Program Logic

Robert M. Keller
March 2014
Please Read

• Bornat, Proof and Disproof in Formal Logic, Part IV: Proof of Programs
Background: 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.”

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

Flowchart from Turing’s Paper
(What does it do?)
Turing’s Notation

- High-level languages hadn’t been invented.

- Rather than an assignment:
  \[ s := s + 1 \]

  Turing used mathematical notation:
  \[ s' = s + 1 \]

  where ‘ indicates the new value of the variable.
Robert W. Floyd

“Assigning meanings to programs”, 1967

(Figure 1. Flowchart of program to compute \( S = \sum_{j=1}^{n} a_j \) (\( n \geq 0 \))

Robert W. Floyd 1936-2001

Floyd attributed the idea to Saul Gorn in part.

http://amturing.acm.org/award_winners/floyd_3720707.cfm
Turing's Program
Translated to Assignment Notation
and Floyd-style Assertions Added
One Type of Verification

• In the present slides, we focus on one type of verification, known as “inductive assertions”.

• Another type is based on structural induction and is relevant for functional programs with list data structures. We don’t discuss it here, but it is the basis for the famous ACL2 prover by Boyer and Moore:  
  http://www.cs.utexas.edu/users/moore/acl2/
Caution: Informality

• We will assume certain properties of integers, etc. are provable, but do not work within a specific formal theory for this aspect.

• The focus is more on program structure than on specific domains.
Program Graph Model

• The first model is conceptual.
• It will enable us to express some basic ideas, that will be used in more typical models.
• For now, we assume imperative programs, without recursion.
• [This is a variant on a model presented in R.M. Keller, Formal Verification of Parallel Programs, CACM 1976. Downloadable here: http://dl.acm.org/citation.cfm?id=360251]
Program Graph Model

• A program is a directed graph with labeled arcs.

• The nodes represent possible stopping points between changes.

• The arcs are annotated with two things:
  • **Gate** predicate: indicating when the arc can be taken.
  • **Assignment** function: indicating change in values of zero or more program variables if the arc is taken.
Assignment

• In some languages assignment is represented by $=\,$

• We will use $:=\,$ instead, reserving $=\,$ for equality.

• Example:

  $x := x + y$

  The RHS is evaluated, and the value assigned to $x$. 
Parallel Assignment

• With more than one variable on the LHS, the RHS should contain the same number of expressions.
  
  \[(x, n) := (x+y, n-1)\]

• All expressions on the right are evaluated first, then assigned simultaneously to the LHS variables (which should each be unique).
Example

Gates are shown in [brackets]. If none is shown, the gate is implicitly True.

Assignments are shown with :=. If none is shown, the assignment function is implicitly the identity function on all variables. The RHS expressions are computed first, then assigned all at once to the LHS variables.
**Example**

Gates are shown in [brackets]. If none is shown, the gate is implicitly True.

Assignments are shown with :=. If none is shown, the assignment function is implicitly the identity function on all variables. The RHS expressions are computed first, then assigned all at once to the LHS variables.
Semantics

• A program represents a labeled-transition system.

• The states of a system consists of:
  • A single node label, representing the node before the next arc to be taken, if any.
  • For each variable: a value.
Some States of the Previous Example

These states represent the sequence of steps taken if the program is started in node 1 with \( n = 3 \), and \( i \) and \( f \) are initially 0.

What states would occur if the program is started in node 1 with \( n = 5 \) with \( n = 0 \)?
Semantics of the Arcs

• The arcs of a program express state-transition semantics.

• That is, each arc determines transitions as follows, where \( v \) stands for the vector of all variables.

\[
(m, v) \rightarrow (n, F(v)), \text{ provided that } P(v) \text{ holds}
\]

\[
[P(v)] \; v := F(v)
\]
Composite Semantics

• Consider composing two consecutive arcs into one:

\[
\begin{align*}
[P_{ab}(v)] \; v & := F_{ab}(v) \\
[P_{bc}(v)] \; v & := F_{bc}(v) \\
[P_{ac}(v)] \; v & := F_{ac}(v)
\end{align*}
\]

Composite Semantics:

\[
\begin{align*}
P_{ac}(v) & = P_{ab}(v) \land P_{bc}(F_{ab}(v)) \\
F_{ac}(v) & = F_{bc}(F_{ab}(v))
\end{align*}
\]
Example of Composite Semantics

• Here all gates are T, so are not shown:

\[
\begin{align*}
  f &= f \times i \\
  i &= i + 1 \\
  (i, f) &= (i + 1, f \times i)
\end{align*}
\]
Example of Composite Semantics

Note that reversing the assignments is not generally equivalent.
Example of Composite Semantics

Note that reversing the assignments is not generally equivalent.

```
(i, f) := (i+1, f*i)
```

Composite version

```
(i, f) := (i+1, f*(i+1))
```

Composite version
Assertions

- Assertions are logical expressions that annotate nodes.

- They describe a condition that is supposed to be true when that node is the state component.
Example with Assertions

1. \( (i, f) := (1, 1) \)
2. \([i \leq n] \ (i, f) := (i+1, f*i)\)
3. \([i > n] \)

Assertions

- \( A_1: n \geq 0 \)
- \( A_2: i \leq n+1 \land f = (i-1)! \land n \geq 0 \)
- \( A_3: f = n! \)
Executable Assertions

• You may be familiar with executable `assert` statements.

• These are used in development to indicate relations among variables hold, as if the program should not continue if they don’t.

• If the assertion does not hold, the program terminates immediately, indicating the source location.
Proving Assertions

• Verification of a program consists of proving that the assertions are valid, without executing the program.

• The assertion at the initial node of the program is assumed to be true, so we call it The Assumption, or the Pre-Condition for the entire program.

• The assertion at the final node is desired to be true, so we call it The Expectation, or the Post-Condition for the entire program.
Verification Conditions

• Once assertions have been assigned, the proof can be accomplished by a set of detachable logic formulas, one for each arc.

• These logic formulas are called Verification Conditions (VCs).

• If the assertions are correct, the VCs are ideally provable, without reference to the original program.
Uniform Verification Conditions for the Graph Model

- Recall the semantics of arcs: There is a transition \( (m, v) \rightarrow (n, F(v)) \), provided that \( P(v) \) holds.
- The corresponding verification condition, given assertions \( A_m \) (the pre-condition) and \( A_n \) (the post-condition), is thus:

\[
VC_{mn}: (A_m(v) \land P(v)) \rightarrow A_n(F(v))
\]
Example VC’s

\( VC_{mn} : A_m(v) \land P(v) \rightarrow A_n(F(v)) \)

- \( VC_{12} : A_1(i, f, n) \land True \rightarrow A_2(1, 1, n) \)

- \( VC_{22} : A_2(i, f, n) \land i <= n \rightarrow A_2(i+1, f*i, n) \)

- \( VC_{23} : A_2(i, f, n) \land i > n \rightarrow A_3(i, f, n) \)
What are the assertions?

• The assertion at the entry node of the program is **The Assumption**, an assumed restriction on initial values.

• The assertion at the exit node of the program is **The Expectation**, i.e. the desired result.

• Other assertions need to be filled in.
Example Assumption and Expectation

• The example program was designed to compute the factorial of a natural number.

