Notes on Program Semantics and Derivation

H. Conrad Cunningham
cunningham@cs.olemiss.edu

Software Architecture Research Group
Department of Computer and Information Science
University of Mississippi
201 Weir Hall
University, Mississippi 38677 USA

August 2006

H. Conrad Cunningham, D.Sc.
Professor and Chair
Department of Computer and Information Science
University of Mississippi
P. O. Box 1848
201 Weir Hall
University, Mississippi 38677
USA

cunningham@cs.olemiss.edu

Preface

This set of lecture notes was developed for use in the course CSCI 550, Program Semantics and Derivation, that I teach in the Department of Computer and Information Science at the University of Mississippi. These notes assume that the student is familiar with a calculational style of predicate logic similar to that sketched in the accompanying notes *A Programmer's Introduction to Predicate Logic*. The course is open to advanced undergraduates and beginning graduate students in computer science.

I first encountered the ideas presented in these notes when I purchased a copy of David Gries' now classic book *The Science of Programming* (Springer-Verlag, 1981) in the early 1980s while I was working in industry. However, I did not give program derivation much attention until in the mid-1980s when I took a graduate course on the topic taught by Jan Tijmen Udding at Washington University in St. Louis. Udding's course did not have a textbook, but it had content and approach similar to the textbook *A Method of Programming* by Edsger W. Dijkstra and W. H. J. Feijen (Addison-Wesley, 1988). My approach to programming has also been influenced by K. Mani Chandy and Jayadev Misra's *Parallel Program Design: A Foundation* (Addison-Wesley, 1988).

I first taught CSCI 550 at the University of Mississippi in the Spring semester of 1990 with an emphasis on programming language semantics and program verification. I used the textbook *Programming Logics: An Introduction to Verification and Semantics* by Raymond Gumb (Wiley, 1989) .

After teaching the course once, I decided it was better to focus more on program derivation. So for the second offering in Spring 1991, I adopted Gries' book as the textbook and also used my class notes from Udding's course and material from the Dijkstra-Feijen book as sources of ideas for lectures and assignments. For the 1991 offering of the course, I developed a handout on predicate logic, which by 1996 had evolved into a set of notes titled *A Programmer's Introduction to Predicate Logic*.

Midway through the second offering of the course, I discovered the book *Programming in the 1990s: An Introduction to the Calculation of Programs* by Edward Cohen (Springer-Verlag, 1990). I liked the presentation style and selection of material in the Cohen book and adopted it as the textbook for the Fall 1992 offering of CSCI 550 and for subsequent offerings in Spring 1994, Spring 1995, Fall 1996, and Spring 1998. I taught mostly from my predicate logic handout and my handwritten lecture notes based on Cohen's book and on adaptations of material I had used previously. I also adapted some material from Anne Kaldewaij's book *Programming: The Derivation of Algorithms* (Prentice-Hall, 1990) and Rob Hoogerwoord's 1989 doctoral dissertation *The Design of Functional Programs: A Calculational Approach* at the Technical University at Eindhoven, the Netherlands.

After the Spring 1998 offering, I did not teach the course for eight years because I focused my attention in other directions. However, I decided to revive the course and offer it in Spring 2006. To my dismay, I discovered that the Cohen textbook had gone out of print. Thus I decided to adopt the classic Gries' textbook again. I also set out to produce my own set of handouts based on my handwritten lecture notes from previous offerings. With considerable assistance from Jian Weng, I created this set of notes.

Acknowledgments

I thank Ms. Jian Weng, my teaching assistant and PhD student, who aided me enormously in the preparation and editing of these notes. She learned LaTeX and created the initial versions of the handouts from photocopies of my sloppy and tersely written personal lecture notes. She also drew the figures in the document. After the semester, she helped me integrate my expanded versions of the various handouts into a single document.

I also thank the students in the Spring 2006 CSCI 550 class, who did not complain too much about the many typos and other errors in the versions of the notes I handed out in class. Their error corrections and comments (particularly those of Chuck Jenkins) assisted me in improving the notes.

In the 1990s my course used Ed Cohen's textbook, so my selection of material and my presentation is similar to that book in several places.

Conrad Cunningham
Oxford, Mississippi, USA
August 2006

Contents

1 Specification

1.1 Hoare Triples

A *Hoare triple* is a boolean expression

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

where Q is the *precondition* predicate, R is the *postcondition* predicate, and S is the *program*.

The Hoare triple notation has higher binding power than the function application operator ".".

$\{Q\}\ S\ \{R\}$ is a *predicate* (actually just a proposition) meaning:

> if execution of S is begun in any state satisfying Q, then S is guaranteed to terminate in a state satisfying R,

Note: This says nothing about executions begun in states not satisfying Q!

This is called *total correctness* because it involves both *termination* and establishing the correct final state. A program is *partially correct* if it establishes the final state but might have the possibility of not terminating.

The Hoare triple above gives the *functional specification* (or just *specification*) of program S.

1.2 Equations

To denote equations, we use the notation

$$u : f.u$$

where u is a list of unknown variables and $f.u$ is a boolean expression.

A *solution* of the equation is a set of values for the unknowns u that make the expression $f.u$ evaluate to $true$.

1.3 Programming

Programming is the process of solving an equation of the form $\{Q\}\ S\ \{R\}$ for S.

In the equational notation, the above Hoare triple would be:

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

A solution of this equation is some program S that makes the Hoare triple true.

1.4 Unknowns

Consider the specification:

$$\{Q\}\ S\ \{x = y\}$$

In the above, program S could consist of any of the following assignments:

- $x \leftarrow y$
- $y \leftarrow x$
- $x \leftarrow v$ and $y \leftarrow v$ for any value v

We can use equation notation in postcondition, e.g.,

$$\{Q\}\ S\ \{x : x = y\}$$

to mean that S can only change the value of x to establish $x\ =\ y$.

Or, alternatively, we can use the following shorthand notation:

$$\{Q\}\ x : x = y$$

1.5 Declarations

1.5.1 Variables

Variable declarations begin with the keyword **var**.

Note the use of bold font for keywords. In handwritten text, keywords are written underlined. For example, var.

var $x, y : int$
var $b : bool$
var $x, y : int\ \{Q\}\ ;\ x : x = y$

1.5.2 Arrays

The declaration

var $b[p\ ..\ q)$: **array of** int

denotes an array b with indices p, $p+1, \cdots, q-1$ if $p \leq q$.

Note that the "[" means that the lower bound p is included in the range and the ")" means that the upper bound q is not included .

The *length* of b is $q - p$ if $p \leq q$ and 0, otherwise.

Similarly, $b[p\ \cdots\ q]$ denotes an array in which both bounds are included in the index range, $b(p\ \cdots\ q)$ denotes an array in which neither bound is included, and $b(p\ \cdots\ q]$ denotes an array in which only the upper bound is included.

Array references use function application notation. For example, $b.1$ refers to the value of te array element at index 1 of b.

1.5.3 Scope of variables and program blocks

A *program block* is a program fragment that begins with |[and ends with]|. Variables declared at the beginning of the block are known throughout the block but not outside it. That is, the block is the *scope* of the variables declared there.

For example,

|[**var** x : int ; x known here]|

Blocks may be nested. The variables declared in the outer block are known in the inner block. However, the variables declared in the inner block are not known in the inner block.

Fresh variable names should be chosen to avoid confusion.

1.5.4 Constant declarations

A *constant* is a "variable" whose values cannot be changed in the block. For example,

con N : int value cannot be changed

1.6 Example Specifications

1. Set x to value 7.

   $|[\ \textbf{var}\ x : int\ \{true\}\ ;\ x : x = 7\ ]|$

   The predicate *true* is satisfied by all states. Hence, it is often omitted from specifications.

2. Set z to the product of natural numbers x and y.

   $|[\ \textbf{var}\ x, y, z : int\ \{x \geq 0 \wedge y \geq 0\}\ ;\ z : z = x * y\ ]|$

3. Set x to the smaller of the values of x and y.

   $|[\ \textbf{var}\ x : int\ \{x = X \wedge y = Y\}\ ;\ x : x = X\ \textbf{min}\ Y\ ]|$

   or

   $|[\ \textbf{var}\ x : int\ \{x = X \wedge y = Y\}\ ;\ x : x \leq X \wedge y \leq Y \wedge (x = X \vee x = Y)\ ]|$

   In the above X and Y denote *specification constants*. They denote the initial values of the variables x and y, respectively. They can be used in the specification, but cannot be accessed in the program.

4. Reverse the order of the elements in array $a[0..N)$ for $N \geq 0$.

   $|[\ \textbf{con}\ N : int\ \{N \geq 0\}\ ;\ \textbf{var}\ a[0 \cdots N) : \textbf{array of}\ int\ \{a = A\}$
   $a : (\forall\ i : 0 \leq i < N : a.i = A.(N - 1 - i))\ ]|$

   The precondition is the conjunction of the listed preconditions: $N \geq 0 \wedge a = A$

1.7 Array and Sequence Terminology

If $f : [p..q) \rightarrow\ int$ is a total function:

$$
\begin{array}{ll}
f \text{ ascending} & \equiv (\forall\ i, j\ :\ p \leq i < j < q\ :\ f.i \leq f.j) \\
f \text{ increasing} & \equiv (\forall\ i, j\ :\ p \leq i < j < q\ :\ f.i < f.j) \\
f \text{ descending} & \equiv (\forall\ i, j\ :\ p \leq i < j < q\ :\ f.i \geq f.j) \\
f \text{ decreasing} & \equiv (\forall\ i, j\ :\ p \leq i < j < q\ :\ f.i > f.j) \\
f \text{ monotonic} & \equiv f \text{ ascending} \vee f \text{ descending} \\
f \text{ constant} & \equiv f \text{ ascending} \wedge f \text{ descending (i.e., all values are equal)}
\end{array}
$$

Let $a[p \cdots q)$ be an array:

- $a[r..s)$ is an *array section* for $p \leq r \leq s \leq q$.

  This is sometimes called an *array segment*.

  The *length* of the section is $s - r$.

- A *plateau* is an array section of equal values.

1.8 More Example Specifications

5. Sort array b into ascending order

   $|[$ **var** $b[0..N)$: **array of** int $\{N \geq 0 \wedge bag.b = B\}$

   $;\ b : bag.b = B \wedge (\forall\, i, j : 0 \leq i < j < N : b.i \leq b.j)\ ]|$

   $bag.b$ denotes the unordered collection of elements of b, with duplicate values allowed. This is variously called a *bag*, a *bunch*, or a *multiset*.

   An alternative formalization of this program specification might be the following:

   $|[$ **var** $b[0..N)$: **array of** int $\{N \geq 0 \wedge\ b = B\}$

   $;\ b : perm.B.b \wedge (\forall\, i, j : 0 \leq i < j < N : b.i \leq b.j)\ ]|$

   Here, $perm.B.b$ asserts that b is a permutation of B.

6. Set z to the length of the longest plateau in integer array b.

   $|[$ **var** $b[0..N)$: **array of** int $\{N \geq 0\}$ $;$ **var** $z : int$

   $;\ z : z = (\mathbf{MAX}\ x, y : 0 \leq x \leq y \leq N\ \wedge\ f.x.y : y - x)\ ]|$

   where $f.x.y \equiv (\forall\, i, j : x \leq i < y \wedge x \leq j < y : b.i = b.j)$

   f is a predicate (boolean function) which is true when $b[x \cdots y)$ is a plateau.

   Introduction of f made the specification easier to read and understand.

2 Guarded Commands

2.1 Guarded Commands Notation

In these notes we use Dijkstra's Guarded Commands notation to express our programs. The notation has the following commands (i.e., statements).

skip: The *skip* command is a "no operation" command. It always terminates, but it does not change the state.

abort: The *abort* command does not terminate or change the state. (It represents a "bug" in the program.)

We seldom will insert *abort* explicitly into our programs, but sometimes we may have other combinations of commands that are equivalent to this command.

havoc: The *havoc* command terminates and changes the state in an arbitrary way. (This might represent a programming "disaster".)

For our purposes, this is mainly a command included for completeness of the theory. We will not explicitly use this command.

assignment $x := E$: The command

$$x := E$$

evaluates the expression E in the state that exists before the execution of the command, then it changes the value of variable x to be the value determined for E.

The assignment operator " := " should be read "becomes".

A *multiple assignment* command

$$x, y := E, F$$

evaluates expressions E and F in the state before execution of the command, and then simultaneously sets the values of variables x and y to the values determined for E and F, respectively.

composition: The command composition operator " ; " composes arbitrary commands in sequence. For example, if S and T are arbitrary commands, then

$$S \ ; \ T$$

composes them into one command. First, S is executed, then T. The state is changed by first modifying it as S modifies it and then as T modifies it.

alternation (or alternative): The alternation command has the following syntax:

$$
\begin{array}{lll}
\textbf{if} & B.0 & \rightarrow S.0 \\
[] & B.1 & \rightarrow S.1 \\
 & & \vdots \\
[] & B.(n-1) & \rightarrow S.(n-1) \\
\textbf{fi} & &
\end{array}
$$

where the $B.i$ (for all i) denote boolean expressions called *guards* and the $S.i$ are arbitrary commands. The $B.i \rightarrow S.i$ construct is called a *guarded command*.

The alternation command executes as follows. All the guards are evaluated in the state preceding the alternation command. Then a guard $B.i$ that evaluates to *true* is chosen for execution and the corresponding guarded command $S.i$ is then executed.

If more than one guard evaluates to *true*, then one of them is chosen *nondeterministically*.

If none of the guards evaluate to *true*, then the command is equivalent to an *abort* command.

Note: By *nondeterministic*, we mean that the choice is totally unpredictable to the programmer. The implementation might chose to evaluate them in any order it wishes. Different implementations or different runs of the program might chose different orders. On a parallel machine, the guards might be evaluated in parallel, with the first *true* guard to complete evaluation being the one chosen for execution. The choice is not necessarily *fair*; it might always chose the same one. Neither is the choice necessarily random.

The Guarded Commands **if** command is a generalization of the **if-then-else** statement found in other languages.

repetition (iteration): The repetition command has the following syntax:

$$
\begin{array}{lll}
\textbf{do} & B.0 & \rightarrow S.0 \\
[] & B.1 & \rightarrow S.1 \\
\vdots & & \\
[] & B.(n-1) & \rightarrow S.(n-1) \\
\textbf{od} & &
\end{array}
$$

The repetition command executes as follows. All the guards are evaluated as in the alternation command. Then a guard $B.i$ that evaluates to *true* is chosen nondeterministically for execution and the corresponding guarded command $S.i$ is executed. Once the guarded command has completed execution, the guards

are reevaluated in the resulting state and a *true* guard is then chosen as before, and so forth.

This process continues until all guards evaluate to *false*. At this point, execution continues with the command after the repetition command. That is, the program exits from the loop.

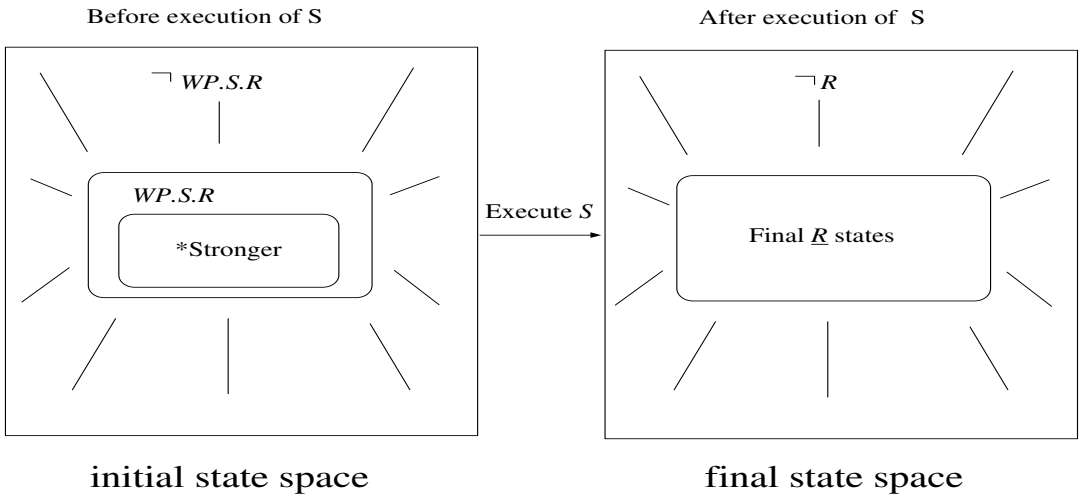
The Guarded Commands **do** command is a generalization of the **while** loop construct found in other languages.

Those are all the control constructs for our simple programming notation for now. We can also introduce procedure calls into the language.

2.2 Weakest Precondition Semantics

For a *terminating* command S and a required postcondition R, $wp.S.R$ symbolizes the *weakest precondition* such that execution of S is certain to establish postcondition R.

This might be pictured as shown in the following figures, where the boxes and ovals represent sets of possible states.



In the above diagram,

- If S is executed from any state satisfying $wp.S.R$, it is guaranteed to terminate in a state satisfying R.

  A *stronger* precondition characterizes a subset of the initial states that will lead to final states satisfying R

- Execution of S from a state in $\neg wp.S.R$ might abort, might establish $\neg R$, or might only sometimes establish R (because of nondeterminism).

- $wp.S$ is sometimes called a *predicate transformer* because it is a function from Predicates to Predicates.

Think about $wp.S.R$ informally in the following situations:

1. $S : i := i + 1$
   $R : i \leq 1$
   $wp.S.R \equiv i \leq 0$

2. S : **if** $x \geq y \rightarrow z := x$
   $[\!]$ $x \leq y \rightarrow z := y$
   fi

   (a) $R : z = x$ **max** y
   $wp.S.R \equiv true$

   That is, command S always establishes $z = x$ **max** y.

   (b) $R' : z = y$
   $wp.S.R' \equiv y \geq x$

   That is, if $y < x$ initially, then $z \neq y$ afterward.

   (c) $R'' : z = y - 1$
   $wp.S.R'' \equiv false$ (i.e., no states)

   That is, execution of S can never establish R''.

   (d) $R''' : z = y + 1$
   $wp.S.R'' \equiv x = y + 1$

3. For any S, $wp.S.true$ denotes the set of all states from which execution of S will terminate.

2.3 Formal Properties of Weakest Preconditions

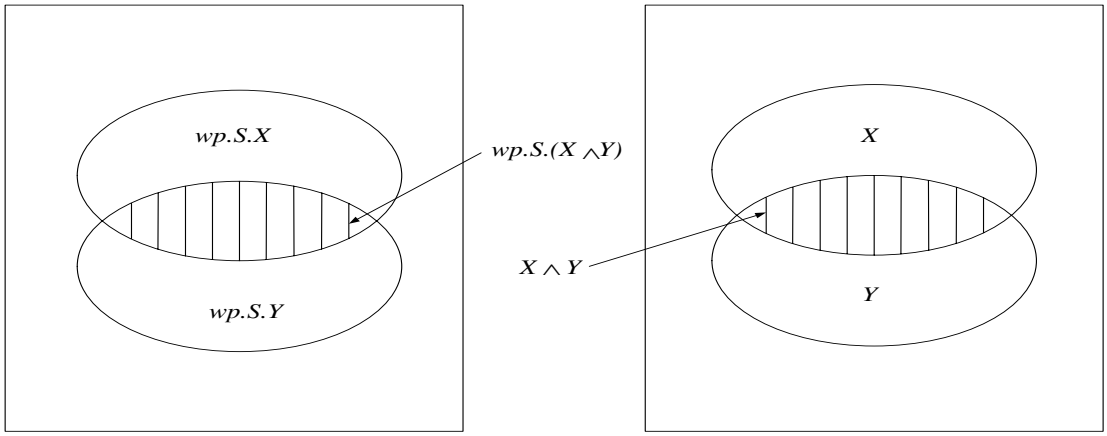
Axiom 1 (Law of the Excluded Miracle)
$wp.S.false \equiv false$

That is, there are no states in which execution of S can begin and guarantee to terminate in a state satisfying $false$

Note: This property is controversial (and not particularly useful). It has been removed from more recent theoretical treatments (Greg Nelson, TOPLAS, Oct 1989).

Axiom 2 (Conjunctivity (distribution of $\wedge$ over $wp.S$))
$wp.S.(X \wedge Y) \equiv wp.S.X \wedge wp.S.Y$

Informally, the weakest precondition of the intersection of two state spaces is the intersection of the weakest preconditions of the two state spaces. This is illustrated in the following diagram.



Theorem 1 (Monotonicity (wp preserves $\Rightarrow$))
$(X \Rightarrow Y) \Rightarrow (wp.S.X \Rightarrow wp.S.Y)$

Proof:

$$
\begin{array}{ll}
 & X \Rightarrow Y \\
\equiv & \quad \langle \ \Rightarrow \text{ to } \wedge \text{ connection } \ \rangle \\
 & X \wedge Y \equiv X
\end{array}
$$

Assume $X \wedge Y \equiv X$, show $wp.S.X \Rightarrow wp.S.Y$.

$$
\begin{array}{ll}
 & wp.S.X \\
\equiv & \quad \langle \text{ assumption } \rangle \\
 & wp.S.(X \wedge Y) \\
\equiv & \quad \langle \text{ conjunctivity } \rangle \\
 & wp.S.X \wedge wp.S.Y \\
\Rightarrow & \quad \langle \text{ conjunct simplification } \rangle \\
 & wp.S.Y
\end{array}
$$

Theorem 2 (Disjunctivity)
$(wp.S.X \vee wp.S.Y) \;\Rightarrow\; wp.S.(X \vee Y)$

Note that this theorem uses $\Rightarrow$ because S may be *nondeterministic*, i.e., for some initial state, more than one final state is possible.

Example: Consider a coin flipping operation `flip`. When a coin is flipped, it may end up in either state *head* or state *tail*.

$$
\begin{array}{l}
wp.\texttt{flip}.head \equiv false \\
wp.\texttt{flip}.tail \;\equiv false
\end{array}
$$

That is, the result of any flip may be *head* or *tail*.

However,

$$wp.\texttt{flip}.(head \vee tail) \equiv true.$$

Proof of Disjunctivity of wp: First, prove the lemma $wp.S.X \Rightarrow wp.S.(X \vee Y)$.

$$
\begin{array}{ll}
 & wp.S.X \Rightarrow wp.S.(X \vee Y) \\
\Leftarrow & \quad \langle \text{ monotonicity, } \Leftarrow \Rightarrow \text{ connection } \rangle \\
 & X \Rightarrow X \vee Y \\
\equiv & \quad \langle \text{ disjunct addition } \rangle \\
 & true
\end{array}
$$

Aside on proof strategy: This is a new proof strategy for $P \equiv true$. The above argument proves $(true \Rightarrow P) \equiv true$ (i.e. is valid). Substituting using the true antecedent rule $(true \Rightarrow P \equiv P)$, we get $P \equiv true$.

By a similar argument, the lemma $wp.S.Y \Rightarrow wp.S.(X \vee Y)$ holds.

