination of parens and |: ( | and |), where I just use { }.

Alan Turing, 1949

• Turing may have been the first to consider proving that a program is correct, in his paper (3 typewritten pages):

  "Checking a Large Routine"

  "How can one check a large routine in the sense that it’s right?

  ... make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

http://www.turingarchive.org/viewer/?id=462&title=01


Robert W. Floyd

• “Assigning meanings to programs”, 1967

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”, CACM, 1969.

Sir Prof. Tony Hoare (FRS)
Microsoft Research Laboratory,
Cambridge, England
One of the Rules from Hoare’s Paper

D2  Rule of Composition
If \( |P|Q_1|R\) and \( |R_1|Q_2|R\) then \( |P|Q_1\cup Q_2|R\)

Program “Dynamics”
- You may be accustomed to thinking of a program as something with “dynamic” behavior.
- A mathematical view is that a program’s behavior is just one of many paths through of a (generally-infinite) static structure, which can be analyzed with mathematics and logic.

Programs States
- Programs work with states.
- Each state is a mapping from program variables into appropriate domains
  
  state: variables \( \rightarrow \) domain

Example: An integer square-root program

<table>
<thead>
<tr>
<th>variables</th>
<th>a</th>
<th>i</th>
<th>r</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = 1; i = 1; r = 0; while( s \leq n )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r = r + 1;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i = i + 2;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s = s + i;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program as a “State Transformer”

starting state

ending state

Note about I/O
- To deal with input streams and files, we will consider the entire file or stream, along with the current position of the reader or writer, to be part of the state.
- We won’t be dealing with such issues in this presentation.
Programs with added Assertions

- An assertion is a predicate-logic expression about the variables in the program.

- Assertions can express two kinds of things:
  - An assumption about the state before a box (also called the pre-condition).
  - An expectation about the state after a box (also called the post-condition).
  - Sometimes, e.g. Huth & Ryan, expectations are called “guarantees”.

Example for the previous program

(i) assumption: \( n \geq 0 \)

(ii) expectation: \( r = \text{isqrt}(n) \)

Relating Expectation to Assumption

- Variables common to both expectation and assumption can be used to relate the two.

- Without such relations, the task of developing and proving a program can become meaningless.

A Program Specification consists of

(i) assumption about the starting state

Program as a gray box.

(ii) expectation about the ending state

Relativity

- Expectations are relative to assumptions.

- Nothing in particular can be expected if the assumption is false when the program is started.

Example

- Assumption:
  \[
  \text{int } x[0..n-1] \text{ is an array of size } n
  \]

- Expectation (x is sorted):
  \[
  \forall i ((i \geq 1 \land i < n) \rightarrow (x[i-1] \leq x[i]))
  \]
A trivial way to meet the specification

```c
for( i = 0; i < n; i++ )
{
    x[i] = 0;
}
```

A More Exacting Specification
(introduces a new array \(X_0\) not part of the program)

- Assumption:
  \[
  \text{int } x[0..n-1] \text{ is an array of size } n \\
  \land \ x = X_0 \\
  \text{(in the sense that } x \text{ and } X_0 \text{ are two arrays with the same elements)}
  \]

- Expectation:
  \[
  \forall i \ ((i \geq 1 \land i < n) \rightarrow (x[i-1] \leq x[i])) \\
  \land \ \text{bagof} (x, X_0)
  \]
  (meaning \(x\) has the same elements, of the same multiplicity as \(X_0\))

Application in Software Engineering

- "Design by Contract" (vs. "Defensive Programming")
- Design a program module as if the assumption were true at the start.
- Design the module to meet the expectation.
- Do not build in extra checks for wrong data. (This helps reduce redundancy in the system overall.)
- Of course, at the external interface, the program should check for "wrong data", but this too could be part of the specification.