• Therefore:
  Assumption: \( n \geq 0 \)

  Expectation: \( f = n! \) \hspace{1cm} \text{(i.e. } 1*2*3*...*n\text{)}

  note: \( 0! = 1 \) by def.
The Boundary Assertions are Given

• Therefore we know:

\[ A_1(i, f, n): n \geq 0 \quad \text{Assumption} \]

\[ A_3(i, f, n): f = n! \quad \text{Expectation} \]
The remaining assertions must be created/discovered.

\[ A_2(i, f, n) \]

This assertion must express two things:

- **what is known** about the state of the program at the indicated node based on the Assumption at entry, and
- **what must be true** to arrive at the exit with the Expectation having been met.
What we have so far

Assertions

1  \[ A_1(i, n, f): n \geq 0 \]

\[(i, f) := (1, 1)\]

2  \[ A_2(i, n, f) \]

\[[i \leq n] \quad (i, f) := (i+1, f*i)\]

3  \[ A_3(i, n, f): f = n! \]

\[[i > n] \]

\[ A_2(i, n, f) \] should include as a conjunct:

\[ f = (i-1)! \quad \text{because we observe } i = n+1 \text{ upon exit.} \]
Is $A_2(i, n, f)$: $f = (i-1)!$ enough?

- $VC_{12}$: $n \geq 0 \land T \rightarrow A_2(1, 1, n)$

- $VC_{22}$: $A_2(i, f, n) \land i \leq n \rightarrow A_2(i+1, f*i, n)$

- $VC_{23}$: $A_2(i, f, n) \land i > n \rightarrow f = n!$
Is $A_2(i, n, f) : f = (i-1)!$ enough?

- Below are the VC’s with the assertions substituted.

- $VC_{12}: n \geq 0 \land T \rightarrow 1 = (1-1)!$

- $VC_{22}: f = (i-1)! \land i \leq n \rightarrow f*i = ((i+1)-1)!$

- $VC_{23}: f = (i-1)! \land i > n \rightarrow f = n!$

- Are these provable?
Is $VC_{12}$ provable?

• $VC_{12}$: $n \geq 0 \land T \rightarrow 1 = (1-1)!$

• reduces to $n \geq 0 \rightarrow 1 = 0!$

• Provable (assuming appropriate axioms for number), since $1 = 0!$
Is $\text{VC}_{22}$ provable?

• $\text{VC}_{22}$: $f = (i-1)! \land i \leq n \rightarrow f \cdot i = ((i+1)-1)!$

• reduces to
  
  $$f = (i-1)! \land i \leq n \rightarrow f \cdot i = i!$$

• If $f = (i-1)!$ then $f \cdot i = i \cdot (i-1)!$, and indeed $i \cdot (i-1)! = i!$

• So yes, provable.
Is $\text{VC}_{23}$ provable?

• $\text{VC}_{23}$: $f = (i-1)! \land i > n \rightarrow f = n!$

• This VC is *not* provable.

• We only have $i > n$, *not* $i = n+1$.

• How can we get $i = n+1$ on exit?
Revisit the assertions

A_2(i, n, f) should include:

\[ f = (i-1)! \text{ and } i \leq n+1. \]
Assertions Revised

1. \( A_1(i, n, f): n \geq 0 \)
   
2. \( A_2(i, n, f): f = (i-1)! \land i \leq n+1 \)
   
3. \( A_3(i, n, f): f = n! \)

\[
(i, f) := (1, 1)
\]

\[
[i \leq n] \quad (i, f) := (i+1, f*i)
\]

\[
[i > n]
\]
Are the revised assertions provable?

• Below are the VC’s with the assertions substituted.

• $VC_{12}$: $n \geq 0 \land T \rightarrow 1 = (1-1)! \land 1 \leq n+1$

• $VC_{22}$: $f = (i-1)! \land i \leq n+1 \land i \leq n$

  $\rightarrow f*i = ((i+1)-1)! \land i+1 \leq n+1$

• $VC_{23}$: $f = (i-1)! \land i \leq n+1 \land i > n \rightarrow f = n!$

• Are these provable?
Is revised VC\(_{12}\) provable?

- VC\(_{12}\): \( n \geq 0 \land T \rightarrow 1 = (1-1)! \land 1 \leq n+1 \)
- reduces (again assuming some properties of numbers) to
  \[
  n \geq 0 \rightarrow 1 = 0! \land 1 \leq n+1
  \]
- This is provable, since \( 1 = 0! \), and
  \[
  n \geq 0 \rightarrow 1 \leq n+1.
  \]
Is revised $VC_{22}$ provable?

- $VC_{22}$: $f = (i-1)! \land i \leq n+1 \land i \leq n$
  
  $\rightarrow f*i = ((i+1)-1)! \land i+1 \leq n+1$

- As $i \leq n$ implies $i \leq n+1$, the latter is subsumed (redundant).

- With some simplification, this VC reduces to:
  
  $f = (i-1)! \land i \leq n \rightarrow f*i = i! \land i+1 \leq n+1$

- which is easily proved.
Is revised $VC_{23}$ provable?

- $VC_{23}$: $f = (i-1)! \land i \leq n+1 \land i > n \rightarrow f = n!$

- Because we are assuming natural numbers, we can show: $i \leq n+1 \land i > n \leftrightarrow i = n+1$.

- Therefore, this VC reduces to: $f = (i-1)! \land i = n+1 \rightarrow f = n!$ which can be proved.
Invariance and Inductiveness

Not all sets of assertions are equal.

- The set of assertions is **invariant** if they are valid for any execution of the program.

- The set of assertions is **inductive** if they are **sufficient** to *prove* the verification conditions.

- Inductive implies invariant, but not vice-versa.
This set is inductive.

 Assertions

1

\(A_1(i, n, f): n \geq 0\)

\((i, f) := (1, 1)\)

\[[i \leq n] \quad (i, f) := (i+1, f*i)\]

2

\(A_2(i, n, f): f = (i-1)! \land i \leq n+1\)

\[[i > n]\)

3

\(A_3(i, n, f): f = n!\)
This set is invariant, but not inductive.

Assertions

1. \( A_1(i, n, f): n \geq 0 \)
   
   \( (i, f) := (1, 1) \)

2. \( A_2(i, n, f): f = (i-1)! \)
   
   \[ [i \leq n] \quad (i, f) := (i+1, f*i) \]

3. \( A_3(i, n, f): f = n! \)
   
   \[ [i > n] \]
Checking Assertions by Execution

• We can do a *sanity check* on the invariance of our assertions by *running* the program for various input data.