Now, using these lemmas, we can prove disjunctivity.

$$
\begin{array}{ll}
 & wp.S.X \vee wp.S.Y \Rightarrow wp.S.(X \vee Y) \\
\equiv & \quad \langle\ \neg \text{ connection, de Morgan } \rangle \\
 & (\neg wp.S.X \wedge \neg wp.S.Y) \vee wp.S.(X \vee Y) \\
\equiv & \quad \langle\ \vee \text{ distribution over } \wedge\ \rangle \\
 & (\neg wp.S.X \vee wp.S.(X \vee Y)) \wedge (\neg wp.S.Y \vee wp.S.(X \vee Y)) \\
\equiv & \quad \langle\ \Rightarrow \neg \text{ connection } \rangle \\
 & (wp.S.X \Rightarrow wp.S.(X \vee Y)) \wedge (wp.S.Y \Rightarrow wp.S.(X \vee Y)) \\
\equiv & \quad \langle\ \text{ above lemmas, } \wedge \text{ identity } \rangle \\
 & true
\end{array}
$$

Now we can state the correctness of our programs in terms of predicate calculus, given the definitions of our programming language statements.

Axiom 3 (Hoare Triple)
$\{Q\}\ S\ \{R\}\ \equiv\ (Q \Rightarrow wp.S.R)$

Theorem 3 $\{wp.S.R\}\ S\ \{R\}$

We take advantage of this theorem in the construction of proofs or derivation of programs.

The following is a useful theorem from predicate calculus.

Theorem 4 $(Q \Rightarrow R)\ \Rightarrow\ (P \wedge Q \Rightarrow P \wedge R)$

Proof: Assume $Q \Rightarrow R$.

$$
\begin{array}{ll}
 & P \wedge Q \\
\equiv & \quad \langle\ \wedge \text{ identity, assumption } \rangle \\
 & P \wedge Q \wedge (Q \Rightarrow R) \\
\equiv & \quad \langle\ \Rightarrow \text{ elimination } \rangle \\
 & P \wedge Q \wedge R \\
\Rightarrow & \quad \langle\ \text{ conjunct simplification } \rangle \\
 & P \wedge R
\end{array}
$$

Theorem 5 (Precondition Rule)
$(P \Rightarrow Q) \wedge \{Q\}\ S\ \{R\}\ \Rightarrow\ \{P\}\ S\ \{R\}$

Proof:

$$
\begin{array}{ll}
 & (P \Rightarrow Q) \wedge \{Q\}\ S\ \{R\} \\
\equiv & \quad \langle\ \text{Hoare triple}\ \rangle \\
 & (P \Rightarrow Q) \wedge (Q \Rightarrow wp.S.R) \\
\Rightarrow & \quad \langle\ \text{transitivity}\ \rangle \\
 & P \Rightarrow wp.S.R \\
\equiv & \quad \langle\ \text{Hoare triple}\ \rangle \\
 & \{P\}\ S\ \{R\}
\end{array}
$$

Theorem 6 (Postcondition Rule)
$\{Q\}\ S\ \{R\} \wedge (R \Rightarrow A)\ \Rightarrow\ \{Q\}\ S\ \{A\}$

Proof:

$$
\begin{array}{ll}
 & \{Q\}\ S\ \{R\} \wedge (R \Rightarrow A) \\
\equiv & \quad \langle\ \text{Hoare triple}\ \rangle \\
 & (Q \Rightarrow wp.S.R) \wedge (R \Rightarrow A) \\
\Rightarrow & \quad \langle\ \text{theorem on previous page, monotonicity of } wp\ \rangle \\
 & (Q \Rightarrow wp.S.R) \wedge (wp.S.R \Rightarrow wp.S.A) \\
\Rightarrow & \quad \langle\ \text{transitivity}\ \rangle \\
 & Q \Rightarrow wp.S.A \\
\equiv & \quad \langle\ \text{Hoare triple}\ \rangle \\
 & \{Q\}\ S\ \{A\}
\end{array}
$$

Theorem 7 (Program Equivalence)
$S = T\ \equiv\ (\forall\ R :: wp.S.R \equiv wp.T.R)$

This is a higher order predicate! It involves quantification over predicates.

2.4 Semantics of Guarded Commands

2.4.1 Skip command

Axiom 4 (Definition of skip)
$wp.skip.R \equiv R$

There is no state change.

In terms of the Hoare triple:

$$\{Q\}\ skip\ \{R\} \equiv Q \Rightarrow R$$

2.4.2 Abort command

Axiom 5 (Definition of abort)
$wp.abort.R \equiv false$

This command never terminates.

In terms of the Hoare triple:

$$\{Q\}\ abort\ \{R\} \equiv Q \equiv false$$

2.4.3 Composition command

Axiom 6 (Definition of composition)
$wp.(S\,;T).R \equiv wp.S.(wp.T.R)$

In terms of the Hoare triple:

$$\{Q\}\ S\,;T\ \{R\} \Leftarrow \{Q\}\ S\ \{H\} \wedge \{H\}\ T\ \{R\} \quad \text{(for some } H\text{)}$$

In the above, H is a predicate that holds in the intermediate state.

2.4.4 Assignment command

Axiom 7 (Definition of assignment (single assignment, simple variable))
$wp.(x := E).R \equiv def.E \wedge R^x_E$

- $def.E$ means "E is defined" or "E can be evaluated".
- $def.E$ is often omitted if obvious. However, always keep it in mind.

- Arrays on left of := (e.g., $b.i := E$) must be handled differently. This is discussed in Section 9.

$def.(x \textbf{ div } y) \equiv y \neq 0$ // i.e., **div** not defined for $y = 0$

$def.(b.i) \equiv 0 \leq i < N$ for some array $b[0..N)$ // i.e., subscript must be in range

We cannot define $def.E$ formally since we have not defined allowable expressions E formally. Thus, we use it informally and omit it from the definition if obviously true.

Remember that R^x_E can also be written as $R(x := E)$ and means the textual replacement of variable x by expression E in expression R.

Ex: Suppose $R \equiv x + y \leq z$

$$\begin{array}{lcl} R^x_{x+1} & = & (x+1) + y \leq z \\ R^y_{y-x} & = & x + (y - x) \leq z \equiv y \leq z \\ R^z_4 & = & x + y \leq 4 \end{array}$$

Axiom 8 (Definition of multiple assignment)
$wp.(x, y := E_1, E_2).R \equiv def.E_1 \wedge def.E_2 \wedge R^{x,y}_{E_1,E_2}$

- $def.E_1 \wedge def.E_2$ are often omitted when it is obvious that they hold.
- Remember that $R^{x,y}_{E_1,E_2}$ is simultaneous substitution for x and y.

Ex: Again suppose $R \equiv x + y \leq z$.

$$\begin{array}{lcl} R^{x,y}_{y,x} & = & y + x \leq z \\ R^{x,z}_{z,y} & = & z + y \leq y \equiv z \leq 0 \\ R^{x,y}_{x+y,-x} & = & (x + y) + (-x) \leq z \equiv y \leq z \end{array}$$

Ex:

$$\begin{array}{ll} & wp.(x, y := 7, x + y).(x = y) \\ \equiv & def.7 \wedge def.(x + y) \wedge (x = y)^{x,y}_{7,x+y} \\ \equiv & (7 = x + y) \end{array}$$

In the above, $def.7 \wedge def.(x + y)$ is trivial and is usually omitted.

Ex: $R \equiv i > 0 \wedge s = (\Sigma\, j : 0 \leq j < i : b.j)$ for $b[0..N)$

$$\begin{array}{ll} & wp.(s, i := s + b.i, i + 1).R \\ \equiv & def.(b.i) \wedge R^{s,i}_{s+b.i,i+1} \\ \equiv & 0 \leq i < N \wedge (i + 1) > 0 \wedge s + b.i = (\Sigma\, j : 0 \leq j < i + 1 : b.j) \\ \equiv & 0 \leq i < N \wedge (s + b.i = (\Sigma\, j : 0 \leq j < i : b.j) + b.i \\ \equiv & 0 \leq i < N \wedge s = (\Sigma\, j : 0 \leq j < i : b.j) \end{array}$$

Thus, in terms of the Hoare triple, we can restate the above axiom as follows:

$$\{Q\}\ x, y := E_1, E_2\ \{R\}\ \equiv\ (Q \Rightarrow def.E_1 \wedge def.E_2 \wedge R^{x,y}_{E_1,E_2})$$

2.4.5 Simple program proof

Let us prove a simple program meets its specification using the semantics we have defined so far. Suppose we have the following specification

$$|[\ \textbf{var}\ x, y, z : int\ \{Q\}\ ; x, y, z : R\ ]|$$

where

$$\begin{array}{ll} Q: & x = X \wedge y = Y \wedge z = Z \\ R: & x = Y \wedge y = X \end{array}$$

Suppose we have the following program:

$$\begin{array}{l} |[\ \ \textbf{var}\ x, y, z : int\ \{Q\}\ ; \\ \quad z := x\ ; \\ \quad x := y\ ; \\ \quad y := z\ ; \\ \quad \{R\} \\ ]| \end{array}$$

Then a *proof outline* can be constructed as follows:

$$\begin{array}{l} |[\ \ \textbf{var}\ x, y, z : int\ \{Q\}\ ; \\ \quad \{((R^y_z)^x_y)^z_x\} \qquad // \text{ i.e., } \{y = Y \wedge x = X\} \\ \quad z := x\ ; \\ \quad \{(R^y_z)^x_y\} \qquad // \text{ i.e., } \{y = Y \wedge z = X\} \\ \quad x := y\ ; \\ \quad \{R^y_z\} \qquad // \text{ i.e., } \{x = Y \wedge z = X\} \\ \quad y := z\ ; \\ \quad \{R\} \\ ]| \end{array}$$

Note, that in the above proof outline, we begin with the postcondition R and construct the weakest precondition of the sequence of assignment commands by using the axioms of assignment and composition. In the proof of this program, we end up with one complicated proof obligation at the point where this back substitution process meets

the precondition Q. We must apply the Precondition Rule and the predicate logic to prove this assertion.

Proof obligation:

$$Q \Rightarrow ((R^y_z)^x_y)^z_x$$

Proof:

$$\begin{array}{ll} & x = X \wedge\ y = Y \wedge\ z = Z \\ \Rightarrow & \langle\ \text{conjunct simplification}\ \rangle \\ & y = Y \wedge\ x = X \end{array}$$

2.4.6 Alternative (or alternation) command

For some finite natural number N, let IF represent

$$\begin{array}{ll} \textbf{if} & B.0 \rightarrow S.0 \\ [] & B.1 \rightarrow S.1 \\ & \vdots \\ [] & B.(N-1) \rightarrow S.(N-1) \\ \textbf{fi} & \end{array}$$

and let

$$BB = (\exists\ i : 0 \leq i < N : B.i)$$

Axiom 9 (Definition of Alternative Command)
$wp.\text{IF}.R \equiv def.BB \wedge BB \wedge (\forall\, i : 0 \leq i < N : B.i \Rightarrow wp.(S.i).R)$

If we make sure that the guards are total functions then the $def.BB$ conjunct can be ignored.

We can restate this axioms in terms of Hoare triples.

Theorem 8

$$\{Q\}\ \text{IF}\ \{R\} \equiv (Q \Rightarrow BB) \wedge (\forall\, i : 0 \leq i < N : \{Q \wedge B.i\}\ S.i\ \{R\})$$

Thus, for an IF, it is required that:

- In the initial state, there is a true guard.
- Each guarded command establishes the postcondition if its guard is true.

Note: Remember that the selection of a true guarded command for execution is nondeterministic.

2.4.7 Calculating the maximum

Suppose we have the following **if** command and postcondition.

$$
\begin{array}{ll}
S: & \textbf{if} \quad x \geq y \rightarrow m := x \\
& [\!] \quad x \leq y \rightarrow m := y \\
& \textbf{fi} \\
R: & (m = x \wedge x \geq y) \vee (m = y \wedge y \geq x)
\end{array}
$$

Now let us calculate the weakest precondition of S.

$$
\begin{array}{ll}
& wp.S.R \\
\equiv & \quad \langle \text{ definition of IF, expanding the quantifiers } \rangle \\
& (x \geq y \vee y \geq x) \wedge (x \geq y \Rightarrow R^m_x) \wedge (x \leq y \Rightarrow R^m_y) \\
\equiv & \quad \langle \text{ arithmetic, } \Rightarrow \neg \text{ connection, textual substitution } \rangle \\
& \mathit{true} \wedge (x < y \vee (x = x \wedge x \geq y) \vee (x = y \wedge y \geq x)) \\
& \wedge (x > y \vee \ (y = x \wedge x \geq y) \vee (y = y \wedge y \geq x)) \\
\equiv & \quad \langle \ \wedge \text{ identity, arithmetic } \rangle \\
& (x < y \vee x \geq y \vee (x = y \wedge y \geq x)) \\
& \wedge (x > y \vee (y = x \wedge x \geq y) \vee y \geq x) \\
\equiv & \quad \langle \text{ excluded middle, } \vee \text{ zero } \rangle \\
& \mathit{true}
\end{array}
$$

Hints on constructing a proof outline for IF:

- Avoid building full wp.IF.R. It tends to be long and complicated.
- Instead, construct it piecemeal. Work backward from the postcondition toward the guards. Also bring the precondition across the guard.

$$
\begin{array}{l}
|[\quad \textbf{var } x, y, m : \mathit{int} \ \{Q\}; \\
\quad \textbf{if } x \geq y \rightarrow \{Q \wedge x \geq y\} \ \{R^m_x\} \ m := x \ \{R\} \\
\quad [\!] \ x \leq y \rightarrow \{Q \wedge x \leq y\} \ \{R^m_y\} \ m := y \ \{R\} \\
\quad \textbf{fi} \\
\quad \{R\} \\
]|
\end{array}
$$

Proof Obligations:

1. $Q \Rightarrow x \geq y \vee x \leq y$
2. $Q \wedge x \geq y \Rightarrow R^m_x$
3. $Q \wedge x \leq y \Rightarrow R^m_y$

2.4.8 Constructing guards

As a preview of how we can use the semantics to construct programs, let us examine the following program and reason how the missing guards need to be defined.

$$
\begin{array}{ll}
|[& \textbf{var } x, y : int\ \{P\}; \\
& \textbf{if } B.0 \to \{P \wedge B.0\} \quad \{P^{x,y}_{y,x}\} \quad x, y := y, x \quad \{P\} \\
& [\!]\ B.1 \to \{P \wedge B.1\} \quad \{P^{x}_{x-y}\} \quad x := x - y \quad \{P\} \\
& \textbf{fi} \\
& \{P\} \\
]|
\end{array}
$$

Assuming $P : x > 0 \wedge y > 0$, find $B.0$ and $B.1$.

Requirements:

1. $P \Rightarrow B.0 \vee B.1$
2. $P \wedge B.0 \Rightarrow P^{x,y}_{y,x}$
3. $P \wedge B.1 \Rightarrow P^{x}_{x-y}$

Consider requirement 2. Assume P, find a $B.0$ such that $B.0 \Rightarrow P^{x,y}_{y,x}$.

$$
\begin{array}{ll}
 & y > 0 \wedge x > 0 \\
\equiv & \quad \langle \text{ assumption } \rangle \\
 & true \\
\Leftarrow & \quad \langle \text{ choose } B.0 \equiv true \ \rangle \\
 & B.0
\end{array}
$$

We could choose anything stronger than *true* for $B.0$, hence, the $\Leftarrow$.

Consider requirement 3. Assume P, find a $B.1$ such that $B.1 \Rightarrow P^{x}_{x-y}$.

$$
\begin{array}{ll}
 & x - y > 0 \wedge y > 0 \\
\equiv & \quad \langle \text{ assumption } \rangle \\
 & x - y > 0 \\
\equiv & \quad \langle \text{ arithmetic } \rangle \\
 & x > y \\
\Leftarrow & \quad \langle \text{ choose } B1 \equiv x > y \ \rangle \\
 & B.1
\end{array}
$$

We could choose a $B.1$ stronger then $x > y$.

Now consider requirement 1, which is $P \Rightarrow B.0 \vee B.1$. Above we determined $B.0$ can be *true* and $B.1$ can be $x > y$. Either can be strengthened. We could strengthen $B.0$ to $x \leq y$ and still meet requirement 2.

This shows how we can use semantics to calculate missing parts of programs.

2.4.9 Repetitive (or repetition) command

Let DO represent the following:

$$
\begin{array}{ll}
\textbf{do}\ B.0 & \rightarrow S.0 \\
[]\ \ B.1 & \rightarrow S.1 \\
\ \ \vdots & \\
[]\ \ B.(N-1) & \rightarrow S.(n-1) \\
\textbf{od} &
\end{array}
$$

In the above, we require that $N \geq 0$ and N is finite.

Let IF represent the same guarded commands with **do** $\cdots$ **od** replaced by **if** $\cdots$ **fi**.

Let $BB \equiv (\exists\, i : 0 \leq i < N : B.i)$.

Let $H.k.R$ (for $k \geq 0$) be the set of states from which execution of DO terminates in k or fewer iterations with postcondition R *true*.

Clearly, for the zero-iteration case:

$$H.0.R = \neg BB \ \wedge \ R$$

Here, $\neg BB$ represents loop termination.

Now consider $H.k.R$ for $k > 0$. There are two cases to examine, the loop:

- terminates after 0 iterations, which is just $H.0.R$.
- is executed at least once.

For the second case, in which the guarded command is executed at least one, the following guarded command is equivalent.

$$
\begin{array}{ll}
\textbf{if}\ B.0 & \rightarrow S.0 \\
[]\ B.1 & \rightarrow S.1 \\
\ \ \vdots & \\
[]\ B.(N-1) & \rightarrow S.(n-1) \\
\textbf{fi}\ \{H.(k-1).R\} & \\
;\ \text{DO}\ \{R\} &
\end{array}
$$

$H.0.R = \neg BB \wedge R$

Thus, for $k > 0$, $H.k.R = H.0.R \vee wp.IF.(H.(k-1).R)$. If we expand this recursively, we get the following definition.

Axiom 10 (Definition of Repetition)
$wp.\text{DO}.R \ \equiv \ (\exists \ k : k \geq 0 : H.k.R)$

The existential quantification means that the interation will terminate in some finite, bounded number of iterations.

This axiom (i.e., definition) is difficult to use. That is, it doesn't give any guidance on how to construct the loop.

Fortunately a stronger predicate than $wp.DO.R$ is usually sufficient for our purposes, the *loop invariant*.

The definition of the repetition in terms of wp is not useful and would serve only to delay us. From its definition, however, an exceedingly useful result can be proved.

Theorem 9 (Invariance)
$\{Q\}$ DO $\{R\}$ is true if there exists a predicate P (the *loop invariant*) and an integer function t (*variant or bound function*) such that the conjunction of the following holds: *(⇐)*

1. $(Q \ \Rightarrow \ P)$ – *initialization*
2. $(\forall \ i : 0 \leq i < N : \{P \ \wedge \ B.i\} \ S.i \ \{P\})$ – *invariance*
3. $(\forall \ i : 0 \leq i < N : \{P \ \wedge \ B.i \ \wedge \ t = T\} \ S.i \ \{t < T\})$ – *progress*
4. $P \wedge BB \ \Rightarrow t \geq 0$ – *boundedness*
5. $P \wedge \neg BB \Rightarrow R$ – *finalization*

In the above, the two $\forall$ expressions sometimes are written as one assertion.

Note: Some writers may use the formulation:

$$P \wedge t < 0 \ \Rightarrow \ \neg BB$$

which is equivalent by shunting, assuming t is a total function.

Heuristics:

- Choosing an invariant P: find the unchanging relationships among the loop's variables that hold at each loop test
- Choosing t: find an integer-valued function that decreases on each iteration and is bounded below by 0

Ex: Given $P \equiv x > 0 \wedge y > 0$. Prove that P is an invariant of the following program:

$|[$ **var** $x, y : int\ \{P\};$
do $x < y \rightarrow x, y := y, x$
$[]\ \ x > y \rightarrow x := x - y$
od
$\{P\ \wedge\ x = y\}$
$]|$

If we construct the proof outline, we get the following annotated program:

$|[$ **var** $x, y : int\{P\};$
do $x < y \rightarrow \{P\ \wedge\ x < y\}$
$\{P^{x,y}_{y,x}\}\ x, y := y, x\ \{P\}$
$[]\ \ x > y \rightarrow \{P\ \wedge\ x > y\}$
$\{P^{x}_{x-y}\}\ x := x - y\ \{P\}$
od $\{P \wedge \neg(x < y \vee x > y)\}$
$\{P \wedge x = y\}$
$]|$

Proof Obligations for invariance:

1. $P \Rightarrow P$.

2. $P \wedge x < y \Rightarrow P^{x,y}_{y,x}$

$$\begin{aligned} & x > 0 \wedge y > 0 \wedge x < y \\ \Rightarrow\ & y > 0 \wedge x > 0 \end{aligned}$$

3. $P \wedge x > y \Rightarrow P^{x}_{x-y}$

$$\begin{aligned} & x > 0 \wedge y > 0 \wedge x > y \\ \Rightarrow\ & x - y > 0 \wedge y > 0 \\ \equiv\ & x > y\ \wedge\ y > 0 \end{aligned}$$

4. $P \wedge \neg(x < y \vee x > y) \Rightarrow P \wedge x = y$

Now consider boundedness and progress. For the bound function, choose function $t : x + 2 * y$. Why?

- Initially $t > 0$ because of the precondition P.
- The first guarded command exchanges x and y when $x < y$.
  - Thus give heavier weight to y relative to x.
  - Thus use a factor of 2 on y.
- The second guarded command decreases x when $x > y$.

Proof Obligations for progress and boundedness:

5. $P \ \wedge \ (x < y \ \vee \ x > y) \ \Rightarrow \ t \geq 0$

   $x > 0 \ \wedge \ y > 0 \ \wedge \ x \neq y \ \Rightarrow \ x + 2 * y \geq 0$

   Assume $x > 0 \ \wedge \ y > 0 \ \wedge \ x \neq y$

   $$\begin{array}{ll} & x + 2 * y \geq 0 \\ \equiv & \quad \langle \ x > y \ \wedge \ y > 0, \text{ arithmetic } \rangle \\ & \mathit{true} \end{array}$$

6. $P \ \wedge \ x < y \ \wedge \ t = T \ \Rightarrow \ (t < T)^{x,y}_{y,x}$

   $x > 0 \ \wedge \ y > 0 \ \wedge \ x < y \ \wedge \ x + 2 * y = T \ \Rightarrow \ y + 2 * x < T$

   Assume $x > 0 \ \wedge \ y > 0 \ \wedge \ x < y$

   $$\begin{array}{ll} & x + 2 * y = T \\ \equiv & \quad \langle \ \text{arithmetic } \rangle \\ & 2 * x + y = T + x - y \\ \Rightarrow & \quad \langle \ \text{assumption } \rangle \\ & 2 * x + y < T \end{array}$$

7. $P \wedge x > y \wedge t = T \ \Rightarrow \ (t < T)^{x}_{x-y}$

   $x > 0 \wedge y > 0 \wedge x > y \wedge x + 2 * y = T \ \Rightarrow \ x - y + 2 * y < T$

   $x > 0 \wedge y > 0 \wedge x > y \wedge x + 2 * y = T \ \Rightarrow \ x + y < T$

   Assume $x > 0 \ \wedge \ y > 0 \ \wedge \ x > y$

   $$\begin{array}{ll} & x + 2 * y = T \\ \equiv & \quad \langle \ \text{arithmetic } \rangle \\ & x + y = T - y \\ \Rightarrow & \quad \langle \ \text{assumption } \rangle \\ & x + y < T \end{array}$$

2.4.10 Array assignment

We will consider assignments to array variables in Section 9.