Specification with Exceptions

- Assumption:
  \[
  \text{valid}(\text{input}) \rightarrow T \\
  \land \ \neg \text{valid}(\text{input}) \rightarrow \text{red_flag}
  \]

- Expectation:
  \[
  \neg \text{red_flag} \rightarrow \ldots \text{normal expectation} \ldots \\
  \land \ \text{red_flag} \rightarrow \text{exception is indicated}
  \]

- The value of \(\text{valid}(\text{input})\) may be something the program itself computes. But it is a predicate just the same.

Ways of Using Logic

- Formal Verification: Create a program that is proved to meet its specification.
- Model Checking: Mechanically check that a program meets its specification (used for finite-state systems).
- Static Analysis: Symbolically check that no erroneous things are being done by the program (incomplete, but useful).
- Program Synthesis: Automate the construction of a program from a logical specification.

What if s

- What if the assumption about the starting state doesn’t hold?
  - We don’t care about the result in this case.
  - However, the assumption can be made very stringent, e.g. \(T\), in which case we will always care.
What if the assumption about the starting state holds, but the expectation doesn’t hold when the program terminates?

- The program is incorrect.

Floyd Assertions

- Annotate program steps with logical assertions between statements.
- Prove that the assertions hold, based on a form of induction.

Floyd Assertions

\[
\begin{align*}
\text{s} &= 1; \quad \text{s}' = s+1 \\
\text{i} &= 1; \quad \text{i}' = i+2 \\
\text{r} &= 0; \quad \text{r}' = \text{r}+1
\end{align*}
\]

while (s < n)

\[
\begin{align*}
\text{r} &= \text{r} + 1; \\
\text{i} &= \text{i} + 2; \\
\text{s} &= \text{s} + \text{i}; \\
\end{align*}
\]

\[
\text{r}^2 \leq n < (r+1)^2
\]

Verification Conditions

- From program + assertions are derived “verification conditions”, which are pure logical statements that can be proved independently of each other.

\[
\begin{align*}
(s < n & \land s \geq 0 & \land s = r^2 & \land i = 2r+1) & \quad \text{pre-condition} \\
(s' = s + i) & \quad \text{statement semantics} \\
\rightarrow (s' < n+i & \land s' \geq 0 & \land s' = r^2+i & \land \text{i} = 2r+1) & \quad \text{post-condition}
\end{align*}
\]

Better Mechanization

- Hoare, and later Dijkstra, demonstrated how the derivation of assertions could be partly automated,

eliminating the need to create verification conditions explicitly.

In particular, Hoare’s method resembled natural deduction.
Hoare Triples

- Consider endowing a program to be designed with its assumption and expectation:

  \{\text{assumption}\} \text{ code } \{\text{expectation}\}

- This is known as a "triple", or "Hoare triple".

- Originally Hoare put the braces around the code, instead of around the assertions. Now the opposite is more common.

Example of a Triple

\{\text{assumption}\} \text{ code } \{\text{expectation}\}

\{x \leq y \land x \leq z\} \text{TBD... } \{x \leq y \land y \leq z\}

[TBD = "To Be Determined"]

Design then becomes the process of filling in the TBD code.

Some triples are more stringent than others.

\{\text{assumption}\} \text{ code } \{\text{expectation}\}

\{x \leq y \land x \leq z\} \text{TBD... } \{x \leq y \land y \leq z\}

\{x \leq y\} \text{TBD... } \{x \leq y \land y \leq z\}

\{T\} \text{TBD... } \{x \leq y \land y \leq z\}

\{T\} \text{TBD... } \{x \leq y \land y \leq z \land z \leq w\}

Stringency

- \{\text{assumption}\} \text{ code } \{\text{expectation}\}

- \text{T}_1: \{A\} \subset \{E\} for short

- \text{T}_2: \{A\} \subset \{E'\}

- \text{T}_3: \{A'\} \subset \{E\}

- If \(E' \rightarrow E\), is \(T_3\) more or less stringent than \(T_1\)?

- If \(A' \rightarrow A\), is \(T_3\) more or less stringent than \(T_1\)?

Rationale

- If \(E' \rightarrow E\), then any state satisfying \(E'\) must also satisfy \(E\), but not necessarily conversely, so \(\{A\} \subset \{E'\}\) is more stringent than \(\{A\} \subset \{E\}\).

- If \(A' \rightarrow A\), then any state satisfying \(A'\) must also satisfy \(A\), so \(\{A'\} \subset \{E\}\) is less stringent than \(\{A\} \subset \{E\}\).

- In other words,

  - \(\{A\} \subset \{E'\}\) meets the expectation \(E\) and possibly more.

  - \(\{A'\} \subset \{E\}\) assumes more than \(A\) to get the same job done.

Extreme Stringency

- \{T\} \subset \{\bot\}

- Here \(c\) would not assume anything, but meets every expectation.
Extreme Leniency

- $\{\bot\} \to \{T\}$

- Assuming everything, don’t expect anything.

First Rule of Inference

- Consequent Rule
  
  $A \to A', \{A'\} \to \{E\}, E' \to E \to \{A\} \to \{E\}$

- Here triples are combined with logic ($\to$ is implies).

- Effectively this says that any triple can be derived from a more stringent one.

- This is called the “Implied” rule in Huth&Ryan, p 270.

Special Cases in JAPE Hoare Logic

- Consequent(L):
  
  $A \to A', \{A'\} \to \{E\} \to \{A\} \to \{E\}$

- Consequent(R):
  
  $\{A'\} \to \{E\}, E' \to E \to \{A\} \to \{E\}$

Composition of Triples

- Suppose we have a triple:
  
  $\{\text{Assumption}\} \to \{\text{Code}\} \to \{\text{Expectation}\}$

- To develop the code, we can break it into two parts:
  
  $\{\text{Assumption 1}\} \to \{\text{Code 1}\} \to \{\text{Expectation 1}\}$

  $\{\text{Assumption 2}\} \to \{\text{Code 2}\} \to \{\text{Expectation 2}\}$

  We want Code = Code 1; Code 2 (concatenation)

  What do we need for this to work?

Composition Rule

We need Expectation1 = Assumption2.

In the form of a natural deduction rule:

$\{A\} \to S1 \{B\} \to \{B\} \to S2 \{C\}$

$\{A\} \to S1;S2 \{C\}$

Example of Composition Rule

1. $\{T\} \to S1 \{x \leq y\}$
2. $\{x \leq y\} \to S2 \{x \leq y \land y \leq z\}$
3. $\{T\} \to S1; S2 \{x \leq y \land y \leq z\}$

Comp. 1, 2
What if Conditions don’t Match

- Sometimes we need to compose segments of code, but the expectation of the first doesn’t match the assumption of the second.

- In this case, we seek help from the consequent rule, together with composition.

Example of Weakening/Strengthening

1. \{T\} S1 \{x < y\}
2. \{x \leq y\} S2 \{x \leq y \land y \leq z\}
3. To compose these we can either use consequent, then composition to get:
   \{T\} S1;S2 \{x \leq y \land y \leq z\}
   since x < y \rightarrow x \leq y

Generalized Composition Rule

\[ \begin{array}{c}
\{A\} \quad \{B\} \\
\begin{array}{c}
B \rightarrow C \\
\{C\} \quad \{D\}
\end{array}
\end{array} \]

\[ \{A\} \quad \{D\} \quad \text{compose} \]

Conditional Rule

\[ \begin{array}{c}
\{A \land P\} \quad \{B\} \\
\{A \land \neg P\} \quad \{S2\} \quad \{B\}
\end{array} \]

\[ \{A\} \quad \text{if } P \text{ else } S2 \quad \{B\} \quad \text{cond}\]

There is a strong resemblance to \(\lor\)-Elimination.

Called "If rule" in H&R, "choice rule" in JAPE.