• This is only a sanity check. It does not prove invariance. It can at best increase our confidence.
Sanity Check: Python Program Execution

```
def fac(n):
    (i, f) = (1, 1)
    while i <= n:
        (i, f) = (i+1, f*i)
    return f

def fac_with_assertions(n):
    assert n >= 0  # A1, assumption
    (i, f) = (1, 1)
    assert i <= n+1 and f == pyfac(i-1)  # A2
    while i <= n:
        assert i <= n+1 and f == pyfac(i-1)  # A2
        (i, f) = (i+1, f*i)
    assert f == pyfac(n)  # A3, expectation
    return f

# pyfac is a different way of computing fac
def pyfac(n):
    return reduce(multiply, range(1, n+1), 1)

def multiply(x, y):
    return x*y
```

Comparing n, fac(n), and pyfac(n):

<table>
<thead>
<tr>
<th>n</th>
<th>fac(n)</th>
<th>pyfac(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>720</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>5040</td>
<td>5040</td>
</tr>
<tr>
<td>8</td>
<td>40320</td>
<td>40320</td>
</tr>
<tr>
<td>9</td>
<td>362880</td>
<td>362880</td>
</tr>
<tr>
<td>10</td>
<td>3628800</td>
<td>3628800</td>
</tr>
</tbody>
</table>
Loop Invariants

• A2 has a special role and name: It is called the loop invariant, because:
  • It is true before every test of the loop condition, including:
    • the first time, and
    • the last time.
  • It is true at the end of the loop body.
  • (It generally won’t be true throughout the body.)
• As we shall see, discovering loop invariants is the most creative part of program verification.
Weakest Preconditions
Partial Automation of Assertion Derivation

• Given one assertion on a node, the assertion on the other connected but otherwise isolated node can be derived.

\[ \text{VC}_{mn}: A_m(v) \land P(v) \rightarrow A_n(F(v)) \]

• If \( A_n \) is given, then the strongest possible \( A_m \) is seen to be

\[ P(v) \rightarrow A_n(F(v)). \]

which is thus called the weakest pre-condition corresponding to \( A_n \).
Weakest Precondition

\[ VC_{mn} : A_m(v) \land P(v) \rightarrow A_n(F(v)) \]

- If \( A_n \) is given, then the **weakest possible** \( A_m \) is seen to be \( P(v) \rightarrow A_n(F(v)) \).

- Abbreviate the above as: \( WP[[P(v)] \ v := F(v), A_n] \)

- So the VC at the top is logically equivalent to
  \[ A_m(v) \rightarrow WP[[P(v)] \ v := F(v), A_n] \]

recalling that \( A \land B \rightarrow C = A \rightarrow (B \rightarrow C) \)
Origin of WP Notation

• The WP notation is due to Edsgar Dijkstra, who called WP a “predicate transformer”.

• See

http://en.wikipedia.org/wiki/Predicate_transformer_semantics

Edsgar W. Dijkstra, 1930-2002
Weak vs. Strong Assertions

• If $A \rightarrow B$ but not $(A \rightarrow B)$ then
  • A is **stronger than** B.
  • B is **weaker than** A.
  • A conveys more information than B.
• The **weakest precondition** is the precondition that implies any other valid precondition.
• The **strongest assertion ever** is $\bot$ (“bottom”, false). It conveys so much information that *everything* is derivable from it.
• The **weakest assertion ever** is $T$ (“top”, true). It conveys no information. Nothing other than $T$ is derivable from it.
Weakest Pre-Condition (WP) Working Backward

All you need to remember is in this box:

\[ WP[[P(v)]v := F(v), A_n(v)] \]

\[ [P(v)] v := F(v) \]

\[ A_n \]

\[ WP[[P(v)]v := F(v), A_n(v)] \]

\[ is \]

\[ P(v) \rightarrow A_n(F(v)) \]

\[ A_n(F(v)) \] is just **substituting** the RHS of \( v := F(v) \) for \( v \).
Special Case when $P = \text{True}$

Omit $[P]

WP[v := F(v), A_n(v)]$

$[T] v := F(v)$

$A_n$

$A_n(F(v))$

$A_n(F(v))$ is just $\text{substituting}$ the RHS of $v := F(v)$ for $v$. 

$A_n(F(v))$
Example of WP Working Backward

• Consider the fragment shown below, where \( A_c \) is given as shown.

• We derive \( A_b \) as WP[\( i := i+1, A_c \)]

\[
A_c(i, f, n): f = (i-1)! \land i \leq n+1
\]

\[
A_b \text{ is WP}[i := i+1, A_c]
\]

which is: \( A_c(i+1, f, n) \) substituting \( i+1 \) for \( i \) in \( A_c \)

which is \( f = ((i+1)-1)! \land i+1 \leq n+1 \)

which simplifies to \( f = i! \land i \leq n \)
Another Example of WP

• Consider the fragment shown below, where $A_b$ is given as shown.

• We derive $A_a$ as $\text{WP}[[i \leq n] \ f := \ f*i, \ A_b]$
Chaining Weakest Pre-Conditions

• The weakest pre-condition allows us to work backward through a chain of arcs.

• This is similar to the composite semantics idea discussed earlier, except now we are deriving assertions, not semantics.
Working backward through chain with WP

\[ a := a+1 \]
\[ b := b \]
\[ i := i+1 \]
\[ f = (i-1)! \land i \leq n+1 \]
Working backward with WP

\[ a := i+1 \]
\[ b \]
\[ f := f \cdot i \]
\[ f = (i-1)! \wedge i \leq n+1 \]
\[ c \]
\[ i := i+1 \]
\[ f = i! \wedge i+1 \leq n+1 \]
Working backward with WP

\[ f \cdot i = i! \land i+1 \leq n+1 \]
\[ f := f \cdot i \]
\[ f = i! \land i+1 \leq n+1 \]
\[ i := i+1 \]
\[ f = (i-1)! \land i \leq n+1 \]
Two Ways to Simplify

• We can wait until the end to simplify, or
• Simplify at each backward step
Simplifying at Each Step

\[ a := i+1 \]

\[ f := f \times i \]

\[ i := i+1 \]

\[ f = (i-1)! \land i \leq n+1 \]
Simplifying at Each Step

\[ a \]

\[ f := f \times i \]

\[ b \quad \quad \quad f = i! \land i+1 \leq n+1 \quad \text{simplifies to} \quad f = i! \land i \leq n \]

\[ i := i+1 \]

\[ c \quad \quad \quad f = (i-1)! \land i \leq n+1 \]
Simplifying at Each Step

\[ a \rightarrow f \cdot i = i! \land i \leq n \quad \text{which simplifies to} \quad f = (i - 1)! \land i \leq n \]

\[ f := f \cdot i \]

\[ b \rightarrow f = i! \land i \leq n \]

\[ i := i + 1 \]

\[ c \rightarrow f = (i-1)! \land i \leq n+1 \]
Derive the missing assertions using WP

\[
x := y \times z - 1
\]

\[
y := x
\]

\[
x := y \times z - 1
\]

\[
x > y
\]
Sequencing Formula

• Suppose that the semicolon in Fragment1 ; Fragment2 means that the two fragments are executed in sequence.

• The backward WP reasoning used in previous slides can be summarized as:

  \[ WP[(\text{Fragment1} ; \text{Fragment2}), A] = WP[\text{Fragment1}, WP[\text{Fragment2}, A]] \]

  where A is the post-condition after Fragment2.

• The next slide illustrates this important formula.
WP for Sequenced Fragments

\[ WP[\text{Fragment1}; \text{Fragment2, A}] = \]

- \[ WP[\text{Fragment1, WP[Fragment2, A]}] \]
- \[ WP[\text{Fragment2, A}] \]
- \[ A \]
Two Options for Sequenced Fragments

• We now have two options for deriving WP for a sequence of fragments:

  • Derive the composition of the fragments, then derive the WP in one step, or

  • Derive the WP stepwise, by using the WP of a later fragment in the WP of its predecessor.

• Usually the second way is easier. Why?
Branching

• When the P part of an arc is not identically T, there is usually a paired arc with the complementary condition.

• This occurs in conditionals and in iteration.

• In this case, both WP’s need to be satisfied, so the WP is the conjunction of the two.
Branching

In this case, both WP’s need to be satisfied, so the WP at 1 is the conjunction of the two.

\[
[P] v := F(v) \\
[\neg P] v := G(v) \\
\text{WP is } P \rightarrow A_m(F(v)) \land \neg P \rightarrow A_n(G(v))
\]
Branching using 3-ary connective

- We could also express WP for 2-way branching using a 3-ary connective __ ? __ : __ as in C, C++, Java:

```
[A_m(F(v)) if [P] v := F(v) else A_n(G(v))]
[¬P] v := G(v)
```

In Python, WP would be the expression

```
A_m(F(v)) if P else A_n(G(v))
```
Looping

- Consider the form of a while loop
  while $P$ do $v := F(v)$
Looping

• We temporarily “unwind” this loop.

• WP for $A_m$ would be
  
  \[ WP[\neg P \text{ noop}, A_n] \land WP[[P \ v := F(v), A_m] \]

  but this would be recursive.
Loop Invariant

• Because
  \( A_m = \text{WP}[[\neg P \text{ noop}, A_n]] \land \text{WP}[[P \ v := F(v), A_m]] \)
  is recursive, \( \text{WP} \) does not necessarily have a closed form,
  we try to \textit{discover} a \textbf{loop invariant} \( I \) that satisfies the above equation for \( A_m \):

  \[
  I = \text{WP}[[\neg P \text{ noop, } A_n]] \land \text{WP}[[P \ v := F(v), I]]
  \]

• \textbf{Sufficient} conditions for this equation to be satisfied are:

  \[
  I \rightarrow \text{WP}[[P \ v := F(v), I]]
  \]

  \[
  I \rightarrow \text{WP}[[\neg P \text{ noop, } A_n]]
  \]
Loop Invariant VCs

We can rewrite the sufficient conditions for the loop invariant $I$:

$I \land P \rightarrow WP[v := F(v), I]$

$I \land \neg P \rightarrow A_n$
Refolding the loop and adding initialization

- Loop Verification Conditions are
  \[ A_r \rightarrow WP[v := G(v), I] \quad \text{Initialization} \]
  \[ I \land P \rightarrow WP[v := F(v), I] \quad \text{Body} \]
  \[ I \land \lnot P \rightarrow A_n \quad \text{Finalization} \]
Example

• Consider the “squaring by addition” program.
• Note that
  • $1^2 = 1$
  • $2^2 = 1 + 3$
  • $3^2 = 1 + 3 + 5$
  • $4^2 = 1 + 3 + 5 + 7$
    etc.

We can square by adding odd numbers.
Squaring Program

\[
\text{Squaring Program} \\
(i, j, s) := (0, 1, 0) \\
[i < n] \quad (i, j, s) := (i+1, j+2, s+j) \\
[i \geq n] \quad s = n^2
\]

**Assertions**

- **A₁(i, j, s, n):** \( n \geq 0 \)
- **A₂(i, j, s, n):** ???
- **A₃(i, j, s, n):** \( s = n^2 \)
VC’s for the Squaring Program

• Loop Verification Conditions I are
  
  \[ \begin{align*}
  n \geq 0 & \rightarrow WP[(i, j, s) := (0, 1, 0), I] \\
  I \land i < n & \rightarrow WP[(i, j, s) := (i+1, j+2, s+j), I] \\
  I \land i \geq n & \rightarrow s = n^2
  \end{align*} \]

  \textit{Init} \hspace{1cm} \textit{Body} \hspace{1cm} \textit{Final}

• What should I be?
• It should include \( i \leq n \).
• Also include \( s = i^2 \).
• And we need something about \( j \). (What?)
Invariant for Squaring Program

- Proposed I: \( i \leq n \land s = i^2 \land \ldots \)

- Check Final VC:
  \[
  I \land i \geq n \rightarrow s = n^2
  \]
  \[
  (i \leq n \land s = i^2) \land i \geq n \rightarrow s = n^2 ??
  \]
  But
  \[
  i < n \land i \geq n \text{ is equivalent to } i = n.
  \]
  
  So the above simplifies to
  \[
  s = i^2 \land i = n \rightarrow s = n^2
  \]
  which is obviously valid.

We only have 2 more VC’s to check, as long as we don’t take away anything from the proposed invariant I.
What to say about j?

• It is maintaining the “next odd number”.

• But the invariant must be true when \((i, j, s) = (0, 1, 0)\).

• So perhaps \(j = 2*i + 1\), making the invariant:

\[
i \leq n \land s = i^2 \land j = 2*i + 1
\]
Checking the **Init VC**

\[ n \geq 0 \rightarrow \text{WP}[(i, j, s) := (0, 1, 0), l] \]

\[ n \geq 0 \rightarrow \text{WP}[(i, j, s) := (0, 1, 0), i \leq n \land s = i^2 \land j = 2*i + 1] \]

\[ n \geq 0 \rightarrow 0 \leq n \land 0 = 0^2 \land 1 = 2*0 + 1 \]

\[ n \geq 0 \rightarrow 0 \leq n \land 0 = 0^2 \land 1 = 1 \quad \text{which is valid} \]
Checking the Body VC

• $l \land i < n \rightarrow WP[(i, j, s) := (i+1, j+2, s+j), l]$

  with $l$ as $i \leq n \land s = i^2 \land j = 2*i + 1$

• We need to verify:

  $(i \leq n \land s = i^2 \land j = 2*i + 1) \land i < n \rightarrow$

  $i+1 \leq n \land s+j = (i+1)^2 \land j+2 = 2*(i+1) + 1$

  (the WP part)
Checking the Body VC, continued

• We need to verify:
  \((i \leq n \land s = i^2 \land j = 2*i + 1) \land i < n\)
  \(\rightarrow i+1 \leq n+1 \land s+j = (i+1)^2 \land j+2 = 2*(i+1) + 1\)

• The LHS simplifies to:
  \(i < n \land s = i^2 \land j = 2*i + 1\)

• The RHS simplifies to:
  \(i \leq n \land s+j = i^2+2*i +1 \land j = 2*i + 1\)

• which further simplifies to T.
The Remarkable Thing about VCs

• Once we have constructed the VCs, we are dealing with purely mathematical statements.

• We do not have to think about program semantics explicitly.
Anchor Variables

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

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

• Anchor variables are also used to tie together the assumption and expectation of the program.
Anchor Variable Example used with Assumption and Expectation

Here the values of $f$ range

1,

$n_0$,

$n_0 \cdot (n_0-1)$,

$n_0 \cdot (n_0-1) \cdot (n_0-2)$, ...

= $n_0! / n_0!$, $n_0! / (n_0-1)!$, $n_0! / (n_0-2)!$...

Assertions with anchor variable $n_0$

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

$f := 1$

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

$n > 0$

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

We use multiplication on the left of $=$, rather than division on the right.

$n \leq 0$

$A_3: f = n_0!$

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

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

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

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

  • \( f^n! = n_0! \land n \geq 0 \land n \leq 0 \rightarrow f = n_0! \) \hspace{1cm} \text{Final}
Partial vs. Total Correctness

• So far, we’ve dealt with “partial correctness”:
  \[
  \text{If the assumption is true and the program terminates, then the expectation will be true.}
  \]
  
  Partial Correctness: Assumption \& Termination \rightarrow Expectation

• Of even greater interest is “total correctness”:
  \[
  \text{If the assumption is true, then the program terminates and the expectation will be true.}
  \]
  
  Total Correctness: Assumption \rightarrow Termination \& Expectation
Partial vs. Total Correctness

• Therefore

  Total Correctness =

  Partial Correctness + Termination

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

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

• Identify a function $\eta$ on states (called a variant or measure):

  $\eta$: States $\rightarrow$ Integers, such that:

  • at the start of any iteration, $\eta > 0$, and

  • between the start of one iteration and the next, $\eta$ properly decreases in value.

• These two conditions imply that iteration cannot go on forever.
More Details on Proving Termination

• We can use the invariant I to characterize when an iteration is possible.

• Once we have proposed the variant \( \eta \):

  • **To show that at the start of any** iteration, \( \eta > 0 \), show
    \[
    I \land P \rightarrow \eta > 0
    \]
    where I is the loop invariant and P is the test condition.

  • **To show that between the start of one iteration and the next,**
    \( \eta \) **properly decreases** in value: conjoin to the assertions at the
    beginning and end of the loop:
    \[
    \eta = K \text{ at the start}
    \]
    \[
    \eta < K \text{ at the end}
    \]
    where K is a new, an **anchor variable**.
Termination Verification Conditions (TVCs)

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

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

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

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

Loop invariant: n ≥ 0

What is an acceptable variant \( \eta \) in this case?
Variant for the Trivial Loop

• $\eta = n$

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

  which simplifies to
  
  $n > 0 \land (n = K) \rightarrow (n-1 < K)$   [valid]

• TVC2: $I \land P \rightarrow (\eta > 0)$ is
  
  $n \geq 0 \land n > 0 \rightarrow n > 0$   [valid]
Termination Example 2

\[ n := 0; \]
\[ \text{while}( n < c ) \quad \text{c is some constant} \]
\[ \{ \]
\[ \quad \ldots \]
\[ \quad n := n+1; \]
\[ \} \]

Suppose I is \( n \leq c \).

What is an acceptable \( \eta \) in this case?
Termination Example 3

\[
n := 1;
\]
\[
\text{while( } n < c \text{ ) \hspace{1cm} c \text{ is some constant}
\]
\[
\{
\]
\[
\quad \ldots
\]
\[
\quad n := 2 \times n;
\]
\[
\}
\]

Suppose I is \( n \leq 2 \times c \).

What is an acceptable \( \eta \) in this case?
Variant for Second Loop

What specifically are these?

• TVC1: $I \land P \land (\eta = K) \rightarrow WP[\text{loop-body}, (\eta < K)]$

• TVC2: $I \land P \rightarrow (\eta > 0)$ is
Hoare Logic (HL)

• C.A.R. (Tony) Hoare discovered a way to put VC’s into an elegant natural-deduction formalism.

• The rules in the Bornat text use HL.
Hoare Logic

- C.A.R. ("Tony") Hoare was the first to express program construction along with proofs of correctness as a single unified logic.

Sir Tony Hoare (b. 1934)
Microsoft Research Laboratory,
Cambridge
Hoare Logic Triples

• A **triple** in HL has the form

  \{\text{Pre}\} \text{ Fragment } \{\text{Post}\}

  • Pre is a predicate logic formula: the **Pre-Assertion**
  • Fragment is a program fragment
  • Post is a predicate logic formula: the **Post-Assertion**

• The idea is that the triple forms a **Verification Condition** for the entire fragment.
HL Rules of Inference

• Hoare gave rules of inference for these triples.

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

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

Originally Hoare put braces around the code, rather than the formulas, in a triple.

D2  Rule of Composition
If \( \vdash P \{ Q_1 \} R_1 \) and \( \vdash R_1 \{ Q_2 \} R \) then \( \vdash P \{ (Q_1 ; Q_2) \} R \)

Most people now put the braces around the formulas rather than the code
\{Pre\} Fragment \{Post\}
rather than
Pre \{Fragment\} Post
as Hoare did.
Composition or Sequence Rule

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

• $\{Q\}$ Fragment$_1$ $\{R\}$ $\{R\}$ Fragment$_2$ $\{S\}$
  $\{Q\}$ Fragment$_1$; Fragment$_2$ $\{S\}$
Consequent Rule

- This rule is based on implications.

\[ P \rightarrow Q \quad \{ Q \} \text{ Fragment } \{ R \} \quad R \rightarrow S \]

\[ \{ P \} \text{ Fragment } \{ S \} \]
Typical Application of Consequent Rule

- Sometimes in a sequential composition, assertions don’t quite match up.
- We want
  \[
  \{P\} \text{ Fragment}_1 \{Q\} \quad \{Q\} \text{ Fragment}_2 \{S\}
  \]
  \[
  \{P\} \text{ Fragment}_1; \text{ Fragment}_2 \{S\}
  \]

  but we only have
  \[
  \{P\} \text{ Fragment}_1 \{Q\} \quad \{R\} \text{ Fragment}_2 \{S\}
  \]
  \[
  \{P\} \text{ Fragment}_1; \text{ Fragment}_2 \{S\}
  \]

  If we can derive \(Q \rightarrow R\), then the composition rule is enabled, since by the consequent rule, we also have
  \[
  Q \rightarrow R \quad \{R\} \text{ Fragment}_2 \{S\}
  \]
  \[
  \{Q\} \text{ Fragment}_2 \{S\}\]
Assignment Rule

The assignment rule has no antecedent. It captures the idea of WP for assignment statements.

It should be read from **right to left**.

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

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

\( Q[t/v] \) means \( Q \) with \( t \) substituted for all free occurrences of \( v \), the same as \( WP[v := t, Q] \).
Assignment Rule Examples

Remember to read from right to left.

\[
\{Q[t/v]\} \; v := t \; \{Q\}
\]

Assignment

Example 1: \(\{y > 0\} \; x := y \; \{x > 0\}\)

Example 2: \(\{2*x + y < y\} \; x := 2*x + y \; \{x < y\}\)
Conditional Rule

• The conditional rule has two antecedents.

\[ \{R \land P\} \text{ Fragment}_1 \{S\} \quad \{R \land \neg P\} \text{ Fragment}_2 \{S\} \]

\{R\} \text{ if } P \text{ then Fragment}_1 \text{ else Fragment}_2 \{S\}

• We can see how to derive a suitable R, given S:

\[ R = wp[\text{if } P \text{ then Fragment}_1 \text{ else Fragment}_2, S] \]

\[ = P \rightarrow wp[\text{Fragment}_1, S] \]
\[ \land \neg P \rightarrow wp[\text{Fragment}_2, S] \]
Conditional Rule Example

\[ \{R \land P\} \text{ Fragment}_1 \{S\} \quad \{R \land \neg P\} \text{ Fragment}_2 \{S\} \]
\{R\} \textbf{ if } P \textbf{ then } \text{ Fragment}_1 \textbf{ else } \text{ Fragment}_2 \{S\}

\[ \{T \land x > y\} \ z := x \ \{z = \max(x, y)\} \quad \{T \land x \leq y\} \ z := y \ \{z = \max(x, y)\} \]

\[ \{T\} \textbf{ if } x > y \textbf{ then } z := x \textbf{ else } z := y \ \{z = \max(x, y)\} \]
1-Armed Conditional Rule

• If there is no else part, it is as if the else part is a noop. We need to account for that.

• \( \{R \land P\} \text{ Fragment } \{S\} \quad R \land \neg P \rightarrow S \)
  \( \{R\} \textbf{ if } P \textbf{ then Fragment } \{S\} \)
While Rule

• This is the most complex rule.
• It involves a loop invariant I.

\[\{I \land P\} \text{Body}\ \{I\}\]

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

• Note carefully how the parts fit together:
  
  • In the antecedent, Body *preserves* the invariant.
  
  • In the consequent, the invariant holds *before and after* the while loop.
While Rule Example

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

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

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

• The statement form is **while** P **do** Body.
• The triple form is

  \( \{I\} \textbf{while} \ P \ \textbf{do} \ \text{Body} \ \{I \land \neg P\} \)

• The test P will always be true before Body.
• The test P will be false after the **while**.
• The invariant will be true before and after everything.

  \( \{I \land P\} \ \text{Body} \ \{I\} \)

• (The invariant will never by the same as the test.)
Tree Presentation of Full Proofs

• There are at least three ways to present:
  • Trees
  • In-line assertions
  • Numbered triples
Tree Presentation of the Simple while Example

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

Consequent

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

Consequent

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

Consequent

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

Trees tend to get rather wide, making it difficult to manage the presentation.
While Rule with Termination

• Let $I$ be an invariant, and $v$ be an integer-valued expression.

• The while rule with termination based on $v$ is

\[
I \land P \rightarrow v > 0 \quad \{I \land P \land v = K\} \text{ Body } \{I \land v = K\}
\]

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

Here $K$ is a new integer-valued constant symbol.
Presentation of Full Proofs

• There are at least three ways to present:
  • Trees
  • In-line assertions
  • Numbered triples

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

• However, the trees tend to get rather wide, making it difficult to manage the presentation.
In-Line Assertions

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

- Instead we may embed the assertions into the code, as if comments.

- In the case of consequent rules, the logical implications are implied by adjacent assertions.
In-Line Assertions Presentation

\{x \leq n\}

\textbf{while}( x < n )

\begin{align*}
\{x \leq n \land x \leq n\} & \quad \text{Assign} \\
\{x+1 \leq n\} & \quad \text{Consequent} \\
x := x+1 & \quad \text{While} \\
\{x \leq n\} & \quad \text{Consequent} \\
\{x \leq n \land \neg(x < n)\} & \\
\{x = n\} & \quad \text{Logic Implicit in Consequent Rules:} \\
(x \leq n \land x < n) \rightarrow x+1 \leq n \\
(x \leq n \land \neg(x < n)) \rightarrow x = n
\end{align*}
Numbered Triples Presentation
This is JAPE’s version of Hoare Logic which includes a termination proof based on variant \( (n-x) \).

```
1: x ≤ n ∧ x < n → x + 1 ≤ n
2: \{x + 1 ≤ n\}(x := x + 1){x ≤ n}
3: \{x ≤ n ∧ x < n\}(x := x + 1){x ≤ n}
4: x ≤ n ∧ x < n → n - x > 0

5: \textbf{integer} \textbf{Km}
6: x ≤ n ∧ x < n ∧ n - x = Km → n - (x + 1) < Km
7: \{n - (x + 1) < Km\}(x := x + 1){n - x < Km}
8: \{x ≤ n ∧ x < n ∧ n - x = Km\}(x := x + 1){n - x < Km}
9: \{x ≤ n\}\textbf{while} x < n \textbf{do} x := x + 1 \textbf{od}\{x ≤ n ∧ \neg (x < n)\} \textbf{while} 3, 4, 5–8
10: x ≤ n ∧ \neg (x < n) → x = n
11: \{x ≤ n\}\textbf{while} x < n \textbf{do} x := x + 1 \textbf{od}\{x = n\}
```

Provided:
DISTINCT n, x
What JAPE Does

• JAPE will not prove the program for you.
• It will set up the necessary verification conditions.
• It will automatically introduce certain needed rules, such as the consequent rules, when you select the right core rule.
• It will handle certain substitutions by unification, to make editing easier.
Hoare Logic Proofs in JAPE

• From the JAPE File menu, open new theory: Examples hoare_logic > hoare.jt

• This opens several menus:
  • Array Programs
  • Comparison
  • Indexing
  • Useful Lemmas
  • Variable Programs

• Start with Variable Programs
Variable Programs

About half of the examples in the menu are shown here.
Each line is single Hoare triple.

\[
\begin{align*}
\{i=2\}&(i:=i+1)\{i=3\} \\
\{i=K_i \land j=K_j\}&(t:=i;i:=j;j:=t)\{i=K_j \land j=K_i\} \\
\{i=K_i \land j=K_j\}&(i:=j;j:=i)\{i=K_j \land j=K_i\} \\
\{T\}&\text{if } j > k \text{ then } i:=j \text{ else } i:=k \text{ fi}\{(j \geq k \rightarrow i=j) \land (k \geq j \rightarrow i=k)\} \\
\{j=K_j \land k=K_k\}&\text{if } j > k \text{ then } i:=j \text{ else } i:=k \text{ fi}\{j=K_j \land k=K_k \land (j > k \rightarrow i=K_j) \land (k \geq j \rightarrow i=K_k)\} \\
\{j=K_j \land k=K_k\}&\text{if } j \geq k \text{ then } i:=j \text{ else } i:=k \text{ fi}\{j=K_j \land k=K_k \land (j \geq k \rightarrow i=K_j) \land (k \geq j \rightarrow i=K_k)\} \\
\{i=K_i\}&\text{if } i \geq 40 \text{ then } r:=\text{pass} \text{ else } r:=\text{fail} \text{ fi}\{i=K_i \land (i < 40 \rightarrow r=\text{fail}) \land (i \geq 40 \rightarrow r=\text{pass})\}
\end{align*}
\]
First example

- A proof for this Hoare triple is being requested:
  \[ \{ i = 2 \} (i := i+1) \{ i = 3 \} \]
- This will need the assignment rule, and possibly the consequent rule.
Upon applying the variable-assignment rule, the use of the consequence(L) rule is inserted automatically.

We are left with showing the implication.
Proving the Implication

The Hoare Logic theory includes the full set of natural deduction rules.
Dealing with Numbers

• All variables are natural numbers in its programming language, called Japeish.

• However, the Hoare logic theory does not include the full theory of natural numbers. So we occasionally need to improvise by adding Useful Lemmas.
Justifying with “Obviously”

- The Obviously solution should only be used when we are certain about the formula.
- It is good to keep such uses contained each in a separate Lemma, for accounting reasons.
Using Equality is Tricky
Sequence Example

• Here $K_i$ and $K_j$ are anchor variables.
• We are trying to prove the sequence of 3 statements interchanges the variables $i$ and $j$.
• $t$ is used as a temporary.
• DISTINCT $I, j, t$ is used to indicate these variables are not aliases for one another.

```plaintext
{i=K_i \land j=K_j}(t:=i;i:=j;j:=t){i=K_j \land j=K_i}

Provided:
DISTINCT i, j, t
```
Apply the Sequence Rule

- New placeholder formulas _B2 and _B4 are introduced when the sequence rule is applied.
- These are best resolved **working backward**, using the variable assignment rule, as with chained WPs.
Working Backward

\[
\begin{align*}
1: & \{i=Ki \land j=Kj\}(t:=i)\{i=Kj,j=Ki\} \\
2: & \{\_B4\}(i:=j)\{i=Kj \land t=Ki\} \\
3: & \{i=Kj \land t=Ki\}(j:=t)\{i=Kj \land j=Ki\} \quad \text{variable-assignment} \\
4: & \{i=Kj \land j=Ki\}(t:=i; i:=j; j:=t)\{i=Kj \land j=Ki\} \quad \text{sequence 1,2,3}
\end{align*}
\]

Provided:
DISTINCT \(i, j, t\)

\[
\begin{align*}
1: & i=Ki \land j=Kj \rightarrow j=Kj \land i=Ki \\
2: & \{j=Kj \land i=Ki\}(t:=i)\{j=Kj \land t=Ki\} \quad \text{variable-assignment} \\
3: & \{i=Kj \land j=Ki\}(t:=i)\{j=Kj \land t=Ki\} \quad \text{sequence (L) 1,2} \\
4: & \{j=Kj \land t=Ki\}(i:=j)\{i=Kj \land t=Ki\} \quad \text{variable-assignment} \\
5: & \{j=Kj \land t=Ki\}(j:=t)\{i=Kj \land j=Ki\} \quad \text{variable-assignment} \\
6: & \{i=Ki \land j=Kj\}(t:=i; i:=j; j:=t)\{i=Kj \land j=Ki\} \quad \text{sequence 3,4,5}
\end{align*}
\]

Provided:
DISTINCT \(i, j, t\)
Final Steps are All Natural Deduction
Conditional Example

• This example uses anchor variables $K_j$ and $K_k$.
• The greater of values $j$ and $k$ is put into $i$.
• Note the Japeish syntax. The `if` statement is terminated with `fi`.

```plaintext
1: \{ j=K_j \land k=K_k \} \text{if } j>k \text{ then } i:=j \text{ else } i:=k \text{ fi}
\{ j=K_j \land k=K_k \land (j>k \rightarrow i=K_j) \land (k \geq j \rightarrow i=K_k) \}
```

Provided:
DISTINCT $i$, $j$, $k$
“Choice” Rule is used for Conditional

- Note that placeholder formulas are introduced. These can be resolved when the assignment rule is invoked.
- As before, these are best resolved working backward.

\[ j = K_j \land k = K_k \rightarrow (j > k \rightarrow _A5) \land (\neg(j > k) \rightarrow _B6) \]

\[ (j > k \rightarrow _A5) \land (\neg(j > k) \rightarrow _B6) \] if \( j > k \) then \( i := j \) else \( i := k \) fi

\[ \text{choice 2.3} \]

\[ j = K_j \land k = K_k \land (j > k \rightarrow i = K_j) \land (k \geq j \rightarrow i = K_k) \]

\[ \text{consequence (1, 4)} \]

Provided:
DISTINCT \( i, j, k \)
Using Assignment Rule

\[
\begin{align*}
1. & \quad j = K_j \land k = K_k \rightarrow (j > k \rightarrow j = K_j \land k = K_k \land (j > k \rightarrow j = K_j) \land (k \geq j \rightarrow k = K_k)) \\
   & \quad \land (\neg(j > k) \rightarrow j = K_j \land k = K_k \land (j > k \rightarrow k = K_j) \land (k \geq j \rightarrow k = K_k)) \\
2. & \quad \{j = K_j \land k = K_k \land (j > k \rightarrow j = K_j) \land (k \geq j \rightarrow k = K_k)\} \\
   & \quad \{i = j\}\{j = K_j \land k = K_k \land (j > k \rightarrow i = K_j) \land (k \geq j \rightarrow i = K_k)\} \quad \text{variable-assignment} \\
3. & \quad \{j = K_j \land k = K_k \land (j > k \rightarrow k = K_j) \land (k \geq j \rightarrow k = K_k)\} \\
   & \quad \{i = k\}\{j = K_j \land k = K_k \land (j > k \rightarrow i = K_j) \land (k \geq j \rightarrow i = K_k)\} \quad \text{variable-assignment} \\
   & \quad \{(j > k \rightarrow j = K_j \land k = K_k \land (j > k \rightarrow j = K_j) \land (k \geq j \rightarrow k = K_k)) \land (\neg(j > k))\} \\
4. & \quad \rightarrow j = K_j \land k = K_k \land (j > k \rightarrow k = K_j) \land (k \geq j \rightarrow k = K_k)) \quad \text{if } j > k \text{ then } \\
   & \quad i = j \quad \text{else} \quad i = k \quad \text{fi} \quad \{j = K_j \land k = K_k \land (j > k \rightarrow i = K_j) \land (k \geq j \rightarrow i = K_k)\} \quad \text{choice 2,3} \\
5. & \quad \{j = K_j \land k = K_k\} \quad \text{if } j > k \text{ then } i = j \quad \text{else} \quad i = k \quad \text{fi} \\
   & \quad \{j = K_j \land k = K_k \land (j > k \rightarrow i = K_j) \land (k \geq j \rightarrow i = K_k)\} \quad \text{consequence(L) 1,4}
\end{align*}
\]

Provided:

\text{DISTINCT } i, j, k
The Rest is Natural Deduction

• We use lots of alternating →I and ∧I steps.
• A few other rules from Comparison are used to resolve inequalities.
• The completed proof is shown on the next page.
1. $j = Kj \land k = Kk$
2. $j = Kj$
3. $k = Kk$
4. $j > k$
5. $\neg (k < j)$
6. $k < j$
7. $\bot$
8. $j = Kk$
9. $k \geq j$
10. $j = Kk$
11. $k \geq j \rightarrow j = Kk$
12. $j = Kj \land k = Kk \land (j > k \rightarrow j = Kj) \land (k \geq j \rightarrow j = Kk)\land (k \geq j \rightarrow k = Kk)\land (\neg (j > k))$
13. $\bot$
14. $j = Kj \land k = Kk \land (j > k \rightarrow j = Kj) \land (k \geq j \rightarrow j = Kk)\land (k \geq j \rightarrow k = Kk)\land (\neg (j > k))$
15. $\neg (j > k)$
16. $\bot$
17. $j = Kj \land k = Kk \land (j > k \rightarrow j = Kj) \land (k \geq j \rightarrow j = Kk)\land (k \geq j \rightarrow k = Kk)\land (\neg (j > k))$
18. $\bot$
19. $\bot$
20. $\bot$
21. $\bot$
22. $\bot$
23. $\bot$
24. $\bot$
25. $\bot$
26. $\bot$
27. $\bot$
28. $\bot$
Programs with Loops

- JAPE does not do partial correctness and termination separately.

- The termination conditions are introduced immediately, resulting in more proof obligations.
Possibly the Simplest Loop Program

• Note again the Japeish syntax:
  \texttt{while } ... \texttt{ do } ... \texttt{ od}

• We will apply the while rule
While Rule Steps

• The While rule generates 4 Verification Conditions, in this order:
  • Body VC
  • TVC2
  • TVC1
  • Exit VC
• There is no Init VC, as that is handled as a separate step.
• These should be proved in reverse order.
• TVC1 is where the variant will be introduced.
Result of Applying **while** Rule

Body VC

Anchor variable

Km for variant

TVC2

TVC1

Exit VC

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

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

3: integer \(Km\)

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

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

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

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

Provided:

Km NOTIN _M
Prove the Exit VC

\[
\begin{align*}
\{n \geq 0\} & \text{while } n > 0 \text{ do } n := n - 1 \text{ od}\{n = 0\} \\
1: & \{n \geq 0 \land n > 0\} (n := n - 1) \{n \geq 0\} \\
2: & n \geq 0 \land n > 0 \rightarrow \_M > 0 \\
3: & \text{integer } K_m \\
4: & \{n \geq 0 \land n > 0 \land \_M = K_m\} (n := n - 1) \{\_M < K_m\} \\
5: & \{n \geq 0\} \text{while } n > 0 \text{ do } n := n - 1 \text{ od}\{n \geq 0 \land \neg(n > 0)\} \text{ while } 1, 2, 3, 4 \\
6: & n \geq 0 \land \neg(n > 0) \quad \text{assumption} \\
7: & n = 0 \\
8: & n \geq 0 \land \neg(n > 0) \rightarrow n = 0 \quad \neg \text{ intro } 6, 7 \\
9: & \{n \geq 0\} \text{while } n > 0 \text{ do } n := n - 1 \text{ od}\{n = 0\} \quad \text{consequence(R) } 5, 8
\end{align*}
\]

Provided:
Km NOTIN \_M
Select a variant

• The variant is n in this case.
• Unify \_M with n.
• Then prove TVC1 using the assignment rule.
• The result is on the next page.
TVC1 proved
TVC2 proved

\[
\begin{align*}
1: \{n \geq 0 \land n > 0\} \{n := n - 1\} \{n \geq 0\} \\
2: n \geq 0 \land n > 0 & \quad \text{assumption} \\
3: n > 0 & \quad \land \text{elim 2} \\
4: n \geq 0 \land n > 0 \rightarrow n > 0 & \quad \rightarrow \text{intro 2-3} \\
5: \text{integer } K_m \\
6: n \geq 0 \land n > 0 \land n = K_m & \quad \text{assumption} \\
7: n = K_m \\
8: n = K_m \rightarrow (n - 1) < K_m & \quad \land \text{elim 6} \\
9: (n - 1) < K_m \\
10: n \geq 0 \land n > 0 \land n = K_m \rightarrow n - 1 < K_m & \quad \rightarrow \text{elim 8,7} \\
11: \{n - 1 < K_m\} \{n := n - 1\} \{n < K_m\} \\
12: \{n \geq 0 \land n > 0 \land n = K_m\} \{n := n - 1\} \{n < K_m\} & \quad \text{variable-assignment} \\
13: \{n \geq 0\} \text{while } n > 0 \text{ do } n := n - 1 \text{ od}\{n \geq 0 \land \neg(n > 0)\} & \text{while 1,4,5-12} \\
14: n \geq 0 \land \neg(n > 0) & \quad \text{assumption} \\
15: n = 0 \\
16: n \geq 0 \land \neg(n > 0) \rightarrow n = 0 & \quad \rightarrow \text{intro 14-15} \\
17: \{n \geq 0\} \text{while } n > 0 \text{ do } n := n - 1 \text{ od}\{n = 0\} & \text{consequence(R) 13,16}
\end{align*}
\]
Body VC Proved

Body VC

1. \(n \geq 0 \land n > 0\)
2. \(n > 0\)
3. \(n - 1 \geq 0\)
4. \(n \geq 0 \land n > 0 \rightarrow n - 1 \geq 0\)
5. \([n - 1 \geq 0 \mid n = n - 1 \mid n \geq 0]\)
6. \([n \geq 0 \land n > 0 \mid n = n - 1 \mid n \geq 0]\)
7. \(n \geq 0 \land n > 0\)
8. \(n > 0\)
9. \(n \geq 0 \land n > 0 \rightarrow n > 0\)
10. \text{Integer } K_m
11. \(n \geq 0 \land n > 0 \land n = K_m\)
12. \(n = K_m\)
13. \(n = K_m \rightarrow (n - 1) < K_m\)
14. \((n - 1) < K_m\)
15. \(n \geq 0 \land n > 0 \land n = K_m \rightarrow n - 1 < K_m\)
16. \([n - 1 < K_m \mid n = n - 1 \mid n < K_m]\)
17. \([n \geq 0 \land n > 0 \mid n = K_m \mid n = n - 1 \mid n < K_m]\)
18. \([n \geq 0 \mid \text{while } n > 0 \text{ do } n := n - 1 \text{ od } n \geq 0 \land \neg (n > 0)\) while 6, 9, 10–17
19. \(n \geq 0 \land \neg (n > 0)\)
20. \(n = 0\)
21. \(n \geq 0 \land \neg (n > 0) \rightarrow n = 0\)
22. \([n \geq 0 \mid \text{while } n > 0 \text{ do } n := n - 1 \text{ od } n = 0]\) consequence(\(\Delta\) 18, 21)
Same example with assertions in-line

\[ \{ n \geq 0 \} \]

while \( n > 0 \) do

\[ \{ n \geq 0 \land n > 0 \land n = Km \} \]

\( n := n-1 \)

\[ \{ n \geq 0 \land n < Km \} \]

od

\[ \{ n \geq 0 \land \neg(n > 0) \} \]

\[ \{ n = 0 \} \]
Lemmas Used

1: \( n \geq 0 \land \neg n > 0 \) premise
2: \( n = 0 \) obviously

1: \( n \geq 0 \land \neg (n > 0) \) premise
2: \( n = 0 \) obviously

1: \( n = K_m \rightarrow (n-1) < K_m \) obviously

1: \( n > 0 \) premise
2: \( (n-1) \geq 0 \) obviously
Total Correctness Example: gcd
(due to Euclid, 300 BCE)

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

\text{while}( \neg (m = n) ) 
\{ 
\text{if}( m < n ) \quad n := n - m; \quad \text{else} \quad m := m - n; 
\}
\{m = \text{gcd}(m_0, n_0)\} \quad // \text{expectation}
Total Correctness of the gcd program

• What is the loop invariant?

• What is an appropriate variant?
Lemmas for gcd

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

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

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

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

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

- \( \text{gcd}(m, n) = \text{gcd}(m_0, n_0) \)
Choice of Variant

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

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

• Quantifiers may be needed in more complex programs.

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

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

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

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

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

• Assignment
  \[ A[i] := v \]

is treated as assignment

\[ A := \text{new_array}\{j = i \rightarrow v \land j \neq i \rightarrow A[j]\} \]

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

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

Km is the anchor variable.

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

Provided:

DISTINCT n, x

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

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

• Then $\text{WP}[x := 2, y = 3]$ is still $y = 3$.

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

• Aliasing is **assumed not to occur** unless provided for otherwise.