2.4.11 Inner blocks

Consider blocks that are nested within other blocks.

Axiom 11 (Inner Block)
$wp.(|[\ \textbf{var}\ y : int; S]|).R \equiv (\forall\, y : y \in int : wp.S.R)$

That is, let the inner block variables take on any arbitrary value of the proper type.

Theorem 10
$\{Q\}|[\ \textbf{var}\ y : int; S]|\{R\} \equiv \{Q\}\ S\ \{R\}$

Note that $\{Q\}S\{R\}$ is an assertion over the outer state space extended by integer variable y.

3 Program Proofs

3.1 Example: Sum an Array

Let $R : s = (\Sigma\, i : 0 \leq i < N : b.i)$. We can specify a program to sum an array of integers as follows:

$|[$ **var** $s, n : int$; **var** $b[0..N)$: **array of** $int\{N \geq 0\}$;
$\quad \{s, n : R\}$
$]|$

Suppose we are given the following program and asked to prove that it satisfies the specification.

$|[$ **var** $s, n : int$; **var** $b[0..N)$: **array of** $int\{N \geq 0\}$;
$\quad s, n := 0, 0;$
$\quad$ **do** $\quad n \neq N \rightarrow$
$\quad\quad s, n := s + b.n, n + 1$
$\quad$ **od**
$\quad \{s, n : R\}$
$]|$

Using the semantic rules, we can write a proof outline for the program as follows, where P and t are an appropriate loop invariant and bound function for the loop.

$|[$ **var** $s, n : int$; **var** $b[0..N)$: **array of** $int\{N \geq 0\}$;
$\quad \{P^{s,n}_{0,0}\}$
$\quad s, n := 0, 0;$
$\quad \{$loop invariant P, bound function $t\}$
$\quad$ **do** $\quad n \neq N \rightarrow \{P \ \wedge \ n \neq N \ \wedge \ t = T\}$
$\quad\quad \{(P \ \wedge \ t < T)^{s,n}_{s+b.n,n+1}\}$
$\quad\quad s, n := s + b.n, n + 1$
$\quad\quad \{P \ \wedge \ t < T\}$
$\quad$ **od** $\{P \wedge \neg(n \neq N)\}$
$\quad \{s, n : R\}$
$]|$

We thus have the following **Proof Obligations**.

1. Initialization. $N \geq 0 \Rightarrow P^{s,n}_{0,0}$
2. Finalization. $P \wedge \neg(n \neq N) \Rightarrow R$
3. Invariance. $P \wedge n \neq N \Rightarrow P^{s,n}_{s+b.n,n+1}$
4. Boundedness. $P \wedge n \neq N \Rightarrow t \geq 0$
5. Progress. $P \wedge n \neq N \wedge t = T \Rightarrow (t < T)^{s,n}_{s+b.n,n+1}$

3.2 Heuristic for Finding Loop Invariant

1. *Start with conjuncts that express the bounds on the loop's control variables.*

   In the example, n is the loop control variable. Note that the loop control variable is initially zero, is incremented by one for each iteration, and is at the size of the array when the loop exits. Thus, we speculate that we need the conjunct

   $$0 \leq n \leq N$$

   as a part of the invariant.

2. *Then add conjuncts that express the relationships among the other variables involved in the loop.* Often the loop invariant is similar in structure to the loop's postcondition. Remember that:

   - the invariant must hold whenever control reaches the guards on any iteration
   - the invariant and the negation of the disjunction of the guards (i.e., $\neg BB$) must imply the postcondition.

   In the example we have program variables n and s, array b that is not changed, and constant N. So, let's track the changes to the two variables when control is at the loop test.

After iteration	n	s
0	0	0
1	1	$b.0$
2	2	$b.0 + b.1$
3	3	$b.0 + b.1 + b.2$

   So we speculate that we need the conjunct

   $$s = (\Sigma i : 0 \leq i < N : b.i)$$

   as part of the invariant. Note that this conjunct is similar in structure to the postcondition R.

   At termination $n = N$. Substituting this in the invariant we get

   $$0 \leq N \leq N \wedge s = (\Sigma i : 0 \leq i < N : b.i)$$

   which implies R. This is proof obligation 2.

3. *Check that the loop invariant holds the first time control reaches the loop. If not, go back to step 2.*

   In the example, initially $s = 0 \ \wedge \ n = 0$. If we substitute these values into the invariant, we get

   $$0 \leq 0 \leq N \ \wedge \ 0 = (\Sigma i : 0 \leq i < 0 : b.i)$$

   which is equivalent to $N \geq 0$. This is proof obligation #1.

4. *Check that the loop invariant is preserved by the body of the loop. If not, go back to step 2.*

   Informally, in a pass through the loop body, the next element is added to s and the loop invariant is reestablished by incrementing n by 1. This is proof obligation 3.

In summary, the following seems to be an appropriate loop invariant P.

$$P : 0 \leq n \leq N \ \wedge \ s = (\Sigma i : 0 \leq i < n : b.i)$$

The heuristics we study in subsequent sections for the derivation of loops will give more insight into the construction of loop invariants for existing programs.

3.3 Heuristic for Finding a Bound Function

1. *Rewrite the termination condition (i.e., $\neg BB$) as an integer expression that is constantly equal to 0 (or some positive constant).* Sometimes the variable bounds or other conjuncts of the loop's invariant may be needed to do this.

   In the example, $\neg BB \equiv \neg(n \neq N) \equiv n = N \equiv N - n = 0$.

   The expression $N - n = 0$ is called a grounded termination condition.

2. *Weaken the relation generated above to form a new relation that is greater than or equal to 0 at initialization and after each iteration.* Often you may need to examine the commands that change the "control" variables of the loop. This new relation is called the *variant condition.*

   Note that $0 \leq n \leq N$ is the range conjunct of the invariant in the example. Also note that the body of the loop decreases the value of $N - n$ for each iteration. Thus we weaken the above equality to:

   $$N - n \geq 0$$

   This is proof obligation #4.

3. *Obtain the bound function directly from the variant condition.*

   In the example, we define:

   $$t : N - n$$

   Proof obligation #5 is to show that this value decreases for each iteration.

Typical forms of variant functions in terms of the bounds on the control variable and expressions involving the control variable include:

- UpperBound − IncreasingExpression
- DecreasingExpression − LowerBound
- DecreasingExpression − IncreasingExpression

3.4 Proofs for Array Summation

1. Initialization. Prove $N \geq 0 \Rightarrow 0 \leq 0 \leq N \wedge 0 = (\Sigma i : 0 \leq i < 0 : b.i)$.

   Assume $N \geq 0$.

   $0 \leq 0 \leq N \ \wedge \ 0 = (\Sigma i : 0 \leq i < 0 : b.i)$
   $\equiv$ $\langle$ arithmetic $\rangle$
   $N \geq 0$
   $\equiv$ $\langle$ $N \geq 0$ assumption $\rangle$
   true

2. Finalization. Prove $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge \neg(n \neq N) \Rightarrow s = (\Sigma\, i : 0 \leq i < N : b.i)$.

   $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n = N$
   $\Rightarrow$ $\langle$ Leibniz, conjunct simplification $\rangle$
   $s = (\Sigma i : 0 \leq i < N : b.i)$

3. Invariance. Prove $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N \Rightarrow 0 \leq n+1 \leq N \wedge s + b.n = (\Sigma i : 0 \leq i < n+1 : b.i)$

   Assume $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N$.

   $0 \leq n+1 \leq N \wedge s + b.n = (\Sigma i : 0 \leq i < n+1 : b.i)$
   $\equiv$ $\langle$ assumption $0 \leq n \leq N \wedge n \neq N$ $\rangle$
   $s + b.n = (\Sigma i : 0 \leq i < n+1 : b.i)$
   $\equiv$ $\langle$ range split, one point, $0 \leq n \leq N$ $\rangle$
   $s + b.n = (\Sigma i : 0 \leq i < n : b.i) + b.n$
   $\equiv$ $\langle$ arithmetic $\rangle$
   $s = (\Sigma i : 0 \leq i < n : b.i)$
   $\equiv$ $\langle$ assumption $\rangle$
   true

4. Boundedness. Prove $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N \Rightarrow N - n \geq 0$.

   Assume $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N$.

   $N - n \geq 0$
   $\equiv$ $\langle$ $n \leq N$ assumption, arithmetic $\rangle$
   true

5. Progress. Prove $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N \wedge N - n = T \Rightarrow N - (n+1) < T$

   Assume $0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i) \wedge n \neq N \wedge N - n = T$.

   $N - (n+1) < T$
   $\equiv$ $\langle$ arithmetic $\rangle$
   $N - n < T + 1$
   $\equiv$ $\langle$ $N - n = T$ assumption, arithmetic $\rangle$
   true

3.5 Example: Minimum of an Array

Consider the following program to find the minimum value in an array.

$$
\begin{array}{l}
|[\ \textbf{var}\ m, n : int\ ;\ \textbf{var}\ b[0..N) : \textbf{array of}\ int\{N > 0\}; \\
\quad\quad m, n := b.0, 1; \\
\quad\quad \textbf{do}\ n \neq N \rightarrow \\
\quad\quad\quad\quad m, n := b.n\ \textbf{min}\ m, n+1 \\
\quad\quad \textbf{od} \\
\quad\quad \{m, n : m = (\textbf{MIN}\ i : 0 \leq i < N : b.i)\} \\
]|
\end{array}
$$

We can construct a proof outline as follows:

$$
\begin{array}{l}
|[\ \textbf{var}\ m, n : int\ ;\ \textbf{var}\ b[0..N) : \textbf{array of} int\{N > 0\}; \\
\quad\quad \{P^{m,n}_{b.0,1}\} \\
\quad\quad m, n := b.0, 1; \\
\quad\quad \{\text{inv. } P, \text{ bound } t\} \\
\quad\quad \textbf{do}\ n \neq N \rightarrow \{P \wedge n \neq N \wedge t = T\} \\
\quad\quad\quad\quad \{(P \wedge t < T)^{m,n}_{b.n\textbf{min}m,n+1}\} \\
\quad\quad\quad\quad m, n := b.n\ \textbf{min}\ m, n+1 \\
\quad\quad\quad\quad \{P \wedge t < T\} \\
\quad\quad \textbf{od}\ \{P \wedge \neg(n \neq N)\} \\
\quad\quad \{m, n : m = (\textbf{MIN}\ i : 0 \leq i < N : b.i)\} \\
]|
\end{array}
$$

Proof Obligations

1. Initialization. $N > 0 \ \Rightarrow\ P^{m,n}_{b.0,1}$
2. Finalization. $P \wedge \neg(n \neq N) \Rightarrow\ m = (\textbf{MIN}\ i : 0 \leq i < N : b.i)$
3. Invariance. $P \wedge n \neq N \Rightarrow P^{m,n}_{b.n\textbf{min}m,n+1}$
4. Boundedness. $P \wedge n \neq N \Rightarrow t \geq 0$
5. Progress. $P \wedge n \neq N \wedge t = T \Rightarrow t < T$

The development of an appropriate invariant and bound function using similar techniques to those in the previous example can yield:

inv. $P : 1 \leq n \leq N\ \wedge\ m = (\textbf{MIN}\ i : 0 \leq i < n : b.i)$

$t : N - n$

3.6 Example: Integer Square Root

Consider the following program to compute the integer square root of an integer.

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\ ;\ \textbf{var}\ i, j, k : int\{N \geq 0\}\ ; \\
\qquad i, j, k := 0, 1, 1; \\
\qquad \textbf{do}\ k \leq N \rightarrow \\
\qquad\qquad i, j := i + 1, j + 2; \\
\qquad\qquad k := k + j \\
\qquad \textbf{od} \\
\qquad \{0 \leq i^2 \leq N < (i+1)^2\} \\
]|
\end{array}
$$

If we construct the proof outline, we get the following annotated program:

$$
\begin{array}{l}
|[\ \textbf{con}\ N : int\ ;\ \textbf{var}\ i, j, k : int\{N \geq 0\}\ ; \\
\qquad \{P^{i,i,k}_{0,1,1}\} \\
\qquad i, j, k := 0, 1, 1; \\
\qquad \{\text{inv. } P, \text{ bound } t\} \\
\qquad \textbf{do}\ k \leq N \rightarrow \{P \wedge k \leq N \wedge t = T\} \\
\qquad\qquad \{((P \wedge\ t < T)^k_{k+j})^{i,j}_{i+1,j+2}\} \\
\qquad\qquad i, j := i + 1, j + 2;\ \{(P \wedge t < T)^k_{k+j}\} \\
\qquad\qquad k := k + j \{P\ \wedge\ t < T\} \\
\qquad \textbf{od}\ \{P \wedge \neg(k \leq N)\} \\
\qquad \{0 \leq i^2 \leq N < (i+1)^2\} \\
]|
\end{array}
$$

Proof Obligations

1. Initialization. $N \geq 0\ \Rightarrow\ P^{i,j,k}_{0,1,1}$
2. Finalization. $P\ \wedge\ \neg(k \leq N)\ \Rightarrow\ 0 \leq i^2 \leq N < (i+1)^2$
3. Invariance. $P\ \wedge\ k \leq N\ \Rightarrow\ (P^k_{k+j})^{i,j}_{i+1,j+2}$
4. Boundedness. $P\ \wedge\ k \leq N\ \Rightarrow\ t \geq 0$
5. Progress. $P\ \wedge\ k \leq N\ \wedge\ t = T\ \Rightarrow\ ((t < T)^k_{k+j})^{i,j}_{i+1,j+2}$

Now let us develop the invariant P. The control variable is k. What is the range on k?

Clearly, $1 \leq k$. But what is the upper bound? Note that on exit from the loop that $k > N$. We will address this issue a bit later.

Let's trace the values of the variables as the loop executes to try to determine the unchanging relationship among the variables.

After iteration	i	j	k
0	0	1	1
1	1	3	4
2	2	5	9
3	3	7	16

Thus, let's choose the invariant $P : j = 2{*}i{+}1 \wedge k = (i{+}1)^2 \wedge$ range_of_control_variable

Now, on exit from the loop, $k = (i+1)^2 > N$, which means that $i^2 \leq N$ is an appropriate upper bound.

The lower bound of $1 \leq k$ determined above can be restated as $1 \leq (i+1)^2$. Noting that i is never negative, let's restate the invariant as follows:

$$P : 0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2$$

Thus the termination condition $P \wedge \neg BB$ is
$0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k > N$.

We can thus choose the bound function $t : N - i^2$

Alternatively, we could choose $N - i$ or $\sqrt{N} - i$ for t.

Proofs for Integer Square Root

1. Initialization. Prove $N \geq 0 \Rightarrow 0 \leq 0 \wedge 0^2 \leq N \wedge 1 = 2 * 0 + 1 \wedge 1 = (0+1)^2$.

   Assume $N \geq 0$.

   $0 \leq 0 \wedge 0^2 \leq N \wedge 1 = 2 * 0 + 1 \wedge 1 = (0+1)^2$
   $\equiv$ $\langle$ arithmetic $\rangle$
   $N \geq 0$
   $\equiv$ $\langle$ assumption $\rangle$
   true

2. Finalization. Prove $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge \neg(k \leq N) \Rightarrow 0 \leq i^2 \leq N < (i+1)^2$.

   Assume $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k > N$.

   $0 \leq i^2 \leq N < (i+1)^2$
   $\equiv$ $\langle$ $0 \leq i \wedge i^2 \leq N$ assumption, arithmetic $\rangle$
   $(i+1)^2 > N$
   $\equiv$ $\langle$ $k = (i+i)^2 \wedge k > N$ assumptions, Leibniz $\rangle$
   true

3. Invariance. Prove $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N \Rightarrow 0 \leq i + 1 \wedge (i+1)^2 \leq N \wedge j + 2 = 2 * (i+1) + 1 \wedge k + (j+2) = ((i+1)+1)^2$.

   Assume $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N$.

   $0 \leq i + 1 \wedge (i+1)^2 \leq N \wedge j + 2 = 2 * (i+1) + 1 \wedge k + (j+2) = ((i+1)+1)^2$
   $\equiv$ $\langle$ $0 \leq i$ assumption, Leibniz, arithmetic $\rangle$
   $(i+1)^2 \leq N \wedge (2 * i + 1) + 2 = 2 * i + 3 \wedge (i+1)^2 + (2 * i + 1) + 2 = ((i+1)+1)^2$
   $\equiv$ $\langle$ assumption, Leibniz, arithmetic $\rangle$
   $k \leq N \wedge i^2 + 4 * i + 4 = i^2 + 4 * i + 4$
   $\equiv$ $\langle$ assumption, arithmetic $\rangle$
   true

4. Boundedness. Prove $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N \Rightarrow N - i^2 \geq 0$

   Assume $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N$.

   $N - i^2 \geq 0$
   $\equiv$ $\langle$ $i^2 \leq N$, arithmetic $\rangle$
   true

5. Progress. Prove $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N \wedge N - i^2 = T \Rightarrow N - (i+1)^2 < T$.

   Assume $0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i+1)^2 \wedge k \leq N \wedge N - i^2 = T$.

   $N - (i+1)^2 < T$
   $\equiv$ $\langle$ arithmetic $\rangle$
   $N - i^2 - 2 * i - 1 < T$
   $\equiv$ $\langle$ $0 \leq i \wedge N - i^2 = T$ assumptions, arithmetic $\rangle$
   true

3.7 Example: Greatest Common Divisor

Note: This section needs additional explanation and proofreading.

Axiom 12 (Definition of Greatest Common Divisor (gcd))
For integers x *and* y, $x > 0$ *and* $y > 0$,
x **gcd** $y = (\mathbf{MAX}\ i : i > 0\ \wedge\ x\ \mathbf{mod}\ i = 0\ \wedge\ y\ \mathbf{mod}\ i = 0 : i)$

We can use a program we examined earlier with different pre- and postconditions to compute the greatest common divisor.

$$
\begin{array}{l}
|[\ \mathbf{var}\ x, y : int\ \{x > 0 \wedge y > 0 \wedge x = X \wedge y = Y\}\ ; \\
\quad\quad \mathbf{do}\ x < y \rightarrow x, y := y, x \\
\quad\quad [\!]\quad x > y \rightarrow x := x - y \\
\quad\quad \mathbf{od} \\
\quad\quad \{x = X\ \mathbf{gcd}\ Y\} \\
]|
\end{array}
$$

What is an appropriate invariant?

Knowing properties of **gcd** will help!

For any $x > 0$ and $y > 0$:

1. x **gcd** $x = x$
2. x **gcd** $y = y$ **gcd** x
3. x **gcd** $y = (x - y)$ **gcd** y

Note: (1) and (2) are easy to see from **gcd** definition. Property (3) follows from:

$$
\begin{array}{ll}
 & x\ \mathbf{mod}\ i = 0\ \wedge\ y\ \mathbf{mod}\ i = 0 \\
\equiv & (x - y)\ \mathbf{mod}\ i = 0\ \wedge\ y\ \mathbf{mod}\ i = 0
\end{array}
$$

Given $x > 0 \wedge y > 0 \wedge i > 0$:

$$
\begin{array}{ll}
 & x\ \mathbf{mod}\ i = 0\ \wedge\ y\ \mathbf{mod}\ i = 0 \\
\equiv & \langle\ \text{Def. of } \mathbf{mod}\text{, assumption}\ \rangle \\
 & (\exists\ m, n :: m * i = x\ \wedge\ n * i = y) \\
\equiv & \langle\ \text{arithmetic}\ \rangle \\
 & (\exists\ m, n :: (m - n) * i = x - y\ \wedge\ n * i = y) \\
\equiv & \langle\ \text{Def. of } \mathbf{mod}\text{, assumption}\ \rangle \\
 & (x - y)\ \mathbf{mod}\ i = 0\ \wedge\ y\ \mathbf{mod}\ i = 0
\end{array}
$$

Now consider an invariant. We note that x **gcd** y remains constant. This suggests the invariant:

$P : x > 0 \ \wedge \ y > 0 \ \wedge \ x \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y$

Now we can consider the proof of the program using the new specification and invariant.

Proof Obligations

1. Initialization.

$$\begin{array}{l} \quad x > 0 \wedge y > 0 \wedge x = X \wedge y = Y \\ \Rightarrow x > 0 \wedge y > 0 \wedge x \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y \end{array}$$

2. Finalization.

$$\begin{array}{l} \quad x > 0 \wedge y > 0 \wedge x \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y \wedge \neg(x < y \vee x > y) \\ \Rightarrow x = X \ \mathbf{gcd} \ Y \end{array}$$

3. Invariance (first leg).

$$\begin{array}{l} \quad x > 0 \ \wedge \ y > 0 \ \wedge \ x \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y \ \wedge \ x < y \\ \Rightarrow \ y > 0 \ \wedge \ x > 0 \ \wedge y \ \mathbf{gcd} \ x = X \ \mathbf{gcd} \ Y \end{array}$$

4. Invariance (second leg).

$$\begin{array}{l} \quad x > 0 \ \wedge \ y > 0 \ \wedge \ x \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y \ \wedge \ x > y \\ \Rightarrow \ (x - y) > 0 \ \wedge \ y > 0 \ \wedge (x - y) \ \mathbf{gcd} \ y = X \ \mathbf{gcd} \ Y \end{array}$$

5. Boundedness and progress are the same as before.

3.8 Example: Maximum Row Sum in a Matrix

Consider the following program to compute the maximum sum of the rows in a two-dimensional matrix.

$$
\begin{array}{l}
|[\ \textbf{var}\ r, s, m, n : int\ ;\ \textbf{con}\ b[0..M)[0..N) : \textbf{array of array of}\ int\ \{Q\}; \\
\quad r, m := 0, 0; \\
\quad \textbf{do}\ m \neq M \rightarrow \\
\qquad s, n := 0, 0; \\
\qquad \textbf{do}\ n \neq N \rightarrow \\
\qquad\quad s, n := s + b.m.n, n + 1 \\
\qquad \textbf{od} \\
\qquad r, m = r\ \textbf{max}\ s, m + 1 \\
\quad \textbf{od} \\
\quad \{r, s, m, n : R\} \\
]|
\end{array}
$$

where:

$$
\begin{array}{l}
Q : M \geq 0\ \wedge\ N \geq 0\ \wedge\ (\forall\ i : 0 \leq i < M\ \wedge\ 0 \leq j < N : b.i.j > 0) \\
R : r = 0\ \textbf{max}\ (\textbf{MAX}\ i : 0 \leq i < M : (\Sigma j : 0 \leq j < N : b.i.j))
\end{array}
$$

Note: To see the rationale for the "0 **max**" part of R, consider the case where M is zero. In the case of an empty range for a **MAX** quantification the result is negative infinity.

For this program, we can construct the following proof outline.

$$
\begin{array}{l}
|[\ \textbf{var}\ r, s, m, n : int\ ;\ \textbf{var}\ b[0..M)[0..N) : \textbf{array of array of}\ int\ \{Q\}; \\
\quad \{P_1{}^{r,m}_{0,0}\} \\
\quad r, m := 0, 0; \\
\quad \{\text{inv. } P_1, \text{ bound } t_1\} \\
\quad \textbf{do}\ m \neq M \rightarrow\ \{P_1\ \wedge\ m \neq M\ \wedge\ t_1 = T_1\} \\
\qquad \{P_2{}^{s,n}_{0,0}\} \\
\qquad s, n := 0, 0; \\
\qquad \{\text{ inv. } P_2, \text{ bound } t_2\} \\
\qquad \textbf{do}\ n \neq N \rightarrow \{P_2\ \wedge\ n \neq N\ \wedge\ t_2 = T_2\} \\
\qquad\qquad \{(P_2\ \wedge\ t_2 < T_2)^{s,n}_{s+b.m.n,\, n+1}\} \\
\qquad\qquad s, n := s + b.m.n, n + 1 \\
\qquad\qquad \{P_2\ \wedge\ t_2 < T_2\} \\
\qquad \textbf{od}\ \{P_2\ \wedge\ \neg(n \neq N)\} \\
\qquad \{(P_1\ \wedge\ t_1 < T_1)^{r,\,m}_{r\ \textbf{max}\ s,\, m+1}\} \\
\qquad r, m = r\ \textbf{max}\ s, m + 1 \\
\qquad \{P_1\ \wedge\ t_1 < T_1\} \\
\quad \textbf{od}\quad \{P_1\ \wedge\ \neg(m \neq M)\} \\
\quad \{r, s, m, n : R\} \\
]|
\end{array}
$$

Proof Obligations

1. Outer loop initialization.

   $$M > 0 \wedge N > 0 \wedge (\forall\ i, j : 0 \leq i < M \wedge 0 \leq j < N : b.i.j > 0) \Rightarrow P_1{}^{r,m}_{0,0}$$

2. Outer loop finalization.

   $$P_1 \wedge \neg(m \neq M) \Rightarrow R$$

3. Outer loop boundedness.

   $$P_1 \wedge m \neq M \Rightarrow t_1 \geq 0$$

4. Inner loop initialization.

   $$P_1 \wedge m \neq M \wedge t_1 = T_1 \Rightarrow P_2{}^{s,n}_{0,0}$$

5. Inner loop invariance.

   $$P_2 \wedge n \neq N \Rightarrow P_2{}^{s,\,n}_{s+b.m.n,\, n+1}$$

6. Inner loop boundedness.

$$P_2 \wedge n \neq N \Rightarrow t_2 \geq 0$$

7. Inner loop progress.

$$P_2 \wedge n \neq N \wedge t_2 = T_2 \Rightarrow (t_2 < T_2)^{s,\,n}_{s+b.m.n,\,n+1}$$

8. Inner loop finalization and outer loop invariance.

$$P_2 \wedge \neg(n \neq N) \Rightarrow P_1\,{}^{r,m}_{r\ \mathbf{max}\ s,\,m+1}$$

9. Inner loop finalization and outer loop progress.

$$P_2 \wedge \neg(n \neq N) \Rightarrow (t_1 < T_1)^{r,m}_{r\ \mathbf{max}\ s,\,m+1}$$

3.8.1 Outer loop

Consider the important variables for outer loop.

- m : control variable
- r : "running" maximum of matrix row sums
- s, n : values used only within loop, not needed from iteration to iteration
- b : constant

The range for the control variable m is $0 \leq m \leq M$.

Variable r is at least 0. It is the maximum row sum for region processed so far.

An appropriate formula for r seems to be:

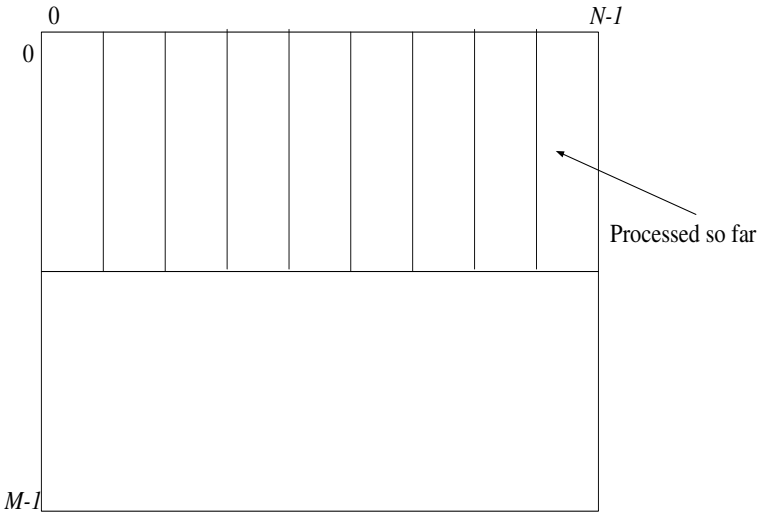
$$r = 0\ \mathbf{max}\ (\mathbf{MAX}\ i : 0 \leq i < m : (\Sigma j : 0 \leq j < N : b.i.j))$$

This argument suggests the following invariant for the outer loop:

$$P_1 : Q \wedge 0 \leq m \leq M \wedge r = 0\ \mathbf{max}\ (\mathbf{MAX}\ i : 0 \leq i < m : (\Sigma j : 0 \leq j < N : b.i.j))$$

Note: Since none of the variables in the precondition are allowed to change, we can take the precondition Q as a part of the invariants. We could leave this implicit.

Because of the invariant range $0 \leq m \leq M$ and progress statement $m := m + 1$, we can take the bound function for the outer loop as $t_1 : M - m$

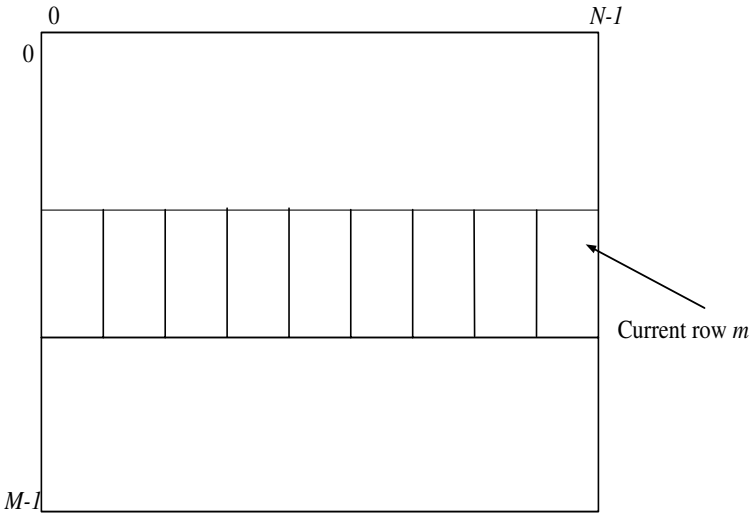


3.8.2 Inner loop

The range on the control variable n is $0 \leq n \leq N$.

Note that $P_1 \ \wedge \ t_1 = T$ holds throughout the inner loop because none of the variables mentioned are changed.

Variable s is the sum of the elements from the current row m that have been processed so far. That is, $s = (\Sigma\, j : 0 \leq j < n : b.m.j)$



Thus we take the following as the inner loop invariant:

$$P_2 : 0 \leq n \leq N \ \wedge \ s = (\Sigma j : 0 \leq j < n : b.m.j) \ \wedge \ P_1 \ \wedge \ t_1 = T_1$$

We also take the following as the bound function for the inner loop because of the range $0 \leq n \leq N$ and progress statement $n := n + 1$.

$t_2 : N - n$

3.9 Example: Counting Adjacent Decreasing Pairs

Consider a program to count the number of adjacent decreasing pairs in an integer array. We might consider this a measure of the "disorder" in the array.

```
|[ con N : int; con b[0..N): array of int ; var n, c : int {Q} ;
      if  N ≤ 0 →
            c := 0;
      []  N > 0 →
            n, c := 1, 0;
            do n ≠ N →
                  if  b.(n − 1) > b.n →
                        n, c := n + 1, c + 1;
                  []  b.(n − 1) ≤ b.n →
                        n := n + 1;
                  fi
            od
      fi
      {n, c : R}
]|
```

where:

$Q : true$
$R : c = (\#i : 1 \leq i < N : b.(i-1) > b.i)$

We can state a proof outline as follows. Note that for **if** statements, we can carry the preconditions forward across the guards. We must, of course, verify that the guards are defined and that at least one of the guards evaluates to *true*.

$|[$ **con** $N : int$; **con** $b[0..N)$: **array of** int ; **var** $n, c : int$ $\{Q\}$;
$\{\ Q \wedge (N \leq 0 \vee N > 0)\}$
if $N \leq 0 \rightarrow \{Q \wedge N \leq 0\}$
$\{R_0^c\}\ c := 0;\ \{R\}$
$[]$ $N > 0 \rightarrow \{Q \wedge N > 0\}$
$\{P_{1,0}^{n,c}\}$
$n, c := 1, 0;$
{invariant P, bound t}
do $n \neq N \rightarrow \{P \wedge n \neq N \wedge t = T\}$
$\{\ P \wedge t = T \wedge 0 < n < N \wedge (b.(n-1) > b.n \vee b.(n-1) \leq b.n)\ \}$
if $b.(n-1) > b.n \rightarrow \{P \wedge 0 < n < N \wedge b.(n-1) > b.n \wedge t = T\}$
$\{(P \wedge t < T)_{n+1,c+1}^{n,c})\}\ n, c := n+1, c+1;\ \{P \wedge t < T\}$
$[]$ $b.(n-1) \leq b.n \rightarrow \{P \wedge 0 < n < N \wedge b.(n-1) \leq b.n \wedge t = T\}$
$\{(P \wedge t < T)_{n+1}^{n}\}\ n := n+1;\ \{P \wedge t < T\}$
fi $\{P \wedge t < T\}$
od $\{\ P \wedge \neg(n \neq N)\ \}$
fi
$\{n, c : R\}$
$]|$

Proof Obligations

1. Nonabortion of outer **if**.

   $true \Rightarrow N \leq 0 \vee N > 0$

2. First leg of outer **if**.

   $N \leq 0 \Rightarrow R_0^c$

3. Second leg of outer **if**. Initialization for the **do**.

   $N > 0 \Rightarrow P_{1,0}^{n,c}$

4. Second leg of outer **if**. Finalization for the **do**.

   $P \wedge \neg(n \neq N) \Rightarrow R$

5. Second leg of outer **if**. Boundedness for the **do**.

   $P \wedge 0 < n < N \Rightarrow t \geq 0$

6. Nonabortion of inner **if**.

   $P \wedge t = T \wedge n \neq N \Rightarrow 0 < n < N \wedge (b.(n-1) > b.n \vee b.(n-1) \leq b.n)$

7. First leg of inner **if**. Invariance and progress for the **do**.

   $P \wedge 0 < n < N \wedge b.(n-1) > b.n \wedge t = T \Rightarrow (P \wedge t < T)_{n+1,c+1}^{n,c}$

8. Second leg of inner **if**. Invariance and progress for the **do**.

   $P \wedge 0 < n < N \wedge b.(n-1) \leq b.n \wedge t = T \Rightarrow (P \wedge t < T)^n_{n+1}$

Note: In previous examples, the invariance and progress proof obligations are shown separately. Above we combine them.

In the loop, since n is initially 1 and is incremented by 1 on each iteration until N is reached, it is easy to see that function $N - n$ is a sufficient bound function and that the invariant probably needs a conjunct $1 \leq n \leq N$.

In the loop, also note that variable c contains the running count of the number of adjacent elements of the array in which the first is larger than the second. At each interation, the pairs $(0, 1)$ through $(n-2, n-1)$ have been considered previously.

Therefore, the invariant P and bound function t below seem appropriate:

$P : 1 \leq n \leq N \wedge c = (\#i : 1 \leq i < n : b.(i-1) > b.i)$
$t : N - n$

Of course, we must prove the eight proof obligations using the above formulation of P and t. It is relatively straightforward to prove proof obligations 1 through 6.

To prove 7, assume $1 \leq n \leq N \wedge c = (\#i : 1 \leq i < n : b.(i-1) > b.i)$
$\wedge\ 0 < n < N \wedge b.(n-1) > b.n \wedge N - n = T$.

$$
\begin{array}{ll}
 & 1 \leq n+1 \leq N \wedge c+1 = (\#i : 1 \leq i < n+1 : b.(i-1) > b.i) \wedge N - (n+1) < T \\
\equiv & \langle \text{ assumption } 1 \leq n < N \ \rangle \\
 & \mathit{true} \wedge c+1 = (\#i : 1 \leq i < n \vee i = n : b.(i-1) > b.i) \wedge (N - n - 1) < T \\
\equiv & \langle \text{ range splitting, assumption } N - n = T \ \rangle \\
 & c+1 = (\#i : 1 \leq i < n : b.(i-1) > b.i) + (\#i : i = n : b.(i-1) > b.i) \wedge \mathit{true} \\
\equiv & \langle \text{ assumption } c = (\#i : 1 \leq i < n : b.(i-1) > b.i), \text{ one point } \rangle \\
 & c+1 = c + \#.(b.(n-1) > b.n) \\
\equiv & \langle \text{ assumption } b.(n-1) > b.n, \#.\mathit{true} = 1 \ \rangle \\
 & c+1 = c+1 \\
\equiv & \langle \text{ arithmetic } \rangle \\
 & \mathit{true}
\end{array}
$$

The proof of obligation 8 differs primarily in that $\#.(b.(n-1) > b.n) = 0$ in this leg of the inner **if** statement.

4 Program Derivation

Program derivation is the calculation of a program from its specification by means of formula manipulation.

4.1 Example: Minimum of an Array Revisited

Consider the loop body of the program for finding the minimum value in the array $b[0..N)$ from Section 3.5. Suppose we have determined (or guessed) that the body of the loop has the following specification and structure, for some unknown expression E, loop invariant P, and loop guard $n \neq N$:

$$\{P \wedge n \neq N\}\ m, n := E, n+1\ \{P\}$$

Now let's calculate the unknown expression E, where P is defined as follows:

$$P : 1 \leq n \leq N \wedge m = (\mathbf{MIN}\ i : 0 \leq i < n : b.i)$$

We can proceed by using the logic, semantics of the guarded commands, and proof techniques to solve the logical equations for the unknown expression.

Assume precondition $P \wedge n \neq N$ holds. Then we can calculate $wp.(m, n := E, n+1).P$ to solve for the unknown E.

$$\begin{array}{ll} & wp.(m, n := E, n+1).P \\ \equiv & \quad \langle\ \text{assignment axiom, substitution}\ \rangle \\ & 1 \leq n+1 \leq N \wedge E = (\mathbf{MIN}\ i : 0 \leq i < n+1 : b.i) \\ \equiv & \quad \langle\ \text{assumption } 1 \leq n \leq N \wedge n \neq N\text{, arithmetic, range splitting, one-point}\ \rangle \\ & E = (\mathbf{MIN}\, i : 0 \leq i < n : b.i)\ \ \mathbf{min}\ \ b.n \\ \equiv & \quad \langle\ \text{assumption } P\text{, Leibniz}\ \rangle \\ & E = m\ \ \mathbf{min}\ \ b.n \end{array}$$

Thus the derived command is $m, n := m\ \mathbf{min}\ b.n, n+1$.

4.2 Example: Integer Square Root Revisited

Now consider the integer square root program we examined in Section 3.6. The body of the loop had the following specification and structure, where P is the loop invariant and $k \leq N$ is the loop's guard, and E and F are unknown expressions:

$$\{P \wedge k \leq N\}\ i, j, k := i + 1, E, F\ \{P\}$$

As determined previously, let the invariant be defined as follows:

$$P : 0 \leq i \wedge i^2 \leq N \wedge j = 2 * i + 1 \wedge k = (i + 1)^2$$

Assume $P\ \wedge\ k \leq N$.

$$\begin{array}{ll} & wp.(i, j, k := i + 1, E, F).P \\ \equiv & \quad \langle \text{ assignment axiom, textual substitution } \rangle \\ & 0 \leq i + 1 \wedge (i + 1)^2 \leq N \wedge E = 2 * (i + 1) + 1 \wedge F = ((i + 1) + 1)^2 \\ \equiv & \quad \langle \text{ precondition assumption } 0 \leq i \wedge i^2 \leq N \wedge k = (i + 1)^2 \wedge k \leq N, \text{ arithmetic } \rangle \\ & E = (2 * i + 1) + 2\ \wedge\ F = (i + 1)^2 + 2 * (i + 1) + 1 \\ \equiv & \quad \langle \text{ precondition assumption, arithmetic } \rangle \\ & E = j + 2 \wedge F = k + (2 * i + 1) + 2 \\ \equiv & \quad \langle \text{ precondition assumption, Leibniz } \rangle \\ & E = j + 2 \wedge F = k + j + 2 \end{array}$$

Note: We can avoid doing the addition $j + 2$ twice by sequencing the command $k := k + j$ after the command $i, j := i + 1, j + 2$.

Thus the derived command is:

$$i, j, k := i + 1, j + 2, k + j + 2$$

Or, using the suggested optimization, the command is the following sequence, which is what we had in the earlier program:

$$i, j := i + 1, j + 2; k = k + j$$

4.3 Example: Absolute Value

Now consider the derivation of a more complex command, one that has the specification

$$|[\ \textbf{var}\ x : int\ \{Q\};\ S\ \{R\}\ ]|$$

where

$$\begin{array}{ll} Q: & x = X \\ R: & x = |X| \end{array}$$

We impose the **restriction** that command S does not contain an absolute value operator.

Note: $x = |X| \ \equiv\ (x = X \vee x = -X) \wedge x \geq 0$

An examination of the precondition Q and the postcondition R yields two possible statements to establish R:

1. *skip*
2. $x := -x$

So, let's try to calculate to determine what the needed program structure is.

1. Let's try a calculation for *skip*.

   Assume Q.

   $$\begin{array}{ll} & wp.skip.R \\ \equiv & \quad \langle\ skip \text{ axiom, def. of absolute value}\ \rangle \\ & (x = X \vee x = -X) \wedge x \geq 0 \\ \equiv & \quad \langle\ \text{assumption } Q\ \rangle \\ & x \geq 0 \end{array}$$

   Thus *skip* can establish R when $x \geq 0$. Hence, we have derived the guarded command "$x \geq 0 \rightarrow skip$".

2. Let's try a calculation for $x := -x$.

   Assume Q.

   $$\begin{array}{ll} & wp.(x := -x).R \\ \equiv & \quad \langle\ \text{assignment axiom, textual substitution}\ \rangle \\ & (-x = X \vee -x = -X) \wedge -x \geq 0 \\ \equiv & \quad \langle\ \text{assumption } Q\text{, arithmetic}\ \rangle \\ & -x \geq 0 \\ \equiv & \quad \langle\ \text{arithmetic}\ \rangle \\ & x \leq 0 \end{array}$$

Thus $x := -x$ can establish R when $x \leq 0$. Hence, we have derived guarded command "$x \leq 0 \rightarrow x := -x$".

3. Note that $Q \Rightarrow (x \geq 0 \vee x \leq 0)$, so we can stop generating guarded commands. Hence, we have derived the **if** command:

   if $x \geq 0 \rightarrow skip$
   $[]$ $x \leq 0 \rightarrow x := -x$
   fi

Note we can strengthen either guard by adding any conjunct (e.g., $x \neq 0$) as long as the non-abortion condition (3) still holds. That is, we might optimize the above guarded command by removing the $x = 0$ case from one of the guards.

4.4 Strategy for Developing an Alternative Command

Remember that in Section 2.4.6 we gave the following definition for an alternative command IF (where BB denotes the disjunction of the guards):

$$wp.\text{IF}.R \;\equiv\; def.BB \wedge BB \wedge (\forall\, i : 0 \leq i < N : B.i \Rightarrow wp.(S.i).R)$$

Consider $\{Q\}\ IF\ \{R\}$. The previous example suggests a strategy for deriving commands of this type.

We can use the following strategy to derive a command to establish postcondition R:

1. Find some command C whose execution will establish postcondition R in at least some cases.
2. Find a Boolean expression B such that B is defined when Q holds and that $Q \wedge B \Rightarrow wp.C.R$.
3. Form a guarded command "$B \rightarrow C$"using the command and guard just found.
4. Repeat steps 1 through 3 until precondition Q implies that at least one of the derived guards is *true*.

Gries' Principle for *if* Commands

All other things being equal, make the guards as* strong *as possible so that some errors will cause abortion. (Here, an "error" means that the precondition is not satisfied.)

We call this *Gries' principle for* **if** *commands*. It follows from the philosophy that it is better for a program to yield no result (i.e., abort) rather than yield an erroneous result.

This differs a bit from the philosophy of defensive programming that we have been taught in our programming classes. As a practical matter, it is still good for our program to be made robust by validating that it is being used properly, especially at the points where it meets the outside world (e.g., parameter values, input data, etc.) However, the stress in this class is that a program should be built internally to be correct for all situations that meet the specification.

In some cases, we may want to consider whether we need to refine the specification of the program to define the needed result for invalid inputs.

4.5 Strategy for Developing a Simple Loop

Consider a simple **do** loop with the following structure:

$\{\ Q\ \}$
initialization;
{ invariant P, bound function t }
do $B \rightarrow$
    decrease t, keeping P true
od $\{\ P \wedge \neg B\}$
$\{\ R\ \}$

We can use the following strategy to construct the loop:

1. Transform postcondition R into a candidate invariant P.
2. Develop a guard such that $P \wedge \neg B \ \Rightarrow\ R$.
3. Develop an initialization "assignment" to establish P in a limiting case.
4. Develop a candidate bound function t such that $P \wedge B \ \Rightarrow\ t \geq 0$.
5. Develop a candidate command to decrease t in the loop body.
6. Use a weakest precondition calculation on the t-decreasing command to identify other commands.
7. If some term cannot be expressed in terms of the current variables, add a fresh variable and refine the invariant, initialization, and bound function accordingly.
8. Continue until $P \wedge B$ logically implies the precondition of the derived body.

Observation: The choice of the invariant P depends upon the *shape* of the postcondition R. (Remember that $P \wedge \neg B \Rightarrow R$.)

In transforming a postcondition to find a candidate invariant and guards, we use various *heuristics*. These include:

1. Deleting a conjunct
2. Replacing a constant by a variable
3. Strengthening an invariant
4. Tail recursion

These are not the *only* heuristics used, but the principal ones we will study.

Gries' Principle for *do* Commands

All other things being equal, make the guards of a loop as* weak *as possible, so that an error may cause an infinite loop.

We call this *Gries' principle for* **do** *commands*. It follows from the philosophy that it is better for a program to yield no result (i.e., not terminate) rather than to yield an erroneous result.

Again, this differs from the philosophy of defensive programming that we have been taught in our programming classes. As with alternative commands, the emphasis here is that a program should be built internally to be correct for all situations that meet the specification.

Perhaps it is better in some situations to change the specification to require "illegal" initial states to be given some reasonable final state or an explicit error message.

5 Deleting a Conjunct

5.1 Heuristic

In this section, we begin to apply our heuristics to develop small programs. This section discusses a technique we call "deleting a conjunct". Some writers call this heuristic "taking a conjunct".

Suppose we are given a postcondition R such as:

$$R : X \wedge Y$$

The first step in developing a loop is to choose an invariant P and a guard B. In particular, for the given postcondition R, the invariant P and loop guard B must be chosen to satisfy the finalization criterion:

$$P \wedge \neg B \Rightarrow R$$

This requirement suggests at least two possible choices for P and B to establish postcondition R, that is, to derive the loop:

$$\{P\}\ \textbf{do}\ B \rightarrow S\ \textbf{od}\ \{P \wedge \neg B\}$$

These choices include:

$$P :\ X$$
$$B : \neg Y$$

and

$$P :\ Y$$
$$B : \neg X$$

Either choice is acceptable as far as the finalization criterion $P \wedge \neg B \Rightarrow R$ is concerned. However, a bad choice could make initialization difficult or impossible.

Strategy: Delete the conjunct of the postcondition that is the most difficult to initialize and take its negation as the guard.

Note: If a postcondition is not a conjunction, it can often be made into one. For example, the unchanging parts of the precondition or properties of the objects in the postcondition can be conjuncted into the postcondition.

5.2 Example: Integer Division-Remainder

Consider a program Divmod, with the following specification, to compute the quotient and remainder from dividing a natural number by a positive integer:

$|[\ \ \textbf{con}\ x, y : int\ \{Q\}\ ;\ \textbf{var}\ q, r : int\ ;\ \text{Divmod}\ \{R\}\ ]|$

where

$$Q : 0 \leq x \wedge 0 < y$$
$$R : 0 \leq r\ \ \wedge\ \ r < y\ \ \wedge\ \ q * y + r = x$$

Restriction: Program Divmod cannot use division or multiplication, but it can use addition $(+)$ and subtraction $(-)$.

Speculation: Try using "deleting a conjunct" to construct a loop. Then we have a program with the following structure:

$\{\ Q\ \}$
$q, r := E, F\ ;$
$\{$ invariant P, bound $t\ \}$
do $B \rightarrow$
$\quad S_1$
od $\{P \wedge \neg B\}$
$\{\ R\ \}$

Which conjunct of R can we delete to form an invariant?

- Delete the third conjunct $q * y + r = x$?

  No, this is the only conjunct naming q and it defines a relation among all variables.

- Delete the first conjunct $0 \leq r$?

  Then our choices for the invariant P and guard B are as follows.

  $$P : r < y \wedge q * y + r = x$$
  $$B : r < 0$$

  Then we calculate on the initialization command to see if appropriate expressions for the unknowns E and F can be found. Assume Q, that is, $0 \leq x \wedge 0 < y$.

  $$\begin{array}{ll} & wp.(q, r := E, F).P \\ \equiv & \quad \langle\ \text{assignment axiom, textual substitution}\ \rangle \\ & F < y\ \ \wedge\ \ E * y + F = x \end{array}$$

  There is no clear choice for the unknown expressions. $E = 0 \wedge F = x$ establishes the right conjunct, but not the left.

- Delete the second conjunct $r < y$?

  Then our choices for the invariant P and guard B are as follows.

  $$P : 0 \leq r \wedge q * y + r = x$$
  $$B : r \geq y$$

  Then we again calculate on the initialization command to see if appropriate expressions for the unknowns E and F can be found.

  Assume Q, that is, $0 \leq x \wedge 0 < y$.

  $$\begin{array}{ll} & wp.(q, r := E, F).P \\ \equiv & \quad \langle \text{ assignment axiom, textual substitution } \rangle \\ & 0 \leq F \ \wedge \ E * y + F = x \\ \equiv & \quad \langle \text{ choose } E = 0 \ \wedge \ F = x \rangle \\ & 0 \leq x \end{array}$$

  This choice is really obvious given the invariant. The program resulting from this choice is the following, with t and S_1 as unknowns:

  $\{Q\}$
  $q, r := 0, x$;
  { invariant P, bound t }
  do $r \geq y \rightarrow$
  $\quad S_1$
  od
  $\{P \wedge r < y$, hence $R\}$

Now, how do we decide the bound function t?

Here the guard is $r \geq y$. What are our choices?

- Increase y?

  No, because y is constant in the loop body.

- Decrease r?

  This should work.

- Do both?

  No, again because y is a constant.

Thus, for bound function t, we choose $r - y$ or, more simply, just r.

Now, consider the loop body S_1. We need:

- a command to decrease t. This can be accomplished by $r := r - k$, for some $k > 0$.

- a command to maintain the invariance of P. This can be accomplished by $q := E$, for some unknown expression E.

Let us speculate that the above commands can be combined into a multiple assignment and solve for E and k ($k > 0$) in the following:

$$\{P \wedge r \geq y\}\ q, r := E,\ r - k\ \{P\}$$

Assume $P \wedge r \geq y$. Also assume Q and $k > 0$.
(Remember that P is $0 \leq r \wedge q * y + r = x$ and that Q is $0 \leq x \wedge 0 < y$.)

$$\begin{array}{ll} & wp.(q, r := E, r - k).P \\ \equiv & \quad \langle \text{ assignment axiom, textual substitution } \rangle \\ & 0 \leq r - k \wedge E * y + (r - k) = x \\ \equiv & \quad \langle \ q * y + r = x \text{ assumption, Leibniz, arithmetic } \rangle \\ & r \geq k \wedge E * y + r - k = q * y + r \\ \equiv & \quad \langle \ 0 < y \wedge k > 0 \text{ assumptions, arithmetic } \rangle \\ & r \geq k \wedge E = q + k \ \mathbf{div}\ y \\ \equiv & \quad \langle \text{ choose } k = y, \text{ the smallest positive multiplier of } y \text{ to avoid } \mathbf{div} \text{ and remainder } \rangle \\ & r \geq y \wedge E = q + 1 \\ \equiv & \quad \langle \ r \geq y \text{ assumption } \rangle \\ & E = q + 1 \end{array}$$

To summarize, the derived program is the following:

$$\begin{array}{l} |[\ \ \mathbf{con}\ x, y : int\ \{Q\}\ ;\ \mathbf{var}\ q, r : int\ ; \\ \quad q, r := 0, x; \\ \quad \{\text{ invariant } P : 0 \leq r \wedge q * y + r = x, \text{ bound function } t : r\} \\ \quad \mathbf{do}\ r \geq y \rightarrow \\ \qquad q, r := q + 1, r - y; \\ \quad \mathbf{od} \\ \quad \{\ P \wedge r < y, \text{ hence } R\ \} \\ ]| \end{array}$$

where

$$\begin{array}{l} Q : 0 \leq x \wedge 0 < y \\ R : 0 \leq r \wedge r < y \wedge q * y + r = x \end{array}$$

How many iterations of the loop are needed?

- The time complexity is $\mathcal{O}(x\ \mathbf{div}\ y)$.
- Each repetition increases q by 1.

5.3 Division-Remainder Revisited: Better Solution?

Can we find a solution to the integer division-remainder problem that is more efficient, that is, that requires fewer iterations of the loop?

Let's go back to the choice of k. We chose $k = y$, but no choice was obvious. A larger value for k might give a more efficient algorithm.

We could postpone the decision on the value of k. To do so, we can introduce a fresh variable d such that $k = d * y$, with $d \geq 1$. That is, we choose a multiple of y to avoid **div** (and a remainder). We may be able to compute d incrementally.

Assume $P \wedge r \geq y$. Also assume Q and $k > 0$.
(Remember that P is $0 \leq r \wedge q * y + r = x$ and that Q is $0 \leq x \wedge 0 < y$.)

$$
\begin{array}{ll}
\ldots & \text{Calculation as above } \ldots \\
 & r \geq k \wedge E = q + k \ \mathbf{div}\ y \\
\equiv & \quad \langle \text{ choose } k = d * y \text{ for } d \geq 1 \ \rangle \\
 & r \geq d * y \wedge E = q + (d * y) \ \mathbf{div}\ y \wedge d \geq 1 \\
\equiv & \quad \langle \text{ arithmetic, } 0 < y \text{ assumption } \rangle \\
 & r \geq d * y \wedge E = q + d \wedge d \geq 1 \\
\equiv & \quad \langle \text{ constrain } d \text{ such that } r \geq d * y \wedge d \geq 1 \ \rangle \\
 & E = q + d
\end{array}
$$

Thus we can introduce an inner block within the loop and get the following program. Note the constraints on d that we require.

$$
\begin{array}{l}
\mathbf{do}\ r \geq y \rightarrow \{P \wedge r \geq y\} \\
\quad |[\ \ \mathbf{var}\ d : int\ ; \\
\quad\quad S_2\ ; \{d : r \geq d * y \wedge d \geq 1 \wedge P \wedge r \geq y\} \\
\quad\quad q, r := q + d, r - d * y \\
\quad]|\ \{\ P\ \} \\
\mathbf{od}
\end{array}
$$

So, the remaining tasks are to determine S_2 and to remove $d * y$ (because of the forbidden multiplication operation).

Now let's eliminate the multiplication in $d * y$. We can introduce a fresh variable dd such that $dd = d * y$. We may be able to compute dd incrementally.

We now have the following program, with variable dd and the constraints upon it.

$$
\begin{array}{l}
\mathbf{do}\ r \geq y \rightarrow \{P \wedge r \geq y\} \\
\quad |[\ \ \mathbf{var}\ d, dd : int\ ; \\
\quad\quad S_2; \{d, dd : r \geq d * y \wedge d \geq 1 \wedge dd = d * y \wedge P \wedge r \geq y\} \\
\quad\quad q, r := q + d, r - dd \\
\quad]|\ \{\ P\ \} \\
\mathbf{od}
\end{array}
$$

We have solved one problem by adding others. However, the new problems offer the possibility of efficient solutions.

Next we need to find S_2 that establishes:

$$R_0 : d \geq 1$$
$$R_1 : r \geq d * y$$
$$R_2 : dd = d * y$$

Note: $P \wedge r \geq y$ holds throughout the inner loop. Its variables are not changed in S_2.

We want to increase d efficiently, so we constrain d to be a power of 2. Thus it can be increased by an efficient "doubling" operation—adding d to itself or doing an arithmetic left shift. Thus we introduce a new requirement:

R_3 : "d is a power of 2" — i.e., $(\exists\ i : i \geq 0 : d = 2^i)$

Note: R_3 is a "trick", but it is a commonly used trick to improve efficiency. Once you use a trick more than once, it becomes a method.

We want d to be as large as possible. Because of R_1 (i.e., $r \geq d * y$), let $d * y$ be the largest power of $2 \leq r$. Thus we add a new requirement:

$$R_4 : 2 * d * y > r$$

Thus we need a new program that establishes:

$$R' : R_0 \wedge R_1 \wedge R_2 \wedge R_3 \wedge R_4$$

Clearly, S_2 is a repetition. We delete a conjunct to find the loop structure. Let's analyze the postcondition to find an appropriate conjunct to delete.

- R_0 (i.e., $d \geq 1$) holds important information about d and cannot be deleted.
- R_1 (i.e., $r \geq d * y$) is a possibility.
- R_2 (i.e., $dd = d * y$) holds the only information about dd.
- R_3 (i.e., "d is a power of 2") holds important information about d.
- R_4 $(2 * d * y > r)$ is a possibility.

The choice between R_1 and R_4 is arbitrary. However, initializing the invariant may be easier if $\neg R_4$ is taken as the guard. Thus our choices for the S_2 loop are:

guard : $r \geq 2 * d * y$
invariant PP : $R_0 \wedge R_1 \wedge R_2 \wedge R_3 \wedge P \wedge r \geq y$

Because of R_2, the guard can be rewritten as $r \geq dd + dd$.

The conjuncts $P \wedge r \geq y$ of the invariant can be left implicit since there is no effect on any of their variables in this inner loop.

Clearly, the command $d, dd := 1, y$ establishes PP given $P \wedge r \geq y$. Thus the inner loop is the following, for some command SS:

$\{\ P \wedge r \geq y\ \}$
$d, dd := 1, y;$
$\{$ invariant $PP : R_0 \wedge R_1 \wedge R_2 \wedge R_3 \wedge P \wedge r \geq y\}$
do $r \geq 2 * d * y \rightarrow$
    SS
od
$\{\ PP \wedge r < 2 * d * y$, hence $R'\ \}$

Consider termination of this inner loop. $r \geq 2 * d * y$ must become false. Because r and y are constants within SS, only d may change.

Because of R_3, we can increase d by doubling. That is, the bound function can be $r - d$ and the progress command can be $d := 2 * d$.

Consider $d, dd := 2 * d, E$ for SS and calculate for the unknown E. We only need to consider requirement R_2 because it is the only conjunct affected by an assignment to variable dd.

Assume $PP \wedge r \geq 2 * d * y$.

$wp.(d, dd := 2 * d, E).R_2$
$\equiv$ $\langle$ assignment axiom $\rangle$
$E = (2 * d) * y$
$\equiv$ $\langle$ assumption $dd = d * y$, arithmetic $\rangle$
$E = 2 * dd$

We can eliminate the multiplication by 2 by replacing the assignment $d, dd := 2 * d, 2 * dd$ by $d, dd := d + d, dd + dd$. We get the following program:

```
|[  con x, y : int { Q : 0 ≤ x ∧ 0 < y } ;
    var q, r : int ;
    q, r := 0, x;
    {invariant P : 0 ≤ r ∧ q * y + r = x, bound function t : r}
    do r ≥ y →  {P ∧ r ≥ y}
        |[ var d, dd : int ;
            d, dd := 1, y ;
            {invariant PP, bound:  r − d}
            do r ≥ dd + dd →
                d, dd := d + d, dd + dd
            od { PP ∧ r < dd + dd, hence R′  }
            q, r := q + d, r − dd
        ]|
    od { P ∧ r < y, hence R }
]|
```

What is the time complexity? The worst case time complexity is:

$$\mathcal{O}((log_2\, x)^2)$$

The justification for the time complexity characterization is:

- The inner loop has $log_2\, r$ iterations.
- The outer loop has $log_2\, r$ iterations.
- $r \leq x$.

Thus we have found a more efficient algorithm as x gets large.

5.4 Example: Linear Search

Consider the following program specification for the linear search problem. This program determines the smallest natural number value x for which some boolean function f is *true*, assuming there is solution.

$$
\begin{array}{l}
|[\ \ \textbf{function}\ f[0..) : \mathit{boolean}\ \{\ Q\}\ ; \\
\quad \textbf{var}\ x : \mathit{int}\ ; \\
\quad \{\ x : R\ \} \\
]|
\end{array}
$$

where

$$
\begin{array}{l}
Q : (\exists\, i : 0 \le i : f.i) \\
R : 0 \le x \wedge f.x \wedge (\forall\ i : 0 \le i < x : \neg f.x)
\end{array}
$$

Clearly, a loop is needed. So let's try deleting a conjunct. But which one is to be deleted?

Delete conjunct $f.x$? If we do not delete $f.x$, then we must initialize it to be *true*, which means we must already know a solution (although not necessarily the smallest).

Thus take $\neg f.x$ as the guard and the remaining conjuncts as parts of invariant P. For convenience, we split P into two named invariants:

$$
\begin{array}{l}
P_0 : 0 \le x \\
P_1 : (\forall\ i : 0 \le i < x : \neg f.i)
\end{array}
$$

Thus we have guessed that the program has the following structure:

initialization;
{invariant P, bound t }
do $\neg f.x \rightarrow$
    S
od { $P \wedge f.x$, hence R }

x is the only variable. So, in the initialization, we need to make $P_0 \wedge P_1$ true by a simple assignment to x. Clearly $x := 0$ will establish $P_0 \wedge P_1$.

Now we need to develop the loop body S.

First, what is the bound function?

- It must decrease.
- A large enough decrease will falsify the guard $\neg f.x$.

From precondition Q, we know that $f.x$ must hold for at least one natural number value. Let X be a value such that $0 \leq X \wedge f.X$. We define the bound function as follows:

$$t : X - x$$

Note: X is not a program variable or constant. It is a specification variable. We know that it exists given the precondition, but we do not know what value it h as. Indeed, finding the value of X is the purpose of the program.

Thus, we speculate that the following is an appropriate "progress" command S (i.e., a command that decreases t) for some value $k > 0$:

$$x := x + k$$

Now, we now need to identify a specific k.

We observe the following:

- P_0 (i.e., $0 \leq x$) is preserved by any positive k.
- P_1 (i.e., $(\forall\ i : 0 \leq i < x : \neg f.i)$ is preserved by just an increase by 1.

Therefore, we choose the following for S:

$$x := x + 1$$

Alternatively, we can calculate this result.

Assume $P_0 \wedge P_1 \wedge \neg f.x$. Also assume $k > 0$.
(That is, $0 \leq x \wedge (\forall i : 0 \leq i < x : \neg f.i) \wedge \neg f.x \wedge k > 0$.)

$$\begin{array}{ll} & wp.(x := x + k).(0 \leq x \wedge (\forall i : 0 \leq i < x : \neg f.i)) \\ \equiv & \quad \langle \text{ assignment axiom } \rangle \\ & 0 \leq x + k \wedge (\forall\ i : 0 \leq i < x + k : \neg f.i) \\ \equiv & \quad \langle \text{ assumptions } k > 0 \text{ and } P_0 \ \rangle \\ & (\forall\ i : 0 \leq i < x + k : \neg f.i) \\ \equiv & \quad \langle \text{ assumption } k > 0, \text{ range splitting, one-point } \rangle \\ & (\forall\ i : 0 \leq i < x : \neg f.i) \wedge \neg f.x \wedge (\forall\ i : x < i < x + k : \neg f.i) \\ \equiv & \quad \langle \text{ assumptions } P_1 \wedge \neg f.x \ \rangle \\ & (\forall\ i : x < i < x + k : \neg f.i) \\ \equiv & \quad \langle \text{ choose } k = 1, \text{ empty range } \rangle \\ & true \end{array}$$

Thus we have derived the following program:

$$
\begin{array}{l}
|[\ \ \textbf{function}\ f[0..) : \mathit{boolean}\ \ \{Q : (\exists\ i : 0 \le i : f.i)\}\ ; \\
\quad \textbf{var}\ x : \mathit{int}\ ; \\
\quad x := 0\ ; \\
\quad \{\ \text{invariant}\ P_0 \wedge P_1, \text{bound}\ t : X - x\ \} \\
\quad \textbf{do}\ \neg f.x \to x := x + 1\ \textbf{od} \\
\quad \{\ P \wedge f.x, \text{hence}\ R\ \} \\
]|
\end{array}
$$

However, we have not yet proved the boundedness of t. That is, our proof obligation is:

$$P_0 \wedge P_1 \wedge \neg f.x \Rightarrow t \ge 0$$

Starting with the most complicated expression we have, namely P_1, we can calculate as shown below.

Assume $P_0 \wedge P_1 \wedge \neg f.x$. Also assume the preconditions $Q \wedge 0 \le X \wedge f.X$. (That is, $0 \le x \wedge (\forall\ i : 0 \le i < x : \neg f.i) \wedge \neg f.x \wedge 0 \le X \wedge f.X$.)

$$
\begin{array}{ll}
 & (\forall\ i : 0 \le i < x : \neg f.i) \\
\equiv & \quad \langle\ \text{trading}\ \rangle \\
 & (\forall\ i :: 0 \le i \wedge i < x \Rightarrow \neg f.i) \\
\Rightarrow & \quad \langle\ \text{instantiation with}\ i := X\ \rangle \\
 & 0 \le X \wedge X < x \Rightarrow \neg f.x \\
\equiv & \quad \langle\ \text{shuffle, to isolate the precondition}\ \rangle \\
 & 0 \le X \wedge f.X \Rightarrow X \ge x \\
\equiv & \quad \langle\ \text{assumption}\ 0 \le X \wedge f.X\ \rangle \\
 & \mathit{true} \Rightarrow X \ge x \\
\equiv & \quad \langle\ \text{true antecedent rule}\ \rangle \\
 & X \ge x \\
\equiv & \quad \langle\ \text{assumption}\ f.X\ \text{and}\ \neg f.x, \text{hence}\ X \ne x\ \rangle \\
 & X > x \\
\equiv & \quad \langle\ \text{arithmetic, definition of}\ t\ \rangle \\
 & t > 0
\end{array}
$$

Thus the loop eventually terminates.

5.5 Specializations of Linear Search

The Linear Search algorithm (program) is defined in terms of an arbitrary boolean function on natural numbers (or at least on a "prefix" of naturals that is known to contain at least one *true*). Thus the program can be used to find the smallest natural for which any boolean function is true. This makes it a general program that can be specialized by defining the function f appropriately.

Examples:

1. Find the smallest x such that $m \leq 2^x$ (that is, the smallest power of 2 that is $\geq m$).

   We can formulate this as linear search problem by defining f as follows.

   $$f.x \;\equiv\; m \leq 2^x$$

2. Find the largest x such that $2^x \leq m$.

   We can formulate this as linear search problem by transforming it to a search for the smallest with the following definition for f.

   $$f.x \;\equiv\; m \leq 2^{x+1}$$

3. Find the integer square root of m.

   We can formulate this as linear search problem by transforming it to a search for the smallest with the following definition for f.

   $$f.x \;\equiv\; m < (x+1)^2$$

4. Search the array $b[0..N)$ for a value z known to occur in the array.

   We can formulate this as linear search problem by defining f as follows.

   $$f.x \;\equiv\; b.x = z$$

6 Replacing a Constant by a Variable

6.1 Heuristic

For a simple loop with postcondition R, we want a loop invariant P and guard B such that:

$$P \wedge \neg B \Rightarrow R$$

If R is a conjunction, then we can often use the heuristic "deleting a conjunct" that we examined in Section 5.

If R is not a conjunction, then we can often transform it into one. One quite useful heuristic is called "replacing a constant by a variable".

Strategy: The method we call *replacing a constant by a variable* has the following general steps.

1. *Identify a constant in postcondition R and replace it by a fresh variable.* By "fresh variable", we mean a variable with a name that is not currently defined in that program.
2. *Add a new conjunct to R, equating the new variable to the constant replaced.*
3. *Now apply the "deleting a conjunct" method.* Usually it is the conjunct just added that is deleted.
4. *Strengthen the invariant to make sure the expression with the new variable is defined.* This usually means to give the domain for the valid values of the fresh variable.

By adding a fresh variable, we *extend* the program's state space. At the end of the execution of the loop, the projection of the program's state space back to the original variables yields a state space that satisfies the original postcondition.

Suppose we have a program specification with postcondition

$$R : x = f.N$$

in which, x is a program variable, f is a function (i.e., some expression), and N is a constant.

1. We identify N as the constant to be replaced.
2. We replace N by a fresh variable n by transforming the postcondition into:

$$R' : n = N \wedge x = f.n$$

3. Now we can apply the "deleting a conjunct" heuristic, obtaining the candidate loop invariant P_0 and guard B as follows:

   $P_0 : x = f.n$
   $B : n \neq N$

4. Finally, we need to make sure that the invariant P_0 and the guard B are well defined. For instance, suppose that f is a function defined on the integer interval $[0..N]$. Then we add a new conjunct P_1 to our invariant. We now have a loop invariants with two conjuncts.

   $P_0 : x = f.n$
   $P_1 : 0 \leq n \leq N$

Thus, we have selected an invariant P where:

$$P : P_0 \wedge P_1$$

We often write an invariant (or other assertion) in several parts as we determine the need for each part and denote the various parts with natural number subscripts on the same base name. The full invariant is the conjunction of the various parts, as we see above for invariant P.

6.2 Example: Array Summation

Consider the array summation program we examined in Section refsec:Sum. It had the specification:

|[**con** $b[0..N)$: **array of** int $\{Q\}$; **var** $s : int$;
Summation
$\{R\}$
]|

where Q and R are defined:

$Q : N \geq 0$
$R : s = (\Sigma i : 0 \leq i < N : b.i)$

We speculate that the body of summation is a simple loop plus any needed initialization and termination code. Our reasoning is as follows:

- The form of postcondition R is a singly quantified expression without any nested quantifications.
- The form of a solution is often similar to the form of the specification.
- The addition operation requires each element to be touched once.

Thus, let's apply the "replace constant by a variable" heuristic. If we examine the postcondition

$R : s = (\Sigma i : 0 \leq i < N : b.i)$

we note that both 0 and N are obvious candidate constants.

Then, we choose to replace N by fresh variable n in the postcondition to get:

$R' : n = N \wedge s = (\Sigma i : 0 \leq i < n : b.i)$

Now we can delete the $n = N$ conjunct we just added and obtain the following candidate invariant P_0 and guard B:

$P_0 : s = (\Sigma i : 0 \leq i < n : b.i)$
$B : n \neq N$

However, we need to make sure that the reference to $b.n$ is defined, so we add an invariant that restricts the values of n to b's defined domain:

$P_1 : 0 \leq n \leq N$

So the invariant P is $P_0 \wedge P_1$ and the form of the program solution is the following, where E and F are unknown expressions and S_1 is an unknown command:

$|[$ **con** $b[0..N)$ **array of** int $\{Q\}$; **var** $s : int$;
$s, n := E, F$;
{ invariant P }
do $n \neq N \rightarrow S_1$ **od**
{ $P \wedge n = N$, hence R }
$]|$

The choices for E and F are obvious (both 0). But for completeness let's calculate them.

Assume precondition Q. (That is, $N \geq 0$.)

$wp.(s, n := E, F).P$
$\equiv$ $\langle$ assignment axiom $\rangle$
$0 \leq F \leq N \wedge E = (\Sigma i : 0 \leq i < F : b.i)$
$\equiv$ $\langle$ choose $F = 0$ because of $n \neq N$ guard and it makes Σ easy to calculate $\rangle$
$0 \leq 0 \leq N \wedge E = (\Sigma i : 0 \leq i < 0 : b.i)$
$\equiv$ $\langle$ arithmetic $\rangle$
$N \geq 0 \wedge E = 0$
$\equiv$ $\langle$ assumption $N \geq 0$ $\rangle$
$E = 0$

This choice is obvious. Usually we won't go to the trouble to calculate in such a situation.

Now let's focus on the loop body S_1. Since addition must proceed one element at a time, the following choices seem reasonable:

bound function $t : N - n$

progress command $n := n + 1$

invariant maintenance $s := G$, for some expression G

Now calculate for G in $s, n := G, n + 1$, assuming that $P \wedge n \neq N$ (the loop body precondition) holds. (That is, $s = (\Sigma i : 0 \leq i < n : b.i) \wedge 0 \leq n \leq N \wedge n \neq N$.)

$wp.(s, n := G, n + 1).P$
$\equiv$ $\langle$ assignment axiom $\rangle$
$0 \leq n + 1 \leq N \wedge G = (\Sigma i : 0 \leq i < n + 1 : b.i)$
$\equiv$ $\langle$ assumptions $P_1 \wedge n \neq N$, range split, one-point for $i = n$ $\rangle$
$G = (\Sigma i : 0 \leq i < n : b.i) + b.n$
$\equiv$ $\langle$ assumption $P_0 \wedge P_1 \wedge n \neq N$ $\rangle$
$G = s + b.n$

Thus command S_1 is the assignment "$s, n := s + b.n, n + 1$" and we have the following program:

$|[$ **con** $b[0..N)$: **array of** int $\{Q\}$; **var** $s : int$;
  $|[$ **var** $n : int$;
    $s, n := 0, 0$;
    $\{$inv. $P : 0 \leq n \leq N \wedge s = (\Sigma i : 0 \leq i < n : b.i)$, bound $t : N - n\}$
    do $n \neq N \rightarrow s, n := s + b.n, n + 1$ **od**
    $\{P \wedge n \neq N$, hence $R\}$
  $]|$
  $\{R\}$
$]|$

Note: Here we added an inner block for the new variables added by the heuristic. In most of the examples that follow, we just add the new variable to the declarations at the beginning of the outer block unless we wish to restrict its visibility to a portion of the program.

We can now reexamine the process we used and the resulting program and make a few observations.

- If we desired to do so, we could easily reconstruct a formal proof of the program from the derivation.
- The time complexity of the program is $\mathcal{O}(N)$.
- If we look back at the postcondition R, we see that the choice of the variable to replace was arbitrary.

On the third point, if we choose to replace the constant 0, instead of N, in

$$R : s = (\Sigma i : 0 \leq i < N : b.i)$$

we obtain a different (but equivalent) program. The strengthened postcondition would be:

$$R'' : n = 0 \wedge s = (\Sigma i : n \leq i < N : b.i)$$

This leads to the following initial choices for the invariant P_0' and guard B:

$$P_0' : s = (\Sigma i : n \leq i < N : b.i)$$
$$B : n \neq 0$$

Again, we would add an invariant to make sure that P_0 is defined:

$$P_1' : 0 \leq n \leq N$$

Thus the invariant is

$$P' : P_0' \wedge P_1'$$

and it is easy to initialize with command "$s, n := 0, N$".

Given the initialization, the bound function t is just expression n and the progress command would be $n := n - 1$. A calculation of the loop body as above would result in the following program fragment for the loop:

$s, n := 0, N;$
$\{$inv. P', bound $t : n\}$
do $n \neq 0 \rightarrow s, n := s + b.(n-1), n-1$ **od**
$\{P' \wedge n = 0$, hence $R\}$

7 Strengthening the Invariant

7.1 Heuristic

A third useful heuristic is called *strengthening the invariant.* It is used in conjunction with the other methods to refine the invariant for a loop.

The strengthening the invariant heuristic is normally applied in the following way:

1. Select a candidate loop invariant and progress command using another technique such as deleting a conjunct.
2. Use calculation to identify the commands needed to re-establish the invariant.
3. If this leads to an expression E that cannot be expressed in terms of the program variables, add a fresh variable.
4. Add a new conjunct to the invariant that required the new variable to equal E.
5. Change the initialization to give a value to the new variable that establishes the new conjunct initially.
6. Use calculation to identify the commands needed to reestablish the invariant (including the new conjunct just added) within the loop body.

7.2 Example: Fibonacci Function (2nd Order)

In this section, let's consider a program to compute the value of the second order Fibonacci function at some natural number value.

For natural number arguments, the second-order Fibonacci function *fib* can be defined as follows:

$$fib.n = \begin{cases} 0, & \textbf{for } n = 0 \\ 1, & \textbf{for } n = 1 \\ fib.(n-2) + fib.(n-1), & \textbf{for } n > 1 \end{cases}$$

The desired program has the following specification, with the restriction that the function *fib* not be called directly:

$|[$ **con** $N : int$ $\{Q\}$; **var** $x : int$;
Fibonacci
$\{R\}$
$]|$

where Q and R are defined

$Q : N \geq 0$
$R : x = fib.N$

Given a definition of *fib* that is in terms of a double recursion, it is clear that the program needs to involve a loop. Also, we might not expect a particularly efficient algorithm to compute the result.

If we replace constant N by fresh variable n, we obtain the following postcondition:

$R' : n = N \wedge x = fib.n$

Applying the deleting a conjunct heuristic, we get an invariant P with the conjuncts P_0 and P_1 and a guard B:

$P_0 : x = fib.n$
$P_1 : 0 \leq n \leq N$
$B : n \neq N$

It is easy to see from the definition of *fib* that the following initialization command is appropriate:

$x, n := 0, 0$

Given the initialization and invariant P_1, the following bound function t and progress command seem appropriate:

$$t : N - n$$
$$\text{progress} : \; n := n + 1$$

Thus, so far, we have derived the following program, for some unknown expression E:

$$\begin{array}{ll} |[& \textbf{con } N : int \; \{Q\} \; ; \; \textbf{var } x, n : int \; ; \\ & x, n := 0, 0; \\ & \{ \text{ invariant } P \; \} \\ & \textbf{do } n \neq N \rightarrow x, n := E, n + 1 \textbf{ od} \\ & \{ \; P \wedge n = N, \text{ hence } R \; \} \\ ]| & \end{array}$$

Now let's use a calculation on the loop body to find the unknown expression E.

Assume loop body precondition $P \wedge n \neq N$. (That is, $x = fib.n \wedge 0 \leq n \leq N \wedge n \neq N$.)

$$\begin{array}{ll} & wp.(x, n := E, n + 1).P \\ \equiv & \quad \langle \text{ assignment axiom } \rangle \\ & E = fib.(n + 1) \wedge 0 \leq n + 1 \leq N \end{array}$$

Now $fib.(n + 1) = fib.(n - 1) + fib.n$ for $n > 0$, but $fib.(n - 1)$ is not defined for $n = 0$. Also $fib.(n - 1)$ is not defined in the invariant. Thus we seem stuck at this point in our simplification.

Thus we strengthen the invariant to solve this problem. We add a fresh variable y to hold the value of $fib.(n + 1)$, which is the expression we do not know how to restate in terms of previously known variables. That is, we add an invariant:

$$P_2 : y = fib.(n + 1)$$

(Henceforth, let $P' \equiv P \wedge P_2$.)

Now the new initialization command for P' is

$$x, y, n := 0, 1, 0$$

and the command to maintain this invariant becomes

$$x, y, n := E, F, n + 1$$

where both E and F are unknown expressions.

Now we can again try calculating the unknown loop body expressions.

Assume $P' \wedge n \neq N$. (That is, $x = fib.n \wedge 0 \leq n \leq N \wedge y = fib.(n+1) \wedge n \neq N$.)

$$
\begin{array}{ll}
 & wp.(x, y, n := E, F, n+1).P' \\
\equiv & \quad \langle \text{ assignment axiom } \rangle \\
 & E = fib.(n+1) \wedge 0 \leq n+1 \leq N \wedge F = fib.(n+2) \\
\equiv & \quad \langle \text{ assumptions } P_2 \text{ and } n \neq N, \text{ def. } fib \rangle \\
 & E = y \wedge F = fib.n + fib.(n+1) \\
\equiv & \quad \langle \text{ assumptions } P_0 \wedge P_2 \rangle \\
 & E = y \wedge F = x + y
\end{array}
$$

Thus we have derived the following program to compute $fib.N$:

$$
\begin{array}{l}
|[\quad \textbf{con } N : int \ \{Q\} \ ; \ \textbf{var } x, y : int \ ; \\
\quad x, y, n := 0, 1, 0; \\
\quad \{ \text{ invariant } P' \ \} \\
\quad \textbf{do } n \neq N \rightarrow \\
\qquad x, y, n := y, \ x + y, \ n + 1 \\
\quad \textbf{od} \\
\quad \{P' \wedge n \neq N, \text{hence } x = fib.N \ \} \\
]|
\end{array}
$$

Note that the time complexity is $\mathcal{O}(N)$. It is a linear algorithm to compute what did not seem initially to have an efficient solution.

The above programming technique is similar to the functional programming techniques called *accumulating parameters* and *tupling*. The technique can be used to improve efficiency by computing some values incrementally, storing values computed in one iteration of a loop to compute updated values needed in the next iteration.

7.3 Example: Evaluating a Polynomial

Consider a program to evaluate a polynomial whose coefficients are stored in an array. A specification is:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ c[0..N)\ \textbf{array of}\ int\ \ \{Q\}\ ;\ \textbf{con}\ x : int\ ;\ \textbf{var}\ f : int\ ; \\
\quad \text{Poly} \\
\quad \{\ R\ \} \\
]|
\end{array}
$$

where

$$
\begin{array}{l}
Q : N \geq 0 \\
R : f = (\Sigma i : 0 \leq i < N : c.i * x^i)
\end{array}
$$

Clearly, we need a loop to compute the desired value in general. If we replace constant N by a fresh variable n, we get the following invariant P and guard B for a simple loop:

$$
\begin{array}{ll}
P_0 : f = (\Sigma i : 0 \leq i < n : c.i * x^i) & \\
P_1 : 0 \leq n \leq N & P : P_0 \wedge P_1 \\
B : n \neq N &
\end{array}
$$

Clearly, the initialization command "$f, n := 0, 0$" seems appropriate to establish the invariant P in a limiting case.

Given invariant P_1 and the guard $n \neq N$, it is not difficult to see that an appropriate bound function t and progress command are the following:

$$
\begin{array}{l}
t : N - n \\
\text{progress} :\ n := n + 1
\end{array}
$$

For some unknown expression E, so far we have the following loop structure:

$$
\begin{array}{l}
f, n := 0, 0\ ; \\
\{\ \text{invariant}\ P\ \} \\
\textbf{do}\ n \neq N \rightarrow f, n := E, n + 1\ \textbf{od} \\
\{\ P \wedge n = N, \text{ hence } R\ \}
\end{array}
$$

Assume $P \wedge n \neq N$. (That is, $f = (\Sigma i : 0 \leq i < n : c.i * x^i) \wedge 0 \leq n \leq N \wedge n \neq N$.)

$$
\begin{array}{ll}
 & wp.(f, n := E, n + 1).P \\
\equiv & \quad \langle\ \text{assignment axiom}\ \rangle \\
 & E = (\Sigma i : 0 \leq i < n + 1 : c.i * x^i) \wedge 0 \leq n + 1 \leq N \\
\equiv & \quad \langle\ \text{assumptions } P_1 \wedge n \neq N\text{, range split and one-point for } i = n\ \rangle \\
 & E = (\Sigma i : 0 \leq i < n : c.i * x^i) + c.n * x^n \\
\equiv & \quad \langle\ \text{assumption } P_0\ \rangle \\
 & E = f + c.n * x^n
\end{array}
$$

This gives us a program that satisfies the specification, but the x^n term is inefficient. It involves n multiplies. Can we find a more efficient solution?

Sure. Note that exponentiation is just a shorthand for multiplication We should be able to complete x^n incrementally.

Thus let's strengthen the invariant. Let's add a fresh variable y and invariant

$$P_2 : y = x^n$$

Let $P' \equiv P \wedge P_2$.

We modify the initialization to be command "$f, y, n := 0, 1, 0$" and choose the new loop body command to be the following, where both E and F are unknown expressions:

$$f, y, n := E, F, n + 1$$

Now we calculate on the loop body again to seek definitions for the two unknowns.

Assume $P' \wedge n \neq N$. (That is, $f = (\Sigma i : 0 \leq i < n : c.i * x^i) \wedge y = x^n \wedge 0 \leq n \leq N \wedge n \neq N$.)

$$\begin{array}{ll} & wp.(f, y, n := E, F, n + 1).P' \\ \equiv & \quad \langle \text{ assignment axiom } \rangle \\ & E = (\Sigma i : 0 \leq i < n + 1 : c.i * x^i) \wedge F = x^{n+1} \wedge 0 \leq n + 1 \leq N \\ \equiv & \quad \langle \text{ assumptions } P_2 \text{ and } n \neq N, \text{ range split and one-point for } i = n \ \rangle \\ & E = (\Sigma i : 0 \leq i < n : c.i * x^i) + (c.n) * x^n \wedge F = x^{n+1} \ \rangle \\ \equiv & \quad \langle \text{ assumptions } P_0 \text{ and } P_2 \ \rangle \\ & E = f + (c.n) * y \wedge F = x * x^n \\ \equiv & \quad \langle \text{ choose } E = f + c.n * y, \text{ assumption } 0 \leq n, \text{ arithmetic } \rangle \\ & F = x * x^n \\ \equiv & \quad \langle \text{ assumption } P_2 \ \rangle \\ & F = x * y \end{array}$$

Therefore, we have derived the following program:

$$\begin{array}{l} |[\quad \textbf{con } c[0..N) \textbf{ array of } int \ \{Q\} \ ; \textbf{ con } x : int \ ; \textbf{ var } f, n, y : int \ ; \\ \quad f, y, n := 0, 1, 0; \\ \quad \{ \text{ invariant } P' \ \} \\ \quad \textbf{do } n \neq N \rightarrow \\ \qquad f, y, n := f + c.n * y, x * y, n + 1; \\ \quad \textbf{od } \ \{ \ P' \wedge n = N, \text{ hence } R \ \} \\ ]| \end{array}$$

Now let us examine this program. An execution of the program uses $2 * N$ multiplies and $2 * N$ additions (half of them being incrementing $n + 1$). Can we find a more efficient version?

Let's go back to beginning and look for new opportunities to exploit. The postcondition is:

$$R : f = (\Sigma i : 0 \leq i < N : c.i * x^i)$$

What constants other than N might we replace?

1. We could replace 0 by a fresh variable n.

   But this seems no more efficient than replacing N. Moreover the sequence of calculations $x^{N-1} \cdots x_0$ seems to defeat the optimization we used for the exponentiation.

2. We could look for another, less obvious, constant 0 to replace, in particular one that has something to do with the exponentiation.

If we rewrite the postcondition as below, we can see a "hidden" constant 0 in the exponent:

$$R : f = (\Sigma i : 0 \leq i < N : c.i * x^{i-0})$$

Let us proceed by replacing both the lower bound 0 and the newly uncovered constant 0 in the exponent by a fresh variable n, getting the new postcondition:

$$R' : n = 0 \wedge f = (\Sigma i : n \leq i < N : c.i * x^{i-n})$$

We delete the new postcondition conjunct $n = 0$ to get the candidate guard B, put the remainder of the postcondition in invariant S_0, and add invariant S_1 to make sure that the accesses to array c are well-defined. The resulting invariant S and guard B are:

$$\begin{array}{ll} S_0 : f = (\Sigma i : n \leq i < N : c.i * x^{i-n}) & \\ S_1 : 0 \leq n \leq N & S \equiv S_0 \wedge S_1 \\ B : n \neq 0 & \end{array}$$

It is easy to see that an appropriate initialization is to make the domain of the quantification "just empty":

$$f, n := 0, N$$

Given invariant S_1 and the initialization, the bound function t and progress command can be chosen as follows:

$$\begin{array}{l} t : n \\ \text{progress} : \ n := n - 1 \end{array}$$

For some unknown expression G, we speculate that the loop body includes the command "$f, n := G, n-1$" to guarantee progress and maintain the invariant. Let us calculate on this command for the unknown expression G.

Assume $S \wedge n \neq 0$. (That is, $f = (\Sigma i : n \leq i < N : c.i * x^{i-n}) \wedge 0 \leq n \leq N \wedge n \neq 0$.)

$$
\begin{array}{ll}
 & wp.(f, n := G, n-1).S \\
\equiv & \quad \langle \text{ assignment axiom } \rangle \\
 & G = (\Sigma i : n-1 \leq i < N : c.i * x^{i-(n-1)}) \wedge 0 \leq n-1 \leq N \\
\equiv & \quad \langle \text{ assumptions } S_1 \text{ and } n \neq 0, \text{ range splitting, one-point for } i = n-1 \rangle \\
 & G = (\Sigma i : n \leq i < N : c.i * x^{i-n+1}) + c.(n-1) * x^0 \\
\equiv & \quad \langle \text{ arithmetic, exponent } i-n+1 \geq 1, \text{ distribution } \rangle \\
 & G = (\Sigma i : n \leq i < N : c.i * x^{i-n}) * x + c.(n-1) \\
\equiv & \quad \langle \text{ assumption } S_0 \rangle \\
 & G = f * x + c.(n-1)
\end{array}
$$

Thus, we have derived the following program:

$$
\begin{array}{l}
|[\quad \textbf{con } c[0..N) \textbf{ array of } int \ \ \{Q\} \ ; \textbf{ con } x : int \ ; \textbf{ var } f, n : int \ ; \\
\quad f, n := 0, N \ ; \\
\quad \{ \text{ invariant } S \ \} \\
\quad \textbf{do } n \neq 0 \rightarrow \\
\qquad f, n := f * x + c.(n-1), n-1 \\
\quad \textbf{od } \ \{ \ S \wedge n = N, \text{ hence } R \ \} \\
]|
\end{array}
$$

Note: The subexpression "$n-1$" appears two places on the right-hand-side of the assignment. Most compilers will compute this value only once and store its value (in a register) for the second usage. Alternatively, we could sequence the command "$n := n-1$" in front of the update of f to achieve a similar effect. Thus we only count half of the subtractions.

Thus, this new program only needs N multiplies, N subtractions (which are all decrementing operations $n-1$), and N additions. Because addition and subtraction are faster operations than multiplication on most platforms, this algorithm is more efficient than the previous one. On some machines, incrementing and decrementing operations may be faster than other addition operations.

This program is an implementation of a technique that is sometimes called *Horner's Rule*:

$$((c_{N-1}x + c_{N-2})x + c_{N-3})x + ... + c_0$$

7.4 Example: Counting Pairs

In this subsection, we derive a program for the following specification:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ b[0..N) :\ \textbf{array of}\ int\ \{Q\}\ ;\ \textbf{var}\ r : int\ ; \\
\quad S \\
\quad \{\ R\ \} \\
]|
\end{array}
$$

where Q and R are defined

$$
\begin{array}{l}
Q : N \geq 0 \\
R : r = (\#\ i, j : 0 \leq i < j < N : b.i \leq 0 \wedge b.j \geq 0)
\end{array}
$$

There is a naive algorithm for this that checks all pairs (i, j) of values from the array b. Using comparisons as our metric, that algorithm would involve nested loops and be $\mathcal{O}(N^2)$.

Let us try to derive an $\mathcal{O}(N)$ program that uses a single loop. If we replace constant N by a fresh variable n in the postcondition R, we can derive the following invariant P and guard B:

$$
\begin{array}{ll}
P_0 : r = (\#\ i, j : 0 \leq i < j < n : b.i \leq 0 \wedge b.i \geq 0) & \\
P_1 : 0 \leq n \leq N & P : P_0 \wedge P_1 \\
B : n \neq N &
\end{array}
$$

Because of the precondition $N \geq 0$, the only value that makes P_1 true for all valid values of N is $n = 0$. Thus we choose the initialization command "$r, n := 0, 0$".

Given this initialization and invariant P_1, we choose the following for the bound function t and progress command in the loop:

$$
\begin{array}{l}
t : N - n \\
\text{progress: } n := n + 1
\end{array}
$$

If the program can be implemented as a single loop, it might have the following structure, where E is some unknown expression for the updated value of variable r:

$$
\begin{array}{l}
r, n := 0, 0\ ; \\
\{\text{invariant } P\ \} \\
\textbf{do}\ n \neq N \rightarrow \{P \wedge n \neq N\} \\
\quad r, n := E, n + 1 \\
\textbf{od}\ \{\ P \wedge n = N, \text{ hence } R\ \}
\end{array}
$$

Now let's try a calculation on the loop body with the invariant P as a postcondition to try to determine an appropriate expression for E.

Assume loop body precondition $P \wedge n \neq N$.
(That is, $r = (\#\, i, j : 0 \leq i < j < n : b.i \leq 0 \wedge b.i \geq 0) \wedge 0 \leq n \leq N \wedge n \neq N$.)

$$
\begin{array}{ll}
 & wp.(r, n := E, n+1).P \\
\equiv & \quad \langle \text{ assignment axiom } \rangle \\
 & E = (\#\, i, j : 0 \leq i < j < n+1 : b.i \leq 0 \wedge b.j \geq 0) \wedge 0 \leq n+1 \leq N \\
\equiv & \quad \langle \text{ assumption } P_1 \wedge n \neq N, \text{ range split and one-point for } j = n \ \rangle \\
 & E = (\#\, i, j : 0 \leq i < j < n : b.i \leq 0 \wedge b.j \geq 0) + (\#\, i : 0 \leq i < n : b.i \leq 0 \wedge b.n \geq 0) \\
\equiv & \quad \langle \text{ assumption } P_0 \ \rangle \\
 & E = r + (\#\, i : 0 \leq i < n : b.i \leq 0 \wedge b.n \geq 0) \\
\equiv & \quad \langle \text{ case analysis using assumption } P_1 \wedge n \neq N \text{ to guarantee guards defined,} \\
 & \quad \text{arithmetic, false term for first case, } \wedge\text{-identity for second } \rangle \\
 & \left\{ \begin{array}{ll} E = r, & \textbf{if } b.n < 0 \\ E = r + (\#\, i : 0 \leq i < n : b.i \leq 0), & \textbf{if } b.n \geq 0 \end{array} \right. \\
\equiv & \quad \langle \text{ strengthen invariant by adding fresh variable } s \text{ and invariant} \\
 & \quad P_2 : s = (\#\, i : 0 \leq i < n : b.i \leq 0) \ \rangle \\
 & \left\{ \begin{array}{ll} E = r, & \textbf{if } b.n < 0 \\ E = r + s, & \textbf{if } b.n \geq 0 \end{array} \right.
\end{array}
$$

Let invariant $P' \equiv P_0 \wedge P_1 \wedge P_2$.

Of course, we must initialize s so that P_2 holds. Clearly, the assignment "s := 0" will do that.

Next we need to consider how to update s in the loop body with r and n. That is, we will consider command "$r, s, n := E, F, n+1$" for the E calculated above and an unknown expression F. Because it is the only invariant affected by changes in the values of variable s, let's just look at P_2 as a postcondition.

Assume loop body precondition $P' \wedge n \neq N$.
That is, $r = (\#\, i, j : 0 \leq i < j < n : b.i \leq 0 \wedge b.i \geq 0) \wedge 0 \leq n \leq N \wedge$
$s = (\#\, i : 0 \leq i < n : b.i \leq 0) \wedge n \neq N$.)

$$
\begin{array}{ll}
 & wp.(r, s, n := E, F, n+1).P_2 \\
\equiv & \quad \langle \text{ assignment axiom } \rangle \\
 & F = (\#\, i : 0 \leq i < n+1 : b.i \leq 0) \\
\equiv & \quad \langle \text{ assumption } P_1, \text{ range split and one-point for } i = n \ \rangle \\
 & F = (\#\, i : 0 \leq i < n : b.i \leq 0) + \#.(b.n \leq 0) \\
\equiv & \quad \langle \text{ assumption } P_2 \ \rangle \\
 & F = s + \#.(b.n \leq 0) \\
\equiv & \quad \langle \text{ case analysis using assumptions } P_1 \wedge n \neq N \text{ to guarantee guards defined,} \\
 & \quad \text{arithmetic } \rangle \\
 & \left\{ \begin{array}{ll} F = s, & \textbf{if } b.n > 0 \\ F = s + 1, & \textbf{if } b.n \leq 0 \end{array} \right.
\end{array}
$$

Combining the case analyses above, we see there are three cases to consider for the value of $b.n$ with respect to 0:

$$\begin{cases} E = r + s \wedge F = s, & \textbf{if } b.n > 0 \\ E = r + s \wedge F = s + 1, & \textbf{if } b.n = 0 \\ E = r \wedge F = s + 1, & \textbf{if } b.n < 0 \end{cases}$$

Instead of an assignment statement, the cases analyses have yielded an **if** command with three legs having different assignments. Thus the program derived is as follows:

```
|[  con b[0..N) :  array of int {Q} ; var r, s, n : int ;
    r, s, n := 0, 0, 0 ;
    { invariant P', bound t : N − n };
    do n ≠ N →
        if b.n > 0 → r, n := r + s, n + 1
        [] b.n = 0 → r, s, n := r + s, s + 1, n + 1
        [] b.n < 0 → s, n := s + 1, n + 1
        fi
    od { P' ∧ n = N, hence R}
]|
```

The program makes one comparison for each pass through the loop. It goes through the loop N times. The time complexity of the program is thus $\mathcal{O}(N)$.

By systematically applying the program derivation heuristics to explore possible program structures, we found an efficient linear algorithm for a problem that, on initial examination, seemed to have only an $\mathcal{O}(N^2)$ algorithm.

7.5 Example: Binary Search (General)

This section considers a generalization of the *binary search* problem that is familiar from the undergraduate data structures and algorithms courses. Its specification is as follows:

$|[$ **con** $N : int$ $\{Q\}$; **function** $b[0..N) : boolean$; **var** $x : int$;
Search
$\{ R \}$
$]|$

where Q and R are defined

$Q : N \geq 0$
$R : b.x \wedge \neg b.(x+1) \wedge -1 \leq x \wedge x < N$

Nothing is assumed about the value of function b. For example, we do not assume that its values are ascending for increasing values of the argument.

Problem: What if $N = 0$, if b is everywhere $false$, or if b is everywhere $true$? In these cases, R would then include undefined references to $b.(n-1)$ and/or $b.N$.

Problem Resolution: We resolve this problem by strengthening the precondition and range of b to enable us to derive a program. We add dummy values just beyond the bounds of array b as defined in precondition Q'.

$Q' : Q \wedge b.(-1) \wedge \neg b.N$

Constraint: The special function values $b.(-1)$ and $b.N$ cannot be referenced in a program. They are for specification and reasoning purposes only.

Now postcondition R can be satisfied in all cases.

Do we need a loop? Yes, because we must search the function's domain for values that satisfy R. There is no way to go to such a state in a fixed number of steps.

Can we delete a conjunct to arrive at a candidate invariant for a loop?

- In general, $b.x$ and $\neg b.(x+1)$ cannot both be in the invariant because there is no way to initialize the invariant when both are present.
- Deleting one of $b.x$ or $\neg b.(x+1)$ will lead us to a linear solution. Let's look for a more efficient solution.

Let's try replacing a constant by a variable.

- Because of the problem noted above for $b.x$ and $\neg b.(x+1)$, we need to modify one of them to find another solution.

- We could replace the "1" in $x+1$ by a fresh variable.
- But, instead, let's use a more general technique. Let's replace expression $x+1$ by fresh variable y.
- The range of the new variable y should be the same as the range for expression $x+1$.

  The strengthened postcondition resulting from this transformation is:

  $$R' : b.x \wedge \neg b.y \wedge -1 \leq x \wedge x < N \wedge y = x+1$$

Proceeding with the derivation, we delete conjunct $y = x+1$ and obtain the following invariant P and guard B:

$$P_0 : b.x \wedge \neg b.y$$
$$P_1 : -1 \leq x \wedge x < y \wedge y \leq N$$
$$B : y \neq x+1 \qquad\qquad P : P_0 \wedge P_1$$

Therefore, we guess that the program has the following structure for unknown expression E and F and an unknown command S_1:

$\{Q'\}$
$x, y := E, F$;
{ invariant P }
do $y \neq x+1 \rightarrow \{P \wedge y \neq x+1\}$
  S_1
od $\{P \wedge y = x+1$, hence $R\}$

Now let's calculate initialization expressions E and F.

Assume precondition Q'. (That is, $N \geq 0 \wedge b.(-1) \wedge \neg b.N$.)

$$\begin{array}{ll} & wp.(x, y := E, F).P \\ \equiv & \quad \langle \text{ assignment axiom } \rangle \\ & b.E \wedge \neg b.F \wedge -1 \leq E \wedge E < F \wedge F \leq N \\ \equiv & \quad \langle \text{ choose } E = -1 \wedge F = N \rangle \\ & b.(-1) \wedge \neg b.N \wedge -1 \leq -1 \wedge -1 \leq N \wedge N \leq N \\ \equiv & \quad \langle \text{ assumption } Q' \rangle \\ & true \end{array}$$

Thus the initialization command is "$x, y := -1.N$".

Next let's focus on the loop body. Given the initialization and invariant P_1, it is appropriate to define the bound function t as $y - x$.

What about a command to guarantee progress? A command to decrease y, to increase x. or to do both would work. Let's choose to either decrease or increase, but not both.

Let m be a value such that:

$$x < m < y$$

This value exists because $x < y$ is guaranteed by invariant P_1 and $y \neq x + 1$ because of the guard.

Therefore, we determine that loop body command S_1 has the following structure for some unknown expression G and guards B_0 and B_1:

$$\begin{array}{l} |[\quad \textbf{var } m : int \ \{P \wedge y \neq x + 1\}; \\ \quad\ m := G \ \{P \wedge y \neq x + 1 \wedge x < m < y\}; \\ \quad\ \textbf{if } B_0 \rightarrow x := m \\ \quad\ [] \ B_1 \rightarrow y := m \\ \quad\ \textbf{fi } \ \{ \ P \ \} \\ ]| \end{array}$$

Note that this program decreases the bound function $y - x$ by some value that is at least one because $x < m < y$.

Now let's calculate B_0, assuming $P \wedge y \neq x + 1 \wedge x < m < y \wedge B_0$. (Remember that P is $b.x \wedge \neg b.y \wedge -1 \leq x \wedge x < y \wedge y \leq N$.)

$$\begin{array}{ll} & wp.(x := m).P \\ \equiv & \quad \langle \ \text{assignment axiom} \ \rangle \\ & b.m \wedge \neg b.y \wedge -1 \leq m \wedge m < y \wedge y \leq N \\ \equiv & \quad \langle \ \text{assumptions } P_0, \ P_1, \ x < m, \ m < y \ \rangle \\ & b.m \\ \equiv & \quad \langle \ \text{choose } B_0 \equiv b.m \ \rangle \\ & true \end{array}$$

Now let's calculate B_1, assuming $P \wedge y \neq x + 1 \wedge x < m < y \wedge B_1$.

$$\begin{array}{ll} & wp.(y := m).P \\ \equiv & \quad \langle \ \text{assignment axiom} \ \rangle \\ & b.x \wedge \neg b.m \wedge -1 \leq x \wedge x < m \wedge m \leq N \\ \equiv & \quad \langle \ \text{assumptions } P_0, \ P_1, \ x < m, \ m < y \ \rangle \\ & \neg b.m \\ \equiv & \quad \langle \ \text{choose } B_1 \equiv \neg b.m \ \rangle \\ & true \end{array}$$

Note that $P \wedge y \neq x + 1 \wedge x < m < y \Rightarrow def.(b.m) \wedge (b.m \vee \neg b.m)$. That is, the preconditions for the **if** guarantee that $0 \leq m < N$. Hence, the **if** will not abort.

Next consider the command $m := G$. We must choose $x < G < y$. What are the possibilities: $x + 1$? $y - 1$? $(x + y)$ **div** 2?

We choose $G = (x + y)$ **div** 2 for symmetry and to seek a loop that terminates more quickly.

Thus the assignment is "$m := (x + y)$ **div** 2". We could use "$m := x + (y - x)$ **div** 2 if integer arithmetic overflow can occur.

We have derived the following program:

```
|[  con N : int {Q}; function b[0..N) : boolean; var x, y : int;
    x, y := −1, N;
    { invariant P, bound t : Y − x };
    do y ≠ x + 1 →
        |[  var m : int { P ∧ y ≠ x + 1 };
            m := (x + y) div 2;
            { P ∧ y ≠ x + 1 ∧ x < m < y }
            if   b.m → x := m
            []   ¬b.m → y := m
            fi
        ]|
    od { P ∧ y = x + 1, hence R}
]|
```

Note that each repetition halves $y - x$. Thus this algorithm does $\mathcal{O}(log_2(N + 1))$ repetitions of the loop.

This general program is quiet useful. It can be specialized in many different ways by defining an appropriate function f.

7.6 Example: Application of Binary Search (Square Root)

Suppose we have the following problem to solve: For $m \geq 0$, compute $\sqrt{m}$ as an integer. Do not use the $\sqrt{\ }$ operation.

We can formulate the desired postcondition as:

$$R : x \leq \sqrt{m} \ \wedge \ \sqrt{m} \ < \ x + 1$$

Given that $m \geq 0$, we can eliminate the $\sqrt{\ }$ operator and restate this postcondition as:

$$R' : x^2 \leq m \wedge m < (x+1)^2$$

R' resembles the postcondition of the general binary search program. We can transform it to binary search by choosing function b and upper bound N appropriately.

To transform a problem Z to a binary search problem, we have the following requirements:

1. The precondition of Z must logically imply the precondition of the general binary search program.
2. The postcondition of the binary search program must logically imply the postcondition of Z.

In this case, Z is the integer square root problem described above. Let's examine the postcondition of binary search to see if we can select b and N so that it can imply the postcondition of the integer square root problem.

$$\begin{array}{ll} & b.x \wedge \neg b.(x+1) \wedge -1 \leq x \wedge x < N \\ \equiv & \quad \langle \ \text{choose } b.x \equiv ((nat.x)^2 \leq m), \text{ Note 1} \ \rangle \\ & x^2 \leq m \wedge m < (x+1)^2 \wedge -1 \leq x \wedge x < N \\ \Rightarrow & \quad \langle \ \text{conjunct simplification} \ \rangle \\ & R' \end{array}$$

Note 1: The Cohen textbook gives the definition $b.x \equiv (x^2 \leq m)$. However, there is a small problem with that definition when $x = -1$ and $m = 0$, then $(-1)^2 \leq 0$ is false. So, instead, we define $b.x \equiv ((nat.x)^2 \leq m)$ where:

$$\begin{array}{l} nat.x = \textbf{if } x \geq 0 \rightarrow x \\ \qquad\quad [] \ x < 0 \rightarrow 0 \\ \qquad\quad \textbf{fi} \end{array}$$

Next consider the precondition of the integer square root problem. We must have:

$$m \geq 0 \Rightarrow N \geq 0 \wedge b.(-1) \wedge \neg b.N$$

Assume $m \geq 0$.

$$\begin{array}{ll} & N \geq 0 \wedge b.(-1) \wedge \neg b.N \\ \equiv & \quad \langle \text{ previous choice } b.x \equiv ((nat.x)^2 \leq m) \ \rangle \\ & N \geq 0 \wedge (nat.(-1))^2 \leq m \wedge m < N^2 \\ \equiv & \quad \langle \text{ choose } N = m+1, \text{ smallest } N \text{ such that } N \geq 0 \text{ and } m < N^2, \ \rangle \\ & m+1 \geq 0 \wedge 0 \leq m \wedge m < (m+1)^2 \\ \equiv & \quad \langle \text{ assumption } m \geq 0, \text{ arithmetic } \rangle \\ & true \end{array}$$

Now just replace $b.m$ and N in the binary search program by $x^2 \leq m$ and $m+1$, respectively, and the binary search program becomes a solution to the integer square root problem.

7.7 Example: Binary Search for m

This subsection looks at another application of the binary search, the common application that we studied in the undergraduate data structures or algorithms course.

Suppose we are given an integer m and an integer array $f[0..L)$ for $L \geq 0$. The goal is to set boolean variable p such that postcondition RR holds:

$$RR : p \equiv (\exists\, k : 0 \leq k < L : f.k = m)$$

Clearly, we must determine where m occurs to set p properly. That is, we must find an x such that:

$$f.x = m \wedge 0 \leq x < L$$

But this is too strong if x does not occur in f. Thus we weaken the first conjunct above and get the following requirement:

$$R : f.x \leq m \wedge m < f.(x+1) \wedge 0 \leq x < L$$

Now, if f is ascending, then x is either where m occurs or where it should occur. Thus the program

$$\text{Establish } R \;;\; p := (f.x = m)$$

will establish RR provided "f is ascending" is a precondition of the assignment to p.

But there is a problem with this solution if $L = 0$, $m < f.0$, or $m > f.(L-1)$. The solution to this problem is to weaken R to R'

$$R' : f.x \leq m \wedge m < f.(x+1) \wedge -1 \leq x < L$$

where $f.(-1)$ and $f.L$ are given values $-\infty$ and $+\infty$ if f is ascending.

Thus, for f ascending, the program becomes:

Establish R' ;
if $0 \leq x \quad \rightarrow p := f.x = m$
$[]\ \ x = -1 \rightarrow p := false$
fi

Now, establishing R' is an application of the general binary search. We can transform the specifications to show this.

First, consider the postcondition of the general binary search to find definitions for b and N that are needed for the transformation. Remember that the postcondition of the general binary search must logically imply the postcondition of the search for m problem.

$$
\begin{array}{ll}
 & b.x \wedge \neg b.(x+1) \wedge -1 \leq x \wedge x < N \\
\equiv & \quad \langle \text{ choose } b.x \equiv f.x \leq m \ \rangle \\
 & f.x \leq m \wedge m < f.(x+1) \wedge -1 \leq x \wedge x < N \\
\equiv & \quad \langle \text{ choose } N \geq L \ \rangle \\
 & R'
\end{array}
$$

Next, consider the precondition of binary search. It must be logically implied by the precondition of the search for m problem.

$$
\begin{array}{ll}
 & b.(-1) \wedge \neg b, N \wedge 0 \leq N \\
\equiv & \quad \langle \text{ previous choice } b(x) \equiv f.x \leq m \ \rangle \\
 & f.(-1) \leq m \wedge f.N > m \wedge 0 \leq N \\
\equiv & \quad \langle \text{ choose } N = L, \text{ since we know nothing about } f \ \rangle \\
 & f.(-1) \leq m \wedge f.L > m \wedge 0 \leq L \\
\equiv & \quad \langle \text{ fictitious } f.(-1) \text{ and } f.L \text{ set appropriately, e.g., } -\infty \text{ and } +\infty \text{ for ascending } f \ \rangle \\
 & 0 \leq L \\
\equiv & \quad \langle \text{ search for } m \text{ program precondition } \rangle \\
 & \mathit{true}
\end{array}
$$

Thus, we have made the following choices:

- $b.m$ is replaced by $f.m \leq m$.
- N replaced by L.

8 Tail Recursion (Tail Invariants)

8.1 Heuristic

Let the following be functions of the specified type

$H : type1 \to type2$
$b : type1 \to boolean$
$c : type1 \to type2$
$d : type1 \to type1$
$t : type1 \to int$

such that

$(\forall\, x : x \in \mathbf{dom}\ H : x \in \mathbf{dom}\ b)$
$(\forall\, x : x \in \mathbf{dom}\ H \wedge \neg b.x : x \in \mathbf{dom}\ c)$
$(\forall\, x : x \in \mathbf{dom}\ H \wedge b.x : x \in \mathbf{dom}\ d \wedge d.x \in \mathbf{dom}\ H)$
$(\forall\, x : x \in \mathbf{dom}\ H \wedge b.x : t.x \geq 0)$ — boundedness
$(\forall\, x : x \in \mathbf{dom}\ H \wedge b.x : t.(d.x) < t.x)$ — progress

and

$H.x =$ **if** $\neg b.x \to c.x$
$\quad [\!]\quad b.x \to H.(d.x)$
fi

where H does not appear in the definitions of b, c, or d.

A function definition H that meets the above criteria has a *tail recursive* definition.

Consider a program with the following specification:

$|[$ **con** $X : type1$ { $X \in \mathbf{dom}\ H$ }; **var** $r : type2$;
evaluate H
{ $r = H.X$ }
$]|$

A general program for the above specification is:

$|[$ **con** $X : type1$ $\{X \in \mathbf{dom}\ H\}$; **var** $x : type1$; **var** $r : type2$;
$x := X$
{invariant $P : H.x = H.X \wedge x \in \mathbf{dom}\ H$, bound t };
do $b.x \to$
$\quad x := d.x$
od
{ $P \wedge \neg b.x$, hence $c.x = H.X$}
$r := c.x$
$\{r = H.X\}$
$]|$

Requirement: t must have the required boundedness and progress properties.

Note: The $x \in \mathbf{dom}\ H$ conjunct of the invariant might be left off if H is defined for all values of $type1$.

Solving a problem by tail recursion involves finding a function H with a definition of the proper shape.

An invariant P of the form $H.x = H.X$ is sometimes called a *tail invariant*.

8.2 Example: Finding the Maximum

Consider a program to find the maximum value in an array.

$$
\begin{array}{l}
|[\ \ \textbf{con}\ a[0..N]:\ \textbf{array of}\ int\ \{N \geq 0\};\ \textbf{var}\ r : int; \\
\quad \text{Maxval} \\
\quad \{\ r = (\textbf{MAX}\, i : 0 \leq i \leq N : a.i)\ \} \\
]|
\end{array}
$$

For $0 \leq x \leq y \leq N$, define:

$$F.x.y = (\textbf{MAX}\ i : x \leq i \leq y : a.i)$$

The postcondition of the Maxval program can be restated as $r = F.0.N$.

F can be given the following tail recursive definition:

$$
\begin{array}{l}
F.x.y = \textbf{if}\ x = y \rightarrow a.x \\
\qquad [\!]\ x < y \rightarrow \textbf{if}\ a.x \leq a.y \rightarrow F.(x+1).y \\
\qquad\qquad\qquad\qquad [\!]\ a.y \leq a.x \rightarrow F.x.(y-1) \\
\qquad\qquad\qquad\qquad \textbf{fi} \\
\qquad \textbf{fi}
\end{array}
$$

If we wish to prove that the recursive definition is equivalent to the quantified definition, we can use induction on the length of the array.

Now we use the tail recursion method to get an interactive program. In this use of the tail recursion method:

- The argument of the tail recursive function F is a pair (x, y), which means that the maximum is to be returned for the segment $[x, y]$ of the array.
- The desired value to compute (i.e., maximum in the entire array) is represented by the argument pair $(0, N)$.
- $(x, y) \in \textbf{dom}\ F\ \equiv\ 0 \leq x \leq y \leq N$. (The **MAX** quantification is undefined for $x > y$ or indices outside the bounds of the array a.)
- The invariant is $P : F.x.y = F.0.N \wedge 0 \leq x \leq y \leq N$.
- The guard $b.x$ for the recursive leg of the function definition is $x \neq y$.
- $t : y - x$ seems to be a good choice for the bound function. (Of course, we are required to prove that the final loop does indeed terminate.)

If we plug these definitions into the general program for tail recursion we get the following program:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ a[0..N]:\ \textbf{array of}\ int\ \{N \geq 0\};\ \textbf{var}\ r, x, y : int; \\
\quad x, y := 0, N; \\
\quad \{\ \text{invariant}\ P,\ \text{bound}\ t\ \} \\
\quad \textbf{do}\ x \neq y \rightarrow \\
\qquad \textbf{if}\ a.x \leq a.y \rightarrow x := x + 1 \\
\qquad [\!]\ a.y \leq a.x \rightarrow y := y - 1 \\
\qquad \textbf{fi} \\
\quad \textbf{od}\ ; \\
\quad \{P \wedge x = y,\ \text{hence}\ a.x = F.0.N\} \\
\quad r := a.x \\
\quad \{\ r = (\textbf{MAX}\ i : 0 \leq i \leq N : a.i)\ \} \\
]|
\end{array}
$$

We still need to prove termination, i.e., the boundedness and progress properties for t. This involves proving the following properties.

1. Boundedness.
   $F.x.y = F.0.N \wedge 0 \leq x \leq y \leq N \wedge x \neq y \Rightarrow y - x \geq 0$

2. Progress (first leg).
   $F.x.y = F.0.N \wedge 0 \leq x \leq y \leq N \wedge x \neq y \wedge a.x \leq a.y \wedge y - x = T \Rightarrow y - (x+1) < T.$

3. Progress (second leg).
   $F.x.y = F.0.N \wedge 0 \leq x \leq y \leq N \wedge x \neq y \wedge a.y \leq a.x \wedge y - x = T \Rightarrow (y-1) - x < T.$

All of these are easy to prove given the $0 \leq x < y$ on the left-hand-side and arithmetic properties.

8.3 Special Case of Tail Recursion

Let the following be functions of the specified type

$F : type1 \to type2$
$b : type1 \to boolean$
$f : type1 \to type2$
$g : type1 \to type1$
$h : type1 \to type2$
$\oplus : type2 \times type2 \to type2$
$t : type1 \to int$

such that

$(\forall\, x : x \in \mathbf{dom}\ F : x \in \mathbf{dom}\ b)$
$(\forall\, x : x \in \mathbf{dom}\ F \wedge \neg b.x : x \in \mathbf{dom}\ f)$
$(\forall\, x : x \in \mathbf{dom}\ F \wedge b.x : x \in \mathbf{dom}\ g \wedge g.x \in \mathbf{dom}\ F)$
$(\forall\, x : x \in \mathbf{dom}\ F \wedge b.x : x \in \mathbf{dom}\ h)$
$\oplus$ is a total operation on $type2$
$\oplus$ is associative and has left identity e
$(\forall\, x : x \in \mathbf{dom}\ F \wedge b.x : t.x \geq 0)$ — boundedness
$(\forall\, x : x \in \mathbf{dom}\ F \wedge b.x : t.(g.x) < t.x)$ — progress

and

$$F.x = \mathbf{if}\ \neg b.x \to f.x \ [\!]\ b.x \to h.x \oplus F.(g.x)\ \mathbf{fi}$$

where F does not appear in the definitions of b, f, g, and h.

F is a *linear recursive* and *backward recursive* definition. A function definition is linear recursive if it involves just one recursive application on any leg of the definition; otherwise it is multiply recursive. A function definition is *forward recursive* if the recursion is the outermost operation in the definition. If the recursive operation is embedded within the expression, it is backward recursive. A *tail recursive* definition is both linear recursive and forward recursive.

As we see in the previous subsection, tail recursions are especially important in programming because they can be converted into loops without the need for a stack to store temporary values. That is, there is no computational work to be done when the program returns from the recursive call.

Both $F.x$ and $h.x \oplus F.(h.x)$ (from the second leg of the definition) have the same general structure:

$$y \oplus F.z$$

Thus we will try to define a tail recursive function G such that:

$$(\forall\, r, x : r \in type2 \wedge x \in \mathbf{dom}\ F : G.r.x = r \oplus F.x)$$

Note that $(\forall\, x : x \in \mathbf{dom}\ F : F.x = G.e.x)$.

To find a recursive definition of $G.r.x$, we proceed by induction on the natural numbers (with the ordering determined by decreasing values of the bound function t).

Base case $\neg b.x$ (i.e., $t.x < 0$ or t undefined).

$$
\begin{array}{ll}
 & G.r.x \\
= & \quad \langle \text{ definition } G \ \rangle \\
 & r \oplus F.x \\
= & \quad \langle \text{ definition } F, \text{ given } \neg b.x \ \rangle \\
 & r \oplus f.x
\end{array}
$$

Inductive case $b.x$ (i.e., $t.x \geq 0$). Assume the induction hypothesis:
$(\forall\, s, y : s \in \mathit{type}2 \wedge y \in \mathbf{dom}\ F \wedge t.y < t.x : G.s.y = s \oplus F.y)$

$$
\begin{array}{ll}
 & G.r.x \\
= & \quad \langle \text{ definition of } G \ \rangle \\
 & r \oplus F.x \\
= & \quad \langle \text{ definition of } F, \text{ given } b.x \ \rangle \\
 & r \oplus (h.x \oplus F.(g.x)) \\
= & \quad \langle \ \oplus \text{ associativity } \ \rangle \\
 & (r \oplus h.x) \oplus F.(g.x) \\
= & \quad \langle \text{ Note 1, induction hypothesis applied right to left } \ \rangle \\
 & G.(r \oplus h.x).(g.x)
\end{array}
$$

Note 1: From the definition of F, we know that $(\forall\, x : x \in \mathbf{dom}\ F \wedge b.x : t.(g.x) < t.x)$. The range of this quantification holds in this inductive case, so $t.(g.x) < t.x$. Thus the induction hypothesis can be applied to $(r \oplus h.x) \oplus F.(g.x)$.

Thus, for any $r \in \mathit{type}2$, $x \in \mathbf{dom}\ F$ and the restrictions on b, f, g, h, $\oplus$, and t given previously, we define:

$$
\begin{array}{l}
G.r.x = \mathbf{if}\ \neg b.x \rightarrow r \oplus f.x \\
\qquad\quad [\!]\quad b.x \rightarrow G.(r \oplus h.x).(g.x) \\
\qquad\quad \mathbf{fi}
\end{array}
$$

To compute $F.x$, we can use $G.e.x$.

Now using the program for tail recursion, we have a program for this linear recursive case:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ X : type1\ \{\ X \in \textbf{dom}\ H\ \};\ \textbf{var}\ x : type1;\ \textbf{var}\ r : type2; \\
\quad r, x := e, X; \\
\quad \{\text{invariant}\ P : r \oplus F.x = F.X \wedge x \in \textbf{dom}\ F, \text{ bound } t\}; \\
\quad \textbf{do}\ b.x \rightarrow \\
\qquad r, x := r \oplus h.x, g.x \\
\quad \textbf{od}\ \{\ P \wedge \neg b.x, \text{ hence } r \oplus f.x = F.X\ \}; \\
\quad r := r \oplus f.x \\
\quad \{\ r = F.X\ \} \\
]|
\end{array}
$$

The above program can be used to compute F.

The technique used above can be used to find more efficient algorithms for tail recursive programs.

8.4 Example: Multiplication

For integers x and y, with $y \geq 0$, multiplication can be defined in terms of addition as follows:

$$\begin{array}{l} x * y = \textbf{if } y = 0 \rightarrow 0 \\ \qquad\quad [\!] \; y > 0 \rightarrow x + x * (y - 1) \\ \qquad\quad \textbf{fi} \end{array}$$

This definition is not tail recursive, but it is linear recursive and can be transformed into a tail recursion using the technique shown on the previous pages.

Note that $x * y$ and $x + x * (y - 1)$ both have the shape $a + b * c$

For integers a, b, and c, with $c \geq 0$, we define $G.a.b.c = a + b * c$.

We can transform this to the tail recursive definition:

$$\begin{array}{l} G.a.b.c = \textbf{if } c = 0 \rightarrow a \\ \qquad\quad [\!] \; c > 0 \rightarrow G.(a + b).b.(c - 1) \\ \qquad\quad \textbf{fi} \end{array}$$

If we apply the tail recursion technique to this, we get the program:

$$\begin{array}{l} |[\quad \textbf{con } X, Y : int \; \{ \; Y \geq 0 \; \}; \textbf{ var } r, y : int; \\ \quad r, y := 0, Y; \\ \quad \{ \text{ invariant } P : G.r.X.y = G.0.X.Y \wedge y \geq 0, \text{ bound } t : y \} \\ \quad \textbf{do } y > 0 \rightarrow r, y := r + x, y - 1 \textbf{ od} \\ \quad \{r = X * Y\} \\ ]| \end{array}$$

This algorithm requires $\mathcal{O}(Y)$ additions and subtractions. Can we find a more efficient algorithm, i.e., a new definition for G that "converges" on the base case in fewer recursive calls?

Remember that a bound function must show progress for each iteration of the algorithm. If y is not sufficient as a bound function, then a different bound function might be found.

Let's try for an $\mathcal{O}(log_2(Y))$ algorithm by dividing argument c of G by 2 instead of subtracting by 1. Note that $t.(c \textbf{ div } 2) < t.c$ for $c > 0$ so progress is still achieved.

What we want to do is to define a tail recursive function $G.a.b.c$ so that its recursive leg is of the form $G.E.F.(c \textbf{ div} 2)$ for some expression E and F. It must be equivalent to the nonrecursive definition ($G.a.b.c = a + b * c$). To do so, we proceed by calculation on two cases, c even or c odd.

Case $c > 0$ and c even.

$$
\begin{array}{ll}
 & G.E.F.(c \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{definition of } G \text{ (nonrecursive)} \ \rangle \\
 & E + F * (c \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{choose } F = 2 * b \ \rangle \\
 & E + (2 * b) * (c \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{assumption } c \text{ even, arithmetic} \ \rangle \\
 & E + b * c \\
= & \quad \langle \ \text{choose } E = a \ \rangle \\
 & a + b * c \\
= & \quad \langle \ \text{definition of } G \text{ (nonrecursive)} \ \rangle \\
 & G.a.b.c
\end{array}
$$

Case $c > 0$ and c odd.

$$
\begin{array}{ll}
 & G.E.F.(c \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{assumption } c \text{ even, arithmetic} \ \rangle \\
 & G.E.F.((c-1) \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{definition of } G \text{ (nonrecursive)} \ \rangle \\
 & E + F * ((c-1) \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{choose } F = 2 * b \ \rangle \\
 & E + (2 * b) * ((c-1) \ \mathbf{div}\ 2) \\
= & \quad \langle \ \text{assumption } c \text{ odd, arithmetic} \ \rangle \\
 & E + b * (c-1) \\
= & \quad \langle \ \text{arithmetic} \ \rangle \\
 & E + b * c - b \\
= & \quad \langle \ \text{choose } E = a + b \ \rangle \\
 & (a + b) + b * c - b \\
= & \quad \langle \ \text{arithmetic} \ \rangle \\
 & a + b * c \\
= & \quad \langle \ \text{definition of } G \text{ (nonrecursive)} \ \rangle \\
 & G.a.b.c
\end{array}
$$

Drawing these two cases into a tail recursive definition, we get:

$$
\begin{array}{l}
G.a.b.c = \mathbf{if}\ c = 0 \to a \\
\qquad [\!]\ c > 0 \to \mathbf{if}\ c \ \mathbf{mod}\ 2 = 0 \to G.a.(2 * b).(c \ \mathbf{div}\ 2) \\
\qquad\qquad\qquad\quad [\!]\ c \ \mathbf{mod}\ 2 = 1 \to G.(a + b).(2 * b).((c-1) \ \mathbf{div}\ 2) \\
\qquad\qquad\qquad\quad \mathbf{fi} \\
\qquad \mathbf{fi}
\end{array}
$$

This function definition uses integer multiplication and division by 2. Although in general multiplication and division are expensive operations on computers, integer multiplication and division by 2 are just arithmetic shifts by one bit position. Arithmetic shifts are normally inexpensive operations to perform on most computers.

The algorithm also includes a test comparing a natural number modulo 2 to 0 or 1 (that is, an even/odd test). This can also be implemented efficiently by just examining the lowest order bit of the integer representation, an operation that is efficient on most computers.

If we plug the definition of G into the general program for tail recursion we get:

$$
\begin{array}{l}
|[\ \ \textbf{con}\ X, Y : int\ \{\ Y \geq 0\ \};\ \textbf{var}\ r, x, y : int; \\
\quad r, x, y := 0, X, Y; \\
\quad \{\ \text{invariant}\ P : G.r.x.y = G.0.X.Y \wedge y \geq 0,\ \text{bound}\ t : y\} \\
\quad \textbf{do}\ y > 0 \rightarrow \\
\qquad \textbf{if}\ y\ \textbf{mod}\ 2 = 0 \rightarrow x, y := 2 * x, y\ \textbf{div}\ 2 \\
\qquad [\!]\ y\ \textbf{mod}\ 2 = 1 \rightarrow r, x, y := r + x, 2 * x, (y - 1)\ \textbf{div}\ 2 \\
\qquad \textbf{fi} \\
\quad \textbf{od} \\
\quad \{\ r = X * Y\ \} \\
]|
\end{array}
$$

The resulting algorithm uses $\mathcal{O}(log_2(Y))$ iterations to multiply an integer by a natural number. Each iteration involves at most one addition, one subtraction (a decrement), a multiplication by 2, a division by 2, and an even/odd test.

9 Array Assignment

9.1 Semantics

In Section 2.4.4, we defined the semantics of assignment where an array expression occurs on the right-hand-side as follows, for some array b and index expression E,

$$wp.(x := b.E).R \;\equiv\; def.E \wedge inrange.b.E \wedge R^{x}_{b.E}$$

where $inrange.b.E$ means that expression E evaluates to an integer value that is within the declared index range of array b.

What about the semantics of assignments of the form $b.E := F$?

One possibility might be:

$$wp.(b.E := F).R \;\equiv\; def.F \wedge def.E \wedge inrange.b.E \wedge R^{b.E}_{F}$$

For $b[0..10)$: **array of** int, using the definition given above:

$$wp.(b.i := 0).(b.2 \neq 0) \;\equiv\; 0 \leq i \leq 10 \wedge b.2 \neq 0$$

But what if the variable i had value 2?

Intuitively, the rule above does not seem to "work" properly because there are several different ways of referring to $b.2$. These include $b.2$, $b.i$ for $i = 2$, $b.(j-2)$ for $j = 4$, etc. That is, there are several *aliases* for $b.2$.

How do we get around this problem and still have a simple, textual substitution rule to define the wp?

Strategy: Treat an assignment to an element of an array as an assignment to the entire array.

First, let's introduce some new notation.

Suppose f is a function $int \rightarrow sometype$. Then let $f(h : E)$ be the same function as f except that at argument h it yields value E instead of what f yields:

$$f(h : E).i = \begin{cases} E & \textbf{if } i = h \\ f.i & \textbf{if } i \neq h \end{cases}$$

Now, an array $b[0..N)$ of $sometype$ is just a function $b : [0..N) \rightarrow sometype$. An assignment to an array element is thus a modification of the function's definition.

Axiom 13 (Array Assignment (single))
$wp.(b.E := F).R \;\equiv\; def.F \wedge def.E \wedge inrange.b.E \wedge R^{b}_{b(E:F)}$

Now let's consider the example above again (where $b[0..10)$ is an array of integers and i is an integer variable).

$$
\begin{array}{ll}
 & wp.(b.i := 0).(b.2 \neq 0) \\
\equiv & \quad \langle \text{ array assignment } \rangle \\
 & def.0 \wedge def.i \wedge inrange.b.i \wedge (b.2 \neq 0)^{\,b}_{b(i:0)} \\
\equiv & \quad \langle \text{ expressions 0 and } i \text{ defined everywhere, } b[0..10), \text{ textual substitution } \rangle \\
 & 0 \leq i < 10 \wedge b(i:0).2 \neq 2 \\
\equiv & \quad \langle \ b(i:0) \text{ elimination } \rangle \\
 & 0 \leq i < 10 \wedge (2 = i \Rightarrow 0 \neq 0) \wedge (2 \neq i \Rightarrow b.2 \neq 0) \\
\equiv & \quad \langle \ P \Rightarrow false \equiv \neg P, \text{ arithmetic } \rangle \\
 & 0 \leq i < 10 \wedge i \neq 2 \wedge (i \neq 2 \Rightarrow b.2 \neq 0) \\
\equiv & \quad \langle \ \Rightarrow \text{elimination } \rangle \\
 & 0 \leq i < 10 \wedge i \neq 2 \wedge b.2 \neq 0
\end{array}
$$

Next let's examine the case of a command S and postcondition R, where $b[0..N)$ for $N \geq 0$ and x, y, and i are integer variables:

$$
\begin{array}{l}
S : b.(i-1) := x+1 \\
R : (\forall\, j : i \leq j < N : b.j = y)
\end{array}
$$

$$
\begin{array}{ll}
 & wp.S.R \\
\equiv & \quad \langle \text{ array assignment, } x+1 \text{ and } i-1 \text{ defined, } b[0..N) \ \rangle \\
 & 0 \leq i-1 < N \wedge (\forall\, j : i \leq j < N : b.j = y)^{\,b}_{b(i-1:x+1)} \\
\equiv & \quad \langle \text{ textual substitution } \rangle \\
 & 0 \leq i-1 < N \wedge (\forall\, j : i \leq j < N : b(i-1:x+1).j = y) \\
\equiv & \quad \langle \text{ arithmetic, } j > i-1, \ b(i-1:x+1) \text{ elimination } \rangle \\
 & 1 \leq i \leq N \wedge (\forall\, j : i \leq j < N : b.j = y)
\end{array}
$$

Now, consider the command S' below for the same postcondition and variables as the previous example:

$$
S' : b.(i-1),\ i := x+1,\ i-1
$$

$$
\begin{array}{ll}
 & wp.S'.R \\
\equiv & \quad \langle \text{ array assignment, } x+1 \text{ and } i-1 \text{ defined, } b[0..N) \ \rangle \\
 & 0 \leq i-1 < N \wedge (\forall\, j : i \leq j < N : b.j = y)^{\,b,i}_{b(i-1:x+1),i-1} \\
\equiv & \quad \langle \text{ arithmetic, textual substitution } \rangle \\
 & 1 \leq i \leq N \wedge (\forall\, j : i-1 \leq j < N : b(i-1:x+1).j = y) \\
\equiv & \quad \langle \text{ range splitting } \rangle \\
 & 1 \leq i \leq N \ \wedge \ (\forall\ j : j = i-1 : b(i-1:x+1).j = y) \\
 & \qquad\qquad \wedge \ (\forall\ j : i \leq j < N : b(i-1:x+1).j = y) \\
\equiv & \quad \langle \text{ one-point, } b(i-1:x+1) \text{ elimination } \rangle \\
 & 1 \leq i \leq N \ \wedge \ x+1 = y \ \wedge \ (\forall\ j : i \leq j < N : b.j = y)
\end{array}
$$

Axiom 14 (Array Assignment (multiple))

$$\begin{array}{ll} & wp.(b.E, b.F := G, H).R \\ \equiv & def.G \wedge def.H \wedge def.E \wedge inrange.b.E \wedge def.F \wedge inrange.F \wedge E \neq F \wedge R^b_{b(E,F:G,H)} \end{array}$$

In the axiom above, we define:

$$b(E, F : G, H).i = \begin{cases} G & \textbf{if } i = E \\ H & \textbf{if } i = F \\ b.i & \textbf{otherwise} \end{cases}$$

9.2 Derivation Example: Partial Sums

Suppose we have the following specification for a program to compute the partial sums of an array:

$$\begin{array}{l} |[\quad \textbf{con } a[0..N) : \textbf{ array of } \ int \ \{Q\}; \textbf{ var } b[0..N) : \textbf{ array of } \ int; \\ \quad \text{PartialSums}; \\ \quad \{ \ R \ \} \\ ]| \end{array}$$

where

$$\begin{array}{l} Q : N \geq 1 \\ R : (\forall\, i : 0 \leq i < N : b.i = (\Sigma\, j : 0 \leq j \leq i : a.j)) \end{array}$$

Clearly, we need a loop to compute the values for b in general.

To derive the loop in the PartialSums program, we replace constant N in the post-condition by a fresh variable n and take the negation of the new $n = N$ conjunct as the guard.

$$\begin{array}{ll} R' : & n = N \wedge (\forall\, i : 0 \leq i < N : b.i = (\Sigma\, j : 0 \leq j \leq i : a.j)) \\ \text{guard } B : & n \neq N \\ \text{invariant } P_0 : & (\forall\, i : 0 \leq i < n : b.i = (\Sigma j : 0 \leq j \leq i : a.j)) \end{array}$$

To make sure that references to arrays a and b are defined, we add an invariant P_1.

$$P_1 : 0 \leq n \leq N$$

Next, we consider the initialization. Note that precondition is $N \geq 1$, so we have two easy possibilities:

1. $n := 0$
2. $b.0, n := a.0, 1$

We choose option 2 because it seems to lead to a slightly more efficient program.

The choice of the initialization allows us to strengthen P_1 to P_1'.

$$P_1' : 1 \leq n \leq N$$

Given invariant P_1' and the initialization, an appropriate bound function is:

$$t : N - n$$

Because addition must take place one element at a time, we choose $n := n + 1$ as the progress command.

To preserve the invariant given the $n := n + 1$ choice above, it seems that an assignment to a single array element should suffice (probably to $b.n$). Thus, for some unknown expressions E and F, the update command is:

$$b.E := F$$

Thus, so far we have the program:

$|[$ **con** $a[0..N)$: **array of** int $\{Q\}$; **var** $b[0..N)$: **array of** int;
$b.0, n := a.0, 1$;
{invariant $P_0 \wedge P_1'$, bound $t : N - n$}
do $n \neq N \rightarrow \{P_0 \wedge P_1' \wedge n \neq N\}$
$b.E, n := F, n + 1$ { $P_0 \wedge P_1'$ }
od $\{P_0 \wedge P_1' \wedge n = N$, hence R }
$]|$

Next we can calculate for the unknown expressions E and F.

Assume $P_0 \wedge P_1' \wedge n \neq N$:

$$
\begin{array}{ll}
 & wp.(b.E, n := F, n+1).(P_0 \wedge P_1') \\
\equiv & \quad \langle \text{ assignment axiom, textual substitution } \rangle \\
 & def.F \wedge def.E \wedge 0 \leq E \leq N \wedge \\
 & (\forall\, i : 0 \leq i < n+1 : b(E:F).i = (\Sigma\, j : 0 \leq j \leq i : a.j)) \wedge 1 \leq n+1 \leq N \\
\equiv & \quad \langle \text{ assumptions } P_1' \wedge n \neq N, \text{ range splitting (outer) for } i = n, \text{ one-point } \rangle \\
 & def.F \wedge def.E \wedge 0 \leq E \leq N \wedge \\
 & (\forall\, i : 0 \leq i < n : b(E:F).i = (\Sigma\, j : 0 \leq j \leq i : a.j)) \wedge \\
 & b(E:F).n = (\Sigma\, j : 0 \leq j \leq n : a.j) \\
\equiv & \quad \langle \text{ choose } E = n,\ b(E:F) \text{ elimination } \rangle \\
 & def.F \wedge 0 \leq n \leq N \wedge \\
 & (\forall\, i : 0 \leq i < n : b.i = (\Sigma\, j : 0 \leq j \leq i : a.j)) \wedge F = (\Sigma\, j : 0 \leq j \leq n : a.j) \\
\equiv & \quad \langle \text{ assumptions } P_0,\ P_1', \text{ range splitting for } j = n, \text{ one-point } \rangle \\
 & def.F \wedge F = (\Sigma\, j : 0 \leq j < n : a.j) + a.n \\
\equiv & \quad \langle \text{ assumptions } P_0,\ P_1', \text{ Note 1 } \rangle \\
 & def.F \wedge F = b.(n-1) + a.n \\
\equiv & \quad \langle \text{ choose } F = b.(n-1) + a.n, \text{ assumption } P_1', \text{ arithmetic } \rangle \\
 & true
\end{array}
$$

Note 1: Here we need P_1' instead of P_1 to make sure $b.(n-1)$ was defined. Otherwise, we would need to add a new variable. say s, with invariant $s = (\Sigma\, j : 0 \leq j < n : a.j)$ and initial value 0.

Thus, the body of the loop has command:

$$b.n, n := b.(n-1) + a.n, n+1$$