Example of Conditional Rule

1. \{x \leq y \land (y > z)\} S1 \{x \leq y \land y \leq z\} same expectations
2. \{x \leq y \land \neg(y > z)\} S2 \{x \leq y \land y \leq z\}
3. \{x \leq y\}
   \quad \text{if}(y > z) \text{ S1 else } S2
   \quad \{x \leq y \land y \leq z\}

One-Sided Conditional Rule

\[ \begin{array}{c}
\{A \land P\} \quad \{B\} \\
(A \land \neg P) \rightarrow B
\end{array} \]

\[ \{A\} \quad \text{if } P \text{ else } S1 \quad \{B\} \quad \text{cond-1} \]
Example of One-Sided Conditional Rule

1. \( \{ x \leq y \land y > z \} \) \( S \) \( \{ x \leq y \land y \leq z \} \)
2. \( ((x \leq y) \land \neg(y > z)) \rightarrow (x \leq y \land y \leq z) \)
3. \( \{ x \leq y \} \)
   \( \text{if}(y > z) \) \( S \)
   \( \{ x \leq y \land y \leq z \} \) \( \text{cond-1, 1, 2} \)

While Rule

\[
\{ I \land P \} \quad S \quad \{ I \}
\]
\[\{ I \}\] \( \text{while}( P ) \) \( S \) \( \{ I \land \neg P \} \)
while

I is known as the "loop invariant"

Example of While Rule

1. \( \{ x \leq y \land y \geq z \} \) \( S \) \( \{ x \leq y \} \)
2. \( \{ x \leq y \} \)
   \( \text{while}(y \geq z) \) \( S \)
   \( \{ x \leq y \land \neg(y \geq z) \} \) \( \text{while, 1} \)

Assignment Statement Rule

\[
\{ A[\epsilon/\nu] \} \quad \nu := \epsilon \quad \{ A \}
\]
\( \nu \) is a variable, an \( \epsilon \) expression.
As in predicate logic, \( A[\epsilon/\nu] \) denotes the result of replacing free occurrences of variable \( \nu \) in \( A \) with \( \epsilon \).

(This rule has an \underline{empty} antecedent.)

Example of Assignment Rule

\[
\{ A[\epsilon/\nu] \} \quad \nu := \epsilon \quad \{ A \}
\]
1. \( \{ x \leq z \} \quad y := z \quad \{ x \leq y \} \) \( \text{assign} \)
   
   Here \( \nu \) is identified with \( y \)
   \( \epsilon \) is identified with \( z \)

   It is easiest to "work backward" from the expectation.

More Examples of Assignment Rule

\[
\{ A[\epsilon/\nu] \} \quad \nu := \epsilon \quad \{ A \}
\]
1. \( \{ x \leq y + 1 \} \quad y := y + 1 \quad \{ x \leq y \} \) \( \text{assign} \)
2. \( \{ x \leq y \} \quad y := x \cdot y \quad \{ y \leq n \} \) \( \text{assign} \)
3. \( \{ x \leq n + 1 \} \quad x := x + 1 \quad \{ x \leq n + 1 \} \) \( \text{assign} \)
Examples of Derivations of Small Programs: Exchange Program

To derive: A program that exchanges the values in variables x and y.

\( \{ x = a \land y = b \} \)

1. \( z := x; x := y; y := z; \) assign

2. \( (x = a \land y = b) \) assign

3. \( z := x; x := z; \) assign

4. \( (x = a \land y = b) \) assign

5. \( (x = a \land y = b) \) assign

In tree form

Examples of Derivations of Small Programs: Ordering two numbers

- \( \{ x = a \land y = b \} \)

  if( \( x > y \) ) { \( z := x; x := y; y := z; \) }

- \( \{ x < n \} \) while( \( x < n \) ) \( x := x + 1 \) \( \{ x = n \} \)

- We can use the while rule here, provided that we can rely on properties of integer arithmetic such as:
  \( \{ x < n \} \) while( \( x < n \) ) \( x := x + 1 \) \( \{ x = n \} \)

