

(Imperative) Program Logic

Robert Keller
February 2010


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



- Turing may have been the first to consider proving that a program is correct, in his 1949 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”, 1969.**



Sir Prof. Tony Hoare (FRS)
Microsoft Research Laboratory,
Cambridge

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"

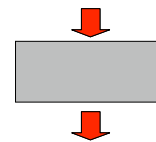
- You may be accustomed to thinking of a program as something with "dynamic" behavior.
- A **mathematical** view is that a program's behavior is just one of many paths through of a (generally-infinite) **static** structure, which can be analyzed with mathematics.

Programs States

- Programs work with **states**.
- Each state is a mapping from program variables into appropriate domains
- A state is much like an **assignment** for an interpretation our discussion of semantics of predicate calculus.

Program as a State Transformer

starting state



ending state

Note about I/O

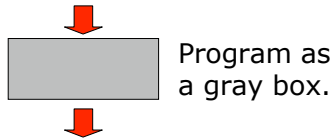
- To deal with input streams and files, we will **consider the entire file or stream**, along with the current position of the reader or writer, to be part of the state.
- We won't be dealing with such issues in this presentation.

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



(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.

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 trivial or meaningless.

Example

- Assumption:

int x[0..n-1] is an array of size n

- Expectation (x is sorted):

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

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 are two arrays with the same elements)

- Expectation:

$\forall i ((i \geq 1 \wedge i < n) \rightarrow (x[i-1] \leq x[i]))$
 $\wedge \mathbf{bagof}(x, x_0)$
(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:
$$\text{valid}(\text{input}) \rightarrow T$$
$$\wedge \neg \text{valid}(\text{input}) \rightarrow \text{red_flag}$$
- Expectation:
$$\neg \text{red_flag} \rightarrow \dots \text{normal expectation} \dots$$
$$\wedge \text{red_flag} \rightarrow \text{exception is indicated}$$
- 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

- **Program Synthesis:** Automate the construction of a program from a specification.
- **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).

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.

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

{assumption} code {expectation}

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

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 $\{x \leq y \wedge y \leq z \wedge z \leq w\}$

Stringency

- {assumption} code {expectation}
- $T_1: \{A\} c \{E\}$ for short
- $T_2: \{A\} c \{E'\}$
- $T_3: \{A'\} c \{E\}$
- If $E' \rightarrow E$, is T_2 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 $\{A\} c \{E'\}$ is more stringent than $\{A\} c \{E\}$.
- If $A' \rightarrow A$, 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 and **possibly more**.
 - $\{A'\} c \{E\}$ **assumes more** to get the same job done.

Maximal Stringency

- $\{T\} c \{\perp\}$
- Not assuming anything, but meeting every expectation.
- Not seen too often, as this implies the program will never terminate under any condition.

Minimal Stringency

- $\{\perp\} c \{T\}$
- Assuming everything, not expecting anything.
- Not seen too often either, as there is no point in running this program. The assumptions cannot be satisfied.

First Rule of Inference

- Consequent Rule

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

- Here triples meet logic (\rightarrow 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

- JAPE's consequent(L):

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

- JAPE's consequent(R):

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

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:

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

Example of Composition Rule

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

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

Conditional Rule

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

There is a strong resemblance to \vee -Elimination, with the \vee -formula being $P \vee \neg P$.

Called "If" in H&R, "choice" 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. $\{x \leq y\}$
 $\text{if } (y > z) S1 \text{ else } S2$
 $\{x \leq y \wedge y \leq z\}$ cond 1, 2

One-Sided Conditional Rule

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

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. $\{x \leq y\}$
 $\text{if } (y > z) S1$
 $\{x \leq y \wedge y \leq z\}$ cond-1, 1, 2

While Rule

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

I is known as the "loop invariant"

Example of While Rule

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

Assignment Statement Rule

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

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.)

"Assignment" here should not be confused with assignment as in the interpretation of logic formulas. Those assignments are like program states.

Example of Assignment Rule

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

- $\{x \leq z\} \quad y := z \quad \{x \leq y\}$ 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

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

- $\{x \leq y+1\} \quad y := y+1 \quad \{x \leq y\}$ assign
- $\{x*y \leq n\} \quad y := x*y \quad \{y \leq n\}$ assign
- $\{x+1 \leq n+1\} \quad x := x+1 \quad \{x \leq n+1\}$ assign

Examples of Derivations of Small Programs: Exchange Program

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

$$\{x = x_0 \wedge y = y_0\} \quad z := x; \quad x := y; \quad y := z; \quad \{y = x_0 \wedge x = y_0\}$$

- $\{z = x_0 \wedge x = y_0\} \quad y := z; \quad \{y = x_0 \wedge x = y_0\}$ assign
- $\{z = x_0 \wedge y = y_0\} \quad x := y; \quad \{z = x_0 \wedge x = y_0\}$ assign
- $\{x = x_0 \wedge y = y_0\} \quad z := x; \quad \{z = x_0 \wedge y = y_0\}$ assign
- $\{z = x_0 \wedge y = y_0\} \quad x := y; \quad y := z; \quad \{y = x_0 \wedge x = y_0\}$ comp 2, 1
- $\{x = x_0 \wedge y = y_0\} \quad z := x; \quad x := y; \quad y := z; \quad \{y = x_0 \wedge x = y_0\}$ comp 3, 4

Examples of Derivations of Small Programs: Ordering two numbers

- $\{x = x_0 \wedge y = y_0\}$
- if($x > y$) { $z := x$; $x := y$; $y := z$ }**
- $\{x \leq y \wedge ((x = x_0 \wedge y = y_0) \vee (y = x_0 \wedge x = y_0))\}$
- We'll obviously be needing the 1-sided conditional rule.
- We'll assume some things about the $<$ and \leq predicates:
 - $\neg(x > y) \rightarrow (x \leq y)$
 - $(y > x) \rightarrow (x \leq y)$
- Similar to the derivation on the previous page, we can derive:
 - $\{x > y \wedge x = x_0 \wedge y = y_0\}$
 - $z := x$; $x := y$; $y := z$;**
 - $\{y > x \wedge y = x_0 \wedge x = y_0\}$
- and using expectation weakening, we can replace the expectation with $\{x \leq y \wedge y = x_0 \wedge x = y_0\}$
- Then identify P in the 1-sided cond rule as: $x > y$

Examples of Derivations of Small Programs

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

Examples of Derivations of Small Programs

- $((x \leq n) \wedge \neg(x < n)) = (x = n)$ Premise
- $(x < n) \rightarrow ((x+1) \leq n)$ Premise
- $\{x+1 \leq n\}$ **$x := x+1$** $\{x \leq n\}$ Assignment
- $\{x < n\}$ **$x := x+1$** $\{x \leq n\}$ Assumption strengthening 3, 2
- $\{x \leq n \wedge x < n\}$ **$x := x+1$** $\{x \leq n\}$ Assumption strengthening 4
- $\{x \leq n\}$ **while($x < n$) $x := x+1$** $\{x \leq n \wedge \neg(x < n)\}$ While 4
- $\{x \leq n\}$ **while($x < n$) $x := x+1$** $\{x = n\}$ Expectation weakening 6

Another Viewpoint: Verification Conditions

- An alternate, less formal, way to view a triple, such as:
 - $\{x+1 \leq n\}$ **$x := x+1$** $\{x \leq n\}$
- 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**:
 - $(x+1 \leq n \wedge (x' = x + 1)) \rightarrow (x' \leq n)$
- 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. This was **Floyd's method**.

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)
skip tilt sequence Ntuple variable-assignment array-element-assignment choice while consequence(L) consequence(R)	A=A A = = B obviously boundedness from (inequality)	A=B Δ B=A A=B Δ -(A=B) A=B Δ B \neq A A \neq B Δ -(A=B) A<B Δ B>A A \leq B Δ A<B \vee A=B A \leq B Δ B \geq A A \leq B Δ -(A>B) A \leq B Δ A<B+1 A+1 \leq B Δ A<B A \geq B Δ -(A<B) A \geq B Δ A>B-1 A-1 \geq B Δ A>B
Array Indexing FROM E=G INFER (A \oplus E \rightarrow F)[G]=F FROM E=G INFER (A \oplus E \rightarrow F)[G]=A[G]		
A = = B		A Δ Δ B

JAPE Hoare Logic ND Rules

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 (inverts 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 (inverts right)
∨ intro (inverts left)
hyp
    
```

JAPE Hoare Logic Examples

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

```

...
1: {i=5 ∧ j=10}(i:=i+j){i=15 ∧ j=10}
...
Provided:
DISTINCT i, j
    
```

Applying the Variable-Assignment Rule

```

...
1: i=5 ∧ j=10 → i+j=15 ∧ j=10
2: {i+j=15 ∧ j=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
...
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 ∧ j=10           assumption
...
2: i+j=15 ∧ j=10
3: i=5 ∧ j=10 → i+j=15 ∧ j=10    → intro 1-2
4: {i+j=15 ∧ j=10}(i:=i+j){i=15 ∧ j=10} variable-assignment
5: {i=5 ∧ j=10}(i:=i+j){i=15 ∧ j=10} consequence(L) 3,4
...
Provided:
DISTINCT i, j
    
```

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

- Using $\wedge I$ and $\wedge E$

```

1: i=5 ∧ j=10           assumption
2: i=5                 ∧ elim 1
3: j=10                ∧ elim 1
...
4: i+j=15
5: i+j=15 ∧ j=10      ∧ intro 4,3
6: i=5 ∧ j=10 → i+j=15 ∧ j=10    → intro 1-5
7: {i+j=15 ∧ j=10}(i:=i+j){i=15 ∧ j=10} variable-assignment
8: {i=5 ∧ j=10}(i:=i+j){i=15 ∧ j=10} consequence(L) 6,7
...
Provided:
DISTINCT i, j
    
```

Full Arithmetic is Not Available

- So we "cheat"...

```

1: i=5 ∧ j=10           assumption
2: i=5                 ∧ elim 1
3: j=10                ∧ elim 1
4: i+j=15              obviously, from 3,2
5: i+j=15 ∧ j=10      ∧ intro 4,3
6: i=5 ∧ j=10 → i+j=15 ∧ j=10    → intro 1-5
7: {i+j=15 ∧ j=10}(i:=i+j){i=15 ∧ j=10} variable-assignment
8: {i=5 ∧ j=10}(i:=i+j){i=15 ∧ j=10} consequence(L) 6,7
...
Provided:
DISTINCT i, j
    
```

Preferred Way to "Cheat"

- 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 "Obviously"

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}          variable-assignment
   (i:=i+j){i=15 ^ j=10}
3: {i=5 ^ j=10}{i:=i+j}{i=15 ^ j=10} consequence(L) 1,2

```

Conditional Example

```
1: {T} 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: T -> (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)}          choice 2,3
   if j>k then i:=j else i:=k fi ((j>k -> i=j) ^ (k>j -> i=k))
5: {T} 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:  $\top \rightarrow (j > k \rightarrow \_A5) \wedge \neg(j > k) \rightarrow (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k)$ 
...
2:  $\{ \_A5 \} (i := j) \{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$ 
3:  $\{ (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k) \} (i := k) \{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$  variable-assignment
4:  $\{ (j > k \rightarrow \_A5) \wedge \neg(j > k) \rightarrow (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k) \}$  choice 2,3
   if  $j > k$  then  $i := j$  else  $i := k$  fi  $\{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$ 
5:  $\{ \top \}$  if  $j > k$  then  $i := j$  else  $i := k$  fi  $\{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$  consequence(L) 1,4

```

Unifying $_A5$ using the assignment rule

was $_A5$

```

...
1:  $\top \rightarrow (j > k \rightarrow (j \geq k \rightarrow j=j) \wedge (k \geq j \rightarrow j=k)) \wedge \neg(j > k) \rightarrow (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k)$ 
2:  $\{ (j \geq k \rightarrow j=j) \wedge (k \geq j \rightarrow j=k) \} (i := j) \{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$  variable-assignment
3:  $\{ (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k) \} (i := k) \{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$  variable-assignment
4:  $\{ (j > k \rightarrow (j \geq k \rightarrow j=j) \wedge (k \geq j \rightarrow j=k)) \wedge \neg(j > k) \rightarrow (j \geq k \rightarrow k=j) \wedge (k \geq j \rightarrow k=k) \}$  choice 2,3
   if  $j > k$  then  $i := j$  else  $i := k$  fi  $\{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$ 
5:  $\{ \top \}$  if  $j > k$  then  $i := j$  else  $i := k$  fi  $\{ (j \geq k \rightarrow i=j) \wedge (k \geq j \rightarrow i=k) \}$  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:  $j > k \rightarrow (j \geq k \rightarrow j=j) \wedge (k \geq j \rightarrow j=k)$ 

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

```

Annotations: "will combine to \perp " (pointing to lines 4-5) and "will combine to $k = j$ " (pointing to lines 12-13).

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**):
 $\eta: \text{States} \rightarrow \mathbb{N}$ (**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**:
 - **$I \wedge P \rightarrow _M > 0$**
meaning that if the loop continues then **_M** is positive.
 - **$\{I \wedge P \wedge _M = Km\} B \{ _M < Km \}$**
meaning that the value of **_M** decreases during the execution of the body.
- **Km** is introduced to represent the value of **_M** before the loop body.
- For comparison, the **partial correctness template** is:
 - **$\{I \wedge P\} B \{I\}$**

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
 - $\text{gcd}(A, A) = A$
 - $\text{gcd}(A, B) = X \mid\text{---} \text{gcd}(B, A) = X$
 - $\text{gcd}(A, B) = X \mid\text{---} \text{gcd}(A-B, B) = X$
 - $\text{gcd}(A, B) = X \mid\text{---} \text{gcd}(A, B-A) = X$

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

- 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

- $\text{gcd}(m, n) = \text{gcd}(m_0, n_0)$
- 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,n)=gcd(m0,n0)}	variable-assignment
8: {m0>0 ∧ n0>0}(m:=m0){gcd(m,n)=gcd(m0,n0)}	consequence(L) 6,7
9: {gcd(m,n)=gcd(m0,n0)}(n:=n0){gcd(m,n)=gcd(m0,n0)}	variable-assignment

Focus on the while loop

```
0: {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 $\neg(m=n)$)

Verification Condition for the loop body:

```
0: {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
...
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: $\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge \neg(m=n) \rightarrow _M > 0$

12: integer Km
...
13: $\{\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge \neg(m=n) \wedge _M = Km\}$
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: $\{\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge \neg(m=n) \wedge m+n=Km\}$
if $m < n$ then $n := n - m$ else $m := m - n$ fi $\{m+n < Km\}$

Template: $\{I \wedge P \wedge _M = Km\} B \{ _M < Km \}$

12: integer Km
...
13: $\{\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge \neg(m=n) \wedge m+n=Km$
 $\neg(m < n \rightarrow m + (n-m) < Km) \wedge \neg(m < n) \rightarrow m - n + n < Km\}$
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: $\{(m < n \rightarrow m + (n-m) < Km) \wedge \neg(m < n) \rightarrow m - n + n < Km\}$ choice 14,15
if $m < n$ then $n := n - m$ else $m := m - n$ fi $\{m+n < Km\}$
17: $\{\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge \neg(m=n) \wedge m+n=Km$
if $m < n$ then $n := n - m$ else $m := m - n$ fi $\{m+n < Km\}$ consequence(U 13,16

Generated Goals

15: $\neg(m=n)$
16: $m+n=Km$
17: $m < n$
...
18: $m + (n-m) < Km$ } need $m > 0$
19: $m < n \rightarrow m + (n-m) < Km$
20: $\neg(m < n)$
...
21: $m - n + n < Km$ } need $n > 0$
22: $\neg(m < n) \rightarrow m - n + n < Km$

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: $(m > 0 \wedge n > 0) \wedge (m = m_0, n = n_0) \wedge \text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge m > 0 \wedge n > 0$ while $\neg(m < n)$ do if $m < n$ then $m := m - n$ else $m := m - n$ fi od $\{m = \text{gcd}(m_0,n_0)\}$

(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: $(m > 0 \wedge n > 0) \wedge (m = m_0, n = n_0) \wedge \text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge m > 0 \wedge n > 0$
...
2: $\{\text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge m > 0 \wedge n > 0\}$ while $\neg(m < n)$ do if $m < n$ then $m := m - n$ else $m := m - n$ fi od $\{m = \text{gcd}(m_0,n_0)\}$
 $\{m > 0 \wedge n > 0\}$
3: $\{(m = m_0, n = n_0) \wedge \text{gcd}(m,n) = \text{gcd}(m_0,n_0) \wedge m > 0 \wedge n > 0\}$ while $\neg(m < n)$ do if $m < n$ then $m := m - n$ else $m := m - n$ fi od [Stack 1.2](#)
 $\{m = \text{gcd}(m_0,n_0)\}$

Section Above the Intermediate Assertion Resolved

```

1 | m0 > 0 ∧ n0 > 0
2 | m0 > 0
3 | n0 > 0
4 | gcd(m0, n0) defined
5 | gcd(m0, n0) = gcd(m0, n0)
6 | gcd(m0, n0) = gcd(m0, n0) ∧ m0 > 0 ∧ n0 > 0
7 | m0 > 0 ∧ n0 > 0 → gcd(m0, n0) = gcd(m0, n0) ∧ m0 > 0 ∧ n0 > 0
8 | (gcd(m0, n0) = gcd(m0, n0) ∧ m0 > 0 ∧ n0 > 0) → (m = m0 ∧ n = n0)
9 | (m0 > 0 ∧ n0 > 0) → (m = m0 ∧ n = n0) → (gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0)
10 | (gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0) → (m = m0 ∧ n = n0)
11 | (m0 > 0 ∧ n0 > 0) → (m = m0 ∧ n = n0) → (gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0)

```

assumption
 \wedge elim 1
 \wedge elim 1
 $m0 > 0, n0 > 0 \vdash \text{gcd}(m0, n0) \text{ defined } 2,3$
 $A-A 4$
 \wedge intro 2,3
 \rightarrow intro 1-6
variable assignment
consequence(U) 7,8
variable assignment
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) ∧ m > 0 ∧ n > 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 (m = gcd(m0, n0) ∧ m > 0 ∧ n > 0)

```

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
...
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 (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) ∧ m > 0 ∧ n > 0)

```

assumption
while 12,13,14-15
consequence(U) 16,17

Partial Correctness of Loop Ending

```

17 | gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0 ∧ (m = n)
18 | gcd(m, n) = gcd(m0, n0)
19 | (m = n)
20 | m = n
21 | m = gcd(m0, n0)
22 | gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0 ∧ (m = n) → m = gcd(m0, n0)

```

assumption
 \wedge elim 17
 \wedge elim 17
 $\rightarrow E$ 19
 $\text{gcd}(A,B) = X, A = B \vdash A = X$ 18,20
 \rightarrow intro 17-21

Partial Correctness Part of Loop Body

```

12 | (gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0) ∧ (m = n)
13 | gcd(m, n) = gcd(m0, n0)
14 | n > 0
15 | (m = n)
16 | m < n
17 | gcd(m, n) = gcd(m0, n0)
18 | n < m
19 | n - m > 0
20 | gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0
21 | m < n → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0
22 | (m < n)
23 | gcd(m, n) = gcd(m0, n0)
24 | m < n
25 | n < m
26 | n - m > 0
27 | gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0
28 | (m < n) → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0
29 | (m < n → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0) ∧ (m < n) → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0
30 | gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0 ∧ (m = n)
31 | (m < n → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0) ∧ (m < n) → gcd(m, n) = gcd(m0, n0) ∧ n - m > 0 ∧ n > 0

```

assumption
 \wedge elim 12
 \wedge elim 12
 \wedge elim 12
assumption
 $\text{gcd}(A,B) = X \vdash \text{gcd}(A,B) = X$ 13
 $A < B \vdash A < B$ 14,16
 $A < B \vdash A - B > 0$ 18
 \wedge intro 17,19,14
 \rightarrow intro 16-20
assumption
 $\text{gcd}(A,B) = X \vdash \text{gcd}(A,B) = X$ 13
 $A < B \vdash A < B$ 14,16
 $A < B \vdash A < B$ 24
 $m < n \vdash n - m > 0$ 25
 \wedge intro 23,26,14
 \rightarrow intro 21-27
 \wedge intro 21,28
 \rightarrow intro 12-29

Termination Part of Loop Body

```

14 | integer Km
15 | gcd(m, n) = gcd(m0, n0) ∧ m > 0 ∧ n > 0 ∧ (m = n) ∧ m + n = Km
16 | gcd(m, n) = gcd(m0, n0)
17 | m > 0
18 | n > 0
19 | m = n
20 | m + n = Km
21 | m < n
22 | n - m + n = Km
23 | m < n → n - m + n = Km
24 | (m < n)
25 | m - n + n = Km
26 | (m < n) → m - n + n = Km
27 | (m < n → n - m + n = Km) ∧ (m < n) → m - n + n = Km
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
29 | (m - n + n = Km) → (m - n) + n = Km
30 | (m - n + n = Km) → (m - n) + n = Km
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)
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)

```

assumption
 \wedge elim 15
 \wedge elim 15
 \wedge elim 15
 \wedge elim 15
assumption
 \rightarrow intro 21-22
assumption
 \rightarrow intro 24-25
 \wedge intro 23,26
 \rightarrow intro 15-27
variable assignment
variable assignment
choice 29,30
consequence(U) 28,31

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 ...
37 m+n<Km
38 m+n=m+n+K
39 ...
40 m+n=Km
41 ...
42 ~(m=n)
43 (m+n-m+n+K) ^ (m+n-m+n+K)
44 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ ~(m-n) ^ m+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 m:=m-n fi od (m=gcd(m0,n0))
    
```

Use Ntuple rule: n:=m-n was m:=m-n

```

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 [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0]
3: (m0>0 ^ n0>0) [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0] while (m<n) do m:=m-n fi od (m=gcd(m0,n0)) skip 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, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0]
3: (m0>0 ^ n0>0) [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0]
4: gcd(m0,n0)=gcd(m0,n0) ^ m0>0 ^ n0>0 [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0]
5: (m0>0 ^ n0>0) [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0]
6: gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 while (m<n) do 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 m:=m-n fi od (m=gcd(m0,n0)) skip 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 assumption
3 n0>0 assumption
4 gcd(m0,n0) defined A-4
5 gcd(m0,n0)=gcd(m0,n0) A-4
6 gcd(m0,n0)=gcd(m0,n0) ^ m0>0 ^ n0>0 intro 1-5
7 m0>0 ^ n0>0 -> gcd(m0,n0)=gcd(m0,n0) ^ m0>0 ^ n0>0 variable assignment
8 gcd(m0,n0)=gcd(m0,n0) ^ m0>0 ^ n0>0 [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0] consequence 7,8
9 (m0>0 ^ n0>0) [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0] variable assignment
10 (gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>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 m:=m-n fi od (m=gcd(m0,n0))
    
