(Imperative) Program Logic

Robert Keller
February 2011

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?

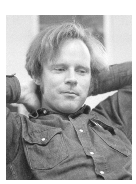
  … make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

A corrected version, with comments, was published by
F.L. Morris and C.B. Jones in Annals of the History of Computing,
(Vol. 6, Apr. 1984)

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

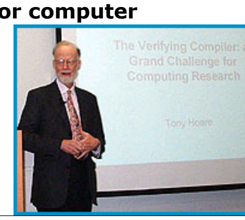
Robert W. Floyd

- "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 $\vdash P\{Q_1\}R_1$ and $\vdash R_1\{Q_2\}R$ then $\vdash P\{(Q_1\,;\,Q_2)\}R$

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 → domain

Example: An integer square-root program

```
s = 1;
i = 1;
r = 0;
while( s ≤ n )
    {
    r = r + 1;
    i = i + 2;
    s = s + i;
    }
```

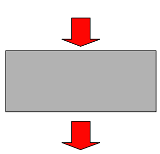
variables

rows are states

s	i	r	n
			10
1			10
1	1		10
1	1	0	10
1	1	1	10
1	3	1	10
4	3	1	10
4	3	2	10
4	5	2	10
9	5	2	10
9	5	3	10
9	7	3	10
16	7	3	10
16	7	3	10

Program as a "State Transformer"

starting state



ending state

Note about I/O

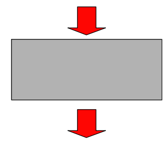
- To deal with input streams and files, we will **consider the entire file or stream**, along with the current position of the reader or writer, to be part of the state.
- We won't be dealing with such issues in this presentation.

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".

A **Program Specification** consists of

(i) **assumption** about the starting state

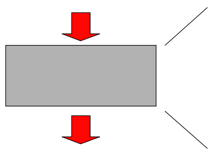


Program as a gray box.

(ii) **expectation** about the ending state

Example for the previous program

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



```
s = 1;
i = 1;
r = 0;
while( s ≤ n )
    {
    r = r + 1;
    i = i + 2;
    s = s + i;
    }
```

(ii) **expectation:** r = isqrt(n)

Relativity

- Expectations are *relative to* assumptions.
- Nothing in particular can be expected if the assumption is false when the program is started.

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.

Example

- Assumption:

  int x[0..n-1] 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

```
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:

  int x[0..n-1] is an array of size n
  $\wedge$ **$x = x_0$**
  (in the sense that x and x_0 are two arrays with the same elements)

- Expectation:

  $\forall i\ ((i \geq 1 \wedge i < n) \rightarrow (x[i-1] \leq x[i]))$
  $\wedge$ **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:

  $\quad$ valid(input) $\rightarrow$ T
  $\wedge$ $\neg$valid(input) $\rightarrow$ red_flag

- Expectation:
  $\neg$red_flag $\rightarrow$ … normal expectation …
  $\wedge$ red_flag $\rightarrow$ **exception** is indicated

- The value of valid(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 ifs

- 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 ifs

- 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

```
         ----- n≥0
s = 1;   ----- n≥0 ∧ s=1
i = 1;   ----- n≥0 ∧ s=1 ∧ i=1
r = 0;   ----- n≥0 ∧ s=(r+1)² ∧ i=2r+1
while( s ≤ n )
    {
    r = r + 1;----- r ²≤n ∧ n≥0 ∧ s=(r+1)² ∧ i=2r+1
    i = i + 2;----- (to be completed)
    s = s + i;
    }
              ----- r² ≤ n < (r+1)²
```

Floyd Assertions

```
         ----- n≥0
s = 0;   ----- n≥0 ∧ s=0
i = 1;   ----- n≥0 ∧ s=0 ∧ i=1
r = 0;   ----- n≥0 ∧ s=r² ∧ i=2r+1
while( s < n )
    {
    s = s + i;----- s<n ∧ n≥0 ∧ s=r² ∧ i=2r+1
    i = i + 2;----- (to be completed)
    r = r + 1;
    }         ----- r²≤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.

  $(s<n \wedge n\geq 0 \wedge s=r^2 \wedge i=2r+1)$ pre-condition
  $\wedge\ (s' = s + i)$ statement semantics
  $\rightarrow (s'<n+i \wedge n\geq 0 \wedge s'=r^2+i \wedge i=2r+1)$ post-condition

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:

  {assumption} code {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

{assumption} code {expectation}

$\{x \leq y \wedge x \leq z\}$ *…TBD…* $\{x \leq y \wedge y \leq z\}$

[TBD = “To Be Determined”]

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

Some triples are more stringent than others.

{assumption} code {expectation}

$\{x \leq y \wedge x \leq z\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{x \leq y\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{T\}$ *TBD* $\{x \leq y \wedge y \leq z\}$

$\{T\}$ *TBD* $\{x \leq y \wedge y \leq z \wedge z \leq w\}$

increasing demands on code

Stringency

- {assumption} code {expectation}
- T_1:{A} c {E} for short
- T_2:{A} c {E’}
- T_3:{A’} c {E}
- If E’ → E, is T_2 more or less stringent than T_1?
- If A’ → A, is T_3 more or less stringent than T_1?

Rationale

- If E’ → E, then any state satisfying E’ must also satisfy E, but not necessarily conversely, so {A} c {E’} is **more** stringent than {A} c {E}.
- If A’ → A, then any state satisfying A’ must also satisfy A, so {A’} c {E} is less stringent than {A} c {E}.
- In other words,
  - {A} c {E’} meets the expectation E and **possibly more**.
  - {A’} c {E} **assumes more** than A to get the same job done.