Examples of Derivations of Small Programs

- \( \{ x < n \} \) while( \( x < n \) ) \( x := x + 1 \) \( \{ x = n \} \)

- We can use the while rule here, provided that we can rely on properties of integer arithmetic such as:
  \( \{ x < n \} \) while( \( x < n \) ) \( x := x + 1 \) \( \{ x = n \} \)

- Another Viewpoint: Floyd Verification Conditions

  An alternate, less formal, way to view a triple, such as:
  \( \{ x < n \} \) while( \( x < n \) ) \( x := x + 1 \) \( \{ x = n \} \)

  - Think of the assignment in terms of primed (after) and unprimed values:
    \( x' = x + 1 \) (mathematical equality)

  Then what we are proving is the following verification condition:
  \( \{ x + 1 \leq n \land (x' = x + 1) \} \rightarrow (x' \leq n) \)

  Proving the program reduces to proving a set of verification conditions, one for each transition in the program.
  Once the VC's are constructed, the program can be forgotten.
Using JAPE

- JAPE’s theory “Hoare logic” contains rules similar to what we have described, in addition to:
  - natural deduction
  - rules for dealing with equalities and inequalities.
- It is not complete, although very usable for instruction.

JAPE Hoare Logic Rules

- Program
- Extra
- Comparison (bi-directional)

Array Indexing

JAPE Hoare Logic Rules from Natural Deduction

- Backward
- Forward

+ Function Symbols

Hoare Logic Examples in JAPE

On-the-Fly Arithmetic Axioms

- Two-Variable Example
  - Triple to be proved
    (We will discuss the DISTINCT issue in a bit.)
Applying the Variable-Assignment Rule

Note: The goal triple (3) is not quite an instance of the assignment rule. Therefore JAPE constructs the instance (2) given the final expectation, and introduces (1) the logical implication needed by the consequence(L) rule to make (2) provable. It is then up to use to prove (1).

Now the program aspect is done; pure logic remains

- Using →E

The HL rules have “all at once” ∧E, ∧I

• Using ∧I and ∧E

Full Arithmetic is Not Available

Best Way to Add Axioms On-the-Fly

• Create a lemma (in Useful Lemmas), then apply it.

• This puts all such assumptions in a common place (the lemmas area) and calls them out by name.

• All “obviously” justifications then appear only inside lemmas.

Using Lemmas Isolates the “Obvious”
What is the PROVIDED ... thing?

• HL rules are sound only if the LHS of an assignment statement is not aliased to another variable.
• JAPE observes this requirement.
• The proviso states this as an assumption.
• Without the proviso, substitutions will be messy.

How to add your own Provisos

• Not well-documented:
  Prefix the triple with the WHERE DISTINCT ... vars... IS at the time of creating your conjecture:

Without Proviso

• You get a mess.
• Here postfix « i+j / i » means the result of substituting i +j for free occurrences of i.

Unifying _B6 using the assignment rule

was _B6

Unifying _A5 using the assignment rule

was _A5
The Implication is All That’s Left

This expression consists of alternating nested implications and conjunctions, and is proved using the respective rules.

The implications and conjunctions can be expanded to the point where their verification is trivial.

Conclusion, using some lemmas

while rule in JAPE

- We need to discuss termination first.
- JAPE will not prove a while program without considering it.

Partial vs. Total Correctness

- So far, only dealt with "partial correctness":
  - If the assumption is true and the program terminates, then the expectation will be true.

- Of greater interest is "total correctness":
  - If the assumption is true, then the program terminates with the expectation being true.
Partial vs. Total Correctness

- Total Correctness = Partial Correctness + Termination

How to Prove Termination?

- A program terminates if it progress inexorably to a final state.
- Identify a function \( \mu \) of state (called a variant):
  \( \eta : \text{States} \rightarrow \mathbb{N} \) (Natural Numbers)
  such that, on every iteration, \( \eta \) decreases in value.
- Because the range of \( \eta \) is non-negative, there is a limit to the number of iterations.