```

Proof below the intermediate assertion, part 1

Template: $\{I \wedge P\} B \{I\}$

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) assumption
14 m>0 assumption
15 n>0 assumption
16 ~(m=n) assumption
17 m+n=Km assumption
18 gcd(m,n)=gcd(m0,n0) assumption
19 m>0 assumption
20 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0 intro 17-20
21 m+n=Km assumption
22 ~(m=n) assumption
23 gcd(m,n)=gcd(m0,n0) assumption
24 m>0 assumption
25 n>0 assumption
26 m+n=Km assumption
27 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0 intro 23,25
28 ~(m=n) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0 intro 22,27
29 (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0
30 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) intro 28,29
31 (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0
32 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0
33 (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ (m+n=Km) ^ gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ m>0
34 (gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0) [(m=m0, n=n0) gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0] consequence 31,33
    
```

Proof below the intermediate assertion, part 2

Template: $I \wedge P \rightarrow (_M > 0)$

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.

```

35 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) assumption
36 m>0 assumption
37 n>0 assumption
38 m+n=Km assumption
39 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ m+n=Km intro 35-38
    
```

Proof below the intermediate assertion, part 3

Template: $\{I \wedge P \wedge _M = Km\} B \{ _M < Km \}$

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 assumption
43 n>0 assumption
44 ~(m=n) assumption
45 m+n=Km assumption
46 m+n=Km assumption
47 m+n=Km assumption
48 m+n=Km assumption
49 m+n=Km assumption
50 m+n=Km assumption
51 m+n=Km assumption
52 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km) assumption
53 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km) assumption
54 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 ^ (m=n) ^ m+n=Km ^ (m+n=Km) ^ (m+n=Km) ^ (m+n=Km)
55 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km)
56 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km)
57 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km)
58 (m+n=Km) ^ (m+n=Km) ^ (m+n=Km)
59 gcd(m,n)=gcd(m0,n0) ^ m>0 ^ n>0 while (m<n) do m:=m-n fi od (m=gcd(m0,n0))
    
```

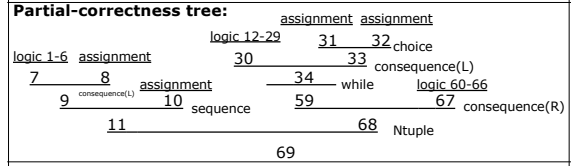
Proof below the intermediate assertion, part 3

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

<pre> 60 gcd(m,n)=gcd(m0,n0) ∧ m > 0 ∧ n > 0 ∧ ¬(m=n) 61 gcd(m,n)=gcd(m0,n0) 62 ¬(m=n) 63 m=n 64 gcd(m,n)=m 65 gcd(m0,n0)=m 66 m=gcd(m0,n0) 67 gcd(m,n)=gcd(m0,n0) ∧ m > 0 ∧ n > 0 ∧ ¬(m=n) ⇒ m=gcd(m0,n0) 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)) 69 (m > 0 ∧ n > 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)) </pre>	<pre> assumption A: elm 60 A: elm 60 ¬E: E 62 m=n ⇒ gcd(m,n)=m 63 A=B, A=C ⇒ B=C E1,64 A=B ⇒ A=C 65 ⇒ Intro 60-66 consequence(L) 59,67 consequence(R) 59,67 E:uple 11,68 </pre>
---	---

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

Numbers refer to line numbers of formulas on previous pages



Termination tree:

