Lecture 1 An introduction to the aims of formal program development

This introduces use of specification and transformation in producing provably correct algorithms.

Concepts:
- Algorithm transformation
  - Recursion and iteration
  - Correctness-preserving transformations
- Complexity
  - Asymptotic complexity and the Big-O notation
- Algorithm derivation
  - An example 2 functions derived from a specification
- Calculi
  - Propositional calculus
  - Predicate calculus
  - Quantifiers - existential, universal and counting

Text Reference:
- Gries: Part 1

Complexity

We need a notation that succinctly expresses the cost as a function of size in an order of “magnitude” sense.

We write \( f = O(g) \) and say \( f \) is order \( g \) if, after the size exceeds some threshold, \( f \) increases with increasing size no faster than \( g \).

Formally:

\[
f = O(g)
\]

if there exist \( k \) and \( m \) such that for all \( n \geq m \),

\[
f(n) \leq k \ast g(n)
\]

Thus:

\[
x = O(3x)
\]

\[
3x = O(x)
\]

\[
x = O(x^2)
\]

But:

\[
x^2 \neq O(x)
\]

Conventionally, we choose \( g \) containing just one term whose coefficient is 1. Thus:

\[
7x^2 + 4x \log x + 10 = O(x^2)
\]
An example of program derivation
Two square root algorithms

A slow one - $O(\sqrt{n})$
a := 0;
{pre: $0 \leq n$}
{inv $P : a^2 \leq n$}
{bound $t : \sqrt(n) - a$}
while $\text{sqr}(a + 1) \leq n$ do
  a := a + 1;
{post: $a^2 \leq n < (a+1)^2$}

The fast one - $O(\log n)$
a := 0;
b := n + 1;
{pre: $0 \leq n$}
{inv $P : a < b \leq n+1 \land a^2 \leq n < b^2$}
{bound $t : b-a+1$}
while $a + 1 \not< b$ do
  begin
    d := $(a + b) \div 2$;
    if $d \times d \leq n$ then
      a := d
    else
      b := d
  end;
{post: $a^2 \leq n < (a+1)^2$}

Pre- and post-conditions, invariants and bound functions

Note that each sequence has 4 added comments.

- **The pre-condition.** This describes the properties assumed of the data. Here the datum $n$ must be positive. If the data does not satisfy the pre-condition, the result of the program will not in general satisfy the post-condition.

- **The post-condition.** This describes the required properties of the results. Here the result, $a$, must satisfy the definition of integer square root $a^2 \leq n < (a+1)^2$

Both pre- and post-conditions are the same for both procedures because they effectively specify the problem.

- **The invariant.** This describes the property of the loop which remains constant over successive traverses. It must be implied by the pre-condition and must imply the post-condition. The derivation of the invariant from the post-condition is one of the key aspects of program derivation. The variant in a sense defines the method used for the solution. By definition, different programs satisfying the same specification will have different invariants.

- **The bound function.** This gives an upper bound on the number of iterations required to terminate. Its existence is crucial for ensuring termination of a loop.

1 Note: As accuracy is of the utmost importance I will use some conventions to remove chances of making an error. The example here is the use of $\not<$ as $\not=$. I always think of values increasing left to right across the page. The convention is, of course, quite unnecessary!
Some examples of pre-conditions

A pre-condition describes the properties assumed of the data. These are in general quite undemanding (but are not irrelevant) and include the following:

Specifying some lower bound on (elements of) the data:

- \(0 \leq n\) The integer square root problem
- \(0 \leq x \land 0 \leq y\) The GCD problem

Specifying the size of the data, generally in an array.

- \(1 \leq n\) Finding the largest element of an array, etc.

Specifying (e.g. ordering) properties of the data.

- \((\forall j : 0 \leq j < n-1 : a_j \leq a_{j+1})\) Searching an ordered list etc.

There are other properties, too, which are usually assumed. For example, that variables have defined values.

Note that a pre-condition can refer only to input variables.

Some examples of post-conditions

The post-condition describes the required properties of the results. It is clearly therefore **VERY** important. The post-condition will necessarily refer to the initial values of the variables and to the final values of some others. As these classes are not disjoint, we will use **upper case for initial values** to distinguish them from their final values.

This part of the course will be filled with post-conditions but we give here a few examples to set the scene. Note that in most textbooks the informal English description is (purposely) ill-defined. We give below such descriptions, followed by a more precise English description, and then a formal definition.

"Summing the elements of an array"

Set \(s\) to the sum of the \(n\) elements of the array \(a[0..n-1]\).

\[ s = (\Sigma j : 0 \leq j < n : a_j) \]

"Sorting the elements of an array"

Rearrange the \(n\) elements of \(a[0..n-1]\) into ascending order.

\[ (\forall j : 0 \leq j < n-1 : a_j \leq a_{j+1}) \land \text{perm}(a, A) \]

"Finding the maximum"

Setting \(p\) to the position of a maximum element of \(a[0..n-1]\).

\[ 0 \leq p < n \land (\forall j : 0 \leq j < n : a_j \leq a_p) \]
Some laws of the propositional calculus

The operators obey laws – which can be derived from the definitions given above. We give some important ones, which you must learn to apply instinctively.

**Commutative Laws**

\[(b \land c) \equiv (c \land b)\]
\[(b \lor c) \equiv (c \lor b)\]
\[(b \equiv c) \equiv (c \equiv b)\]

**Associative Laws**

\[b \land (c \land d) \equiv (b \land c) \land d \equiv b \land c \land d\]
\[b \lor (c \lor d) \equiv (b \lor c) \lor d \equiv b \lor c \lor d\]

**Distributive laws**

\[b \lor (c \land d) \equiv (b \lor c) \land (b \lor d)\]
\[b \land (c \lor d) \equiv (b \land c) \lor (b \land d)\]

**De Morgan’s Laws**

\[\neg (b \land c) \equiv \neg b \lor \neg c\]
\[\neg (b \lor c) \equiv \neg b \land \neg c\]

**Law of Negation**

\[\neg \neg b \equiv b\]

**Law of the Excluded middle**

\[b \lor \neg b \equiv \text{true}\]

Some laws of the propositional calculus (cont)

**Law of Contradiction**

\[b \land \neg b \equiv \text{false}\]

**Law of Implication**

\[b \Rightarrow c \equiv \neg b \lor c\]

**Law of Equality**

\[(b \equiv c) \equiv (b \Rightarrow c) \land (c \Rightarrow b)\]

**Laws of or-simplification**

\[b \lor b \equiv b\]
\[b \lor \text{true} \equiv \text{true}\]
\[b \lor \text{false} \equiv b\]
\[b \lor (b \land c) \equiv b\]

**Laws of and-simplification**

\[b \land b \equiv b\]
\[b \land \text{true} \equiv b\]
\[b \land \text{false} \equiv \text{false}\]
\[b \land (b \lor c) \equiv b\]

**Law of identity**

\[b \equiv b\]
Equivalences and reasoning

• A **tautology** is a proposition that is true for all values of its arguments. For example:

  \[ b \lor \neg b \]

  is a tautology because either \( b \) is true or \( \neg b \) is.

• Two propositions \( p_1 \) and \( p_2 \) are **equivalent** iff \( p_1 \equiv p_2 \) is a tautology.

• **Rule of Substitution**: If \( p_1 \equiv p_2 \) is an equivalence and \( P(p) \) is a proposition (which is a function of \( p \)) then \( P(p_1) \equiv P(p_2) \) is also an equivalence.

• **Rule of Transitivity**: If \( p_1 \equiv p_2 \) and \( p_2 \equiv p_3 \) are equivalences then so too is \( p_1 \equiv p_3 \).

The proof of some of these equivalences

**By truth table**

<table>
<thead>
<tr>
<th>( b )</th>
<th>( c )</th>
<th>( b \rightarrow c )</th>
<th>( \neg c )</th>
<th>( \neg b )</th>
<th>( \neg c \rightarrow \neg b )</th>
<th>( (b \Rightarrow c) \equiv \neg c \rightarrow \neg b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

**By substitution of equivalent terms using the laws.**

\[ b \Rightarrow c \]

\[ \equiv \neg b \lor c \] (Implication)

\[ \equiv c \lor \neg b \] (Commutativity)

\[ \equiv \neg c \lor \neg b \] (Negation)

\[ \equiv \neg c \Rightarrow \neg b \] (Implication)
Predicate Calculus

We now extend the notion of a proposition in 2 ways:

We allow expressions which return Boolean values:

- $0 \leq n$
- $a^2 \leq n$
- $x \in \{1..10\}$
- pattern is a substring of message

We introduce quantifiers:

- The Existential Quantifier:
  $$(\exists j : m \leq j < n : P_j)$$
- The Universal Quantifier
  $$(\forall j : m \leq j < n : P_j)$$
- The Counting Quantifier
  $$(\# j : m \leq j < n : P_j)$$

We also use familiar mathematical quantifier:

- The Summation Quantifier
  $$(\Sigma j : m \leq j < n : E_j)$$
- The Product Quantifier
  $$(\Pi j : m \leq j < n : E_j)$$

The Existential Quantifier

$$(\exists j : m \leq j < n : P_j)$$

- This is read as:
  $$(\exists j \text{ such that } j \in \text{range } m..n-1 \text{ for which the following holds } P_j)$$

- Provided $m < n$, it is a shorthand for:
  $$P_m \lor P_{m+1} \lor P_{m+2} \lor ... \lor P_{n-1}$$

  but if $m \geq n$ then it is false.

- This can be defined recursively.
  $$(\exists j : m \leq j < m : P_j) = false$$
  $$(\exists j : m \leq j < k+1 : P_j) = (\exists j : m \leq j < k : P_j) \lor P_{k+1}, \ k \geq m$$

- Some equivalent examples:
  $$(\exists i : 0 \leq i < 100 : (\exists j : 0 \leq j < 100 : \text{prime} (i) \land i^*j = 1079))$$
  $$(\exists i : 0 \leq i < 100 : \text{prime} (i) \land (\exists j : 0 \leq j < 100 : i^*j = 1079))$$
  $$(\exists i, j : 0 \leq i, j < 100 : \text{prime} (i) \land i^*j = 1079)$$
The Universal Quantifier

\((\forall j : m \leq j < n : P_j)\)

- This is read as:

  \[(\forall j \text{ such that } m \leq j < n \text{ is in the range } m..n-1 \text{ such that } P_j)\]

  but the converse is not true.

- Provided \(m < n\) it is a shorthand for:

  \[P_m \land P_{m+1} \land P_{m+2} \land ... \land P_{n-1}\]

  but if \(m \geq n\) then it is true.

There is a close relationship between existential and universal quantification because of de Morgan’s Laws.

\[\begin{align*}
(\forall j : m \leq j < n : P_j) & \equiv \sim (\exists j : m \leq j < n : \sim P_j) \\
(\exists j : m \leq j < n : P_j) & \equiv \sim (\forall j : m \leq j < n : \sim P_j)
\end{align*}\]

\[\]
A notation for array segments

We will adopt the convention that the subscripts of an array start from 0, so that for an array of \( n \) elements the subscript of the final element is \( n-1 \). Thus such an array might be referred to as \( a[0..n-1] \). (Dromey does not follow this convention.)

It is invariably the case when processing an array, that we need to define the properties of some section of it: an initial segment, for example, or a middle section. We use the conventions:

- We refer to the \( i \)th element \( a_i \) as \( a[i] \)(Pascal style). There is some movement towards using a functional style of notation \( a.i \), which you may see in some of the papers we might look at.
- We refer to the \( n+m \) consecutive elements \( a_m, a_{m+1}, a_{m+2}, \ldots, a_{n-1} \) as \( a[m..n-1] \).

It is often convenient to use a pictorial representation. For example the following, extracted from a sorting example,

\[
0 \leq k < n \land \text{ordered}(b[0:k-1]) \land b[0:k-1] \leq x \leq b[k+1:n-1])
\]

might be drawn like this:

\[
\begin{array}{cccc}
0 & k-1 & k & k+1 & n-1 \\
\hline
0 \leq k < n & \text{ordered, } & \leq x & \geq x & \\
\end{array}
\]

Some more post-conditions

- Set \( x \) to the factorial of \( n \).
  \[
x = n!
\]
- Swap the values of \( x \) and \( y \).
  \[
x = Y \land y = X
\]
- Set \( x \) to the smaller of \( x \) and \( y \).
  \[
  (x = X \lor x = Y) \land x \leq X \land x \leq Y
\]
- Given an ordered array \( a[0..n-1] \) and a value \( x \) return the value of the subscript, \( s \), of the element equal to \( x \) or where it would be if it were added to the array.
  \[
  -1 \leq s < n+1 \land a[0..s] < x \\
  \land x \leq a[s+1..n]
\]
- Given a value \( x \), find a value \( p \) and rearrange the elements of an array \( a[0..n-1] \) so that all the elements in \( a[0] \) to \( a[p-1] \) are less than or equal to \( x \) and all those in \( a[p] \) to \( a[n-1] \) are greater than or equal to \( x \).
  \[
  0 \leq p < n \land a[0..p-1] \leq x \\
  \land x \leq a[p..n-1] \land \text{perm}(a, A)
\]
Lecture 2  Predicate transformers and \textit{wp}.

This introduces the fundamental notions of a state, and of the predicate transformer, which is used in deriving programs from their specifications. We start our study of procedural languages by considering the simplest statements.

Concepts:

- States
- Proof outlines
- Predicate transformers and weakest precondition
- Basic commands
  - \texttt{skip}
  - \texttt{abort}
  - \texttt{composition}

Text Reference:

Gries: Chapters 6, 7 and 8.

States

- A \textit{state associates identifiers with values} and corresponds to a configuration of the computer’s memory at a given point in time.

- Essentially, a \textit{state is a function from a set of identifiers to a set of values}. For predicates the values may be \textit{Boolean}, \textit{(T and F)}\(^1\), \textit{integer}, \textit{natural} or \textit{real}. The state can be represented by a set of ordered pairs.

Given the Pascal-like declaration:

\begin{verbatim}
var
  p, q1 : Boolean;
  i, j : natural;
\end{verbatim}

a particular state may be defined by the sets

\begin{verbatim}
{(p,T), (q1,F), (i,3), (j,2)}
{(p,T), (q1,T), (i,3), (j,0)}
\end{verbatim}

then \(s(a)\) denotes the value determined by applying state \(s\) to identifier \(a\). For example, using the first set above:

\begin{verbatim}
s(p) = T
s(q1) = F.
s(i) = 3
\end{verbatim}

\(^1\) In what follows we use \textit{T} and \textit{F} for \textit{true} and \textit{false}
Evaluation in a state

- Proposition $e$ is **well defined in state** $s$ if each identifier in $e$ is associated with a value in state $s$. For example in state:

$$s = \{ (p, T), (q1, F), (i, 3), (j, 2) \}$$

the following predicate is well defined:

$$\neg p \land (i = j - 1)$$

while the next one is not.

$$q \land (i = j - 1)$$

- Let proposition $e$ be well defined in state $s$. Then $s(e)$ the **value of $e$ in state** $s$, is the value obtained by replacing all occurrences of identifiers $b_i$ in $e$ by their values $s(b_i)$ and evaluating the resulting constant proposition.

Consider the evaluation of

$$\neg p \land (i = j - 1)$$

in the same state

$$s(\neg p \land (i = j - 1)) = (\neg T \land (3 = 2-1))$$

$$= (F \land F)$$

$$= F$$

Assertions

Assertions are predicates that describe a set of states. (Pre- and post-conditions are special cases!)

Consider the declarations:

$$p : \text{Boolean};$$

$$i, j : 0..3;$$

Then the predicate

$$p \land (i = j)$$

represents the states:

$$\{ (p, T), (i, 0), (j, 0) \}$$

$$\{ (p, T), (i, 1), (j, 1) \}$$

$$\{ (p, T), (i, 2), (j, 2) \}$$

$$\{ (p, T), (i, 3), (j, 3) \}$$

We introduce the notation:

$$\{Q\} S \{R\}$$

where $Q$ and $R$ are predicates and $S$ is a sequence of commands. If execution of $S$ is begun in a state satisfying $Q$, then it is guaranteed to terminate in a finite amount of time in a state satisfying $R$. $Q$ is called the **precondition or input assertion** of $S$, $R$ is called the **postcondition, output assertion or result assertion**.
Proof Outlines

Proof outlines consist of the program code interspersed with assertion at key points:

Swapping the values of x and y
{x = X ∧ y = Y}
t := x;
{t = X ∧ x = X ∧ y = Y}
x := y;
{t = X ∧ x = Y ∧ y = Y}
y := t
{x = Y ∧ y = X}

Setting x to its absolute value
{x = X}
if x < 0 then
{x = X ∧ x < 0}
x := −x
{x = −X ∧ x > 0} {x = |x|}
else
{x = X ∧ x ≥ 0}
skip
{x = X ∧ x ≥ 0} {x = |x|}
{x = |x|}

Assertions in proof outlines may be named by preceding them with an identifier and a colon. Two assertions P1, P2 appearing on a line means P1 ⇒ P2.

Strength of predicates

A predicate p1 is weaker than predicate p2:
• if p1 ⇔ p2.¹
• if p1 makes fewer restrictions on the values of its identifiers.
• if p1’s states includes at least p2’s states.
(The reverse property is stronger than!)

The weakest predicate is T, since it represents the set of all states. Many pre-conditions are as weak as this.

The strongest predicate is F, since it represents no states. No post-condition is as strong as this!

¹ ⇔ often, called consequence, is the reverse of implication ⇒.
The predicate transformer \( wp \)

For any command \( S \) and predicate \( R \), which describes the desired result of executing \( S \), we will define another predicate denoted \( wp(S, R) \) as follows:

\[
wp(S, R) \text{ represents the set of all states such that execution of } S \text{ begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying } R.
\]

\( wp(S, R) \) is the weakest pre-condition for the execution of \( S \) to result in a state satisfying \( R \).

Thus the notations:

\[
\{Q\}S\{R\} \\
Q \Rightarrow wp(S, R)
\]

are equivalent.

Note that if we “Curry” \( wp(S, R) \) to \( wp(S) (R) \), which Gries writes \( wp_3(R) \), then \( wp_3 \) is a predicate transformer because it transforms any predicate into another predicate.

Some examples of \( wp \)

Example 1

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>( wp(S, R) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i := i + 1 )</td>
<td>( i &lt; 1 )</td>
<td>( i &lt; 0 )</td>
</tr>
</tbody>
</table>

That is \( wp(“i := i + 1”, i < 1) = (i < 0) \)

If \( i < 0 \), execution of \( i := i + 1 \) terminates with \( i < 1 \);
if \( i \geq 0 \) execution cannot make \( i < 1 \).

Example 2

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>( wp(S, R) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x \geq y ) then ( z := x ) else ( z := y )</td>
<td>( z = \max(x, y) ) {( z = x \uparrow y )} ( ^5 )</td>
<td>( T )</td>
</tr>
</tbody>
</table>

Execution of \( S \) always terminates with \( z = \max(x, y) \).

\(^5\) The operators for \( \max(\cdot) \) and \( \min(\cdot) \) exist in Sigrid and will be used throughout.
Some further examples of \( wp \)

**Example 3**

<table>
<thead>
<tr>
<th>( S )</th>
<th>( R )</th>
<th>( wp(S,R) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x \geq y ) then z := x</td>
<td>z = y</td>
<td>( y \geq x )</td>
</tr>
<tr>
<td>else z := y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution of \( S \) beginning with \( y \geq x \) sets \( z \) to \( y \) and execution of \( S \) beginning with \( y < x \) sets \( z \) to \( x \) which is not equal to \( y \).

**Example 4**

<table>
<thead>
<tr>
<th>( S )</th>
<th>( R )</th>
<th>( wp(S,R) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x \geq y ) then z := x</td>
<td>z = y − 1</td>
<td>( F )</td>
</tr>
<tr>
<td>else z := y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution of \( S \) can never set \( z \) less than \( y \).

The properties of \( wp \)

If we are to define a programming notation using the concept of \( wp \), then we need to show that \( wp \) is well behaved.

**Law of the Excluded Miracle**

For any command \( S \), \( wp(S, F) = F \)

In English, there are no initial states for which a program can terminate in no state.

**Distributivity of Conjunction**

For any command \( S \) and predicates \( Q \) and \( R \) the following is true, \( wp(S, Q) \land wp(S, R) = wp(S, Q \land R) \)

**Proof**

- Consider any state \( s \) that satisfies the lhs of the rule. By definition execution of \( S \) begun in \( s \) will terminate with both \( Q \) and \( R \) true, hence \( Q \land R \) will also be true and \( s \) is in \( wp(S, Q \land R) \), the rhs of the rule.
- Now suppose \( s \) is in \( wp(S, Q \land R) \). Again by definition, execution of \( S \) begun in \( s \) is guaranteed to terminate in some state \( s' \) of \( Q \land R \). Any such \( s' \) must be in both \( Q \) and \( R \), so that \( s \) is in \( wp(S, Q) \) and in \( wp(S, R) \), the left hand side of the rule.
- As we have shown the lhs \( \Rightarrow \) rhs and rhs \( \Rightarrow \) lhs we can conclude that lhs = rhs.
The properties of \( wp \) (cont)

**Law of Monotonicity**

For any command \( S \) and predicates \( Q \) and \( R \), if 
\[
Q \implies R \text{ then } wp(S, Q) \implies wp(S, R).
\]

**Proof**

For any state \( s \) in \( wp(S, Q) \) we have \( \{s\} S \{s'\} \) where \( s' \) is 
some state of \( Q \). As \( Q \implies R \) we can conclude that \( s \) is in \( wp(S, R) \) 
hence \( wp(S, Q) \implies wp(S, R) \)

**Law of Distributivity of Disjunction**

For any command \( S \) and predicates \( Q \) and \( R \) the 
following is true 
\[
wp(S, Q) \lor wp(S, R) \implies wp(S, Q \lor R)
\]

**Proof**

Consider any state \( s \) that satisfies the left-hand side of the rule. 
By definition execution of \( S \) begun in \( s \) will terminate with at 
least one of \( Q \) or \( R \) true, hence \( Q \lor R \) will also be true and \( s \) is 
in \( wp(S, Q \lor R) \), the right-hand side of the rule.

NB. The implication (rather than equivalence) allows 
the command \( s \) to be non-deterministic. (See later lectures.)

Some simple commands

**Skip**

- For any predicate \( R \), \( wp(skip, R) = R \).
- Its predicate transformer is the identity function.
- **Skip** is used in the alternative command (next week), since 
  all alternatives must have a consequent action.

**Abort**

- For any predicate \( R \), \( wp(abort, R) = F \).
- **Abort** is the only possible command whose predicate 
  transformer is a constant.
- As no state satisfies \( F \), **abort** should never be executed.
- **Abort** is used to specify the action of the iterative 
  command.
Sequential composition

This is one way of composing larger program segments from smaller segments.

Definition

Let $S1$ and $S2$ be commands, then for any predicate $R$,

$$wp("S1; S2", R) = wp(S1, wp(S2, R))$$

Example

$$wp("skip; skip", R) = wp(skip, wp(skip, R))$$
$$= wp(skip, R)$$
$$= R$$

Now consider a sequence of three commands $S1; S2; S3$.

Sequential composition is associative, hence:

$$wp("S1; (S2; S3)", R) = wp("(S1; S2); S3", R)$$
Assignment

Written (as in Pascal)

\[ x := e \]

(NB Gries’s convention is to put a space before \( e \) but not after \( x \). It’s not one that we will use.)

Can be evaluated only in a state in which \( e \) can be evaluated.

Evaluation consists of

1. evaluating \( e \)
2. storing this result in \( x \).

Is formally defined:

\[ \wp(“x := e”, R) = \text{domain}(e) \text{ cand } R_x \]

where \( \text{domain}(e) \) is a predicate which defines that set of states in which \( e \) may be evaluated, and \( \text{cand} \) and \( R_x \) are about to be described.

\( \text{domain}(e) \) is often omitted.
The operators \textit{cand} and \textit{cor}

\textbf{Problem}

There is a well-known problem in Pascal (and other languages) in that the following is undefined when \(y = 0\):

\[
\text{if } (y = 0) \text{ or } (x \div y = 5) \text{ then } s1 \\
\text{else } s2
\]

\textbf{Solution}

This is, of course, a pain, and operators have been postulated to overcome the problem. They are included in the language we are defining and are given below.

\[
\begin{align*}
\text{b cand c} &= \text{ if b then c else F} \\
\text{b cor c} &= \text{ if b then T else c}
\end{align*}
\]

These operators have properties like \(\land\) and \(\lor\) and can be combined with them. These are Java’s and C’s \&\& and //.

\textbf{Textual substitution}

- The notation \(E',e\), where \(x\) is an identifier and \(E\) and \(e\) are expressions, denotes the predicate created by \textbf{replacing every free occurrence} of \(x\) in \(E\) by \(e\).

- It may be \textbf{necessary to put parentheses} around \(e\) to maintain the precedence of operators.

- If any identifiers in which would become bound due to the substitution, a \textbf{replacement of bound identifiers} in \(E\) must take place before the substitution occurs.

- \textbf{Bound identifiers} are those which are specified by a quantifier. For example, in:

\[
(\forall i : m \leq i < n : x^i > 0)
\]

\(i\) is bound to the quantifier \(\forall\), while \(m\), \(n\) and \(x\) are free.
Some examples of textual substitution

Consider

\[ E = x < y \land (\forall i : 0 \leq i < n : b[i] < y) \]

Then we have the following substitutions:

1. \( E'_x = z < y \land (\forall i : 0 \leq i < n : b[i] < y) \)
2. \( E'_x y = x < y + \land (\forall i : 0 \leq i < n : b[i] < x + y) \)
3. \( E'_i = E \)
4. \( E'_i = x < i \land (\forall j : 0 \leq j < n : b[j] < i) \)

Some examples of wp on assignments

**Example 1**

wp(“x := 5”, x = 5) = (5 = 5) = T

(The execution of “x := 5” always establishes x = 5.)

**Example 2**

wp(“x := 5”, x ≠ 5) = (5 ≠ 5) = F

(The execution of “x := 5” never establishes x ≠ 5.)

**Example 3**

wp(“x := x+1”, x < 0) = (x + 1 < 0) = x < –1

**Example 4**

wp(“x := c”, x = c) = (e = c)

**Example 5**

wp(“x := c”, y = c) = (y = c)

(The execution of “x := c” doesn’t affect y. That is there is a prohibition on side effects!)

A more significant example – interchanging 2 variables
The sequence:

\[
\begin{align*}
t & := x; \\
x & := y; \\
y & := t
\end{align*}
\]

is often used to swap the values of \(x\) and \(y\). We now prove that it does do so.

\[
\begin{align*}
wp(“t := x; x := y; y := t”, x = Y \land y = X) \\
& = wp(“t := x; x := y”, wp(“y := t”, x = Y \land y = X)) \\
& = wp(“t := x; x := y”, x = Y \land t = X) \\
& = wp(“t := x”, wp(“x := y”, x = Y \land t = X)) \\
& = wp(“t := x”, y = Y \land t = X) \\
& = y = Y \land x = X
\end{align*}
\]

This is tedious to read and we prefer proof outlines:

\[
\begin{align*}
\{x = Y \land y = X\} & \quad \{x = Y \land y = X\} \\
t := x; & \quad t := x; \\
\{x = Y \land t = X\} & \quad x := y; \\
x := y; & \quad y := t \\
\{y = Y \land t = X\} & \quad \{y = Y \land x = X\} \\
y := t & \quad y := t \\
\{y = Y \land x = X\} & \quad \{y = Y \land x = X\}
\end{align*}
\]

Compare these to those given in the Gries.

---

**Multiple assignment**

**Written**

\[
x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n
\]

where the \(x_i\) are distinct variables and \(e_i\) are expressions

**Can be evaluated only in a state in which \(e\) can be evaluated.**

**Evaluation consists of**

1. evaluating the expressions in any order and assigning them to \(v_1, v_2, \ldots, v_n\).
2. assigning \(v_1, v_2, \ldots, v_n\) to \(x_1, x_2, \ldots, x_n\) in that order.

**Is formally defined:**

\[
wp(“x := e”, R) = \text{domain}(\mathfrak{e}) \textbf{ cand } R^x_{\mathfrak{e}}
\]

where \(\text{domain}(\mathfrak{e})\) is a predicate which defines that set of states in which \(\mathfrak{e}\) may be evaluated.

\[
\text{domain}(\mathfrak{e}) = (\forall i: 1 \leq i \leq n : \text{domain}(e_i))
\]

---

\*\(\mathfrak{e}\) represents a vector of expressions. I shall use the underline in place of Gries’s overscore.
Some examples of multiple assignment

Example 1

\[ x, y := y, x \]  
(Interchanges the values of \( x \) and \( y \).)

Example 2

\[ x, y, z := y, z, x \]  
(Rotates the values of \( x \), \( y \) and \( z \).)

Example 3

\[
\begin{align*}
wp( & "z, y := z \cdot x, y - 1", y \geq 0 \land z \cdot x^y = c) \\
& = y - 1 \geq 0 \land (z \cdot x)^{y-1} = c \\
& = y \geq 1 \land z \cdot x^y = c
\end{align*}
\]

Assignment to an array element

- Consider an array, \( b \), as a simple variable which contains a function.

- Consider \((b; i; e)\) as a function which is the same as \( b \) except that at the argument \( i \) it yields \( e \).

\[
\begin{align*}
b[i] & := e \\
b & := (b; i : e)
\end{align*}
\]

are equivalent.

\[
\begin{align*}
wp( & "b[i] := c", R) \\
& = wp("b := (b; i : e)", R) \\
& = \text{domain } ((b; i : e)) \land \text{ domain } R_b^{(b; i : e)} \\
& = \text{ inrange } (b, i) \land \text{ domain } (e) \land \text{ domain } R_b^{(b; i : e)} \\
& = R_b^{(b; i : e)}
\end{align*}
\]
Some examples of assignment to array elements

Example 1

wp("b[i] := 5", b[i] = 5)
   = (b[i] = 5)\(_{(b; i: 5)}\)
   = (b; i: 5)[i] = 5
   = 5 = 5
   = T

(The execution of "b[i] := 5" always establishes b[i] = 5.)

Example 2

wp("b[i] := 5", b[i] = b[j])
   = (b[i] = b[j])\(_{(b; i: 5)}\)
   = (b; i: 5)[i] = (b; i: 5)[j]
   = (i \neq j \land 5 = b[j]) \lor (i = j \land 5 = 5)
   = (i \neq j \land 5 = b[j]) \lor (i = j)
   = (i \neq j \lor i = j) \land (5 = b[j] \lor i = j)
   = T \land (5 = b[j] \lor i = j)
   = i = j \lor 5 = b[j]

The general multiple assignment command

The assignment command is augmented to handle

- multiple assignments of array elements,

b[i], b[j] := b[j], b[i]

- arrays of more than 1 dimension.

b[i][j] := b[j][i]

We will assume the contents of Gries's Chapter 9.4 as appropriate.
Lecture 4 Alternatives.

The second of the three fundamental composing structures is the alternative.

Concepts:

• Guarded commands
• Some theorems about the if-command
• Development of some simple algorithms
• Some general principles
• A strategy for the development of the if-command
• Non-determinism

Text Reference:
Gries: Chapter 10.

The Alternative Command

Conditional statements allow execution to be dependent on the current state of the program variables.

In Pascal:

\[ \text{if } x \geq 0 \text{ then } z := x \]
\[ \text{else } z := -x \]

In Java (or C):

\[ \text{if } (x \geq 0) \]
\[ z = x; \]
\[ \text{else } z = -x; \]

In our programming notation, we would express this as the following alternative command:

\[ \text{if } x \geq 0 \rightarrow z := x \]
\[ [] x \leq 0 \rightarrow z := -x \]
\[ \text{fi} \]

Using the following notation:

\[ \text{guard} \]
\[ \overset{\text{gate}}{\text{if } x \geq 0 \rightarrow z := x} \]
\[ \text{guarded command} \]

to execute the command, find a true guard and execute its corresponding guarded command.
The general form of the alternative command

The general form is:

\[
\text{if } B_1 \rightarrow S_1
\]
\[
[] B_2 \rightarrow S_2
\]
\[
\ldots
\]
\[
[] B_n \rightarrow S_n
\]
\[
\text{fi}
\]

where:

\( n \geq 0 \)

- each \( B_i \) is a Boolean expression (called a guard) for command \( S_i \). The guard ensures that the command is executed only under the right conditions.

For abbreviation, let the general alternative command be referred to as \textbf{IF}, and let \( \text{BB} = B_1 \lor B_2 \lor \ldots \lor B_n \)

\textbf{Command IF can be executed as follows:-}

1. If a guard \( B_i \) is not well-defined, abortion may occur because the order of evaluation of the guards is arbitrary.

2. If all guards are false, execution aborts.

3. Any guarded statement \( B_i \rightarrow S_i \) with true guard \( B_i \) is selected and executed.

\textbf{Notes}

- Any number of alternatives are allowed, so this command corresponds both to the if-statement, and to the case-statement of Pascal and the SwitchStatement of Java.

- There are no defaults (no else) so every statement must be preceded by a guard that specifies the conditions under which it may be executed.

- If more than one guard is true, the selection of the statement to execute is non-deterministic.

- The lack of defaults imposes a symmetry that many believe is mathematically pleasing.
The weakest precondition, \( wp(IF, R) \)

This is defined as:

\[
wp(IF, R) = \text{domain}(BB) \land \\
BB \land \\
(B_i \Rightarrow wp(S_i, R)) \land \\
\ldots \\
(B_n \Rightarrow wp(S_n, R))
\]

This requires:

- the guards to be well-defined,
- at least one of them to be true,
- the execution of each command \( S_i \) with a true guard \( B_i \) must terminate with \( R \) true.

Omitting the first conjunct, \( \text{domain}(BB) \), this can be written:

\[
wp(IF, R) = (\exists i: 1 \leq i \leq n : B_i) \land \\
(\forall i: 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R))
\]

An example

Let us show that, under all conditions, the sequence of p1:

\[
\begin{align*}
\text{if } x & \geq 0 \rightarrow z := x \\
[] (x & \leq 0 \rightarrow z := -x \\
\text{fi}
\end{align*}
\]

sets \( z \) to the absolute value of \( x \). Call it IF1.

\[
wp(IF, z = |x|) = (x \geq 0 \lor x \leq 0) \land \\
(x \geq 0 \Rightarrow wp(“z := x”, z = |x|)) \land \\
(x \leq 0 \Rightarrow wp(“z := -x”, z = |x|))
\]

\[
= T \land \\
(x \geq 0 \Rightarrow x = |x|) \land \\
(x \leq 0 \Rightarrow -x = |x|)
\]

\[
= T \land T \land T
\]

\[
= T
\]
Another example

The following, IF2, is supposed to be the body of a loop that counts the number of positive values, \( p \), in the array \( b[0:m-1] \).

\[
\begin{align*}
&\text{if } b[i] > 0 \rightarrow p, i := p+1, i+1 \\
&[] b[i] < 0 \rightarrow i := i+1
\end{align*}
\]

Clearly the post-condition\(^7\) is \( R : i \leq m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0) \).

\[
\begin{align*}
\text{wp}(\text{IF2}, R) &= (b[i] > 0 \lor b[i] < 0) \land \\
&\quad (b[i] > 0 \Rightarrow \text{wp}(“p, i := p+1, i+1”, R)) \land \\
&\quad (b[i] < 0 \Rightarrow \text{wp}(“i := i+1”, R)) \\
&= b[i] \neq 0 \land \\
&\quad (b[i] > 0 \Rightarrow i+1 \leq m \land \\
&\quad \quad p+1 = (\mathbb{N} j : 0 \leq j < i+1 : b[j] > 0) \\
&\quad \land \\
&\quad (b[i] < 0 \Rightarrow i+1 \leq m \land \\
&\quad \quad p = (\mathbb{N} j : 0 \leq j < i+1 : b[j] > 0) \\
&= b[i] \neq 0 \land \\
&\quad i < m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0) \land \\
&\quad i < m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0)
\end{align*}
\]

So that the array should not contain 0 (more likely the second guard should be weakened) and \( p \) will be correct after the statement only if it is correct before.

---

\(^7\) In lecture 12, I used \#; Gries and Sigrid use \( \mathbb{N} \), so let’s do that from here on in.

---

Proving IF does the required job

How can we tell if an alternative statement will work? Quite often we have a precondition \( Q \) (it is the post-condition of the previous statement, for example). We therefore need to prove that for a statement \( IF \) and a desired postcondition \( R \):

\[ \{ Q \} \text{ IF } \{ R \} \]

or equivalently

\[ Q \Rightarrow \text{wp}(IF, R) \]

Pictorially we have:

---

Formal methods 55 Program Correctness

Formal methods 56 Program Correctness
A (useful) theorem about IF

The following theorem helps.

**Theorem**
If predicate Q satisfies:

- Q \Rightarrow BB
- Q \land B_i \Rightarrow \text{wp}(S_i, R) \forall i, 1 \leq i \leq n

then (and only then) Q \Rightarrow \text{wp}(IF, R).

**Proof**
First, leaving out the limits purely for space reasons, transform:

\( (\forall i : Q \land B_i \Rightarrow \text{wp}(S_i, R)) \)
\( = (\forall i : \neg(Q \land B_i) \lor \text{wp}(S_i, R)) \) [Implication]
\( = (\forall i : \neg Q \lor B_i \lor \text{wp}(S_i, R)) \) [de Morgan]
\( = \neg Q \lor (\forall i : B_i \lor \text{wp}(S_i, R)) \) [Q independent of i]
\( = Q \Rightarrow (\forall i : B_i \lor \text{wp}(S_i, R)) \) [Implication]
\( = Q \Rightarrow (\forall i : B_i \Rightarrow \text{wp}(S_i, R)) \) [Implication]

Then we have:

\( (Q \Rightarrow BB) \land (\forall i : Q \land B_i \Rightarrow \text{wp}(S_i, R)) \) [Assumptions]
\( = (Q \Rightarrow BB) \land (Q \Rightarrow (\forall i : B_i \Rightarrow \text{wp}(S_i, R))) \) [above]
\( = Q \Rightarrow (BB \land (\forall i : B_i \Rightarrow \text{wp}(S_i, R))) \) [Distributivity]
\( = Q \Rightarrow \text{wp}(IF, R) \) [Definition]

**Example**
Consider a binary search for a value x known to be in array \( b[0 : n-1] \). Suppose we are in a state satisfying predicate Q:

\( Q = \text{ordered}(B[0 : n-1]) \land 0 \leq i < k < n \land x \in b[i : j] \)

**We want to prove:**

\{Q\}
\( \text{if } b[k] \leq x \rightarrow i:= k \)
\( \text{else } b[k] > x \rightarrow j:= k \)
\( \text{fi} \)
\{x \in b[i : j]\}

**Proof:**

\[- BB = (b[k] \leq x) \lor (b[k] \geq x) \]
\[= T \]

So Q \Rightarrow BB and the first assumption of the theorem holds.

\[- Q \land b[k] \leq x \rightarrow x \in b[k : j] \]
\[= \text{wp}("i := k", x \in b[i : j]) \]
\[- Q \land b[k] \geq x \rightarrow x \in b[i : k] \]
\[= \text{wp}("j := k", x \in b[i : j]) \]

Thus all assumptions are satisfied and the theorem allows to conclude what we wished to prove.
Development of a simple program

Consider the following problem:

Write a program that, given fixed integers \( x \) and \( y \), sets \( z \) to the maximum of \( x \) and \( y \). Clearly, a command \( S \) is required that satisfies

\[
\{ T \} S \{ R : z = x \uparrow y \}
\]

- Before the program can be developed, \( R \) must be refined by replacing \( \uparrow \) by its definition: without knowing what \( \uparrow \) means we cannot write the program. (If \( \uparrow \) were a primitive – as it is in Sigrid – the problem reduces to a single assignment!)

\[
R : z \geq x \land z \geq y \land (z = x \lor z = y)
\]

What command could possibly be executed in order to establish this? \( R \) seems to suggest, amongst others, \( z := x \)

- Under what conditions will execution of \( z := x \) establish \( R \)? Simply calculate \( wp(\"z := x\", R) \):

\[
wp(\"z := x\", z \geq x \land z \geq y \land \ (z = x \lor z = y))
\]

- \( x \geq x \land x \geq y \land (x = x \lor x = y) \)
- \( T \land x \geq y \land (T \lor x = y) \)
- \( T \land x \geq y \land T \)
- \( x \geq y \)

---

\( ^* \) “Fixed” implies that the value should not be changed by execution of the program.

---

Development of a simple program (cont)

- Our first attempt at a program can be:

\[
\text{if } x \geq y \rightarrow z := x
\]

\[
\text{fi}
\]

- Is that all? To prevent abortion \( Q \Rightarrow BB \).

\[
Q \Rightarrow BB
\]

- \( T \Rightarrow x \geq y \)
- \( F \)

so the answer is NO! At least one more guard is needed. Try \( z := y \).

- Now calculate \( wp(\"z := y\", R) \). It’s \( y \geq x \).

- Our second attempt is:

\[
\text{if } x \geq y \rightarrow z := x
\]

\[
[ ] y \geq x \rightarrow z := y
\]

\[
\text{fi}
\]

- Is that all?

\[
Q \Rightarrow BB
\]

- \( T \Rightarrow (x \geq y) \lor (y \geq x) \)
- \( T \)

---

\( ^* \) “Fixed” implies that the value should not be changed by execution of the program.
A dud try

It must seem that I just picked the two commands to try by magic. Let’s see what would happen if I tried something else. The post-condition is:

\[ R : z \geq x \land z \geq y \land (z = x \lor z = y) \]

and I chose the two disjuncts of the last conjunct in turn. Why not choose one of the other conjuncts? Let’s try the first:

\[ z \geq x \]

and proceed by setting \( z = x + 37 \). We calculate the weakest pre-condition:

\[
\begin{align*}
wp(“z := x + 37”, & \quad z \geq x \land z \geq y \land (z = x \lor z = y)) \\
= \quad & x+37 \geq x \land x+37 \geq y \land (x+37 = x \lor x+37 = y) \\
= \quad & T \land x+37 \geq y \land (F \lor x+37 = y) \\
= \quad & T \land x+37 \geq y \land x+37 = y \\
= \quad & x+37 = y
\end{align*}
\]

This leads to the guarded command:

\[
\text{if } x+37 = y \rightarrow z := x+37 \\
\text{fi}
\]

which, whilst true, is hardly useful.

Some principles

**Principle**: Programming is a **goal-oriented activity**.

By this we mean that the desired result, or goal, \( R \), plays a more important role in the development of a program than the precondition \( Q \). This is obviously true since many pre-conditions are just \( T \).

**Principle**: Before attempting to solve a problem, make absolutely sure you **know what the problem is**.

**Principle**: Before developing a program, **make precise and refine the pre- and post-conditions**.

The form of the specification can influence algorithmic development, so that striving for **simplicity and elegance** should be helpful. With some problems, the major difficulty is making the specification simple and precise, and subsequent development of the program is fairly straightforward.
A strategy for developing an alternative command

To invent a guarded command:

- find a command \( C \) whose execution will establish postcondition \( R \) in at least some cases,
- find a Boolean \( B \) satisfying \( B \Rightarrow wp(C, R) \),
- put them together to form \( B \rightarrow C \).

Continue to invent guarded commands until the precondition of the construct implies that at least one guard is true.

Note the symmetry in:

\[
\text{if } x \geq y \rightarrow z := x \\
\text{fi}
\]

which is possible because of the nondeterminism. If there is no reason to choose between \( z := x \) and \( z := y \) when \( x = y \), one should not be forced to choose.

Nondeterminism is an important feature even if the final program turns out to be deterministic, for it allows us to devise a good programming methodology. One is free to develop many different guarded commands completely independently of each other. Any form of determinism, such as evaluating the guards in order of occurrence, drastically affects the way one thinks about developing alternatives.

A second example

Write a program that permutes (interchanges) the values of integer variables \( x \) and \( y \) so that \( x \preceq y \). We will reason more informally here, but still use the strategy above.

First step: write a suitable precondition \( Q \) and postcondition \( R \).

The problem is slightly harder than the first one, for it requires us to consider the initial and final values of variables. We use the “upper case” convention:

\[
Q : x = X \land y = Y \\
R : x \preceq y \land (x = X \land y = Y \lor x = Y \land y = X)
\]

What simple commands could cause \( R \) to be established, at least under some conditions?

Try: \( \text{skip} \)

The guard \( B_i \) of a guarded command \( B_i \rightarrow S_j \) of an alternative construct must satisfy \( Q \land B_i \Rightarrow \wp(S_j, R) \), according to the theorem about IF.

For the command \( \text{skip} \) we have \( \wp(\text{skip}, R) = R \), so that \( B \) of the command \( B \rightarrow \text{skip} \) must satisfy \( Q \land B \Rightarrow R \). Since \( Q \) implies the second conjunct of \( R \), the first conjunct, \( x \preceq y \), of \( R \) can be the guard, so we have:

\[
\text{if } x \preceq y \rightarrow \text{skip}
\]
A second example (cont)

Now try:

\[ x, y := y, x \]

Calculating the weakest precondition we have:

\[
\text{wp}(\langle x, y := y, x \rangle, R) = y \leq x \land (y = X \land x = y \lor y = Y \land x = X).
\]

Again, the second conjunct of this weakest precondition is implied by \( Q \), so that the first conjunct \( y \leq x \) can be the guard.

This yields the alternative construct

\[
\text{if } x \leq y \rightarrow \text{skip} \\
[] y \leq x \rightarrow x, y := y, x \\
\text{fi}
\]

Since the disjunction of the guards, \( x \leq y \lor y \leq x \), is always true, the program is correct (with respect to the given \( Q \) and \( R \)).
CITS4221
Formal Methods in Software Engineering
Program Correctness

Lecture 4 Alternatives.

The second of the three fundamental composing structures is the alternative.

Concepts:
- Guarded commands
- Some theorems about the if-command
- Development of some simple algorithms
- Some general principles
- A strategy for the development of the if-command
- Non-determinism

Text Reference:
Gries: Chapter 10.

The Alternative Command

Conditional statements allow execution to be dependent on the current state of the program variables.

In Pascal:

```
if x >= 0 then
  z := x
else
  z := -x
```

In Java (or C):

```
if (x>=0)
  z = x;
else
  z = -x;
```

In our programming notation, we would express this as the following alternative command:

```
if x ≥ 0 → z := x
[] x ≤ 0 → z := -x
fi
```

Using the following notation:

```
guard
  if x ≥ 0 → z := x
```

to execute the command, find a true guard and execute its corresponding guarded command.
The general form of the alternative command

The general form is:

\[ \text{if } B_1 \rightarrow S_1 \]
\[ \ldots \]
\[ \text{if } B_n \rightarrow S_n \]

where:

\[ n \geq 0 \]

– each \( B_i \) is a Boolean expression (called a guard) for command \( S_i \). The guard ensures that the command is executed only under the right conditions.

For abbreviation, let the general alternative command be referred to as \( \text{IF} \), and let \( \text{BB} = B_1 \lor B_2 \lor \ldots \lor B_n \)

**Command IF can be executed as follows:-**

4. If a guard \( B_i \) is **not well-defined**, abortion may occur because the order of evaluation of the guards is arbitrary.

5. If all guards are false, execution aborts.

6. Any guarded statement \( B_i \rightarrow S_i \) with true guard \( B_i \) is selected and executed.

**Notes**

- **Any number of alternatives are allowed**, so this command corresponds both to the if-statement, and to the case-statement of Pascal and the SwitchStatement of Java.

- There are no defaults (no \textit{else}) so **every statement must be preceded by a guard** that specifies the conditions under which it may be executed.

- If more than one guard is true, the selection of the statement to execute is **non-deterministic**.

- The lack of defaults imposes a **symmetry** that many believe is mathematically pleasing.
The weakest precondition, \( wp(IF, R) \)

This is defined as:

\[
\begin{align*}
wp(IF, R) & = \text{domain}(BB) \land \\
& \quad \text{BB} \land \\
& \quad (B_i \Rightarrow wp(S_i, R)) \land \\
& \quad \ldots \\
& \quad (B_n \Rightarrow wp(S_n, R))
\end{align*}
\]

This requires:

- the guards to be well-defined,
- at least one of them to be \textit{true},
- the execution of each command \( S_i \) with a true guard \( B_i \) must terminate with \( R \) true.

Omitting the first conjunct, \( \text{domain}(BB) \), this can be written:

\[
\begin{align*}
wp(IF, R) & = (\exists i: 1 \leq i \leq n : B_i) \land \\
& \quad (\forall i: 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R))
\end{align*}
\]

An example

Let us show that, under all conditions, the sequence of \( p1: \)

\[
\begin{align*}
\text{if } x \geq 0 & \rightarrow z := x \\
[] x \leq 0 & \rightarrow z := -x \\
\text{fi}
\end{align*}
\]

sets \( z \) to the absolute value of \( x \). \textit{Call it IF1}.

\[
\begin{align*}
wp(\text{IF1, } z = |x|) & = (x \geq 0 \lor x \leq 0) \land \\
& \quad (x \geq 0 \Rightarrow wp("z := x", z = |x|)) \land \\
& \quad (x \leq 0 \Rightarrow wp("z := -x", z = |x|)) \\
& = T \land \\
& \quad (x \geq 0 \Rightarrow x = |x|) \land \\
& \quad (x \leq 0 \Rightarrow -x = |x|) \\
& = T \land T \land T \\
& = T
\end{align*}
\]
Another example

The following, IF2, is supposed to be the **body of a loop** that counts the number of positive values, \( p \), in the array \( b[0:m-1] \).

\[
\text{if } b[i] > 0 \rightarrow p, i := p+1, i+1 \\
[] b[i] < 0 \rightarrow i := i+1 \\
\text{fi}
\]

Clearly the post-condition\(^9\) is \( R : i \leq m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0) \).

\[
\text{wp(IF2, } R) = (b[i] > 0 \lor b[i] < 0) \land \\
(b[i] > 0 \Rightarrow \text{wp}(“p, i := p+1, i+1”, R)) \land \\
(b[i] < 0 \Rightarrow \text{wp}(“i := i+1”, R)) \\
= b[i] \neq 0 \land \\
(b[i] > 0 \Rightarrow i+1 \leq m \land \\
p+1 = (\mathbb{N} j : 0 \leq j < i+1 : b[j] > 0) \\
\land \\
(b[i] < 0 \Rightarrow i+1 \leq m \land \\
p = (\mathbb{N} j : 0 \leq j < i+1 : b[j] > 0) \\
= b[i] \neq 0 \land \\
i < m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0) \land \\
i < m \land p = (\mathbb{N} j : 0 \leq j < i : b[j] > 0)
\]

So that the array should not contain \( 0 \) (more likely the second guard should be weakened) and \( p \) will be correct after the statement only if it is correct before.

---

\(^9\) In lecture 12, I used \#: Gries and Sigrid use \( \mathbb{N} \), so let’s do that from here on in.
**A (useful) theorem about IF**

The following theorem helps.

**Theorem**
If predicate $Q$ satisfies:

- $Q \Rightarrow BB$
- $Q \land B_i \Rightarrow wp(S_i, R)$ for all $i$, $1 \leq i \leq n$

then (and only then) $Q \Rightarrow wp(IF, R)$.

**Proof**
First, leaving out the limits purely for space reasons, transform:

$$(\forall i : Q \land B_i \Rightarrow wp(S_i, R))$$

$$(\forall i : \neg(Q \land B_i \lor wp(S_i, R)))$$

$$(\forall i : \neg Q \lor \neg B_i \lor wp(S_i, R))$$

$$(Q \Rightarrow (\forall i : B_i \lor wp(S_i, R)))$$

$$(Q \Rightarrow (\forall i : B_i \Rightarrow wp(S_i, R)))$$

[Implication] [Implication] [de Morgan] [Implication] [Implication]

Then we have:

$$(Q \Rightarrow BB) \land (\forall i : Q \land B_i \Rightarrow wp(S_i, R))$$

$$(Q \Rightarrow BB) \land (Q \Rightarrow (\forall i : B_i \Rightarrow wp(S_i, R)))$$

$$Q \Rightarrow (BB \land (\forall i : B_i \Rightarrow wp(S_i, R)))$$

$$Q \Rightarrow wp(IF, R)$$

[Assumptions] [above] [Distributivity] [Definition]

**Example**

Consider a binary search for a value $x$ known to be in array $b[0 : n-1]$. Suppose we are in a state satisfying predicate $Q$:

$$Q = \text{ordered}(B[0 : n-1]) \land 0 \leq i < k < n \land x \in b[i : j]$$

**We want to prove:**

$$\{Q\}$$
$$\text{if } b[k] \leq x \rightarrow i := k$$
$$[\] b[k] \geq x \rightarrow j := k$$
$$\text{fi}$$
$$\{x \in b[i : j]\}$$

**Proof:**

- $BB = (b[k] \leq x) \lor (b[k] \geq x) = T$

So $Q \Rightarrow BB$ and the first assumption of the theorem holds.

- $Q \land b[k] \leq x \Rightarrow x \in b[k : j]$ $= wp(\text{“}i := k\text{”}, x \in b[i : j])$

- $Q \land b[k] \geq x \Rightarrow x \in b[i : k]$ $= wp(\text{“}j := k\text{”}, x \in b[i : j])$

Thus all assumptions are satisfied and the theorem allows to conclude what we wished to prove.
Consider the following problem:

Write a program that, given fixed\footnote{“fixed” implies that the value should not be changed by execution of the program.} integers \( x \) and \( y \), sets \( z \) to the maximum of \( x \) and \( y \). Clearly, a command \( S \) is required that satisfies

\[
\{ T \} \; S \; \{ R : z = x \uparrow \, y \}
\]

- Before the program can be developed, \( R \) must be refined by replacing \( \uparrow \) by its definition: without knowing what \( \uparrow \) means we cannot write the program. (If \( \uparrow \) were a primitive – as it is in Sigrid – the problem reduces to a single assignment!)

\[
R : z \geq x \land z \geq y \land (z = x \lor z = y)
\]

What command could possibly be executed in order to establish this? \( R \) seems to suggest, amongst others, \( z := x \)

- Under what conditions will execution of \( z := x \) establish \( R \)? Simply calculate \( wp("z := x", R) \):

\[
wp("z := x", z \geq x \land z \geq y \land (z = x \lor z = y))
\]

\[
\begin{align*}
&= x \geq x \land x \geq y \land (x = x \lor x = y) \\
&= T \land x \geq y \land (T \lor x = y) \\
&= T \land x \geq y \land T \\
&= x \geq y
\end{align*}
\]

Our first attempt at a program can be:

\[
\text{if } x \geq y \rightarrow z := x \\
\text{fi}
\]

- Is that all? To prevent abortion \( Q \Rightarrow BB \).

\[
Q \Rightarrow BB
\]

\[
\begin{align*}
&= T \Rightarrow x \geq y \\
&= F
\end{align*}
\]

so the answer is NO! \textbf{At least one more guard} is needed. Try \( z := y \).

- Now calculate \( wp("z := y", R) \). It’s \( y \geq x \).

- Our second attempt is:

\[
\text{if } x \geq y \rightarrow z := x \\
[ ] y \geq x \rightarrow z := y \\
\text{fi}
\]

- Is that all?

\[
Q \Rightarrow BB
\]

\[
\begin{align*}
&= T \Rightarrow (x \geq y) \lor (y \geq x) \\
&= T
\end{align*}
\]
**A dud try**

It must seem that I just picked the two commands to try by magic. Let’s see what would happen if I tried something else. The post-condition is:

\[ R : z \geq x \land z \geq y \land (z = x \lor z = y) \]

and I chose the two disjuncts of the last conjunct in turn. Why not choose one of the other conjuncts? Let’s try the first:

\[ z \geq x \]

and proceed by setting \( z \) to \( x + 37 \). We calculate the weakest pre-condition:

\[
\begin{align*}
wp(“z := x + 37”, z \geq x \land z \geq y \land (z = x \lor z = y)) & = x+37 \geq x \land x+37 \geq y \land (x+37 = x \lor x+37 = y) \\
& = T \land x+37 \geq y \land (F \lor x+37 = y) \\
& = T \land x+37 \geq y \land x+37 = y \\
& = x+37 = y
\end{align*}
\]

This leads to the guarded command:

\[
\begin{align*}
\text{if } x+37 = y & \rightarrow z := x+37 \\
\text{fi}
\end{align*}
\]

which, whilst true, is hardly useful.

**Some principles**

**Principle:** Programming is a **goal-oriented activity.**

By this we mean that the desired result, or goal, \( R \), plays a more important role in the development of a program than the precondition \( Q \). This is obviously true since many pre-conditions are just \( T \).

**Principle:** Before attempting to solve a problem, make absolutely sure you **know what the problem is.**

**Principle:** Before developing a program, **make precise and refine the pre- and post-conditions.**

The form of the specification can influence algorithmic development, so that striving for **simplicity and elegance** should be helpful. With some problems, the major difficulty is making the specification simple and precise, and subsequent development of the program is fairly straightforward.
A strategy for developing an alternative command

To invent a guarded command:

- find a command $C$ whose execution will establish postcondition $R$ in at least some cases,
- find a Boolean $B$ satisfying $B \Rightarrow wp(C, R)$,
- put them together to form $B \rightarrow C$.

Continue to invent guarded commands until the precondition of the construct implies that at least one guard is true.

Note the symmetry in:

```plaintext
if \ x \geq y \rightarrow \ z := x
\[
\]
fi
```

which is possible because of the nondeterminism. If there is no reason to choose between $z := x$ and $z := y$ when $x = y$, one should not be forced to choose.

Nondeterminism is an important feature even if the final program turns out to be deterministic, for it allows us to devise a good programming methodology. One is free to develop many different guarded commands completely independently of each other. Any form of determinism, such as evaluating the guards in order of occurrence, drastically affects the way one thinks about developing alternatives.

A second example

Write a program that permutes (interchanges) the values of integer variables $x$ and $y$ so that $x \leq y$. We will reason more informally here, but still use the strategy above.

**First step:** write a suitable precondition $Q$ and postcondition $R$.

The problem is slightly harder than the first one, for it requires us to consider the initial and final values of variables. We use the “upper case” convention:

$Q : x = X \land y = Y$
$R : x \leq y \land (x = X \land y = Y \lor x = Y \land y = X)$

**What simple commands could cause $R$ to be established,** at least under some conditions?

Try : `skip`

The guard $B_i$ of a guarded command $B_i \rightarrow S_i$ of an alternative construct must satisfy $Q \land B_i \Rightarrow wp(S_i, R)$, according to the theorem about IF.

For the command `skip` we have $wp(skip, R) = R$, so that $B$ of the command $B \rightarrow skip$ must satisfy $Q \land B \Rightarrow R$. Since $Q$ implies the second conjunct of $R$, the first conjunct, $x \leq y$, of $R$ can be the guard, so we have:

```plaintext
if \ x \leq y \rightarrow \ skip
```
A second example (cont)

Now try:

\[ x, y := y, x \]

Calculating the weakest precondition we have:

\[
\text{wp}("x, y := y, x", R) = y \leq x \land (y = X \land x = y \lor y = Y \land x = X).\]

Again, the second conjunct of this weakest precondition is implied by \( Q \), so that the first conjunct \( y \leq x \) can be the guard.

This yields the alternative construct

\[
\text{if } x \leq y \rightarrow \text{skip} \\
[] y \leq x \rightarrow x, y := y, x \\
\text{fi}
\]

Since the disjunction of the guards, \( x \leq y \lor y \leq x \), is always true, the program is correct (with respect to the given \( Q \) and \( R \)).
Lecture 5 Iteration.

The third of the three fundamental composing structures is iteration. It is, of course, the key to all procedural programming.

Concepts:
- The do-command
- Invariants
- Termination and the bound function
- Non-determinism
- A theorem about the do-command

Text Reference:
Gries: Chapter 11.

The iterative command

This allows statements to be repeated (iterated) so long as certain conditions hold.

Pascal: Java

while B do while (B)
  S
  S

In our programming notation:

We express a simple loop as the following iterative command:

do B -> S
od

In general, we may have \( n \) guarded commands \( (n \geq 0) \) as shown below. This general form of the iterative command will be called DO:

do \( B_1 \rightarrow S_1 \)
[ ] \( B_2 \rightarrow S_2 \)
... 
[ ] \( B_n \rightarrow S_n \)
od
Semantics: what does it mean?

Command DO can be executed as follows:

1. If any guard $B_i$ is **not well-defined**, **abortion** may occur because the order of evaluation of the guards is arbitrary.

2. **Repeat until no longer possible**: choose a guard $B_i$ that is true and execute the corresponding statement, $S_i$.

Notes

- **Any number** of guarded commands are **allowed**.

- Since two or more of the guards may be true, **non-determinism** is allowed.

- The **general form**:

  ```
  do $B_1 \rightarrow S_1$
  [ ] $B_2 \rightarrow S_2$
  ...
  [ ] $B_n \rightarrow S_n$
  od
  ```

  can be recast:

  ```
  do $BB \rightarrow if B_1 \rightarrow S_1$
  [ ] $B_2 \rightarrow S_2$
  ...
  [ ] $B_n \rightarrow S_n$
  fi
  od
  ```

  where $BB$ is as before, $B_1 \lor B_2 \lor \ldots \lor B_n$. Equivalently:

  ```
  do $BB \rightarrow IF od$
  ```

  The general form often arises in program derivation.
The weakest precondition, $wp(DO, R)$

Now to derive the weakest precondition for the DO. (The difficulty is that the number of iterations of the loop is unknown.)

The predicate, $H_k(R)$, represents the set of all states in which execution of DO terminates in $k$ or fewer iterations with $R$ true.

- Thus $H_0(R) = \neg BB \land R$
  Here all guards are initially false ($BB$ is false) and after 0 iterations $R$ is satisfied.

- Now remember that DO is the same as:
  `do BB -> IF od`

  so we can define $H_k(R)$ recursively, in terms of $H_{k-1}(R)$, remembering that the loop might not be executed at all.

  $H_k(R) = H_0(R) \lor wp(IF, H_{k-1}(R))$ for $k > 0$,

- Finally we can define $wp(DO, R)$:

  $wp(DO, R) = (\exists k : 0 \leq k : H_k(R))$

An introduction to invariants

This definition is difficult to use, and we will reason about loops an alternative way - using invariants.

Consider the following sequence for storing in $z$ the product of $a \times b$, for $b \geq 0$, assuming we have no multiplication order.

(NB The division is really a right shift!)

```plaintext
{Q : b \geq 0}
x, y, z := a, b, 0;
d o y > 0 \land even(y) \rightarrow y, x := y + 2, x + x
[] odd(y) \rightarrow y, z := y - 1, z + x
od
{R : z = a \times b}
```

The essence is to find an invariant, $P$. If:

- $P$ is true before execution of DO;
- each iteration of the loop leaves $P$ true,

then $P$ is true before and after each iteration and on termination.

If $Q \Rightarrow P$ and $P \land \neg BB \Rightarrow R$, then $Q \Rightarrow wp(DO, R)$.

If DO terminates then the sequence has been proved.
**An introduction to invariants (cont)**

Let us confirm the invariant (plucked out of the air):

\[ P : y \geq 0 \land z + x \cdot y = a \cdot b \]

\{ Q : b \geq 0 \}

\[ x, y, z := a, b, 0; \]

**DO**

\[ y > 0 \land \text{even}(y) \rightarrow y, x := y \div 2, x + x \]

\[ [] \text{odd}(y) \rightarrow y, z := y - 1, z + x \]

**OD**

\{ R : z = a \cdot b \}

1. **Confirm that \( P \) is true before execution of \( \text{DO} \).** That is, confirm \( Q \Rightarrow P \). We assume it true and find the weakest precondition for the initialisation to establish it.

\[
\wp(“x, y, z := a, b, 0”, y \geq 0 \land z + x \cdot y = a \cdot b)
\]

\[ = b \geq 0 \land 0 + a \cdot b = a \cdot b \]

\[ = Q \]

2. **Confirm that each iteration of the loop leaves \( P \) true.**

We consider the guarded commands in turn. The first:

a) \( \wp(“y, x := y \div 2, x \cdot x”, y \geq 0 \land z + x \cdot y = a \cdot b) \)

\[ = y + 2 \geq 0 \land z + (x \cdot x) \cdot y + 2 = a \cdot b \]

and this is implied by the conjunction of \( P \) and the guard:

\[ y \geq 0 \land z + x \cdot y = a \cdot b \land y > 0 \land \text{even}(y) \]

The second guarded command:

b) \( \wp(“y, z := y - 1, z \cdot x”, y \geq 0 \land z + x \cdot y = a \cdot b) \)

\[ = y - 1 \geq 0 \land z + x \cdot (y - 1) = a \cdot b \]

\[ = y \geq 1 \land z + x \cdot (y - 1 + 1) = a \cdot b \]

\[ = y \geq 1 \land z + x \cdot y = a \cdot b \]

and this is implied by the conjunction of \( P \) and the guard:

\[ y \geq 0 \land z + x \cdot y = a \cdot b \land \text{odd}(y) \]
An introduction to invariants (concluded)

3. Confirm that $P \land \neg BB \rightarrow R$.

$$y \geq 0 \land z + x \cdot y = a \cdot b \land \neg(y > 0 \land \text{even}(y)) \land \neg\text{odd}(y)$$

$$\Rightarrow y \geq 0 \land z + x \cdot y = a \cdot b \land \neg(y > 0)$$

$$\Rightarrow y = 0 \land z + x \cdot y = a \cdot b$$

$$\Rightarrow z = a \cdot b$$

Remember we have proved that, only provided the sequence terminates, does it achieve the post-condition.

Annotation

A full annotation might go like this:

$$\{Q : b \geq 0\}$$

$$x, y, z := a, b, 0;$$

$$\{P : y \geq 0 \land z + x \cdot y = a \cdot b\}$$

$$\begin{array}{c}
\text{do } y > 0 \land \text{even}(y) \rightarrow \{P \land y > 0 \land \text{even}(y)\} y, x := y + 2, x + x \{P\} \\
\text{end}
\end{array}$$

$$\{P \land y \leq 0\}$$

$$\{R : z = a \cdot b\}$$

and indeed this is a useful aid until you get some experience.

A more appropriate form of annotation is:

$$\{Q : b \geq 0\}$$

$$x, y, z := a, b, 0;$$

$$\{\text{inv } P : y \geq 0 \land z + x \cdot y = a \cdot b\}$$

$$\begin{array}{c}
\{\text{bound } t : y\} \\
\text{do } y > 0 \land \text{even}(y) \rightarrow y, x := y + 2, x + x \\
\text{end}
\end{array}$$

$$\{R : z = a \cdot b\}$$

---

11 We consider the bound shortly.
**Non-determinism**

**Non-determinism** means that the programmer cannot determine which of \( n \) commands, whose guards are true, will be **chosen for execution**.

This does **not** imply that the result will be **unpredictable**. It might be, but it might not.

But note that “**unpredictable**” **results might be useful**. For example, in an algorithm that searches for a **maximum** element in an array, it might be acceptable to return the position of any **maximum** element in the array if there are more than one.

Here the post-condition is

\[
R: 0 \leq k < n \land (\forall i : 0 \leq i < n : b[i] \leq b[k])
\]

**or**

\[
R: 0 \leq k < n \land b[0..n-1] \leq b[k]
\]

To specify the **first** maximum requires a further conjunct,

\[
b[0..k-1] < b[k]
\]

which **may** imply a more **complex** algorithm.

(It suggests, too, that more computationally oriented quantifiers might be added to this methodology.)

---

**Dijkstra’s first problem**

Remember the urn with \( B \) black marbles and \( W \) white marbles, and the “game”: you put your hand into the urn and at random (ie non-deterministically) choose two marbles. Then if:

- If they are of **different colours**, **throw away the black** one, and **return the white** one.
- If they are **both black**, **throw one away** and **put one back**.
- If they are **both white**, **throw them away** and **add a black** one to the urn.

**Repeat** until no further move is possible – that is until there is only one marble left.

It’s not much of a game: it always finishes! The interest is in the **colour of the marble that is left**.

Here is the program - a **constructive proof** that if there were an **even number of whites** to start with the **final marble** will be **black**, and vice versa. Note that the program simulates the player. It’s the annotation that is of interest.

\[
\begin{align*}
\{ \text{inv } P: & \text{ even } w = \text{ even } W \\ & \{ \text{bound: } b+w-1 \} \\
\text{do } & \ b \geq 1 \land w \geq 1 \rightarrow b := b-1 \\
\text{[] } & \ b \geq 2 \rightarrow b := b-1 \\
\text{[] } & \ w \geq 2 \rightarrow b, w := b+1, w-2 \\
\text{od} ; \\
\{ \text{Post } & \ b+w=1 \land \text{ even } W \Rightarrow w=0 \land b=1 \} 
\end{align*}
\]
The importance of the bound function

In the final version of the product algorithm, we includes a comment defining the bound, which we traditionally call t. It is a key component in algorithms, being the means by which we determine whether an algorithm terminates. Its name is derived from its being an upper bound on the number of iterations of a loop yet to go.

Remember Dijkstra’s second problem about the lines joining the points of two distinct sets of points! Is there an end to the process of replacing pairs of crossing lines with lines that do not intersect?

For us the problem is less one of testing whether an algorithm terminates than one of designing algorithms so that they do.

So we see that we need to consider simultaneously both the invariant (which must be maintained) and the bound (which must be reduced) in designing loops.

A theorem about a loop DO

Suppose predicate P satisfies:

- $P \wedge B_i \Rightarrow \text{wp}(S_i, P)$ for all $i, 1 \leq i \leq n$

Suppose further that an integer function t satisfies the following where tl is a fresh identifier:

- $P \wedge BB \Rightarrow (t > 0)$,
- $P \wedge B_i \Rightarrow \text{wp}("t1 := t; S_i", t < tl) \forall i, 1 \leq i \leq n$

then

- $P \Rightarrow \text{wp}(DO, P \wedge \sim BB)$

[For the proof see the exercises in Gries.]

The predicate, P, above is the invariant of the loop as it is true at the start and end of each iteration of the loop.

The integer function, t, is called the bound function and ensures that the loop terminates. At any stage in the execution of DO, t is an upper bound on the number of iterations still to be performed, and decreases by at least 1 on every iteration. As it is bounded below by 0, the loop must terminate.
CITS4221  
Formal Methods in Software Engineering  
Program Correctness

Lecture 6 Iteration 2.

The loop is the key to all programming. We need to be able to prove that a given loop achieves its objective – and to design a loop given this objective.

Concepts:

• A check list for understanding a loop
• Development given an invariant and a bound
• A strategy for uni-guarded loops
• Some principles of program design
• A strategy for multi-guarded loops

Text Reference:  
Gries: Chapter 11.

Checklist for Understanding a Loop

3. Show \( P \) is true before execution of the loop begins.

3. Show that  
\[ \{ P \land B_i \} S_i \{ P \} \quad \text{for all } i, 1 \leq i \leq n \]

That is, execution of each guarded command terminates with \( P \) true, so that \( P \) is indeed a variant of the loop.

7. Show that  
\[ P \land \neg BB \Rightarrow R \]

so that on termination the desired result is true.

3. Show that  
\[ P \land BB \Rightarrow (t > 0) \]

so that \( t \) is bounded from below so long as the loop has not terminated.

2. Show that  
\[ \{ P \land B_i \} t_1 := t; S_i \{ t < t_1 \} \quad \text{for all } i, 1 \leq i \leq n \]

so that each loop iteration is guaranteed to decrease the bound function.
An example

We will use the Checklist to prove the following algorithm which finds the $n$th Fibonacci number $f_n$ for $n > 0$. That is, $f_0 = 0, f_1 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n > 1$.

$$
\begin{align*}
&\{0 < n\} \\
i, a, b := 1, 1, 0; \\
&\text{inv } P: 1 \leq i \leq n \land a = f_i \land b = f_{i-1} \\
&\text{bound } t: n - i \\
do \ i < n \rightarrow i, a, b := i + 1, a + b, a \\
&\text{od} \\
&\{R: a = f_n\}
\end{align*}
$$

3. Show $P$ is true before execution of the loop begins.

$$
\begin{align*}
&\text{wp(“i, a, b := i + 1, a + b, a”, P)} \\
&= \text{wp(“i, a, b := i + 1, a + b, a”, 1 \leq i \leq n \land a = f_i \land b = f_{i-1})} \\
&= 1 \leq 1 \leq n \land 1 = f_i \land 0 = f_0 \\
&= 0 < n \land T \land T \\
&= 0 < n
\end{align*}
$$

Thus

$$\begin{align*}
&\{0 < n\} \ i, a, b := 1, 1, 0 \ \{P\}
\end{align*}$$

holds, and $P$ is true before execution of the loop begins.

An example (cont)

b) Show that $\{P \land B_i\} S_i \{P\}$ for all $i, 1 \leq i \leq n$

That is, execution of each guarded command terminates with $P$ true, so that $P$ is indeed an invariant of the loop.

$$
\begin{align*}
&\text{wp(“i, a, b := i + 1, a + b, a”, P)} \\
&= \text{wp(“i, a, b := i + 1, a + b, a”, 1 \leq i \leq n \land a = f_i \land b = f_{i-1})} \\
&= 1 \leq i + 1 \leq n \land a + b = f_{i-1} \land a = f_i \\
&= 1 \leq i + 1 \leq n \land b = f_{i-1} \land a = f_i \\
&= 0 \leq i < n \land b = f_{i-1} \land a = f_i
\end{align*}
$$

Hence $\{P \land B_i\} S_i \{P\}$ for all $i$ (ie the only one!)

c) Show that $P \land \sim BB \Rightarrow R$ so that on termination the desired result is true.

$$
\begin{align*}
P \land \sim BB \\
= 1 \leq i \leq n \land a = f_i \land b = f_{i-1} \land \sim (i < n) \\
= i = n \land a = f_i \land b = f_{i-1} \\
= i = n \land a = f_n \land b = f_{n-1} \\
\Rightarrow R
\end{align*}
$$
An example (concluded)

4. Show that $P \land \neg BB \Rightarrow (t > 0)$ so that $t$ is bounded from below so long as the loop has not terminated.

$$P \land \neg BB$$

\[
\begin{align*}
1 \leq i \leq n \land a & = f_i \land b = f_i - i \land i < n \\
1 \leq i < n \land a & = f_i \land b = f_i - i \\
1 \leq i \land i < n \land a & = f_i \land b = f_i - i \\
1 \leq i \land 0 < n - i \land a & = f_i \land b = f_i - i \\
1 \leq i \land 0 < t \land a & = f_i \land b = f_i - i \\
\Rightarrow & \quad 0 < t
\end{align*}
\]

5. Show that $\{P \land B_i\} t1 := t; S_i \{t < t1\}$ all $i, 1 \leq i \leq n$ so that each loop iteration is guaranteed to decrease the bound function.

$$wp("t1:= n - i; i, a, b := i + 1, a + b, a", n - i < t1)$$

\[
\begin{align*}
& \quad wp("t1:= n - i", wp("i, a, b := i + 1, a + b, a", n - i < t1)) \\
& \quad wp("t1 := n - i", n - (i + 1) < t1) \\
& \quad n - (i + 1) < n - i \\
& \quad T
\end{align*}
\]

and so the bound function decreases on each iteration.

Thus the checklist is complete and the algorithm proved.

The development of a simple program (given the invariant and the bound function)

Consider the problem “Write a program that, given a fixed integer $n \geq 0$, and a fixed integer array $b[0 : n-1]$, stores in variable $s$ the sum of the elements of $b$.”

Clearly we have:

- $Q : n \geq 0$
- $R : s = (\sum j : 0 \leq j < n : b[j])$

We seek a loop with the invariant $P$ and bound function $t$:

$$P : 0 \leq i \leq n \land s = (\sum j : 0 \leq j < i : b[j])$$

$$t : n - i$$

1. The assignment $i, s := 0, 0$ obviously satisfies $P$, so we use it as an initialization.

2. Now choose the guard $B$ of the loop $\text{do } B \rightarrow S \text{ od}$. We require $P \land \neg B \Rightarrow R$, so let us choose $\neg B$.

$$P : 0 \leq i \leq n \land s = (\sum j : 0 \leq j < i : b[j])$$

$$R : s = (\sum j : 0 \leq j < n : b[j])$$

Clearly $i = n$ will do for $\neg B$. So we have the program:

\[
\begin{align*}
i, s & := 0, 0; \\
\text{do } & i \neq n \rightarrow ??? \\
\text{od}
\end{align*}
\]
The development of a simple program (cont)

Now the command must do two things.

- It must **decrease the bound function**.
- It must **maintain the invariant**.

The obvious way of decreasing the bound function is to increment \( i \) — but that also destroys the invariant! So it must be re-established. Clearly we must increase \( s \) — but by how much?

\[
i, s := 0, 0; \\
do \ i \neq n \rightarrow i, s := i + 1, s + e \\
o d
\]

Let us determine \( e \) by calculation.

\[
\text{wp}(\text{“i, s := i + 1, s + e”, P}) \\
= \text{wp}(\text{“i, s := i + 1, s + e”, 0 \leq i \leq n \land s = (\sum j : 0 \leq j < i : b[j])}) \\
= 0 \leq i + 1 \leq n \land s + e = (\sum j : 0 \leq j < i + 1 : b[j]) + b[i] \\
= 0 \leq i + 1 \leq n \land s + e = s + b[i]
\]

Thus we have \( e = b[i] \)

Further we have that this \( \text{wp} \) is implied by \( P \) and \( i \neq n \). This results in:

\[
i, s := 0, 0; \\
do \ i \neq n \rightarrow i, s := i + 1, s + b[i] \\
o d
\]

A strategy for (uni-guarded) loops

Every loop with a single guard can be expressed:

Initialization;
\{ \text{inv} : P \} \\
\{ \text{bound} : t \} \\
do \ B \rightarrow \text{Decrease } t, \text{ keeping } P \text{ true} \\
o d \\
\{ P \land \neg B \}

Thus a strategy is:

- **Determine an initialization** of the variables that satisfy \( P \).
- **Develop a guard** \( B \) such that \( P \land \neg B \Rightarrow R \).
- **Develop the body** so that it decreases the bound function \( t \) while re-establishing the loop invariant \( P \).
Two more principles

The emphasis so far (and for some time to come) is on correctness. This implies that incorrect programs should fail!

• All other things being equal, make the guards of an alternative command as strong as possible, so that some errors will cause abortion.

• All other things being equal, make the guards of an iterative command as weak as possible, so that some errors will cause infinite loops.

Note that this does not conform to traditional practice, where rather than test for a specific limiting value we often test for an open-ended interval, “in case we got it wrong”!

The development of a another simple program

Consider the following problem:

Sort 4 integer variables $q_0, q_1, q_2, q_3$ into ascending order. Clearly we can specify this formally:

$Q : q_0 = Q_0 \land q_1 = Q_1 \land q_2 = Q_2 \land q_3 = Q_3$

$R : q_0 \leq q_1 \leq q_2 \leq q_3 \land \text{perm}((q_0 .. q_3),(Q_0 .. Q_3))$

We seek a loop with the invariant, $P$, which maintains the $qs$ as a permutation of the $Q$s:

$P : \text{perm}((q_0 .. q_3),(Q_0 .. Q_3))$

and bound function, $t$, which is the number of inversions in the permutation. Taking some liberties with notation this is:

$t : (N \ i, j : 0 \leq i < j < 4 : q_j < q_i)$

Now to develop the program:

• **First the initialization.** None is required since $Q \Rightarrow P$.

• **Now the guard.** It has to satisfy $P \land \sim B \Rightarrow R$.

$B \equiv q_1 > q_0 \lor q_2 > q_1 \lor q_3 > q_2$.

Clearly this is going to lead to an IF as the body of the DO. [Since the general form of the DO can be expressed in this form let us develop the general form instead!]
The development of a another simple program (cont)

- Develop guarded commands each of which makes progress towards termination (ie reduces \( t \)) while it maintains the invariant, \( P \).

The only simple command which maintains \( P \) is the swap, such as :

\[
q_0, q_1 := q_1, q_0
\]

(There are of course 6 of these.) The swap is as likely to increase \( t \) as reduce it. BUT the guarded command must reduce it, so we seek an appropriate guard. A swap reduces \( t \) only if the variables of the swap represent an inversion. In the swap above this if \( q_1 < q_0 \). The symmetry suggests the program:

\[
\text{do } \begin{align*}
& q_1 < q_0 \rightarrow q_0, q_1 := q_1, q_0 \\
& q_2 < q_0 \rightarrow q_0, q_2 := q_2, q_0 \\
& q_3 < q_0 \rightarrow q_0, q_3 := q_3, q_0 \\
& q_2 < q_1 \rightarrow q_1, q_2 := q_2, q_1 \\
& q_3 < q_1 \rightarrow q_1, q_3 := q_3, q_1 \\
& q_3 < q_2 \rightarrow q_2, q_3 := q_3, q_2
\end{align*} \text{ od}
\]

\{Q: do you think all of the commands are required?\}

The development of a another simple program (concluded)

- Prove that \( P \land \neg BB \Rightarrow R \).

\( P \) and \( R \) are:

\[
P : \text{perm } ((q_0 .. q_3),(Q_0 .. Q_3))
\]

\[
R : q_0 \leq q_1 \leq q_2 \leq q_3 \land \text{perm } ((q_0 .. q_3),(Q_0 .. Q_3))
\]

\( P \Rightarrow \) the second conjunct of \( R \), and the negation of \( B_1, B_4 \) and \( B_6 \Rightarrow \) the first.

Note that this program is very non-deterministic in action - but always gives the same result.
Another strategy for (multi-guarded) loops

The strategy used above can be expressed:

- **Determine an initialization** of the variables that satisfy $P$.

- **Develop guarded commands** $B_i \rightarrow S_i$ such that $S_i$ makes progress towards termination, and $B_i$ maintains the invariant.

- **Continue inventing guarded commands** until enough of them have been developed to ensure $P \land \sim BB \Rightarrow R$.

Note that in the ordering program we didn’t adopt quite this strategy: we wrote down 6 guarded commands and then tested their sufficiency at the end. There is reason to suspect that not all are required. We may discuss later on how this program may be “improved”. For the moment note that the non-determinism does not prejudice the program’s correctness.
Lecture 7 Developing invariants.

The key to program design can be seen as the creation of the loop invariants. We consider now how a loop invariant may be derived from the post-condition

Concepts:

- The balloon theory
- Weakening predicates
  Deleting a conjunct
- Constructive post-conditions
  Replacing a constant by a variable

Text Reference:
  Gries: Chapter 16.

Developing invariants

So far the invariants have been pulled out of the hat. They clearly had something to do with the post-condition, of course, but it is not clear precisely what. We consider that now.

The balloon theory

Given a set of initial states, IS, for the loop and the set of states represented by the post-condition $R$, the invariant, $P$, of the loop is true before and after each iteration. Each iteration of the loop deflates $P$, until termination with $R$.

Thus the invariant is weaker than the post-condition.

We can weaken a predicate by:

- Deleting a conjunct
- Replacing a fixed value for a variable by a range of values.
Deleting a conjunct

If the post-condition, \( R \), is a conjunction, \textbf{an appropriate invariant,} \( P \), \textbf{can often be obtained by deleting a conjunct.}

The \textit{deleted conjunct} can be used in the \textit{determination of the guard(s),} \( B_i \), and the \textit{bound function,} \( t \).

We have already seen the example of sorting 4 integer variables \( q0, q1, q2, q3 \) into ascending order.

\textbf{The formal specification was:}
\begin{align*}
Q: & q0 = Q0 \land q1 = Q1 \land q2 = Q2 \land q3 = Q3 \\
R: & q0 \leq q1 \leq q2 \leq q3 \land \text{perm}((q0 .. q3),(Q0 .. Q3))
\end{align*}

\textbf{The invariant,} \( P \), \textbf{was:}
\( P : \text{perm}((q0 .. q3),(Q0 .. Q3)) \)

which is clearly \( R \) without its first conjunct.

\textbf{The guards:}
\begin{align*}
q1 < q0 & \quad q2 < q0 & \quad q3 < q0 \\
q2 < q1 & \quad q3 < q1 & \quad q3 < q2
\end{align*}

and bound function, \( t \), the number of inversions in the permutation, are derived from that first conjunct.

\textbf{Strategy}

When deleting a conjunct from \( R \) to produce an invariant \( P \), try using the complement of the deleted conjunct for the guard of the loop.
Another example: integer square root of an integer

Formally this can be specified:
Q: $0 \leq n$
R: $0 \leq a^2 \leq n < (a+1)^2$

We first rewrite $R$ as a set of conjuncts:
R: $0 \leq a^2 \land a^2 \leq n \land n < (a+1)^2$

Now we delete the third conjunct to get an invariant.
P: $0 \leq a^2 \land a^2 \leq n$

Let us now develop the loop using the 3-step procedure.

- **Initialization.** $P$ is established by the assignment $a := 0$.

- **The guard.** Take the complement of the deleted conjunct as the guard of the loop so on termination the guard is false, so the deleted conjunct is true. ($P \land \sim B \Rightarrow R$)

\[
a := 0; \\
do \ (a+1)^2 \leq n \rightarrow ??? \\
\od
\]

- **The body.** Since $a$ is bounded above by $\sqrt{n}$, a possible bound function $t$ is $[\sqrt{n} - a]$. The purpose of the body of the loop is to decrease $t$, while maintaining $P$.

\[
a := 0; \\
do \ (a+1)^2 \leq n \rightarrow a := a + 1 \\
\od
\]

Another example: quotient and remainder

The quotient, $q$, and remainder, $r$, of the division of $x \geq 0$ by $y > 0$, can be specified:

Q: $0 \leq x \land 0 < y$
R: $0 \leq r < y \land q \cdot y + r = x$

We first rewrite $R$ as a set of conjuncts:
R: $0 \leq r \land r < y \land q \cdot y + r = x$

Now we can delete the second conjunct to get an invariant and use its negation as a guard:

P: $0 \leq r \land q \cdot y + r = x$ and $r \geq y$

Now design the loop

- **Initialization.** $P$ is established by the $q, r := 0, x$.

- **The guard.** As above:

\[
q, r := 0, x; \\
do \ r \geq y \rightarrow ??? \\
\od
\]

- **The bound.** Decreasing the bound must falsify the guard: $y$ is fixed so we must decrease $r$. The bound is $r$. 

Quotient and remainder (cont)

- **The body.** Clearly, within the body $q$ and $r$ must be changed. $r$ must be decreased to reduce the bound and $q$ increased accordingly:

  $q, r := 0, x;
  \textbf{do } r \geq y \rightarrow q, r := E, r - k
  \textbf{od}$

To determine $E$ and $k$, let us calculate the weakest pre-condition:

\[
\wp("q, r := E, r - k", 0 \leq r \land q^*y + r = x)\\
= 0 \leq r - k \land E^*y + r - k = x
\]

but $P$ must be maintained:

\[
= 0 \leq r - k \land E^*y + r - k = q^*y + r\\
= 0 \leq r - k \land E^*y - k = q^*y\\
= 0 \leq r - k \land E = q + k/y
\]

To avoid division (!!!) we choose $k = y$, when $E = q + 1$.

$q, r := 0, x;
\textbf{do } r \geq y \rightarrow q, r := q + 1, r - y
\textbf{od}$

A final example: linear search

Determine the position of the first occurrence of $x$ in array $b$.

$Q: 0 < m \land x \in b[0:m - 1]\\
R: 0 \leq i < m \land (\forall j: 0 \leq j < i \land x \neq b[j]) \land x = b[i]$

The first two conjuncts of $R$ are established by the assignment $i := 0$. So delete the third conjunct, giving:

$P: 0 \leq i < m \land (\forall j: 0 \leq j < i \land x \neq b[j])$

**Now the loop itself.**

- **Initialization.** Just done.
- **The guard.** Use the complement of the deleted conjunct as the loop guard.

  $i := 0;
  \textbf{do } x \neq b[i] \rightarrow ???
  \textbf{od}$

- **The body.** $i$ is bounded above by $m$, so a possible bound function $t$ is $m - i$. To make progress towards termination we increase $i$ by 1.

  $i := 0;
  \textbf{do } x \neq b[i] \rightarrow i := i + 1
  \textbf{od}$

  This is the well-known Linear Search.
A cautionary tale

We have used informal arguments, within our design paradigm, but we really should prove the program correct. (It is.)

3. $Q \Rightarrow P$

$\text{wp(“i := 0”, P): } 0 \leq i < m \land (\forall j : 0 \leq j < i : x \neq b[j])$

$= 0 \leq 0 < m \land (\forall j : 0 \leq j < 0 : x \neq b[j])$

$= 0 < m \land T$

$= 0 < m$

$\Leftarrow Q$

4. $\{P \land B\} \text{S} \{P\}$

We must prove:

$P \land x \neq b[i] \Rightarrow \text{wp(“i := i + 1”, P)}$

$0 \leq i < m \land x \not\in b[0 : i - 1] \land x \neq b[i] \Rightarrow$

$0 \leq i + 1 < m \land x \not\in b[0 : i + 1 - 1]$

$0 \leq i < m \land x \not\in b[0 : i] \Rightarrow$

$0 \leq i + 1 < m \land x \not\in b[0 : i]$

which is, of course, not a tautology! What is wrong? The answer is that the post-condition, and therefore the invariant does not include all the relevant conjuncts, in particular the conjunct, $x \in b[0 : m - 1]$, from the pre-condition should have been added.

Linear Search Principle

To find a minimum value with some property, investigate values starting at the lower bound in increasing order. Similarly, when looking for a maximum value investigate values in a decreasing order.
Strengthening a post-condition

Most post-conditions are not in a form where an invariant can be derived by the deletion of a conjunct. This is not surprising because the post-condition describes a desired outcome, not a process to be followed to achieve it. In other words all trace is lost of any variables which might be required to perform the task. These must somehow recovered – by transforming the post-condition. Such post-conditions are said to be in constructive form. Since they introduce a new variable, which has a definite value, we consider a constructive precondition as stronger than the original, where the variable was unconstrained. This is clearly a creative step in the process, but there a number of possible strategies.

c) Replace a constant by a variable.

\[ x \in b[1 : N] \iff x \in b[1 : i] \land i = N \]

d) Replace an expression by a variable

\[ 0 \leq a^2 \leq n < (a+1)^2 \iff 0 \leq a^2 \leq n < b^2 \land b = a+1 \]

5. Introduce another variable

\[ x = \gcd(X, Y) \iff x = y = \gcd(X, Y) \]

6. Eliminate the existential quantifier

\[
(\exists p : 0 \leq p < n : x = b[p]) \land b[0 : n - 1] \leq x
\]
\[
\iff 0 \leq p < n \land x = b[p] \land b[0 : n - 1] \leq x
\]

Replacing a constant by a variable - summing an array

The problem is to set \( s \) to the sum the \( n \geq 0 \) elements of the fixed array \( b[0 : n-1] \). Formally:

\[
Q: 0 \leq n,
\]
\[
R: s = (\sum_j 0 \leq j < n : b[j])
\]

We introduce a fresh variable \( i \), in place of \( n \), yielding

\[
R: s = (\sum_j 0 \leq j < i : b[j]) \land i = n
\]

To derive an invariant, place bounds on \( i \). Motivating the choice of bounds is the need to establish the invariant initially, and the final value of \( i \). This latter is clearly \( n \). So we have:

\[
P: s = (\sum_j 0 \leq j < i : b[j]) \land ?? \leq i \leq n
\]

The first conjunct is established by \( i, s := 0, 0 \).

So a possible lower bound for \( i \) is 0, yielding the invariant

\[
P: s = (\sum_j 0 \leq j < i : b[j]) \land 0 \leq i \leq n
\]

Now the second conjunct of \( R \) requires \( i = n \) at termination, suggesting the bound function \( t = n - i \). Using this invariant and bound function the following was developed in lecture 16:

\[
i, s := 0, 0;
do\ i \neq n \rightarrow i, s := i + 1, s + b[i]
\]
\od
Lecture 8. Developing invariants (cont) and bounds.

We continue with deriving invariants, and consider important concepts related to bounds.

Concepts:

• Constructive post-conditions
  Replacing an expression by a variable
  Introducing another variable
  Eliminating the existential quantifier

• Bounds

Text Reference:
  Gries: Chapter 16 and 17, and Dromey.

Strengthening a post-condition (reprise)

Most post-conditions are not in a form where an invariant can be derived by the deletion of a conjunct. This is not surprising because the post-condition describes a desired outcome, not a process to be followed to achieve it. In other words all trace is lost of any variables which might be required to perform the task. These must somehow recovered – by transforming the post-condition. Such post-conditions are said to be in constructive form. Since they introduce a new variable, which has a definite value, we consider a constructive postcondition as stronger than the original, where the variable was unconstrained. This is clearly a creative step in the process, but there are a number of possible strategies.

4. Replace a constant by a variable.

\[ x \in b[1 : N] \iff x \in b[1 : i] \land i = N \]

4. Replace an expression by a variable

\[ 0 \leq a^2 \leq n < (a+1)^2 \iff 0 \leq a^2 \leq n < b^2 \land b = a+1 \]

8. Introduce another variable

\[ x = \text{gcd}(X, Y) \iff x = y = \text{gcd}(X, Y) \]

4. Eliminate the existential Quantifier

\[ (\exists p : 0 \leq p < n : x = b[p]) \land b[0 : n-1] \leq x \]
\[ \iff 0 \leq p < n \land x = b[p] \land b[0 : n-1] \leq x \]
Replacing an expression by a variable - integer square root (again)

Formally as before we have the **specification**:

Q: \(0 \leq n\)
R: \(0 \leq a^2 \leq n < (a+1)^2\)

Introducing \(b\) and using it in place of \(a + 1\) we have the **strengthened post-condition**:

R: \(0 \leq a^2 \leq n < b^2 \land b = a + 1\)

To derive an invariant, we place bounds on \(b\). (Remember, the choice of bounds should help to establish the invariant initially, and must reflect the final value of \(i\).) Clearly \(b\) is greater than \(a\). So we have:

P: \(0 \leq a^2 \leq n < b^2 \land a < b \leq ??\)

Also, the first conjunct of the post-condition can be established by the assignment \(a, b := 0, n + 1\)

This leads to the **invariant**:

P: \(0 \leq a^2 \leq n < b^2 \land a < b \leq n + 1\)

We now move on to the second and third parts of the design. (The first, initialization is already done!)

By investigating \(P \land \lnot B \Rightarrow R\), an appropriate guard for the loop is \(a + 1 \neq b\).

---

**Integer square root (cont)**

The program thus far is:

\[
a, b := 0, n + 1;
d o a + 1 \neq b \rightarrow ???
do
\]

Since the loop should terminate with \(a + 1 = b\), the task of the loop is to bring \(a\) and \(b\) closer together. Hence an appropriate bound function is \(t = b - a - 1\).

The interval \((a, b)\) could be reduced by one at each iteration, but, since there are two variables involved, it may be possible to halve the interval at each iteration by setting \(a\) or \(b\) to the midpoint \((a + b) / 2\). If so, the command of the loop could be:

\[
\text{if } ??? \rightarrow a := (a + b) / 2 \\
\text{[ ] } ??? \rightarrow b := (a + b) / 2 \\
\text{fi}
\]

Each command must maintain the invariant \(P\). To find a suitable guard for the first command, calculate

\[
\text{wp}(“a := (a + b) / 2”, 0 \leq a^2 \leq n < b^2 \land a < b \leq n + 1) = 0 \leq ((a + b) / 2)^2 \leq n < b^2 \land (a + b) / 2 < b \leq n + 1
\]

This must be implied by the conjunction of the pre-condition, \((P \land \text{the guard of the loop we’re in})\) and the guard \(B\) of the IF that we are seeking, that is:

\[
0 \leq a^2 \leq n < b^2 \land a < b \leq n + 1 \land a + 1 \neq b \land B
\]
**Integer square root (concluded)**

For this to imply the wp above, \( B = ((a+b) \div 2)^2 \leq n \).

Similarly, the guard for the second command is found to be \(((a+b) \div 2)^2 > n\).

We are led to the following:

\[
\begin{align*}
a, b & := 0, n + 1; \\
\{\text{inv} \; P: 0 \leq a^2 \leq n < b^2 \land a < b \leq n + 1\} \\
\{\text{bound} \; t: b - a + 1\} \\
d & a + 1 \neq b \rightarrow \\
\quad \text{if } ((a+b) \div 2)^2 \leq n \rightarrow a := (a+b) \div 2 \\
\quad \text{[if ] } ((a+b) \div 2)^2 > n \rightarrow b := (a+b) \div 2 \\
\quad \text{fi} \\
\text{od}
\end{align*}
\]

We may, of course, use the usual programming device of introducing a variable to save redundant calculations:

\[
\begin{align*}
a, b & := 0, n + 1; \\
\{\text{inv} \; P: 0 \leq a^2 \leq n < b^2 \land a < b \leq n + 1\} \\
\{\text{bound} \; t: b - a + 1\} \\
d & a + 1 \neq b \rightarrow \\
\quad d := (a+b) \div 2; \\
\quad \text{if } d^2 \leq n \rightarrow a := d \\
\quad \text{[if ] } d^2 > n \rightarrow b := d \\
\quad \text{fi} \\
\text{od}
\end{align*}
\]

**Introduce another variable – gcd**

Consider the greatest common divisor, \( gcd \), problem specified:

\[
\begin{align*}
Q: X > 0 \land Y > 0 \\
R: x = gcd(X, Y)
\end{align*}
\]

There is not much to go on, even when we include the definition of \( gcd \) as follows:

\[
\begin{align*}
gcd \; (x, y) & = gcd \; (x, y - x) \\
gcd \; (x, y) & = gcd \; (x - y, y) \\
gcd \; (x, x) & = x
\end{align*}
\]

Note that in procedures which operate over a few scalars, it is traditional for the original values to be destroyed. Here, \( x \) is; but what about \( y \)? Considering the properties above we strengthen \( R \) to include the final value of \( y \).

\[
\begin{align*}
R: x & = y = gcd(X, Y) \\

\end{align*}
\]

This is still not enough, and we introduce the notion that \( x \) and \( y \) will always be such that their \( gcd \) will remain constant. We get:

\[
\begin{align*}
R: x & = y = gcd(X, Y) \land gcd(x, y) = gcd(X, Y)
\end{align*}
\]

This is over-specified and can be simplified to:

\[
\begin{align*}
R: x & = y \land gcd(x, y) = gcd(X, Y)
\end{align*}
\]
gcd (cont)

Now to the design.

• The first conjunct can be deleted from $R$ to give $P$:

  
  $P$: gcd(x, y) = gcd(X, Y)

• The first conjunct can be negated to get the guard:

  
  \[
  \text{do } x \neq y \rightarrow \text{???} \\
  \text{od}
  \]

• Since it is clear from the definition that either $x$ and $y$ can be decremented by the other an appropriate bound function $t$ is $x-y$. And the quickly leads to:

  
  \[
  \text{do } x \neq y \rightarrow \\
  \quad \text{if } x < y \rightarrow y := y - x \\
  \quad [ ] y < x \rightarrow x := x - y \\
  \quad \text{fi} \\
  \text{od}
  \]

Note that since $x \neq y \equiv x < y \lor y < x$, this can be rewritten:

  
  \[
  \text{do } x < y \rightarrow y := y - x \\
  \quad [ ] y < x \rightarrow x := x - y \\
  \text{od}
  \]

which is the neatest form of this algorithm yet to appear!

Eliminating the existential quantifier – finding the largest element of an array

Consider finding the maximum element of an array.

Formally:

$Q$: $0 < n$  
$R$: $(\exists p : 0 \leq p < n : b[0 : n-1] \leq b[p] \land x = b[p])$

$R$ can be strengthened to:

$R$: $0 \leq p < n \land b[0 : n-1] \leq b[p] \land x = b[p]$

This is still inadequate and we introduce another variable $i$.

$R$: $0 \leq p < i \land b[0 : n-1] \leq b[p] \land x = b[p] \land i = n$

We are now on familiar territory – and can write the invariant by putting a range on $i$. Clearly its maximum value is $n$ and, since the initialization $i, p, x := 0, 0, b[0]$ satisfies the first 3 conjuncts of $R$, we have:

$P$: $0 \leq p < i \land b[0 : n-1] \leq b[p] \land x = b[p] \land 0 \leq i \leq n$

This leads to the program:

\[
\begin{align*}
\text{i, p, x := 0, 0, b[0];} \\
\text{do } i \neq n \rightarrow \\
\quad \text{if } a[i] < x \rightarrow i := i + 1 \\
\quad [ ] a[i] \geq x \rightarrow i, p, x := i + 1, i, a[p] \\
\quad \text{fi} \\
\text{do}
\end{align*}
\]
Bound functions

A bound function:

- shows that a loop terminates,
- provides an upper bound on the number of iterations before the termination of the loop. Hence, it can be used to approximate the time required to execute the program.

As a result, different bound functions may be used for the same program, depending on whether the programmer wishes to prove termination or show that the program is faster than another.

Usually the invariant of the loop will suggest a bound function. However we give two pointers to finding bound functions.

Using the notation of the problem and its solution

Consider searching a non-empty two-dimensional array $b[0:m-1, 0:n-1]$, where $0<m$ and $0<n$, for a fixed value $x$. On termination either $x = b[i, j]$, if the element is there, or $i = m$, if not.

The post-condition is:

$$R: (0 \leq i < m \land 0 \leq j < n \land x = b[i, j]) \lor (i = m \land x \notin b)$$

Using a diagram for part of it, an invariant is:

A possible bound function $t$ is the number of elements in the untested section which is $(m - i)^*n - j$.

Strategy: Express the bound function, in words, as a simple property of the invariant and the problem, and then formalize it (if necessary) as a mathematical expression.
**Lexicographic ordering**

Consider a pair of integers \((i, j)\). We can define lexicographical ordering by saying that one pair \((i, j)\) precedes another pair \((h, k)\), written \((i, j) < (h, k)\) if either \(i < h\) or \(i = h \land j < k\).

A similar definition exists for \(n\)-tuples \(- (3, 5, 5) < (4, 5, 5) < (4, 6, 0) < (4, 6, 1)\)

Consider the following simple loop:

\[
\{0 < m \land 0 < n\} \\
i, j := m - 1, n - 1; \\
\textbf{do} j \neq 0 \implies j := j - 1 \\
\textbf{od} \\
\textit{\textbf{[]}} i \neq 0 \land j = 0 \implies i, j := i - 1, n - 1 \\
\textbf{od}
\]

Execution is guaranteed to terminate, because:
- Variables \(i\) and \(j\) satisfy \(0 \leq i < m\) and \(0 \leq j < n\).
- Each iteration transforms \((i, j)\) into a lexicographically smaller pair, which can happen a only finite number of times.

A suitable bound function \(t\) is \(i^n m + j\).

**Theorem:** Consider a pair \((i, j)\), where \(i\) and \(j\) are expressions containing variables in a loop. Suppose each iteration of the loop lexicographically decreases \((i, j)\). Then execution of the loop must terminate and a suitable bound function is \((i - \text{lower}(i)) \ast \text{range}(j) + j - \text{lower}(j)\).

**Reducing the bound**

Just as the creation of a bound function, is fairly straightforward, so, too, is the reduction in its value required on each iteration. The following observations reflect this.

- It is usually the case that the bound is reduced by one on each iteration. This is certainly the case where the bound function contains only one variable. The variable may be either increased or diminished, depending on the form of the bound function.

- There are some cases where it could be reduced by 2, but these generally involve symmetry.

- As we saw with the integer square root, we can sometimes halve the bound, leading to a gain that is logarithmic.
Lecture 9. Program transformation.

The constructs of Sigrid (and of the methodology on which it is based) are meant for ease of derivation. This often means that the procedures are non-deterministic. We look now at strategies for transforming the procedures so that they are more easily transcribed into traditional languages. On the way we consider questions of efficiency.

Concepts:

- Transformation of parallel assignments
- Reducing the number of guards in a loop
  9. Subset selection
  5. Strengthening a guard

Text Reference:
Gries: Chapter 19.

Program transformation

- The program design techniques lead to programs which are certainly correct.

- Their efficiency primarily depends on the strategy adopted as we saw with the “integer square root” procedures – one was \( O(\sqrt{n}) \), the other \( O(\log n) \). That is the order of magnitude improvements are often made at this stage.

- Nevertheless, at the detailed level, their performance may not be optimal. We now consider techniques for converting a correct procedure into another, more efficient procedure.

- Sometimes transformations are necessary to translate into a deterministic language like Pascal, C or Java.
**Common sub-expressions**

The design techniques, which often concern themselves with developing the guards one at a time, are likely to lead to the **same expression appearing in a number of places**. For example in lecture 8 we derived the following logarithmic procedure for integer square root.

```
proc IntSqrt (value n : nat, result a : nat)
pre Q: true
var b : nat
a, b := 0, n+1;
inv P : 0 ≤ a^2 ≤ n < b^2 ∧ a < b ≤ n + 1
bound t : b−a+1
do a + 1 ≠ b →
   if ((a+b) / 2)^2 ≤ n → a := (a+b) / 2
   [] ((a+b) / 2)^2 > n → b := (a+b) / 2
fi
od
post R: 0 ≤ a^2 ≤ n < (a+1)^2
```

**Notes**
- Although I have not lectured on procedures, I will express the algorithms that way from here on in, so that you may run them in Sigrid. (The syntax follows Gries.)

- The form of the parameters is fairly straightforward. There are three modes: **value**, **result** and **value result**.

- Sigrid has the type **natural**, which makes pre-conditions of the form n≥0 unnecessary.

**Local variables for common sub-expressions**

As we noted then, we may use the usual programming device of **introducing a variable** to avoid unnecessary calculations of the same expression:

```
proc IntSqrt (value n : nat, result a : nat)
pre Q: true
var b : nat
a, b := 0, n+1;
inv P : 0 ≤ a^2 ≤ n < b^2 ∧ a < b ≤ n + 1
bound t : b−a+1
do a + 1 ≠ b →
   var d : nat
   d := (a+b) / 2;
   if d^2 ≤ n → a := d
   [] d^2 > n → b := d
fi
od
post R: 0 ≤ a^2 ≤ n < (a+1)^2
```

**Notes**
- We can introduce **local variables** within any loop.
- The strategy often introduces a sequencing of statements.
Linearization of assignments

Multiple assignments are usually unavailable in programming languages, and therefore they must be decomposed into their components during the transliteration. Furthermore the generality of the multiple assignment statement implies a potential loss of efficiency. Let us investigate.

• Clearly we can transform:
  
a, b := 0, n + 1  \rightarrow  \ a := 0;
  \ b := n + 1

• But equally clearly we cannot transform:
  
i, p := i + 1, i  \rightarrow  \ i := i + 1;
  \ p := i

because this causes $p$ to be assigned a value that’s 1 too high.

• We can though transform:
  
i, p := i + 1, i  \rightarrow  \ p := i;
  \ i := i + 1

• Likewise we cannot transform:
  
x, y := y, x  \rightarrow  \ x := y;
  \ y := x

because it just gives the wrong answer!

General transformation of assignments

• A general transformation is:
  
$$
\begin{align*}
  v_1, v_2, \ldots, v_n & := e_1, e_2, \ldots, e_n \\
  t_1 & := e_1; \\
  t_2 & := e_2; \\
  \vdots \\
  t_n & := e_n;
\end{align*}
$$

where $\mathcal{I}$ is a set of new variables. Remember that the evaluation of $e$ (and therefore the assignment to $t$) can be in any order, but the assignment to $y$ must be left-to-right.

• Thus a completely safe transformation is:
  
$$
\begin{align*}
  i, p & := i + 1, i \\
  t_1 & := i + 1; \\
  t_2 & := i; \\
  i & := t_1; \\
  p & := t_2
\end{align*}
$$

• We can then:
  – move any assignment provided it does not move over statements that affect any variable in its (rhs) expression.  

5. substitute for any $t$-variable the value assigned to it.

Successively from left to right we have:

$$
\begin{align*}
t_1 & := i + 1; \\
t_2 & := i; \\
i & := t_1; \\
p & := t_2;
\end{align*}
$$
Partitioning an array

Suppose we have an integer array \( b[0..n–1] \), and we wish to permute the entries and assign \( p \), so that all those entries \( \leq 0 \) are in \( b[0..p–1] \) and those strictly positive are in \( b[p..n–1] \):

Q: \( 0 \leq n \)
R: \( 0 \leq p \leq n \land (\forall k : 0 \leq k < p : b[k] \leq 0) \land (\forall k : p \leq k < n : b[k] > 0) \);

\( p \) clearly divides the array into 2 parts.

\[
\begin{array}{c|c|c|c}
  & 0 & p-1 & p & n-1 \\
\hline
b & \leq 0 & > 0 &
\end{array}
\]

One strategy is to introduce another variable \( q \), such that \( b[0..p–1] \leq 0 \) and \( b[q..n–1] > 0 \). Thus the constructive postcondition is:

\[
0 \leq p \leq q \leq n \land (\forall k : 0 \leq k < p : b[k] \leq 0) \land (\forall k : q \leq k < n : b[k] > 0) \land p=q
\]

which leads to

- the guard \( p \neq q \),
- the bound \( q–p \)
- and the invariant, \( P: 0 \leq p \leq q \leq n \land (\forall k : 0 \leq k < p : b[k] \leq 0) \land (\forall k : q \leq k < n : b[k] > 0) \)

The body of the loop must simultaneously reduce the bound while maintaining the invariant. Reduce the bound involves decrementing \( q \) or incrementing \( p \).

When can we increment \( p \)? The wp("p:=p+1", \( P \)) is \( b[p] \leq 0 \). Therefore if \( b[p] \) is already \( \leq 0 \), we simply increment \( p \). How else may we ensure that \( b[p] \) is \( \leq 0 \)? If \( b[q–1] \) is \( \leq 0 \), then we can simply swap it with \( b[p] \) and increment \( p \).

These 2 are insufficient (\( BB \) is not identically true), so we develop a similar pair for \( q \).

\[
\text{proc Partition (value result b : array of int, value n : nat, result p : nat)}
\]

\[
\begin{align*}
\text{pre } & Q: 0 \leq n \\
\text{var } & q := n \\
\text{p := 0;} \\
\text{inv } & P: 0 \leq p \leq q \leq n \land (\forall k : 0 \leq k < p : b[k] \leq 0) \land (\forall k : q \leq k < n : b[k] > 0) \\
\text{bound } & t : q-p \\
\text{do } & p \neq q \rightarrow \\
\text{if } & b[p] \leq 0 \rightarrow p := p+1 \\
\text{[] } & b[q-1] \leq 0 \rightarrow b[p],b[q-1],p := b[q-1],b[p],p+1 \\
\text{[] } & b[q-1] > 0 \rightarrow q := q-1 \\
\text{[] } & b[p] > 0 \rightarrow b[p],b[q-1],q := b[q-1],b[p],q-1 \\
\text{fi} \\
\text{od}
\end{align*}
\]

\[
\text{post } R: 0 \leq p \leq n \land (\forall k : 0 \leq k < p : b[k] \leq 0) \land (\forall k : p \leq k < n : b[k] > 0)
\]

This is, of course, non-deterministic.
Selection a subset of guards

Do we need all the guards?

The criterion that is relevant here is that BB must be true. We need only as many guards as are necessary to achieve this. Note that the outer pair \( b[p] \leq 0 \land b[p] > 0 \) achieve this on their own. The inner pair can be removed.

\[
\text{proc Partition (value result b : array of int, value n : nat,}
\text{ result p : nat)
\]

\[
\begin{align*}
\text{pre Q: } & 0 \leq n \\
\text{ var } & q := n \\
\text{ p := 0;} \\
\text{ inv } P: & 0 \leq q \leq n \land (\forall k: 0 \leq k < p: b[k] \leq 0) \land \\
& (\forall k: q \leq k < n: b[k] > 0) \\
\text{ bound } t: & q \vdash p \\
\text{ do } & p \neq q \rightarrow \\
\text{ if } & b[p] \leq 0 \rightarrow p := p + 1 \\
\text{ if } & b[p] > 0 \rightarrow b[p], b[q - 1], q := b[q - 1], b[p], q - 1 \\
\text{ fi } \\
\text{ od } \\
\text{ post R: } & 0 \leq q \leq n \land (\forall k: 0 \leq k < p: b[k] \leq 0) \land \\
& (\forall k: p \leq k < n: b[k] > 0)
\end{align*}
\]

Notes

- We derived the first 2 guards, then the second pair (by symmetry), and finally chose one from each pair.
- We could have retained the inner pair, and eliminated the outer two.

Reducing the number of guards by strengthening

- As long as \( Q \rightarrow BB \), we can strengthen any guard, \( B_i \), without changing the result. (We may, of course, change the order of computation!) If the strengthening produces \( F \) then the guarded command can be removed.

- For any pair of predicates (here guards) \( B_i, B_j \)

\[
B_i \lor B_j = B_i \lor (B_j \land \sim B_i)
\]

Therefore we can always conjoin the negation of any guard with any other. For example, we can transform the second guard in:

\[
\begin{align*}
do & q_1 < q_0 \rightarrow q_0, q_1 := q_1, 0 \\
\[ & q_2 < q_0 \rightarrow q_0, q_2 := q_2, 0 \\
\[ & q_3 < q_0 \rightarrow q_0, q_3 := q_3, 0 \\
\[ & q_2 < q_1 \rightarrow q_1, q_2 := q_2, 1 \\
\[ & q_3 < q_1 \rightarrow q_1, q_3 := q_3, 1 \\
\[ & q_3 < q_2 \rightarrow q_2, q_3 := q_3, q_2 \\
\od & q_2 < q_0 \land \sim(q_1 < q_0) \land \sim(q_2 < q_1) \land q_0 \land \sim(q_2 < q_1) = F
\end{align*}
\]

and so the whole guarded command can be deleted\(^1\). Note that this process often eliminates the non-determinism.

\(^1\) I have no procedure for deciding in general which is the minimal number of guards.
Powering

Consider the problem of setting \( z \) to the \( b \)th power of \( a \), where \( b \) is a natural number.

Q: true (provided that \( b \) is a natural number)
R: \( z = a^b \)

By replacing the constant \( b \) by a variable \( y \) we transform \( R \):

R: \( z = a^y \land y = b \)

from which the invariant follows:

P: \( z = a^y \land 0 \leq y \leq b \)

which in turn inevitably leads to the (linear) procedure:

```plaintext
proc Power(value a : int, value b : nat, result z : int)
pre Q : true
  var y : int
  y, z := 0, 1;
  inv P: z = a^y \land 0 \leq y \leq b
  bound t: b-y
  do y \neq b \rightarrow
      z, y := z*a, y+1
  od
post: z = a^b
```

Logarithmic powering

Let us now see whether we can improve the speed!

- We replaced \( b \) by a constant:
  \[
  z = a^y \land y = b
  \]
- Note that we may equally have used the transformation:
  \[
  z \ast a^y = a^b
  \]
  which produces a procedure in which \( y \) counts down rather than up. Think about this equation for a while.

- Now, for the logarithmic version. The only way forward is to replace \( a \), too! The post-condition becomes:
  \[
  z \ast x^y = a^b
  \]
- This suggests an invariant of
  \[
  P: 0 \leq y \land z \ast x^y = a^b
  \]
  and a bound function of \( y \).
Maybe logarithmic powering

There are two ways of reducing the bound function:

\[
y := y - 1 \quad \text{and} \quad y := y \div 2
\]

Calculating the weakest precondition of these commands leads to:

\[
\text{proc Power(value } a : \text{ int, value } b : \text{ nat, result } z : \text{ int)}
\]

**pre Q : true**

\[
\begin{align*}
\text{var x : int} \\
\text{var y : nat} \\
x, y, z := a, b, 1; \\
\text{inv } P: 0 \leq y \land z \times x^y = a^b \\
\text{bound } t: y \\
\text{do } 0 < y \land \text{even}(y) \rightarrow \\
\quad x, y := x \times x, y + 2 \\
\text{[] } 0 < y \rightarrow \\
\quad z, y := z \times x, y - 1 \\
\text{od} \\
\text{post: } z = a^b 
\end{align*}
\]

Note that this is **non-deterministic** and could take linear time, if the implementation always chose the second guard.

Logarithmic powering

We try to **strengthen the second guard**:

\[
0 < y \land \sim (0 < y \land \text{even}(y)) \\
= 0 < y \land (\sim (0 < y) \lor \sim \text{even}(y)) \quad \text{de Morgan} \\
= 0 < y \land (0 < y) \lor 0 < y \land \sim \text{even}(y) \quad \text{distributivity} \\
= F \lor 0 < y \land \sim \text{even}(y) \quad \text{contradiction} \\
= 0 < y \land \sim \text{even}(y) \quad \text{unit of disjunction} \\
= 0 < y \land \text{odd}(y) \quad \text{definition of odd} \\
= \text{odd}(y) \quad \text{definition of odd}
\]

This leads to:

\[
\text{proc Power(value } a : \text{ int, value } b : \text{ nat, result } z : \text{ int)}
\]

**pre Q : true**

\[
\begin{align*}
\text{var x : int} \\
\text{var y : nat} \\
x, y, z := a, b, 1; \\
\text{inv } P: 0 \leq y \land z \times (\prod i : 0 \leq i \times y : x) = (\prod i : 0 \leq i < b : a) \\
\text{bound } t: [\log(y+1)] \\
\text{do } 0 < y \land \text{even}(y) \rightarrow \\
\quad x, y := x \times x, y + 2 \\
\text{[] } \text{odd}(y) \rightarrow \\
\quad z, y := z \times x, y - 1 \\
\text{od} \\
\text{post R: } z = (\prod i : 0 \leq i < b : a) 
\end{align*}
\]

Note that it is now **deterministic**, and **logarithmic** so that the bound function is tighter. (I have, for no particular reason, expressed \(\land\) in terms of \(\prod\).)
Multi-guarded do-statements

We noted in lecture 15 that the **multiply-guarded DO-statement** can be expressed as a **singly-guarded one**. The general form:

\[
\text{do } B_1 \rightarrow S_1 \\
[] B_2 \rightarrow S_2 \\
\vdots \\
[] B_n \rightarrow S_n \\
\text{od}
\]

can be recast as:

\[
\text{do } BB \rightarrow \\
\quad \text{if } B_1 \rightarrow S_1 \\
\quad [] B_2 \rightarrow S_2 \\
\quad \vdots \\
\quad [] B_n \rightarrow S_n \\
\quad \text{fi} \\
\text{od}
\]

where \( BB = B_1 \lor B_2 \lor \ldots \lor B_n \)

The general form is the more “efficient”, since the alternative form requires the evaluation of \( BB \) as well as some of the guards on each iteration — and we used the transformation in reverse with \( gcd \). On the other hand with “real” languages there is no equivalent to the general form and we have to use the forwards transformation.
Gcd (again)

Consider the gcd program again.

The multi-guard version is:

\begin{verbatim}
proc GCD(value a, b : nat, result y : nat)
const A, B = a, b
pre Q: 1\leq a \land 1\leq b
  inv P: 0 < a \land 0 < b \land gcd(a,b) = gcd(A,B)
  bound t: a \uparrow b
  do a > b →
    a := a - b
  [] b > a →
    b := b - a
od;
y := a
post R: y = gcd(a,b)
\end{verbatim}

Notes (relating to Sigrid)
• We need to express gcd formally,
• There are quantifiers for the first, α, and last, Ω, values in a range that satisfy a criterion.
• There is a neat construct for capturing the “capital letters for initial values” convention.

The uni-guarded version

Of course, multi-guarded loops have no direct equivalence in traditional languages.

Remembering that the multi-guarded DO command can be expressed as an IF within a uni-guarded DO, we can recast this:

\begin{verbatim}
proc GCD(value a, b : nat, result y : nat)
const A, B = a, b
pre Q: 1\leq a \land 1\leq b
  inv P: 0 < a \land 0 < b \land gcd(a,b) = gcd(A,B)
  bound t: a \uparrow b
  do a \neq b →
    if a > b →
      a := a - b
  [] b > a →
    b := b - a
  fi
od;
y := a
post R: y = gcd(a,b)
\end{verbatim}
Gcd (nested do loops)

In this procedure at least, we can go a step further and replace the inner IF with a pair of DOs.

```plaintext
proc GCD(value a, b : nat, result y : nat)
const A, B = a, b
pre Q: 1≤a ∧ 1≤b
   inv P: 0 < a ∧ 0 < b ∧ gcd(a,b) = gcd(A,B)
   bound t: a ↑ b
   do a ≠ b →
      inv P1: P
      bound t1: a
      do a > b →
         a := a - b
      od;
      inv P2: P
     bound t2: b
      do b > a →
         b := b - a
      od;
   od;
   y := a
post R: y = gcd(a,b)
```

Notes
- This procedure should be faster because it capitalizes on sequences of subtractions of either \( a \) or \( b \).
- Non-determinism has been removed.

Logarithmic powering (again)

We can apply the same technique to the logarithmic powering procedure. In lecture 9 we had:

```plaintext
proc Power(value a : int, value b : nat, result z : int)
pre Q : true
   var x : int
   var y : nat
   x, y, z := a, b, 1;
   inv P: 0 ≤ y ∧ z * x^y = a^b
   bound t : \[ \log(y+1) \]
   do 0 < y ∧ even(y) →
      x, y := x * x, y ÷ 2
   [] odd(y) →
      z, y := z * x, y - 1
   od
post R: z = a^b
```
The uni-guarded form

Here is the uni-guarded form, which can be easily transliterated into Pascal or C.

```plaintext
proc Power(value a : int, value b : nat, result z : int)
pre Q : true
  var x : int
  var y : nat
  x, y, z := a, b, 1;
  inv P: 0 ≤ y ∧ z * x^y = a^b
  bound t : [log(y+1)]
  do y ≠ 0 →
    if 0 < y ∧ even(y) →
      x, y := x * x, y ÷ 2
    [] odd(y) →
      z, y := z * x, y - 1
    fi
  od
post R: z = a^b
```

Logarithmic powering (the end!)

We now replace the first guarded command of the IF by a DO, the second command becoming unconditional:

```plaintext
proc Power(value a : int, value b : nat, result z : int)
pre Q : true
  var x : int
  var y : nat
  x, y, z := a, b, 1;
  inv P: 0 ≤ y ∧ z * x^y = a^b
  bound t : [log(y+1)]
  do y ≠ 0 →
    inv P1: P
    bound t1:
      do 0 < y ∧ even(y) →
        x, y := x * x, y ÷ 2
      od;
      z, y := z * x, y - 1
    od
post R: z = a^b
```
Adding variables and strengthening the invariant

Perhaps the most gains arise with the introduction of new variables. (Beyond those introduced for the common subexpression problem.) Consider the linear integer square root problem yet again. We left it like this:

```
proc IntSqRt(value n : nat, result a : nat)
pre Q: true
    a := 0;
    inv P: 0 ≤ a² ≤ n
    bound t: √n-a
    do (a + 1)² ≤ n →
        a := a + 1
    od
post R: 0 ≤ a² ≤ n < (a+1)²
```

Multiplication, which is, $O(n^2)$ is more costly than addition, which is $O(n)$. Let us see if we can eliminate the multiplication by:

- introducing a new variable, $b$,
- expanding the program so that it is maintained as $(a + 1)^2$,
- adding $b = (a + 1)^2$ to our invariant.

```
proc IntSqRt(value n : nat, result a : nat)
pre Q: true
    var b : nat
    pred P1: 0 ≤ a² ≤ n
    pred P2: b=(a+1)²
    a,b := 0,1;
    inv P: P1 ∧ P2
    bound t: √n-a
    do b ≤ n →
        a,b := a+1, b+2*a+3
    od
post R: 0 ≤ a² ≤ n < (a+1)²
```

Note

- Sigrid has a facility for naming predicates, and therefore keeping the size of complex predicates (here the invariant) under control. It is traditional to have one for each stage of the development.

Integer square root (cont)

We can initialize $b$ to 1, and, assuming that we maintain $b$ in the same assignment as $a$:

```
a,b := a+1,b+E
```

we simply use the $wp$ calculation to determine $E$. 

```
proc IntSqRt(value n : nat, result a : nat)
pre Q: true
    var b : nat
    pred P1: 0 ≤ a² ≤ n
    pred P2: b=(a+1)²
    a,b := 0,1;
    inv P: P1 ∧ P2
    bound t: √n-a
    do b ≤ n →
        a,b := a+1, b+2*a+3
    od
post R: 0 ≤ a² ≤ n < (a+1)²
```
**Integer square root (concluded)**

This still contains a multiplication, though since it’s by 2 it can be implemented by a shift. Nevertheless we can eliminate it by a second application of the same principle, maintaining \( c = 2a + 3 \).

\[
\text{proc IntSqRt(value n : nat, result a : nat)} \\
\text{pre Q: true} \\
\text{ var b,c : nat} \\
\text{ pred P1: 0 \leq a^2 \leq n} \\
\text{ pred P2: b=(a+1)^2} \\
\text{ pred P3: c=2*a+3} \\
\text{ a,b,c := 0,1,3;} \\
\text{ inv P : P1 \land P2 \land P3} \\
\text{ bound t : \forall n-a} \\
\text{ do b \leq n \rightarrow} \\
\text{ a,b,c := a+1, b+c, c+2} \\
\text{ od} \\
\text{ post R: 0 \leq a^2 \leq n < (a+1)^2 }
\]

Note that after these transformations, the invariants are necessary to understand the resulting procedure.

**Evaluating polynomials**

Consider evaluating the polynomial:

\[
y = a_0 + a_1x + a_2x^2 + \ldots a_nx^n.
\]

This is just the summation of some function of the elements of an array, which leads directly to:

\[
\text{proc Poly(value a : array of int,} \\
\text{ value n : nat,} \\
\text{ value x : int,} \\
\text{ result y :int)} \\
\text{ pre Q: true} \\
\text{ var i: nat} \\
\text{ i,y := 0,0;} \\
\text{ inv P : 0\leq i\leq n \land y = (\Sigma p : 0\leq p\leq i : a[p]*x^p)} \\
\text{ bound t :n-i} \\
\text{ do i \neq n \rightarrow} \\
\text{ i,y := i+1, y + a[i]*x^i} \\
\text{ od} \\
\text{ post R: y = (\Sigma p : 0\leq p\leq n : a[p]*x^p)}
\]

A further step, which introduces an inner loop, is required to evaluated the power within this outer loop. Thus the algorithm is \( O(n^2) \).
Evaluating polynomials in $O(n)$

We can, however maintain $x^i$ as an invariant of the loop!

```plaintext
proc Poly(value a : array of int,
    value n : nat,
    value x : int,
    result y : int)

pre Q: true
    var i : nat
    var z : int
    pred P1 : 0 ≤ i ≤ n ∧ y = (Σ p : 0 ≤ p < i : a[p] * x^p)
    pred P2 : z = x^i
    i, y, z := 0, 0, 1;
    inv P : P1 ∧ P2
    bound t : n - i
    do i ≠ n →
        i, y, z := i + 1, y + a[i] * z, z * x
    od
post R: y = (Σ p : 0 ≤ p < n : a[p] * x^p)
```

The resulting procedure is linear.