Termination Example

\[
\begin{align*}
n &:= n_0; \\
\text{while}( n > 0 ) & \\
\{ & \\
\quad \ldots & \\
\quad n &:= n-1; \\
\} \\
\end{align*}
\]

What is an acceptable \( \eta \) in this case?

Termination Example 2

\[
\begin{align*}
n &:= 0; \\
\text{while}( n < n_0 ) & \\
\{ & \\
\quad \ldots & \\
\quad n &:= n+1; \\
\} \\
\end{align*}
\]

What is an acceptable \( \eta \) in this case?

Termination Example 3

\[
\begin{align*}
\{ m_0 > 0 \land n_0 > 0 \} & \text{// assumption} \\
m &:= m_0; n := n_0; \\
\text{while}( -(m = n) ) & \\
\{ & \\
\quad \text{if}( m < n ) n := n-m; \text{ else } m := m-n; \\
\} \\
\{ m = \gcd(m_0, n_0) \} & \text{// expectation} \\
\end{align*}
\]

What is an acceptable \( \eta \) in this case?

Termination Variants in JAPE

- JAPE uses an expression, say \( _M \), giving the value of \( \eta \).
- It is up to the user to specify \( _M \).
- It sets up two termination templates for while \( P \) do \( B \):
  - \( 1 \land P \rightarrow ( _M > 0 ) \) meaning that if the loop continues then \( _M \) is positive.
  - \( 1 \land P \land _M = Km \land ( _M < Km ) \)
    meaning that the value of \( _M \) decreases during the execution of the body.
- \( Km \) is introduce to represent the value of \( _M \) before the loop body.
- For comparision, the partial correctness template is:
  - \( 1 \land P \) \( \Rightarrow ( 1 ) \)
Proof of the previous program

- What is the loop invariant?
- What is an appropriate variant?

JAPEish Version

WHERE DISTINCT m,n,m0,n0,gcd IS

\( \{ m0 > 0 \land n0 > 0 \} \)

\( m:=m0; n:=n0; \)

while \( \neg (m=n) \)

\( \text{do if } m < n \text{ then } n:=n-m \text{ else } m:=m-n \text{ fi} \)

\( \text{od} \)

\( \{ m = \text{gcd}(m0,n0) \} \)

JAPE proof

- Some Lemmas
  - \( \text{gcd}(A, A) = A \)
  - \( \text{gcd}(A, B) = X \text{ if } \text{gcd}(B, A) = X \)
  - \( \text{gcd}(A, B) = X \text{ if } \text{gcd}(A-B, B) = X \)
  - \( \text{gcd}(A, B) = X \text{ if } \text{gcd}(A, B-A) = X \)

Informal Proof of \( \text{gcd}(A,B) = X \text{ if } \text{gcd}(A-B,B) = X \)

- Show that pairs \( \{A, B\} \) and \( \{A-B, B\} \) have the same divisors. Therefore they have the same \( \text{gcd} \).
- If \( d \) divides both \( A \) and \( B \), then there are \( A' \) and \( B' \) such that \( A = dA' \) and \( B = dB' \).
  - But then \( A-B = d(A'-B') \), so \( d \) divides \( A-B \) as well.
  - Conversely, if \( d \) divides both \( A-B \) and \( B \), then \( d \) divides \( (A-B)+B \), which is \( A \).

GCD Program Proof in JAPE

\[
\begin{align*}
\text{gcd}(m, n) & = \text{gcd}(m0, n0) \\
\text{Apply the sequence rule} & \\
\text{gcd}(m, n) & = \text{gcd}(m0, n0) \\
\text{Proposed GCD Loop Invariant} & \\
\text{gcd}(m, n) & = \text{gcd}(m0, n0) \\
\text{Unify this with _B2} &
\end{align*}
\]
Resolve the Initialization Steps

Mostly this is automated with the assignment rule.