Extreme Stringency

- $\{T\}$ c $\{\perp\}$
- Here c would not assume anything, but meets every expectation.

Extreme Leniency

- $\{\perp\}$ c $\{T\}$
- Assuming everything, don't expect anything.

First Rule of Inference

- Consequent Rule

$$\frac{A \to A',\ \{A'\}\ c\ \{E'\},\ E' \to E}{\{A\}\ c\ \{E\}}\ \text{consequent}$$

- 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):

$$\frac{A \to A',\ \{A'\}\ c\ \{E\}}{\{A\}\ c\ \{E\}}\ \text{consequent(L)}$$

- consequent(R):

$$\frac{\{A'\}\ c\ \{E\},\ E' \to E}{\{A\}\ c\ \{E\}}\ \text{consequent(R)}$$

Composition of Triples

- Suppose we have a triple: {Assumption} Code {Expectation}
- To develop the code, we can break it into two parts:

  {Assumption 1} Code 1 {Expectation 1}

  {Assumption 2} Code 2 {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:

$$\frac{\{A\}\ S1\ \{B\} \qquad \{B\}\ S2\ \{C\}}{\{A\}\ S1;S2\ \{C\}}\ \text{composition}$$

Example of Composition Rule

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

   $\{T\}$ *S1;S2* $\{x \leq y \wedge y \leq z\}$

   since $x < y \rightarrow x \leq y$

Generalized Composition Rule

$$\frac{\{A\}\ S1\ \{B\} \qquad B \rightarrow C \qquad \{C\}\ S2\ \{D\}}{\{A\}\ S1;S2\ \{D\}}\ \text{compose}$$

Conditional Rule

$$\frac{\{A \wedge P\}\ S1\ \{B\} \qquad \{A \wedge \neg P\}\ S2\ \{B\}}{\{A\}\ \textbf{if(}\ P\ \textbf{)}\ S1\ \textbf{else}\ S2\ \ \{B\}}\ \text{cond}$$

There is a strong resemblance to $\vee$–Elimination.

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

Example of Conditional Rule

1. $\{x \leq y \wedge (y > z)\}$ *S1* $\{x \leq y \wedge y \leq z\}$ same expectations
2. $\{x \leq y \wedge \neg(y > z)\}$ *S2* $\{x \leq y \wedge y \leq z\}$
3. $\{x \leq y\}$

   if(y > z) S1 **else** S2

   $\{x \leq y \wedge y \leq z\}$ cond 1, 2

One-Sided Conditional Rule

$$\frac{\{A \wedge P\}\ S1\ \{B\} \qquad (A \wedge \neg P) \rightarrow B}{\{A\}\ \textbf{if(}\ P\ \textbf{)}\ S1\ \{B\}}\ \text{cond-1}$$

Example of One-Sided Conditional Rule

1. $\{x \leq y \wedge y > z\}$ *S1* $\{x \leq y \wedge y \leq z\}$
2. $((x \leq y) \wedge \neg(y > z)) \rightarrow (x \leq y \wedge y \leq z)$
3. $\{x \leq y\}$

   if(y > z) S1

   $\{x \leq y \wedge y \leq z\}$ cond-1, 1, 2

While Rule

$$\frac{\{I \wedge P\}\ S\ \{I\}}{\{I\}\ \textbf{while(P) S}\ \ \{I \wedge \neg P\}}\ \text{while}$$

I is known as the "loop invariant"

Example of While Rule

1. $\{x \leq y \wedge y \geq z\}$ *S* $\{x \leq y\}$
2. $\{x \leq y\}$

   while($y \geq z$) S

   $\{x \leq y \wedge \neg(y \geq z)\}$ while, 1

Assignment Statement Rule

$$\frac{}{\{A[\varepsilon/v]\}\quad v := \varepsilon \qquad \{A\}}\ \text{assign}$$

v is a variable, an ε expression.
As in predicate logic, $A[\varepsilon/v]$ denotes the result of replacing free occurrences of variable v in A with ε.

(This rule has an **empty** antecedent.)

Example of Assignment Rule

$$\{A[\varepsilon/v]\}\quad v := \varepsilon \qquad \{A\}$$

1. $\{x \leq z\}$ **y := z** $\{x \leq y\}$ assign

   Here v is identified with y

   ε is identified with z

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

More Examples of Assignment Rule

$$\{A[\varepsilon/v]\}\quad v := \varepsilon \qquad \{A\}$$

1. $\{x \leq y+1\}$ **y := y+1** $\{x \leq y\}$ assign
2. $\{x*y \leq n\}$ **y := x*y** $\{y \leq n\}$ assign
3. $\{x+1 \leq n+1\}$ **x := x+1** $\{x \leq n+1\}$ assign

Examples of Derivations of Small Programs: **Exchange Program**

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

{x = a ∧ y = b} **z := x; x := y; y := z;** {y = a ∧ x = b}

1. {z = a ∧ x = b} **y := z;** {y = a ∧ x = b} assign
2. {z = a ∧ y = b} **x := y;** {z = a ∧ x = b} assign
3. {x = a ∧ y = b} **z := x;** {z = a ∧ y = b} assign
4. {z = a ∧ y = b} **x := y; y := z;** {y = a ∧ x = b} comp 2, 1
5. {x = a ∧ y = b} **z := x; x := y; y := z;** {y = a ∧ x = b} comp 3, 4

In tree form

{z = a ∧ y = b} **x := y;** {z = a ∧ x = b}   {z = a ∧ x = b} **y := z;** {y = a ∧ x = b}

{x = a ∧ y = b} **z := x;** {z = a ∧ y = b}   {z = a ∧ y = b} **x := y; y := z;** {y = a ∧ x = b}

{x = a ∧ y = b} **z := x; x := y; y := z;** {y = a ∧ x = b}

Examples of Derivations of Small Programs: Ordering two numbers

- {x = a ∧ y = b}

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

  {x ≤ y ∧ ((x = a ∧ y = b) ∨ (y = a ∧ x = b))}

- We'll obviously be needing the 1-sided conditional rule.
- We'll assume some things about the < and ≤ predicates:
  - ¬(x > y) → (x ≤ y)
  - (y > x) → (x ≤ y)
- Similar to the derivation on the previous page, we can derive:
  {x > y ∧ x = a ∧ y = b}
  z := x; x := y; y := z;
  {y > x ∧ y = a ∧ x = b}
  and using expectation weakening, we can replace the expectation with
  {x ≤ y ∧ y = a ∧ x = b}
- Then identify P in the 1-sided cond rule as: x > y

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) → ((x+1) ≤ n)

  ((x ≤ n) ∧ ¬(x < n)) ≡ (x = n)

Examples of Derivations of Small Programs

1. ((x ≤ n) ∧ ¬(x < n)) ≡ (x = n) Premise
2. (x < n) → ((x+1) ≤ n) Premise
3. {x+1 ≤ n} **x := x+1** {x ≤ n} Assignment
4. {x < n} **x := x+1** {x ≤ n} Assumption strengthening 3, 2
5. {x ≤ n ∧ x < n} **x := x+1** {x ≤ n} Assumption strengthening 4
6. {x ≤ n} while(x < n) **x := x+1** {x ≤ n ∧ ¬(x < n)} While 4
7. {x ≤ n} while(x < n) **x := x+1**{x = n} Expectation weakening 6

Another Viewpoint: **Floyd Verification Conditions**

- An alternate, less formal, way to view a triple, such as:

  {x+1 ≤ 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 ≤ n ∧ (x' = x + 1)) → (x' ≤ 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

skip
tilt
sequence
Ntuple
variable-assignment
array-element-assignment
choice
while

consequence(L)
consequence(R)

Extra

$A=A$

$A = ..$
$.. = B$

obviously

boundedness from (in)equality

Array Indexing

FROM $E=G$ INFER $(A\oplus E\mapsto F)[G]=F$
FROM $E\neq G$ INFER $(A\oplus E\mapsto F)[G]=A[G]$

$A = \ldots$ $\ldots = B$

Comparison (bi-directional)

$A=B \triangleq B=A$
$A=B \triangleq \neg(A\neq B)$
$A\neq B \triangleq B\neq A$
$A\neq B \triangleq \neg(A=B)$
$A<B \triangleq B>A$
$A\leq B \triangleq A<B \vee A=B$
$A\leq B \triangleq B\geq A$
$A\leq B \triangleq \neg(A>B)$
$A\leq B \triangleq A<B+1$
$A+1\leq B \triangleq A<B$
$A\geq B \triangleq \neg(A<B)$
$A\geq B \triangleq A>B-1$
$A-1\geq B \triangleq A>B$

$A \triangleq \ldots$ $\ldots \triangleq B$

JAPE Hoare Logic Rules from Natural Deduction

Backward

↔ intro
∧ intro (all at once)
∧ intro (one step)
→ intro (makes assumption)
∨ intro (preserving left)
∨ intro (preserving right)
¬ intro (makes assumption A)
∀ intro (introduces variable)
∃ intro (needs formula)
truth

contra (classical; makes assumption ¬A)
contra (constructive)
¬ elim (invents formulae)

hyp

Forward

∧ elim (all at once)
∧ elim (preserving left)
∧ elim (preserving right)
→ elim
∨ elim (makes assumptions)
¬ elim
∀ elim (needs formula)
∃ elim (assumption & variable)
contra (constructive)

∧ intro
∨ intro (invents right)
∨ intro (invents left)

hyp

\+ Function Symbols

Hoare Logic Examples in JAPE

...
1: $\{i=2\}(i:=i+1)\{i=3\}$

...
1: $i=2\rightarrow i+1=3$
2: $\{i+1=3\}(i:=i+1)\{i=3\}$ variable-assignment
3: $\{i=2\}(i:=i+1)\{i=3\}$ consequence(L) 1,2

1: $i=2$ assumption
...
2: $i+1=3$
3: $i=2\rightarrow i+1=3$ → intro 1-2
4: $\{i+1=3\}(i:=i+1)\{i=3\}$ variable-assignment
5: $\{i=2\}(i:=i+1)\{i=3\}$ consequence(L) 3,4

On-the-Fly Arithmetic Axioms

1: $i=2$ assumption
2: $i+1=3$ $i=2 \vdash i+1=3$ 1
3: $i=2\rightarrow i+1=3$ → intro 1-2
4: $\{i+1=3\}(i:=i+1)\{i=3\}$ variable-assignment
5: $\{i=2\}(i:=i+1)\{i=3\}$ consequence(L) 3,4

Proof of Lemma

1: $i=2$ premise
2: $i+1=3$ obviously

Two-Variable Example

- Triple to be proved
  (We will discuss the DISTINCT issue in a bit.)

...
1: $\{i=5\wedge j=10\}(i:=i+j)\{i=15\wedge j=10\}$

Provided:
DISTINCT i, j

Applying the Variable-Assignment Rule

	...	
1:	$i=5 \land j=10 \rightarrow i+j=15 \land j=10$	
2:	$\{i+j=15 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	variable-assignment
3:	$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	consequence(L) 1,2

Provided:
DISTINCT i, j

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 $\rightarrow$E

1:	$i=5 \land j=10$	assumption
	...	
2:	$i+j=15 \land j=10$	
3:	$i=5 \land j=10 \rightarrow i+j=15 \land j=10$	$\rightarrow$ intro 1-2
4:	$\{i+j=15 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	variable-assignment
5:	$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	consequence(L) 3,4

Provided:
DISTINCT i, j

The HL rules have "all at once" $\land$E, $\land$I

- Using $\land$I and $\land$E

1:	$i=5 \land j=10$	assumption
2:	$i=5$	$\land$ elim 1
3:	$j=10$	$\land$ elim 1
	...	
4:	$i+j=15$	
5:	$i+j=15 \land j=10$	$\land$ intro 4,3
6:	$i=5 \land j=10 \rightarrow i+j=15 \land j=10$	$\rightarrow$ intro 1-5
7:	$\{i+j=15 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	variable-assignment
8:	$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	consequence(L) 6,7

Provided:
DISTINCT i, j

Full Arithmetic is Not Available

1:	$i=5 \land j=10$	assumption
2:	$i=5$	$\land$ elim 1
3:	$j=10$	$\land$ elim 1
4:	$i+j=15$	obviously, from 3,2
5:	$i+j=15 \land j=10$	$\land$ intro 4,3
6:	$i=5 \land j=10 \rightarrow i+j=15 \land j=10$	$\rightarrow$ intro 1-5
7:	$\{i+j=15 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	variable-assignment
8:	$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	consequence(L) 6,7

Provided:
DISTINCT i, j

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"

$i=5,\ j=10 \vdash i+j=15$

1:	$i=5,\ j=10$	premises
2:	$i+j=15$	obviously

$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$

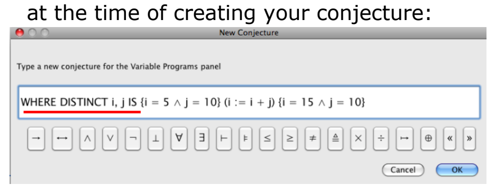
1:	$i=5 \land j=10$	assumption
2:	$i=5$	$\land$ elim 1
3:	$j=10$	$\land$ elim 1
4:	$i+j=15$	$i=5, j=10 \vdash i+j=15$ 2,3
5:	$i+j=15 \land j=10$	$\land$ intro 4,3
6:	$i=5 \land j=10 \rightarrow i+j=15 \land j=10$	$\rightarrow$ intro 1-5
7:	$\{i+j=15 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	variable-assignment
8:	$\{i=5 \land j=10\}(i:=i+j)\{i=15 \land j=10\}$	consequence(L) 6,7

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.

. . .
1: i=5∧j=10→i+j=15∧j«i+j/i»=10
2: {i+j=15∧j«i+j/i»=10} (i:=i+j){i=15∧j=10}   variable-assignment
3: {i=5∧j=10}(i:=i+j){i=15∧j=10}   consequence(L) 1,2

Conditional Example

. . .
1: {⊤}if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}

Using the ‘choice’ rule introduces two triples and a logical implication. The triples contain **un-unified formulas**, the assumptions for the two branches. Those formulas may be derivable automatically. The implication is essentially a variant on the consequent(L) rule.

. . .
1: ⊤→(j>k→_A5)∧(¬(j>k)→_B6)
. . .
2: {_A5}(i:=j){(j≥k→i=j)∧(k≥j→i=k)}
. . .
3: {_B6}(i:=k){(j≥k→i=j)∧(k≥j→i=k)}
4: {(j>k→_A5)∧(¬(j>k)→_B6)} if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   choice 2,3
5: {⊤}if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   consequence(L) 1,4

Unifying _B6 using the assignment rule

was _B6

. . .
1: ⊤→(j>k→_A5)∧(¬(j>k)→(j≥k→k=j)∧(k≥j→k=k))
. . .
2: {_A5}(i:=j){(j≥k→i=j)∧(k≥j→i=k)}
3: {(j≥k→k=j)∧(k≥j→k=k)}(i:=k){(j≥k→i=j)∧(k≥j→i=k)}   variable-assignment
4: {(j>k→_A5)∧(¬(j>k)→(j≥k→k=j)∧(k≥j→k=k))} if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   choice 2,3
5: {⊤}if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   consequence(L) 1,4

Unifying _A5 using the assignment rule

was _A5

. . .
1: ⊤→(j>k→(j≥k→j=j)∧(k≥j→j=k))∧(¬(j>k)→(j≥k→k=j)∧(k≥j→k=k))
2: {(j≥k→j=j)∧(k≥j→j=k)}(i:=j){(j≥k→i=j)∧(k≥j→i=k)}   variable-assignment
3: {(j≥k→k=j)∧(k≥j→k=k)}(i:=k){(j≥k→i=j)∧(k≥j→i=k)}   variable-assignment
4: {(j>k→(j≥k→j=j)∧(k≥j→j=k))∧(¬(j>k)→(j≥k→k=j)∧(k≥j→k=k))} if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   choice 2,3
5: {⊤}if j>k then i:=j else i:=k fi{(j≥k→i=j)∧(k≥j→i=k)}   consequence(L) 1,4

The Implication is All That's Left

...

1: $\top\rightarrow(j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k))\wedge(\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k))$

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

1: $\top$
2: $j>k$
...
3: $(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$
4: $j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$
5: $\neg(j>k)$
...
6: $(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$
7: $\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$
8: $(j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k))\wedge(\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k))$

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

2: $j>k$
3: $j\geq k$
...
4: $j=j$ trivial (A=A)
5: $j\geq k\rightarrow j=j$
6: $k\geq j$
...
7: $j=k$
8: $k\geq j\rightarrow j=k$
9: $(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$
10: $j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$

11: $\neg(j>k)$
12: $j\geq k$
...
13: $k=j$
14: $j\geq k\rightarrow k=j$
15: $k\geq j$
...
16: $k=k$ trivial (A=A)
17: $k\geq j\rightarrow k=k$
18: $(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$
19: $\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$

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

2: $j>k$
3: $j\geq k$
4: $j=j$
5: $j\geq k\rightarrow j=j$
6: $k\geq j$
...
7: $j=k$
8: $k\geq j\rightarrow j=k$
9: $(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$
10: $j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$

will combine to $\perp$

11: $\neg(j>k)$
12: $j\geq k$
...
13: $k=j$
14: $j\geq k\rightarrow k=j$
15: $k\geq j$
16: $k=k$
17: $k\geq j\rightarrow k=k$
18: $(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$
19: $\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$

will combine to k = j

Conclusion, using some lemmas

2: $j>k$
3: $j\geq k$
4: $j=j$
5: $j\geq k\rightarrow j=j$
6: $k\geq j$
7: $\perp$
8: $j=k$
9: $k\geq j\rightarrow j=k$
10: $(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$
11: $j>k\rightarrow(j\geq k\rightarrow j=j)\wedge(k\geq j\rightarrow j=k)$

12: $\neg(j>k)$
13: $j\geq k$
14: $j=k$
15: $k=j$
16: $j\geq k\rightarrow k=j$
17: $k\geq j$
18: $k=k$
19: $k\geq j\rightarrow k=k$
20: $(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$
21: $\neg(j>k)\rightarrow(j\geq k\rightarrow k=j)\wedge(k\geq j\rightarrow k=k)$

---

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 μ of state (called a **variant**):

  η: States→N (**Natural Numbers**)

  such that, **on every iteration**, η **decreases** in value.
- Because the range of η is **non-negative**, there is a limit to the number of iterations.

Termination Example

```
n := n0;
while( n > 0 )
    {
    …
    n := n-1;
    }
```

What is an acceptable η in this case?

Termination Example 2

```
n := 0;
while( n < n0 )
    {
    …
    n := n+1;
    }
```

What is an acceptable η in this case?

Termination Example 3

```
{m0 > 0 ∧ n0 > 0} // assumption

m := m0; n := n0;
while( ¬(m = n) )
    {
    if( m < n ) n := n-m; else m := m-n;
    }
{m = gcd(m0, n0)} // expectation
```

What is an acceptable η in this case?

Termination Variants in JAPE

- JAPE uses an **expression**, say **_M**, giving the value of η.
- It is up to the user to specify _M.
- It sets up two **termination templates** for **while P do B**:
  - **I ∧ P → (_M > 0)**
    meaning that if the loop continues then _M is positive.
  - **{I ∧ P ∧ _M = Km} B {_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:

    {I ∧ P} B {I}

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 ∧ n0>0}
 (
 m:=m0; n:=n0;
 while ¬(m=n)
  do
  if m < n then n:= n - m else m := m-n fi
  od
  )
 {m = gcd(m0,n0)}
```

JAPE proof

- Some Lemmas
  - gcd(A, A) = A
  - gcd(A, B) = X |— gcd(B, A) = X
  - gcd(A, B) = X |— gcd(A-B, B) = X
  - gcd(A, B) = X |— gcd(A, B-A) = X

Informal Proof of gcd(A, B) = X |— gcd(A-B, B) = X

- 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 = 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

```
 ...
1: {m0>0∧n0>0}(m:=m0;n:=n0;while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od){m=gcd(m0,n0)}
```

Apply the sequence rule

```
 ...
1: {m0>0∧n0>0}(m:=m0){_B4}
 ...
2: {_B4}(n:=n0){_B2}
 ...
3: {_B2}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
4: {m0>0∧n0>0}(m:=m0;n:=n0;while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od){m=gcd(m0,n0)} sequence 1,2,3
```

Figure out the Loop Invariant _B2

Proposed GCD Loop Invariant

- gcd(m, n) = gcd(m0, n0)
- Unify this with _B2

Resolve the Initialization Steps

```
  ...
1: {m0>0∧n0>0}(m:=m0){_B4}
  ...
2: {_B4}(n:=n0){gcd(m,n)=gcd(m0,n0)}
```

Mostly this is automated with the assignment rule.

```
1: m0>0∧n0>0                                          assumption
2: m0>0                                               ∧ elim 1
3: n0>0                                               ∧ elim 1
4: gcd(m0,n0)defined                                  m0>0, n0>0 ⊢ gcd... 2,3
5: gcd(m0,n0)=gcd(m0,n0)                              A=A 4
6: m0>0∧n0>0→gcd(m0,n0)=gcd(m0,n0)                    → intro 1-5
7: {gcd(m0,n0)=gcd(m0,n0)}(m:=m0){gcd(m,n0)=gcd(m0,n0)}   variable-assignment
8: {m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)}           consequence(L) 6,7
9: {gcd(m,n0)=gcd(m0,n0)}(n:=n0){gcd(m,n)=gcd(m0,n0)}     variable-assignment
```

Focus on the while loop

```
   ...
10: {gcd(m,n)=gcd(m0,n0)}
    while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
```

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.

```
15: gcd(m,n)=gcd(m0,n0)∧¬¬(m=n)→m=gcd(m0,n0)
```

I ∧ ¬P (since P is ¬(m=n))

Verification Condition for the loop body:

```
10: {gcd(m,n)=gcd(m0,n0)∧¬(m=n)}
    if m<n then n:=n−m else m:=m−n fi{gcd(m,n)=gcd(m0,n0)}
```

Template: **{I ∧ P} B {I}**

Termination Goals

Verification Condition for the loop end (an implication):

```
11: gcd(m,n)=gcd(m0,n0)∧¬(m=n)→_M>0
```

Template: **I ∧ P → (_M > 0)**

_M is a variant expression, to be determined

Verification Condition for the loop body (a triple):

```
12: integer Km                                        assumption
    ...
13: {gcd(m,n)=gcd(m0,n0)∧¬(m=n)∧_M=Km}
    if m<n then n:=n−m else m:=m−n fi{_M<Km}
```

Template: **{I ∧ P ∧ _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?

```
11: gcd(m,n)=gcd(m0,n0)∧¬(m=n)→_M>0
```

```
12: integer Km
    ...
13: {gcd(m,n)=gcd(m0,n0)∧¬(m=n)∧_M=Km}
    if m<n then n:=n−m else m:=m−n fi{_M<Km}
```

Try proving the body triple with _M = m+n

```
12: integer Km
    ...
13: {gcd(m,n)=gcd(m0,n0)∧¬(m=n)∧m+n=Km}
    if m<n then n:=n−m else m:=m−n fi{m+n<Km}
```

Template: **{I ∧ P ∧ _M = Km} B {_M < Km}**

```
12: integer Km                                                     assumption
    ...
13: gcd(m,n)=gcd(m0,n0)∧¬(m=n)∧m+n=Km
    →(m<n→m+(n−m)<Km)∧(¬(m<n)→m−n+n<Km)
14: {m+(n−m)<Km}(n:=n−m){m+n<Km}                                   variable-assignment
15: {m−n+n<Km}(m:=m−n){m+n<Km}                                     variable-assignment
16: {(m<n→m+(n−m)<Km)∧(¬(m<n)→m−n+n<Km)}                           choice 14,15
    if m<n then n:=n−m else m:=m−n fi{m+n<Km}
17: {gcd(m,n)=gcd(m0,n0)∧¬(m=n)∧m+n=Km}                             consequence(L) 13,16
    if m<n then n:=n−m else m:=m−n fi{m+n<Km}
```

Generated Goals

```
15: ¬(m=n)
16: m+n=Km
17: m<n
    ...
18: m+(n−m)<Km          } need m > 0
19: m<n→m+(n−m)<Km
20: ¬(m<n)
    ...
21: m−n+n<Km            } need n > 0
22: ¬(m<n)→m−n+n<Km
```

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

```
   ...
1: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od){m=gcd(m0,n0)}
```

(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

```
   ...
1: {m0>0∧n0>0}(m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}
   ...
2: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
   {m0>0∧n0>0}
3: ((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od) Ntuple 1,2
   {m=gcd(m0,n0)}
```

Section Above the Intermediate Assertion Resolved

```
 1: m0>0∧n0>0                                                       assumption
 2: m0>0                                                            ∧ elim 1
 3: n0>0                                                            ∧ elim 1
 4: gcd(m0,n0)defined                                               m0>0, n0>0 ⊢ gcd(m0,n0)defined 2,3
 5: gcd(m0,n0)=gcd(m0,n0)                                           A=A 4
 6: gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0                                 ∧ intro 5,2,3
 7: m0>0∧n0>0→gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0                       → intro 1-6
 8: {gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}   variable-assignment
 9: {m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}               consequence(L) 7,8
10: {gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}(n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}       variable-assignment
11: {m0>0∧n0>0}(m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}           sequence 9,10
```

Section below intermediate assertion is left

```
    ...
12: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
    {m0>0∧n0>0}
13: ((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od) Ntuple 11,12
    {m=gcd(m0,n0)}
```

while rule applied

```
...
12: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)}
    if m<n then m:=n−m else m:=m−n fi{gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}
...
13: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)→_M>0
14: integer Km                                                                       assumption
    ...
15: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧_M=Km}if m<n then m:=n−m else m:=m−n fi{_M<Km}
16: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od   while 12,13,14-15
    {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)}
...
17: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)→m=gcd(m0,n0)
18: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then m:=n−m else m:=m−n fi od{m=gcd(m0,n0)}   consequence(R) 16,17
    {m0>0∧n0>0}
```

Partial Correctness of Loop Ending

```
17: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)        assumption
18: gcd(m,n)=gcd(m0,n0)                         ∧ elim 17
19: ¬¬(m=n)                                     ∧ elim 17
20: m=n                                          ¬¬E ⊢ E 19
21: m=gcd(m0,n0)                                 gcd(A,B)=X, A=B ⊢ A=X 18,20
22: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)→m=gcd(m0,n0)   → intro 17-21
```

Partial Correctness Part of Loop Body

```
12: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)             assumption
13: gcd(m,n)=gcd(m0,n0)                            ∧ elim 12
14: n>0                                            ∧ elim 12
15: ¬(m=n)                                         ∧ elim 12
16: m<n                                            assumption
17: gcd(n−m,n)=gcd(m0,n0)                          gcd(A,B)=X ⊢ gcd(B−A,B)=X 13
18: n>m                                            A<B≜B>A 16
19: n−m>0                                          A>B ⊢ A−B>0 18
20: gcd(n−m,n)=gcd(m0,n0)∧n−m>0∧n>0                ∧ intro 17,19,14
21: m<n→gcd(n−m,n)=gcd(m0,n0)∧n−m>0∧n>0            → intro 16-20
22: ¬(m<n)                                         assumption
23: gcd(m−n,n)=gcd(m0,n0)                          gcd(A,B)=X ⊢ gcd(A−B,B)=X 13
24: m>n                                            ¬(A=B), ¬(A<B) ⊢ A>B 15,22
25: n<m                                            A<B≜B>A 24
26: m−n>0                                          m<n ⊢ n−m>0 25
27: gcd(m−n,n)=gcd(m0,n0)∧m−n>0∧n>0                ∧ intro 23,26,14
28: ¬(m<n)→gcd(m−n,n)=gcd(m0,n0)∧m−n>0∧n>0         → intro 22-27
29: (m<n→gcd(n−m,n)=gcd(m0,n0)∧n−m>0∧n>0)∧(¬(m<n)→gcd(m−n,n)=gcd(m0,n0)∧m−n>0∧n>0)   ∧ intro 21,28
30: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)                                                 → intro 12-29
    →(m<n→gcd(n−m,n)=gcd(m0,n0)∧n−m>0∧n>0)∧(¬(m<n)→gcd(m−n,n)=gcd(m0,n0)∧m−n>0∧n>0)
```

Termination Part of Loop Body

```
14: integer Km                                      assumption
15: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km        assumption
16: gcd(m,n)=gcd(m0,n0)                             ∧ elim 15
17: m>0                                             ∧ elim 15
18: n>0                                             ∧ elim 15
19: ¬(m=n)                                          ∧ elim 15
20: m+n=Km                                          ∧ elim 15
21: m<n                                             assumption
    ...
22: n−m+n<Km
23: m<n→n−m+n<Km                                    → intro 21-22
24: ¬(m<n)                                          assumption
    ...
25: m−n+n<Km
26: ¬(m<n)→m−n+n<Km                                 → intro 24-25
27: (m<n→n−m+n<Km)∧(¬(m<n)→m−n+n<Km)                ∧ intro 23,26
28: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km→(m<n→n−m+n<Km)∧(¬(m<n)→m−n+n<Km)   → intro 15-27
29: {n−m+n<Km}(m:=n−m){m+n<Km}                      variable-assignment
30: {m−n+n<Km}(m:=m−n){m+n<Km}                      variable-assignment
31: {(m<n→n−m+n<Km)∧(¬(m<n)→m−n+n<Km)}if m<n then m:=n−m else m:=m−n fi{m+n<Km}   choice 29,30
32: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km}if m<n then m:=n−m else m:=m−n fi{m+n<Km}   consequence(L) 28,31
```

Missing before

Termination Part of Loop Body

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

```
29: integer Km
30: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km
31: gcd(m,n)=gcd(m0,n0)
32: m>0
33: n>0
34: ¬(m=n)
35: m+n=Km
36: m<n
    ...
37: n−m+n<Km
38: m<n→n−m+n<Km
39: ¬(m<n)
40: m>n
41: m−n+n<Km
42: ¬(m<n)→m−n+n<Km
43: (m<n→n−m+n<Km)∧(¬(m<n)→m−n+n<Km)
44: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km
    →(m<n→n−m+n<Km)∧(¬(m<n)→m−n+n<Km)
```

This gap cannot be closed.

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

Corrected Program

```
...
1: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od){m=gcd(m0,n0)}
```

was m:=n-m

Use Ntuple rule:

```
...
1: {m0>0∧n0>0}(m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}
...
2: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
3: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od){m=gcd(m0,n0)}   Ntuple 1,2
```

Use sequence, then assignment rule twice:

```
...
1: m0>0∧n0>0→gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0
2: {gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}   variable-assignment
3: {m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}                        consequence(L) 1,2
4: {gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}(n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}     variable-assignment
5: {m0>0∧n0>0}(m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}                     sequence 3,4
...
6: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od{m=gcd(m0,n0)}
7: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n−m else m:=m−n fi od){m=gcd(m0,n0)}   Ntuple 5,6
```

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.

```
 1: m0>0∧n0>0                                          assumption
 2: m0>0                                               ∧ elim 1
 3: n0>0                                               ∧ elim 1
 4: gcd(m0,n0)defined                                  m0>0, n0>0 ⊢ gcd(m0,n0)defined 2,3
 5: gcd(m0,n0)=gcd(m0,n0)                              A=A 4
 6: gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0                    ∧ intro 5,2,3
 7: m0>0∧n0>0→gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0          → intro 1-6
 8: {gcd(m0,n0)=gcd(m0,n0)∧m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}   variable-assignment
 9: {m0>0∧n0>0}(m:=m0){gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}  consequence(L) 7,8
10: {gcd(m,n0)=gcd(m0,n0)∧m>0∧n0>0}(n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}   variable-assignment
11: {m0>0∧n0>0}(m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}   sequence 9,10
```

This is the proved triple for the initialization 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.

```
69: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n-m else m:=m-n fi od)   Ntuple 11,68
    {m=gcd(m0,n0)}
```

Proof below the intermediate assertion, part 1

Template: **{I ∧ P} 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.

```
12: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)                 assumption
13: gcd(m,n)=gcd(m0,n0)                                ∧ elim 12
14: m>0                                                ∧ elim 12
15: n>0                                                ∧ elim 12
16: ¬(m=n)                                             ∧ elim 12
17: m<n                                                assumption
18: gcd(m,n-m)=gcd(m0,n0)                              gcd(m,n)=X ⊢ gcd(m,n-m)=X 13
19: n-m>0                                              m<n ⊢ n-m>0 17
20: gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0                    ∧ intro 18,14,19
21: m<n→gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0                → intro 17-20
22: ¬(m<n)                                             assumption
23: gcd(m-n,n)=gcd(m0,n0)                              gcd(m,n)=X ⊢ gcd(m-n,n)=X 13
24: m>n                                                ¬(m=n), ¬(m<n) ⊢ m>n 16,22
25: n<m                                                A<B≜B>A 24
26: m-n>0                                              m<n ⊢ n-m>0 25
27: gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0                    ∧ intro 23,26,15
28: ¬(m<n)→gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0             → intro 22-27
29: (m<n→gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0)∧(¬(m<n)→gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0)   ∧ intro 21,28
30: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)                 → intro 12-29
    →(m<n→gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0)∧(¬(m<n)→gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0)
31: {gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0}(n:=n-m){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}   variable-assignment
32: {gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0}(m:=m-n){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}   variable-assignment
33: {(m<n→gcd(m,n-m)=gcd(m0,n0)∧m>0∧n-m>0)∧(¬(m<n)→gcd(m-n,n)=gcd(m0,n0)∧m-n>0∧n>0)}   choice 31,32
    if m<n then n:=n-m else m:=m-n fi{gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}
34: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)}if m<n then n:=n-m else m:=m-n fi{gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}   consequence(L) 30,33
```

Proof below the intermediate assertion, part 2

Template: **I ∧ P → (_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.

```
35: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)                 assumption
36: m>0                                                ∧ elim 35
37: n>0                                                ∧ elim 35
38: m+n>0                                              m>0, n>0 ⊢ m+n>0 36,37
39: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)→m+n>0           → intro 35-38
```

Proof below the intermediate assertion, part 3

Template: **{I ∧ P ∧ _M = Km} 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.

```
40: integer Km                                         assumption
41: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km          assumption
42: m>0                                                ∧ elim 41
43: n>0                                                ∧ elim 41
44: ¬(m=n)                                             ∧ elim 41
45: m+n=Km                                             ∧ elim 41
46: m<n                                                assumption
47: m+(n-m)<Km                                         m+n=Km, m>0 ⊢ m+(n-m)<Km 45,42
48: m<n→m+(n-m)<Km                                     → intro 46-47
49: ¬(m<n)                                             assumption
50: m>n                                                ¬(m=n), ¬(m<n) ⊢ m>n 44,49
51: m-n+n<Km                                           m+n=Km, n>0 ⊢ m-n+n<Km 45,43
52: ¬(m<n)→m-n+n<Km                                    → intro 49-51
53: (m<n→m+(n-m)<Km)∧(¬(m<n)→m-n+n<Km)                 ∧ intro 48,52
54: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km→(m<n→m+(n-m)<Km)∧(¬(m<n)→m-n+n<Km)   → intro 41-53
55: {m+(n-m)<Km}(n:=n-m){m+n<Km}                       variable-assignment
56: {m-n+n<Km}(m:=m-n){m+n<Km}                         variable-assignment
57: {(m<n→m+(n-m)<Km)∧(¬(m<n)→m-n+n<Km)}if m<n then n:=n-m else m:=m-n fi{m+n<Km}   choice 55,56
58: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬(m=n)∧m+n=Km}if m<n then n:=n-m else m:=m-n fi{m+n<Km}   consequence(L) 54,57
59: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n-m else m:=m-n fi od   while 34,39,40-58
    {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)}
```

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.

```
60: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)                assumption
61: gcd(m,n)=gcd(m0,n0)                                ∧ elim 60
62: ¬¬(m=n)                                            ∧ elim 60
63: m=n                                                ¬¬E ⊢ E 62
64: gcd(m,n)=m                                         m=n ⊢ gcd(m,n)=m 63
65: gcd(m0,n0)=m                                       A=B, A=C ⊢ B=C 61,64
66: m=gcd(m0,n0)                                       A=B≜B=A 65
67: gcd(m,n)=gcd(m0,n0)∧m>0∧n>0∧¬¬(m=n)→m=gcd(m0,n0)   → intro 60-66
68: {gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n-m else m:=m-n fi od{m=gcd(m0,n0)}   consequence(R) 59,67
69: {m0>0∧n0>0}((m:=m0;n:=n0){gcd(m,n)=gcd(m0,n0)∧m>0∧n>0}while¬(m=n)do if m<n then n:=n-m else m:=m-n fi od)   Ntuple 11,68
    {m=gcd(m0,n0)}
```

Derivation as trees (see also: Huth & Ryan fig. 4.2)

Numbers refer to line numbers of formulas on previous pages