Focus on the while loop

Using the while rule introduces multiple new goals:

- Goals relating to partial correctness
- Goals relating to termination

Partial Correctness Goals

Consequent implication after the loop. This states that the loop end condition implies the overall expectation.

\[
I \land \neg P \quad \text{since } P = \neg (m=n)
\]

Verification Condition for the loop body:

\[
\begin{align*}
I \land P \land _M = Km & \Rightarrow _M < Km \\
\text{if } m > n & \text{ then } _M = Km
\end{align*}
\]

Template: \( \{ I \land P \} \ B \ \{ I \} \)

Termination Goals

Verification Condition for the loop end (an implication):

\[
I \land P \Rightarrow (_M > 0) \quad \text{Template: } I \land P \Rightarrow (_M > 0)
\]

\_M is a variant expression, to be determined

Verification Condition for the loop body (a triple):

\[
\begin{align*}
I \land P \land _M = Km & \Rightarrow _M < Km \\
\text{if } m > n & \text{ then } _M = Km
\end{align*}
\]

Template: \( \{ I \land P \land _M = Km \} \ B \ \{ _M < Km \} \)

Choice of variant

- The variant must be chosen so that the two goals are provable.
- It may be necessary to revisit the invariant, to add to it conditions that make the goals provable.

Termination Goals

A feasible choice for \(_M\) is \(m+n\). But will these be provable for that \(_M\)? Or do we need more?
Try proving the body triple with \( _M = m+n \)

Generated Goals

If we are correct in these needs, we would have to introduce them into the invariant and reprove it.

Partial Correctness Redone

Program with Added Intermediate Assertion

(This program contains a typographical error. Can you spot it? I didn’t discover it until half-way through the proof, and I am leaving it in for illustration. It is a good example of why proving is helpful. I will correct the program later in these slides.)

Use of the “Ntuple” Rule when intermediate assertions are included

The “Ntuple” Rule "hinges" the proof at the intermediate assertion

Section Above the Intermediate Assertion Resolved

Section below intermediate assertion is left
while rule applied

Partial Correctness of Loop Ending

Partial Correctness Part of Loop Body

Termination Part of Loop Body

It was in failing to complete the proof of this part that I detected the error.

This gap cannot be closed.

n - m is positive, and is not necessarily less than m.

Corrected Program

Use Ntuple rule:

On the next slides, the completed proof is discussed.
Proof above the intermediate assertion
This section (lines 1-11) proves the initialization steps.
No separate termination proof is required, as there are no loops.

This is the proved triple for the initial part.
The expectation of this triple becomes the assumption
for the triple for rest of the program, as shown in line 68.
The two pieces are composed using the Ntuple rule in line 69.

Proof below the intermediate assertion, part 1
Template: \( \{I \land P\} \text{ B } \{I\} \)
Lines 12-34 comprise the partial correctness proof of the while body (lines 12-34).
The assumption is the expectation from line 11, conjoined with the loop test.

Proof below the intermediate assertion, part 2
Template: \( I \land P \rightarrow (_M > 0) \)
Lines 35-39 are part of the termination proof, using the variant \( m+n \).
It states that if the invariant and the test condition of the while are true,
then the variant is \( > 0 \). This is pure logic, not a triple.

Proof below the intermediate assertion, part 3
Template: \( \{I \land P \land _M = Km\} \text{ B } \{_M < Km\} \)
Lines 40-59 comprise the part of the termination proof, using the variant \( m+n \),
relating to the body of the while. It shows that the variant strictly decreases
as a result of the body being executed.
The assumption is that the variant has a value \( > 0 \),
along with the test condition and the invariant.

Proof below the intermediate assertion, part 3
Lines 60-68 provide the implication used in the consequence(R) rule,
to link the expectation of the while loop with the overall expectation.

Derivation as trees (see also: Huth & Ryan fig. 4.2)
Numbers refer to line numbers of formulas on previous pages

Partial-correctness tree:

Termination tree: