

The Science of Programming, Revisited

Margaret E. Myers

Robert A. van de Geijn

Draft

September 13, 2011

Copyright 2008
Margaret E. Myers
Robert A. van de Geijn

Contents

Preface

Preliminaries

1.1 Proofs using Equivalence Style

Example 1 Prove Counterpositive: $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$.

Start from the more complicated side: it is often easier to simplify.

$$\begin{aligned}
 & \neg q \rightarrow \neg p \\
 \Leftrightarrow & \quad \langle \textit{implication} \rangle \\
 & \neg(\neg q) \vee \neg p \\
 \Leftrightarrow & \quad \langle \textit{negation} \rangle \\
 & q \vee \neg p \\
 \Leftrightarrow & \quad \langle \textit{commutativity} \rangle \\
 & \neg p \vee q \\
 \Leftrightarrow & \quad \langle \textit{implication} \rangle \\
 & p \rightarrow q
 \end{aligned}$$

We will use this specific style: by speaking the same language, we will be able to make more progress.

Example 2 Prove $p \rightarrow (q \rightarrow p)$.

What do we want to prove? That it is always true, in other words, that it is a tautology.

$$\begin{aligned}
 & p \rightarrow (q \rightarrow p) \\
 \Leftrightarrow & \quad \langle \textit{implication} \rangle \\
 & p \rightarrow (\neg q \vee p) \\
 \Leftrightarrow & \quad \langle \textit{implication} \rangle \\
 & \neg p \vee (\neg q \vee p) \\
 \Leftrightarrow & \quad \langle \textit{commutativity} \rangle \\
 & \neg p \vee (p \vee \neg q) \\
 \Leftrightarrow & \quad \langle \textit{associativity} \rangle \\
 & (\neg p \vee p) \vee \neg q
 \end{aligned}$$

$$\begin{aligned}
&\leftrightarrow \text{ < commutativity >} \\
&\quad (p \vee \neg p) \vee \neg q \\
&\leftrightarrow \text{ < exclusive middle >} \\
&\quad T \vee \neg q \\
&\leftrightarrow \text{ < commutativity >} \\
&\quad \neg q \vee T \\
&\leftrightarrow \text{ < } \vee \text{ - simplification >} \\
&\quad T
\end{aligned}$$

Hence, by transitivity, $p \rightarrow (q \rightarrow p)$ is equivalent to true. *QED*

Note: At this stage of the game, we are being very picky here: we include all steps along the way. Eventually, we will streamline the proves, combining steps.

1.2 Quantifiers

Let us quickly review predicates that use quantification. We do so by example.

Example 3 Evaluate $(\exists n | 0 \leq n \leq 5 : \text{odd}(n))$.

Here $\text{odd}(x)$ returns true iff integer x is true. This predicate is read as “There exists n in the range $0 \leq n \leq 5$ such that n is odd”.

$$\begin{aligned}
&(\exists n | 0 \leq n \leq 5 : \text{odd}(n)) \\
&\leftrightarrow \text{ < definition of } \exists \text{ >} \\
&\quad \text{odd}(0) \vee \text{odd}(1) \vee \text{odd}(2) \vee \text{odd}(3) \vee \text{odd}(4) \vee \text{odd}(5) \\
&\leftrightarrow \text{ < definition of odd >} \\
&\quad F \vee T \vee F \vee T \vee F \vee T \\
&\leftrightarrow \text{ < definition of } \vee \text{ >} \\
&\quad T
\end{aligned}$$

1.2.1 Splitting off a term

Clearly the above would get very messy if the range of the quantification is very large. We offer the following alternative proof, which uses the fact that a single term can be split from the quantification.

$$\begin{aligned}
&(\exists n | 0 \leq n \leq 5 : \text{odd}(n)) \\
&\leftrightarrow \text{ < split term >} \\
&\quad (\exists n | 0 \leq n \leq 4 : \text{odd}(n)) \vee \text{odd}(5) \\
&\leftrightarrow \text{ < definition of odd >} \\
&\quad (\exists n | 0 \leq n \leq 4 : \text{odd}(n)) \vee T \\
&\leftrightarrow \text{ < definition of } \vee \text{ >} \\
&\quad T
\end{aligned}$$

The following are some examples of how quantifications can be split:

Example 4

$$\begin{aligned}
(\exists n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\exists n|0 \leq n \leq 4 : \text{odd}(n)) \vee \text{odd}(5) \\
(\forall n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\forall n|0 \leq n \leq 4 : \text{odd}(n)) \wedge \text{odd}(5) \\
\left(\sum n|0 \leq n \leq 5 : n^2\right) &\leftrightarrow \left(\sum n|0 \leq n \leq 4 : n^2\right) + 5^2 \\
\left(\prod n|0 \leq n \leq 5 : n^2\right) &\leftrightarrow \left(\prod n|0 \leq n \leq 4 : n^2\right) \cdot 5^2
\end{aligned}$$

1.2.2 One-point rule

If the range only includes a single element, the quantification can be replaced by substituting that element for the variable in the expression.

Example 5

$$\begin{aligned}
(\exists n|5 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow \text{odd}(5) \\
(\forall n|5 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow \text{odd}(5) \\
\left(\sum n|5 \leq n \leq 5 : n^2\right) &\leftrightarrow 5^2 \\
\left(\prod n|5 \leq n \leq 5 : n^2\right) &\leftrightarrow 5^2
\end{aligned}$$

1.2.3 Splitting the range

Similar to splitting off a single term, one can split the range into two subranges, where the union of the subranges has to equal the original range, and their intersection must be empty.

Example 6

$$\begin{aligned}
(\exists n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\exists n|0 \leq n \leq k : \text{odd}(n)) \vee (\exists n|k < n \leq 5 : \text{odd}(n)) \\
(\forall n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\forall n|0 \leq n \leq k : \text{odd}(n)) \wedge (\forall n|k < n \leq 5 : \text{odd}(n)) \\
\left(\sum n|0 \leq n \leq 5 : n^2\right) &\leftrightarrow \left(\sum n|0 \leq n \leq k : n^2\right) + \left(\sum n|k < n \leq 5 : n^2\right) \\
\left(\prod n|0 \leq n \leq 5 : n^2\right) &\leftrightarrow \left(\prod n|0 \leq n \leq k : n^2\right) \cdot \left(\prod n|k < n \leq 5 : n^2\right)
\end{aligned}$$

Notice that in the above example, $0 \leq k \leq 5$ or else the subranges do not have the required properties.

1.2.4 Empty range

The next question is what a quantification equals if the range is empty. We reason through this question by looking at the above discussion regarding the splitting of the range. If $k = 5$, then the second quantification on the right of the \leftrightarrow has an empty range and the expression on the right of the \leftrightarrow has to be equal to the one on its left.

$$\begin{aligned}
(\exists n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\exists n|0 \leq n \leq 5 : \text{odd}(n)) \vee \underbrace{(\exists n|5 < n \leq 5 : \text{odd}(n))}_F \\
(\forall n|0 \leq n \leq 5 : \text{odd}(n)) &\leftrightarrow (\forall n|0 \leq n \leq 5 : \text{odd}(n)) \wedge \underbrace{(\forall n|5 < n \leq 5 : \text{odd}(n))}_T \\
\left(\sum n|0 \leq n \leq 5 : n^2\right) &\leftrightarrow \left(\sum n|0 \leq n \leq 5 : n^2\right) + \underbrace{\left(\sum n|5 < n \leq 5 : n^2\right)}_0
\end{aligned}$$

$$\left(\prod n|0 \leq n \leq 5 : n^2\right) \leftrightarrow \left(\prod n|0 \leq n \leq 5 : n^2\right) \cdot \underbrace{\left(\prod n|5 < n \leq 5 : n^2\right)}_1$$

Example 7 Simplify $(\forall j|0 \leq j \leq i : b[j] = b[i])$

$$\begin{aligned} & (\forall j|0 \leq j \leq i : b[j] = b[i]) \\ \leftrightarrow & \quad < \text{split term} > \\ & (\forall j|0 \leq j < i : b[j] = b[i]) \wedge (b[i] = b[i]) \\ \leftrightarrow & \quad < \text{identity} > \\ & (\forall j|0 \leq j < i : b[j] = b[i]) \wedge T \\ \leftrightarrow & \quad < \wedge\text{-simplification} > \\ & (\forall j|0 \leq j < i : b[j] = b[i]) \end{aligned}$$

Example 8 Simplify $(\exists j|0 \leq j < 0 : b[j] \leq b[i])$.

We leave this as an exercise.

1.3 Weakening/Strengthening

Consider the predicate $E1 \rightarrow E2$. The predicate $E1$ is stronger than $E2$ in the sense that if $E1$ is true then $E2$ must also be true: $E1$ is more restrictive.

Given two expressions, $E1$ and $E2$, ask question “Is the implication $E1 \rightarrow E2$ always true?” If the answer is “yes”, then $E1$ is stronger than $E2$.

Example 9 Consider $x > 0$ and $x \geq 0$. Is the implication $(x > 0) \rightarrow (x \geq 0)$ always true? The answer is “yes” and therefore $x > 0$ is stronger than $x \geq 0$.

Here is a theorem that we will use a lot in later chapters:

Theorem 10 (Weakening/Strengthening Theorems) The following are all true:

1. $p \wedge q \rightarrow p$.
2. $p \rightarrow p \vee r$.
3. $p \wedge q \rightarrow p \vee r$.

Example 11 Which one is stronger: $x \geq 1$ or $x > 1$?

Answer:

$$\begin{aligned} & (x > 1) \rightarrow (x \geq 1) \\ \leftrightarrow & \quad < \text{math} > \\ & (x > 1) \rightarrow (x > 1 \vee x = 1) \\ \leftrightarrow & \quad < \text{weakening/strengthening} > \\ & T \end{aligned}$$

Thus, $x > 1$ is stronger than $x \geq 1$.

Example 12 Which one is stronger: $0 \leq x < 5$ or $x < 5$?

Answer:

$$(0 \leq x < 5) \rightarrow (x < 5)$$

↔ < math >

$$((0 \leq x) \wedge (x < 5)) \rightarrow (x < 5)$$

↔ < weakening/strengthening >

T

Thus, $0 \leq x < 5$ is stronger than $x < 5$.

Example 13 Which one is stronger: $0 \leq x \leq 3$ or $x < 3$?

Answer:

$$(0 \leq x \leq 3) \rightarrow (x < 3)$$

↔ < math >

$$((0 \leq x) \wedge (x \leq 3)) \rightarrow (x < 3)$$

↔ < weakening/strengthening >

$$((0 \leq x) \wedge (x < 3 \vee x = 3)) \rightarrow (x < 3)$$

Here none of the weakening/strengthening theorems apply, and one cannot say that one is stronger than the other.

Proving Programs Correct: Basics

In this chapter, we motivate where we are headed.

2.1 Annotating Algorithms

For now, we will view programs strictly as a sequence of commands, to be executed in a specified order:

$$\begin{array}{l} S_0 \\ S_1 \\ S_2 \\ \vdots \\ S_{n-1} \end{array}$$

where each S_i is a command. A command can itself be comprised of multiple commands. Sometimes we will write such a program as $S_0; S_1; S_2; \dots; S_{n-1}$.

When writing a program, it is good habit to insert a comment. Since English (or any other language) is inherently ambiguous, we will insert predicates that are supposed to evaluate to *true* at a given point in the algorithm.

For example, consider a loop that, given an integer array $b[0..(n-1)]$, computes the index of the first entry that equals zero:

```
 $i := 0$   
do  
     $b[i] \neq 0 \rightarrow i := i + 1$   
od
```

Here

- $x := exp$ indicates that expression exp is assigned to x (x becomes exp),
- The loop executes until $b[i] \neq 0$ becomes *false*, and
- If $b[i] \neq 0$ then the variable i is incremented by one.

We will define commands in our language more precisely, later.

One may comment this program as follows:

$b[0..n-1]$ is a nonempty array with at least one zero in it
$i := 0$
do
$0 \leq i < n$ and $b[i..(n-1)]$ is a nonempty array with at least one zero in it
$b[i] \neq 0 \rightarrow i := i + 1$
$0 \leq i < n$ and $b[i..(n-1)]$ is a nonempty array with at least one zero in it
od
$0 \leq i < n$ and $b[i] = 0$

The problem is that English is not a particularly unambiguous language. Therefore, we annotate (comment) the code with predicates instead:

$\{n \geq 0 \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
$i := 0$
$\{0 \leq i < n \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
do
$\{0 \leq i < n \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
$b[i] \neq 0 \rightarrow i := i + 1$
$\{0 \leq i < n \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
od
$0 \leq i < n$ and $b[i] = 0$

An arbitrary program can similarly be annotated:

$$\begin{array}{l} \{Q_0\} \\ S_0 \\ \{Q_1\} \\ S_1 \\ \{Q_2\} \\ S_2 \\ \vdots \\ \{Q_{n-1}\} \\ S_{n-1} \\ \{Q_n\} \end{array}$$

This can also be written as

$$\{Q_0\}S_0\{Q_1\}S_1\{Q_2\}\cdots\{Q_{n-1}\}S_{n-1}\{Q_n\}.$$

2.2 Hoare Triples

Consider $\{Q\}S\{R\}$ where S is a command in our language (yet to be defined) and Q and R are predicates. Here

- S can be a single command, or itself be a sequence of commands.
- Q is a predicate that indicates the state of the variables before execution of S . It is called the *precondition*.
- R is a predicate that indicates the state of the variables after execution of S . It is called the *postcondition*.

The triple $\{Q\}S\{R\}$ is itself a predicate that evaluates to *true* if and only if the command S , when executed starting in a state where Q is true, completes in a state where R is true. It is known as a *Hoare triple*. You can read $\{Q\}S\{R\}$ as “The command S , when started in a state where Q holds completes in a state where R holds” or as “ S , when started in a state where Q holds *establishes* R ”

Let us assume that S is a program that is supposed to leave variables in a state where R is *true* if executed starting in a state where Q is *true*. Then S is a correct program if $\{Q\}S\{R\}$ is *true*.

2.3 When is a Program Correct?

Let us assume that S itself consists of the sequence of commands

$$S : S_0; S_1; S_2; \cdots; S_{n-1}$$

If we can determine predicates Q_0, \cdots, Q_n such that

$$\{Q : Q_0\}S_0\{Q_1\}S_1\{Q_2\} \cdots \{Q_{n-1}\}S_{n-1}\{R : Q_n\},$$

and $\{Q_i\}S_i\{Q_{i-1}\}$, $0 \leq i \leq n$, can be shown to be *true*, then $\{Q\}S\{R\}$ is *true* and the program is correct.

What we will learn next is that a command in our language, S , can be defined by prescribing the weakest precondition transformation for it: $wp(S, R)$. In this section, we will illustrate this for a few simple commands, with more to following in subsequent sections.

Definition of the Language

In this chapter, we show how a careful definition of the language enables rigorous proofs of correctness of programs.

3.1 The Weakest Precondition

Let us consider a command S together with the Hoare triple $\{Q\}S\{R\}$. The way we will define commands is by noting that a predicate, Q' , can be used to characterize *all* states of variables such that execution of S , when started in such a state, completes in a state where R holds. Now, consider the annotated program

$$\begin{array}{c} \{Q\} \\ \{Q'\} \\ S \\ \{R\} \end{array}$$

No computation occurs between the assertions Q and Q' . Thus, if the program is to be in a state that satisfies Q' before S is executed, Q better have the property that Q' also holds. In other words, if and only if $Q \rightarrow Q'$ (Q is stronger than Q') will it be the case that execution of S completes in a state where R holds.

Recall that for the Hoare triple $\{Q\}S\{R\}$, Q is the precondition. The state Q' is known as the *weakest precondition*. If and only if command S is started in a state that satisfies Q' will it complete in a state where R holds. This weakest precondition, which is a function of the command S and the postcondition R , is indicated by the predicate transformer $wp(S, R)$.

Example 14 Consider the following

- $wp("x := y'', x = 3) = (y = 3)$
- $wp("x := x + 1'', 0 \leq x \leq 1) = (-1 \leq x \leq 0)$
- $wp("x := y'', x = y) = T$
- $wp("\text{if } x \geq y \text{ then } z := x \text{ else } z := y'', z \geq x \wedge z \geq y) = T$
- $wp("x := 3'', x = 4) = F$
- $wp(S, T) =$ the set of all states starting from which the execution of S is guaranteed to terminate.

3.2 Properties of $wp(S, R)$.

Let us list a few properties of the weakest precondition transformer. As we do so, it is important to understand what $wp(S, R)$ means:

- The predicate $wp(S, R)$ is *true* if and only if the command S , when executed starting in a state where $wp(S, R)$ holds, completes in a state where R holds.
- The Hoare triple $\{Q\}S\{R\}$ is *true* if and only if $Q \rightarrow wp(S, R)$.

3.2.1 Law of Excluded Miracle

The observation is that our language cannot include a command that leaves us in a state described by the predicate “F”: Let us assume that such a command S does exist. Then that command, when executed, would complete in a state where *false* is *true*.

3.3 Assignment Command

In the previous chapter, we previewed the assignment command $var := exp$ where var is a variable and exp is an expression the result of which is to be assigned to var .

The semantics of the assignment command are defined by the weakest preconditioner transformer. Before we do so, let us gain some insight. Consider

$$\begin{array}{l} \{Q'\} \\ i := i + 1 \\ \{i < 0\} \end{array}$$

What state, Q' must i be in before the assignment if $i < 0$ is to hold after its execution? The intuitive answer is that $i + 1 < 0$ must hold before the assignment.

Similarly, if some predicate that has i as a variable in it, $R(i)$, must hold after execution of the assignment $i := i + 1$, before the execution of that assignment $R(i + 1)$ must hold. More generally, if $R(i)$ is to hold after the assignment $i := exp$, $R(exp)$ must hold before its execution. This motivates the following definition of the assignment command:

$$wp(\text{“}var := exp\text{”}, R) = R_{var}^{exp},$$

where R_{var}^{exp} indicates the predicate that is derived from predicate R by replacing every occurrence of variable var with expression exp .

3.4 Composition of Commands

We next discuss how to define composition of commands. Consider the program that consists of two commands that are executed one after the other:

$$\begin{array}{l} \{wp()\} \\ S_0 \\ S_1 \\ \{R\} \end{array}$$

or, alternatively, $S : S_0; S_1$. The question becomes what the weakest precondition is of S . Note that we would like the following to evaluate to *true*:

$$\begin{array}{l} \{wp(S, R)\} \\ S_0 \\ S_1 \\ \{R\} \end{array}$$

Now, for the second command, S_1 , to leave us in a state where R holds after its executing, the program must be in the state $Q_1 : wp(S_1, R)$ before the execution of S_1 :

$$\begin{array}{l} \{wp(S, R)\} \\ S_0 \\ \{Q_1 : wp(S_1, R)\} \\ S_1 \\ \{R\} \end{array}$$

For the first command, S_0 , to leave us in a state where Q_1 holds after its executing, the program must be in the state $Q : wp(S_0, Q_1)$ before the execution of S_0 :

$$\begin{array}{l} \{wp(S_0, Q_1)\} \\ S_0 \\ \{Q_1 : wp(S_1, R)\} \\ S_1 \\ \{R\} \end{array}$$

Thus, we conclude that $wp("S_0; S_1", R) = wp(S_0, wp(S_1, R))$.

The IF Command

We discuss how to establish the correctness of the IF command.

4.1 Syntax

In the programming languages used in our class, the IF command has the following syntax:

```

if  $B_1 \rightarrow S_1$ 
 $\parallel$   $B_2 \rightarrow S_2$ 
 $\vdots$   $\vdots$   $\vdots$ 
 $\parallel$   $B_n \rightarrow S_n$ 
fi

```

The B_i s are called the *guards*.

The execution of this command is as follows:

- If at the time of execution *any* of the B_i s is not well defined, **abort** is possible.
- If at the time of execution *all* of the B_i s are not well defined, **abort** is inevitable.
- If at the time of execution *all* of the B_i s are *false*, **abort** is inevitable.
- If at the time of execution only one guard is *true* then the command associated with that guard is executed.
- If at the time of execution more than one guard is *true*, then one of the commands associated with a *true* guard is executed. In this case the choice is nondeterministic (random).

4.2 Weakest precondition of the IF command

Consider the IF command

```

if  $B_1 \rightarrow S_1$ 
 $\parallel$   $B_2 \rightarrow S_2$ 
 $\vdots$   $\vdots$   $\vdots$ 
 $\parallel$   $B_n \rightarrow S_n$ 
fi

```

which we will also write as

$$\mathbf{IF} : \mathbf{if} \ B_1 \rightarrow S_1 \ \|\ \dots \ \| B_n \rightarrow S_n \ \mathbf{fi}$$

Now let us consider under what conditions this command completes in a state where predicate R is *true*. We will assume that all of the guards are well defined.

$$wp(\mathbf{IF}, R) = \underbrace{(\exists i | 1 \leq i \leq n : B_i)}_{\text{one of the guards is true}} \ \wedge \ \underbrace{(\forall i | 1 \leq i \leq n : B_i \rightarrow wp(S_i, R))}_{\substack{\text{if } B_i \text{ is } \textit{true}, \text{ then} \\ \text{before the IF statement the} \\ \text{variables must be in a state} \\ \text{such that } S_i \text{ establishes } R}}$$

Example 15 *Determine*

$$wp(\mathbf{“if} \ x \geq 0 \rightarrow y := x \ \| \ x \leq 0 \rightarrow y := -x \ \mathbf{fi}'' , y = |x|).$$

$$\begin{aligned} & wp(\mathbf{“if} \ x \geq 0 \rightarrow y := x \ \| \ x \leq 0 \rightarrow y := -x \ \mathbf{fi}'' , y = |x|) \\ \leftrightarrow & \text{ < definition of IF >} \\ & (x \geq 0 \vee x \leq 0) \wedge (x \geq 0 \rightarrow wp(\mathbf{“}y := x'' , y = |x|)) \\ & \quad \wedge (x \leq 0 \rightarrow wp(\mathbf{“}y := -x'' , y = |x|)) \\ \leftrightarrow & \text{ < Excluded middle, definition of := >} \\ & T \wedge (x \geq 0 \rightarrow (x = |x|)) \wedge (x \leq 0 \rightarrow (-x = |x|)) \\ \leftrightarrow & \text{ < } \wedge\text{-simplification >} \\ & (x \geq 0 \rightarrow (x = |x|)) \wedge (x \leq 0 \rightarrow (-x = |x|)) \\ \leftrightarrow & \text{ < definition of absolute value >} \\ & T \wedge T \\ \leftrightarrow & \text{ < } \wedge\text{-simplification >} \\ & T \end{aligned}$$

End of example.

Example 16 *If $x < 0$ then set $x := 0$:*

$$\begin{aligned} & \{Q : T\} \\ & \mathbf{if} \quad x < 0 \rightarrow x := 0 \\ & \quad \|\quad x \geq 0 \rightarrow \mathbf{skip} \\ & \mathbf{fi} \\ & \{R : x \geq 0\} \end{aligned}$$

Our approach will be to show that $T \rightarrow wp(\mathbf{IF}, x \geq 0)$.

$$\begin{aligned} & T \rightarrow wp(\mathbf{IF}, x \geq 0) \\ \leftrightarrow & \text{ < definition >} \\ & T \rightarrow (x < 0 \vee x \geq 0) \wedge (x < 0 \rightarrow wp(\mathbf{“}x := 0'' , x \geq 0)) \\ & \quad \wedge (x \geq 0 \rightarrow wp(\mathbf{“}skip'' , x \geq 0)) \\ \leftrightarrow & \text{ < Excluded middle; } \wedge\text{-simp; } wp \text{ := and skip >} \\ & T \rightarrow (x < 0 \rightarrow 0 \geq 0) \wedge (x \geq 0 \rightarrow x \geq 0) \\ \leftrightarrow & \text{ < } \rightarrow\text{-simp; } \wedge\text{-simp >} \end{aligned}$$

$$\begin{array}{l}
T \rightarrow T \\
\leftrightarrow \quad < \rightarrow\text{-simp} > \\
T
\end{array}$$

Therefore, program segment holds (is correct).

End of example.

4.3 The IF-Theorem

We start by motivating a theorem that will make things a lot less messy.

Consider

$$\begin{array}{l}
\{Q\} \\
\mathbf{if} \quad B_1 \rightarrow S_1 \\
\quad \parallel \quad B_2 \rightarrow S_2 \\
\mathbf{fi} \\
\{R\}
\end{array}$$

To prove this correct, we need to show that

$$Q \rightarrow (B_1 \vee B_2) \wedge (B_1 \rightarrow wp(S_1, R)) \wedge (B_2 \rightarrow wp(S_2, R)).$$

This has the form

$$E_1 \rightarrow E_2 \wedge E_3 \wedge E_4.$$

How does this simplify?

In one of the homework exercises you showed something slightly simpler: $E_1 \rightarrow E_2 \wedge E_3$.

$$\begin{array}{l}
E_1 \rightarrow E_2 \wedge E_3 \\
\leftrightarrow \quad < \text{implication} > \\
\neg E_1 \vee (E_2 \wedge E_3) \\
\leftrightarrow \quad < \text{distribution} > \\
(\neg E_1 \vee E_2) \wedge (\neg E_1 \vee E_3) \\
\leftrightarrow \quad < \text{implication} > \\
(E_1 \rightarrow E_2) \wedge (E_1 \rightarrow E_3).
\end{array}$$

From this we conclude that

$$Q \rightarrow (B_1 \vee B_2) \wedge (B_1 \rightarrow wp(S_1, R)) \wedge (B_2 \rightarrow wp(S_2, R))$$

is equivalent to

1. $Q \rightarrow (B_1 \vee B_2)$; **and**
2. $Q \rightarrow wp(B_1 \rightarrow wp(S_1, R))$; **and**
3. $Q \rightarrow wp(B_2 \rightarrow wp(S_2, R))$.

We would like to further simplify 2. and 3. We do so by examining $E_1 \rightarrow (E_2 \rightarrow E_3)$.

$$\begin{aligned}
& E_1 \rightarrow (E_2 \rightarrow E_3) \\
\leftrightarrow & \text{ < implication >} \\
& \neg E_1 \vee (\neg E_2 \vee E_3) \\
\leftrightarrow & \text{ < associativity >} \\
& (\neg E_1 \vee \neg E_2) \vee E_3 \\
\leftrightarrow & \text{ < DeMorgan's Law >} \\
& \neg(E_1 \wedge E_2) \vee E_3 \\
\leftrightarrow & \text{ < implication >} \\
& E_1 \wedge E_2 \rightarrow E_3
\end{aligned}$$

From this we conclude that

$$Q \rightarrow (B_1 \vee B_2) \wedge (B_1 \rightarrow wp(S_1, R)) \wedge (B_2 \rightarrow wp(S_2, R))$$

is equivalent to checking that all of the following are *true*:

1. $Q \rightarrow (B_1 \vee B_2)$;
2. $Q \wedge B_1 \rightarrow wp(S_1, R)$; and
3. $Q \wedge B_2 \rightarrow wp(S_2, R)$.

Thus, checking if all three of these conditions hold is equivalent to checking whether the $\{Q\}\text{IF}\{R\}$ is *true*, in other words, whether the program segment is correct.

This can be summarized in the IF-Theorem:

Theorem 17 (IF-Theorem) *If a predicate Q satisfies*

1. $Q \rightarrow (\exists |1 \leq i \leq n : B_i)$; **and**
2. $(\forall i | 1 \leq i \leq n : Q \wedge B_i \rightarrow wp(S_i, R))$

then

$$Q \rightarrow wp(\text{IF}, R)$$

(Q guarantees that IF will establish R regardless of which guard holds) and hence $\{Q\}\text{IF}\{R\}$.

Or, think of it as a check list:

Theorem 18 (IF-Theorem (alternative)) *If a predicate Q satisfies*

- $Q \rightarrow (\exists | 1 \leq i \leq n : B_i)$; **and**
- $Q \wedge B_1 \rightarrow wp(S_1, R)$; **and**
- \vdots
- $Q \wedge B_n \rightarrow wp(S_n, R)$

then $Q \rightarrow wp(\text{IF}, R)$ and hence $\{Q\}\text{IF}\{R\}$ is true.

Example 19 *Prove the following program segment correct:*

$$\begin{array}{l}
\{Q : 0 < i < n \wedge (\exists j|0 < j < i : m = b[j]) \wedge (\forall j|0 < j < i : m \leq b[j])\} \\
\mathbf{if} \ b[i] \leq m \rightarrow m := b[i] \quad (\text{call this } S_{00}) \\
\quad \square \ b[i] \geq m \rightarrow \mathbf{skip} \quad (\text{call this } S_{01}) \\
\mathbf{fi} \\
i := i + 1 \\
\{R : 0 < i \leq n \wedge (\exists j|0 < j < i : m = b[j]) \wedge (\forall j|0 < j < i : m \leq b[j])\}
\end{array}$$

We will let S_1 denote the IF command, while S_2 denotes the assignment $i := i + 1$.

To show this, we insert an extra predicate:

$$\begin{array}{l}
\{Q : 0 < i < n \wedge (\exists j|0 < j < i : m = b[j]) \wedge (\forall j|0 < j < i : m \leq b[j])\} \\
\mathbf{if} \ b[i] \leq m \rightarrow m := b[i] \quad (\text{call this } S_{11}) \\
\quad \square \ b[i] \geq m \rightarrow \mathbf{skip} \quad (\text{call this } S_{12}) \\
\mathbf{fi} \\
\{R^*\} \\
i := i + 1 \\
\{R : 0 < i \leq n \wedge (\exists j|0 < j < i : m = b[j]) \wedge (\forall j|0 < j < i : m \leq b[j])\}
\end{array}$$

Notice that $\{R^*\}i := i + 1\{R\}$ must be *true* and $\{Q\}S_1\{R^*\}$ must be *true*. (Why?)

Step 1 First find R^* .

$$\begin{array}{l}
wp(S_2, R) \\
\leftrightarrow \quad < \text{instantiation} > \\
wp("i := i + 1", 0 < i \leq n \wedge (\exists j|0 < j < i : m = b[j]) \wedge (\forall j|0 < j < i : m \leq b[j])) \\
\leftrightarrow \quad < \text{definition } := > \\
0 < i + 1 \leq n \wedge (\exists j|0 < j < i + 1 : m = b[j]) \wedge (\forall j|0 < j < i + 1 : m \leq b[j])
\end{array}$$

Step 2 Prove $\{Q\}S_1\{R^*\}$.

Here we apply the IF-Theorem:

1. $Q \rightarrow B_1 \vee B_2$:

$$\begin{array}{l}
Q \rightarrow B_1 \vee B_2 \\
\leftrightarrow \quad < \text{instantiation} > \\
Q \rightarrow b[i] \leq m \vee b[i] \geq m \\
\leftrightarrow \quad < \text{excluded middle (or arithmetic)} > \\
Q \rightarrow T \\
\leftrightarrow \quad < \text{simplification} > \\
T
\end{array}$$

Note: Delaying instantiation for Q saved a lot of writing!

2. $Q \wedge B_1 \rightarrow wp(S_{11}, R^*)$:

$$\begin{array}{l}
Q \wedge B_1 \rightarrow wp(S_{11}, R^*) \\
\leftrightarrow \quad < \text{instantiation} > \\
Q \wedge B_1 \rightarrow
\end{array}$$

$$\begin{aligned}
& wp("m := b[i]", 0 < i + 1 \leq n \wedge (\exists j | 0 < j < i + 1 : m = b[j]) \wedge (\forall j | 0 < j < i + 1 : m \leq b[j])) \\
\leftrightarrow & \text{ < definition := >} \\
& Q \wedge B_1 \rightarrow \\
& \quad 0 < i + 1 \leq n \wedge (\exists j | 0 < j < i + 1 : b[i] = b[j]) \wedge (\forall j | 0 < j < i + 1 : b[i] \leq b[j]) \\
\leftrightarrow & \text{ < arithmetic, split-off term } \times 2 \text{ >} \\
& Q \wedge B_1 \rightarrow \\
& \quad 0 \leq i \leq n \wedge ((\exists j | 0 < j < i : b[i] = b[j]) \vee b[i] = b[i]) \wedge ((\forall j | 0 < j < i : b[i] \leq b[j]) \wedge b[i] \leq b[i]) \\
\leftrightarrow & \text{ < identity } \times 2 \text{ >} \\
& Q \wedge B_1 \rightarrow \\
& \quad 0 \leq i \leq n \wedge ((\exists j | 0 < j < i : b[i] = b[j]) \vee T) \wedge ((\forall j | 0 < j < i : b[i] \leq b[j]) \wedge T) \\
\leftrightarrow & \text{ < } \wedge\text{-simplification, } \vee\text{-simplification >} \\
& Q \wedge B_1 \rightarrow \\
& \quad 0 \leq i \leq n \wedge (\forall j | 0 < j < i : b[i] \leq b[j]) \\
\leftrightarrow & \text{ < instantiation >} \\
& 0 < i < n \wedge (\exists j | 0 < j < i : m = b[j]) \wedge (\forall j | 0 < j < i : m \leq b[j]) \wedge b[i] \leq m \rightarrow \\
& \quad 0 \leq i \leq n \wedge (\forall j | 0 < j < i : b[i] \leq b[j]) \\
\leftrightarrow & \text{ < transitivity, arithmetic >} \\
& 0 < i < n \wedge (\exists j | 0 < j < i : m = b[j]) \wedge (\forall j | 0 < j < i : b[i] \leq m \leq b[j]) \rightarrow \\
& \quad (0 = i \vee 0 < i \leq n) \wedge (\forall j | 0 < j < i : b[i] \leq b[j]) \\
\leftrightarrow & \text{ < weakening/strengthening theorem >} \\
& T
\end{aligned}$$

Let's examine this last step carefully. Below, we underline the expressions both sides of the implication have in common:

$$\frac{0 < i < n \wedge (\exists j | 0 < j < i : m = b[j]) \wedge (\forall j | 0 < j < i : b[i] \leq m \leq b[j])}{(0 = i \vee 0 < i \leq n) \wedge (\forall j | 0 < j < i : b[i] \leq b[j])} \rightarrow$$

- All the expressions that are not underlined on the left of the implication strengthen the common expression (further restrict the states that satisfy the expression).
- All the expressions that are not underlined on the right of the implication weaken the common expression (lessen the restriction on the states that satisfy the expression).

3. $Q \wedge B_2 \rightarrow wp(S_{12}, R^*)$:

$$\begin{aligned}
& Q \wedge B_2 \rightarrow wp(S_{12}, R^*) \\
\leftrightarrow & \text{ < instantiation >} \\
& Q \wedge B_2 \rightarrow \\
& \quad wp("skip", 0 < i + 1 \leq n \wedge (\exists j | 0 < j < i + 1 : m = b[j]) \wedge (\forall j | 0 < j < i + 1 : m \leq b[j])) \\
\leftrightarrow & \text{ < definition skip; arithmetic >} \\
& Q \wedge B_2 \rightarrow \\
& \quad 0 \leq i < n \wedge (\exists j | 0 < j < i + 1 : m = b[j]) \wedge (\forall j | 0 < j < i + 1 : m \leq b[j]) \\
\leftrightarrow & \text{ < arithmetic, split-off term } \times 2 \text{ >} \\
& Q \wedge B_2 \rightarrow \\
& \quad 0 \leq i < n \wedge ((\exists j | 0 < j < i : m = b[j]) \vee m = b[i]) \wedge ((\forall j | 0 < j < i : m \leq b[j]) \wedge m \leq b[i])
\end{aligned}$$

↔ < instantiation >

$$0 < i < n \wedge (\exists j | 0 < j < i : m = b[j]) \wedge (\forall j | 0 < j < i : m \leq b[j]) \wedge m \leq b[i] \rightarrow \\ 0 \leq i < n \wedge ((\exists j | 0 < j < i : m = b[j]) \vee m = b[i]) \wedge ((\forall j | 0 < j < i : m \leq b[j]) \wedge m \leq b[i])$$

↔ < split-off term >

$$\frac{0 < i < n \wedge (\exists j | 0 < j < i : m = b[j]) \wedge (\forall j | 0 < j < i : m \leq b[j]) \wedge m \leq b[i]}{(0 < i < n \vee i = 1) \wedge ((\exists j | 0 < j < i : m = b[j]) \vee m = b[i])} \rightarrow \\ \wedge ((\forall j | 0 < j < i : m \leq b[j]) \wedge m \leq b[i])$$

↔ < weakening/strengthening theorem >

T

Hence, by the IF-Theorem the example holds.

End of example.

4.4 Strategy

In the proof of the last example, we employed the weakening/strengthening theorem, which was introduced very briefly in the second lecture of the semester:

Theorem 20 (Weakening/Strengthening Theorems) *The following are all true:*

1. $p \wedge q \rightarrow p$.
2. $p \rightarrow p \vee r$.
3. $p \wedge q \rightarrow p \vee r$.

In Step 2 of the proof of Example ?? we use Result 3 in the theorem as follows: The result to be proven is of the form $E_1 \rightarrow E_2$. We manipulate E_1 and E_2 so that there is a common part to both, let us call it p . We show that E_2 can be written as $p \vee r$ and E_1 can be written as $p \wedge q$, so that $p \wedge q \rightarrow p \vee r$. We then invoke Result 3 from the Weakening/Strengthening Theorems to deduce that the predicate is *true*.

If it turns out that q or r are empty (as in Step 3 of Example ??), then one of the other results from the Theorem can be invoked.

This will become a very frequently employed tool as we prove the correctness of program segments.

The DO Command

In the programming languages used in our class, the DO command has the following syntax:

```

do  $B_1 \rightarrow S_1$ 
   $B_2 \rightarrow S_2$ 
   $\vdots$ 
   $B_n \rightarrow S_n$ 
od

```

The B_i s are called the (loop) *guards*. Together, this command will be indicated by DO.

The symbol BB will be used to denote $B_1 \vee B_2 \vee \dots \vee B_n$, in other words, $(\exists i | 1 \leq i \leq n : B_i)$. Thus, BB is the condition under which the loop does not yet terminate. The command executes as follows:

- The loop is executed until none of the guards evaluate to *true*, in other words, until $\neg BB$.
- Each time through the loop, a guard that evaluates to *true* is randomly selected, and the statement associated with it is executed.

5.1 Weakest precondition

The weakest precondition of the DO command is very messy to define. We will not bother, since the DO-Theorem, discussed next, is what is used in practice.

5.2 Motivation

To motivate theorems regarding a DO command (which we will not prove), let us consider the simpler loop:

```

do
   $BB \rightarrow S$ 
od

```

We would like to establish

```

{ $Q$ }
do
   $BB \rightarrow S$ 

```

```

od
  {R}

```

To do so, we introduce a predicate P , the invariant, which will be true before and after each iteration of the loop. If you think about this a bit, this means P must hold at all the places indicated below:

```

  {Q}
  {P}
do
  {P}
  BB → S
  {P}
od
  {P}
  {R}

```

Now,

- If $Q \rightarrow P$, then P hold immediately before the **do**.
- If P is *true* immediately before the **do** it is *true* immediately after the **do** the first time the loop is executed, since no assignment to any variables occurs in between.
- If BB holds, S is executed. Then S is executed in a state where $P \wedge BB$ holds. We want to show that it completes in a state where P again holds. This means that $P \wedge BB \rightarrow wp(S, P)$ must hold. If this can be shown, then P will again hold just before **od**. And thus, it will hold again right after **do** the second time the loop is executed. By Mathematical Induction, it will hold *every time* just after **do** and just before **od**.
- If BB ever becomes *false*, then the program ends up in a state where $P \wedge \neg BB$ holds just after **od**.
- If $P \wedge \neg BB \rightarrow R$ then *if* the loop ever completes, the desired postcondition holds.

Example 21 Compute the sum of the elements of array $b[0 \dots 9]$ and store the result in variable s :

```

S0 :  i, s := 0, 0
do
  i < 10 →
  S1 :  i, s := i + 1, s + b[i]
od
  {R :  s = (∑ j | 0 ≤ j < 10 : b[j])}

```

We begin by annotating the program with some assertions. We will need to prove that the assertions involving the invariant P are actually true at the indicated point in the program.

```

  {Q :  T}
  S0 :  i, s := 0, 0
  {P :  0 ≤ i ≤ 10 ∧ s = (∑ j | 0 ≤ j < i : b[j])}
do
  {P :  0 ≤ i ≤ 10 ∧ s = (∑ j | 0 ≤ j < i : b[j])}
  i < 10 →
  S1 :  i, s := i + 1, s + b[i]
  {P :  0 ≤ i ≤ 10 ∧ s = (∑ j | 0 ≤ j < i : b[j])}
od
  {P :  0 ≤ i ≤ 10 ∧ s = (∑ j | 0 ≤ j < i : b[j])}
  {R :  s = (∑ j | 0 ≤ j < 10 : b[j])}

```

Show that $Q \rightarrow wp(S_0, P)$:

$$\begin{aligned}
& Q \rightarrow wp(S_0, P) \\
\leftrightarrow & \text{ < instantiation >} \\
& T \rightarrow wp("i, s := 0, 0'', 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j])) \\
\leftrightarrow & \text{ < } \rightarrow \text{-simplification, wp := >} \\
& 0 \leq 0 \leq 10 \wedge 0 = (\sum j | 0 \leq j < 0 : b[j]) \\
\leftrightarrow & \text{ < arithmetic, empty range >} \\
& T \wedge 0 = 0 \\
\leftrightarrow & \text{ < arithmetic, } \wedge \text{-simplification >} \\
& T
\end{aligned}$$

We conclude that $P : 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j])$ holds just before the loop.

Show that $P \wedge BB \rightarrow wp(S_1, P)$:

$$\begin{aligned}
& P \wedge BB \rightarrow wp(S_1, P) \\
\leftrightarrow & \text{ < instantiation >} \\
& P \wedge BB \rightarrow wp("i, s := i + 1, s + b[i]''', 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j])) \\
\leftrightarrow & \text{ < instantiation, wp := >} \\
& 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j]) \wedge i < 10 \\
& \quad \rightarrow 0 \leq i + 1 \leq 10 \wedge s + b[i] = (\sum j | 0 \leq j < i + 1 : b[j]) \\
\leftrightarrow & \text{ < arithmetic, split range >} \\
& 0 \leq i < 10 \wedge s = (\sum j | 0 \leq j < i : b[j]) \\
& \quad \rightarrow (-1 \leq i < 10 \wedge s + b[i] = (\sum j | 0 \leq j < i : b[j]) + b[i]) \\
\leftrightarrow & \text{ < arithmetic, split range >} \\
& \frac{0 \leq i < 10 \wedge s = (\sum j | 0 \leq j < i : b[j])}{\rightarrow (-1 = i \vee 0 \leq i < 10) \wedge s = (\sum j | 0 \leq j < i : b[j])} \\
\leftrightarrow & \text{ < weakening/strengthening >} \\
& T
\end{aligned}$$

We conclude that if $P : 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j])$ holds before an iteration, it again holds after that iteration. By Mathematical Induction, it thus holds before and after every iteration.

Show that $P \wedge \neg BB \rightarrow R$:

$$\begin{aligned}
& P \wedge \neg BB \rightarrow R \\
\leftrightarrow & \text{ < instantiation >} \\
& 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j]) \wedge \neg(i < 10) \rightarrow s = (\sum j | 0 \leq j < 10 : b[j]) \\
\leftrightarrow & \text{ < arithmetic >} \\
& 0 \leq i \leq 10 \wedge s = (\sum j | 0 \leq j < i : b[j]) \wedge i \geq 10 \rightarrow s = (\sum j | 0 \leq j < 10 : b[j]) \\
\leftrightarrow & \text{ < arithmetic >}
\end{aligned}$$

$$\begin{aligned}
& i = 10 \wedge s = (\sum j | 0 \leq j < i : b[j]) \rightarrow s = (\sum j | 0 \leq j < 10 : b[j]) \\
& \leftrightarrow \quad \text{< substitution >} \\
& i = 10 \wedge \underline{s = (\sum j | 0 \leq j < 10 : b[j])} \rightarrow \underline{s = (\sum j | 0 \leq j < 10 : b[j])} \\
& \leftrightarrow \quad \text{< weakening/strengthening >} \\
& T
\end{aligned}$$

We conclude that if the loop terminates, it terminates in a state where R holds, and the program is thus correct *if the loop terminates*.

End of example.

5.3 DO-Theorem, partial correctness

The insights can be summarized in the following theorem:

Theorem 22 (DO-Theorem, Partial Correctness) *Let predicate P , which we call the (loop) invariant, satisfy*

- $Q \rightarrow P$; **and**
- $P \wedge B_1 \rightarrow \text{wp}(S_1, P)$; **and**
- \vdots
- $P \wedge B_n \rightarrow \text{wp}(S_n, P)$; **and**
- $P \wedge (\forall i | 1 \leq i \leq n : \neg B_i) \rightarrow R$.

*If the loop is entered in a state where Q is true **and** the loop terminates, then it will terminate in a state where R is true.*

5.4 DO-Theorem, Total Correctness

The discussion so far allows for the possibility that the loop never ends, which is usually a error. The next step is to motivate a mechanism for showing that a loop finishes.

In Example ??, one can reason that the loop ends because the variable i increases in each iteration and cannot exceed 10. This motivates a general approach: Find some integer valued function of the variables used in the loop, let us call it t , with the following properties:

- The function decreases every iteration; **and**
- The function can be shown to be bounded below.

In Example ??, the function can be chosen as $t = (10 - i)$ and it can be shown that $t \geq 0$.

The next question becomes how to show rigorously that a chosen function t has the desired property. Again consider the simple annotated loop

$$\begin{array}{l}
\{P\} \\
\text{do} \\
\quad \{P\} \\
\quad BB \rightarrow S_1
\end{array}$$

$$\begin{array}{l} \{P\} \\ \mathbf{od} \\ \{P\} \end{array}$$

What we want to show is that if the invariant P holds as well as the guard BB , then the value of function t decreases. We do so by replacing

$$\begin{array}{l} \{P\} \\ BB \rightarrow S_1 \\ \{P\} \end{array}$$

by

$$\begin{array}{l} \{P\} \\ BB \rightarrow t^{\text{old}} := t; \quad S_1 \\ \{P \wedge t < t^{\text{old}}\} \end{array}$$

Since (as part of proving Partial Correctness) we already show that $P \wedge BB \rightarrow wp("S_1", P)$, it remains to be shown that

$$P \wedge BB \rightarrow wp("t^{\text{old}} := t; S_1, t < t^{\text{old}})$$

Note: In our lectures we have used t' for t^{old} .

Example 23 Show that $P \wedge BB \rightarrow wp("t^{\text{old}} := t; S_1', t < t^{\text{old}})$ for the loop in Example ??, with $t = 10 - i$.

$$\begin{array}{l} P \wedge BB \rightarrow wp("t^{\text{old}} := t; S_1', t < t^{\text{old}}) \\ \leftrightarrow \quad \langle \text{instantiation} \rangle \\ P \wedge BB \rightarrow wp("t^{\text{old}} := t; S_1'', t < t^{\text{old}}) \\ \leftrightarrow \quad \langle \text{instantiation, wp ; } \rangle \\ P \wedge BB \rightarrow wp("t^{\text{old}} := 10 - i'', wp("S_1'', 10 - i < t^{\text{old}})) \\ \leftrightarrow \quad \langle \text{wp} := \rangle \\ P \wedge BB \rightarrow wp("t^{\text{old}} := 10 - i'', 10 - (i + 1) < t^{\text{old}}) \\ \leftrightarrow \quad \langle \text{wp} := \rangle \\ P \wedge BB \rightarrow 10 - (i + 1) < 10 - i \\ \leftrightarrow \quad \langle \text{arithmetic} \rangle \\ P \wedge BB \rightarrow T \\ \leftrightarrow \quad \langle \rightarrow\text{-simplification} \rangle \\ T \end{array}$$

End of example.

Note: Rarely does one need to instantiate P and BB on the left of the implication.

Also, we want to show that if the invariant P holds as well as the guard BB , then the value of the chosen integer function t is bounded below. For simplicity, we will assume that we need to show $t \geq 0$. We do so by checking that $P \wedge BB \rightarrow t > 0$.

Example 24 Show that $P \wedge BB \rightarrow t > 0$ for the loop in Example ??, with $t = 10 - i$.

$$P \wedge BB \rightarrow t > 0$$

↔ < instantiation >

$$P \wedge BB \rightarrow 10 - i > 0$$

↔ < arithmetic >

$$P \wedge \underline{BB} \rightarrow \underline{i < 10}$$

↔ < weakening/strengthening >

$$T$$
End of example.

Note: Rarely does one need to instantiate P on the left of the implication.

We summarize the above observations in the following Theorem:

Theorem 25 (DO-Theorem, total correctness) *Let predicate P , which we call the (loop) invariant, satisfy*

- $Q \rightarrow P$; **and**
- $P \wedge B_1 \rightarrow \text{wp}(S_1, P)$; **and**
- \vdots
- $P \wedge B_n \rightarrow \text{wp}(S_n, P)$; **and**
- $P \wedge (\forall i | 1 \leq i \leq n : \neg B_n) \rightarrow R$.
- *There exists t , a function of the variables that occur in the loop, such that*
 - $(\forall i | 1 \leq i \leq n : P \wedge B_i \rightarrow \text{wp}("t^{\text{old}} := t; S_i'', t < t^{\text{old}}));$ **and**
 - $(\forall i | 1 \leq i \leq n : P \wedge B_i \rightarrow t > 0)$.*(Note: the lower bound 0 is arbitrary. Any lower bound can be used.)*

If the loop is entered in a state where Q is true then the loop terminates in a state where R is true.

Goal-Oriented Programming

So far, we have discussed how to prove program segments correct. What we show next is that the proof of correctness can be performed hand-in-hand with the development of the program, making programming goal-oriented. We will focus on developing loops.

6.1 General structure of a loop-based program

Experience tells us that a loop-based program, annotated with assertions, will have the structure

Step	Annotated algorithm	
1a	$\{Q\}$	precondition
4	S_I	initialization command
2	$\{P\}$	invariant holds before the loop
	do	
2	$\{P\}$	invariant holds before each iteration
3	$B \rightarrow$	guard
2, 3	$\{P \wedge B\}$	state if guard holds
5	S_L	update
2	$\{P\}$	invariant holds after each iteration
	od	
2,3	$\{P \wedge \neg B\}$	invariant holds after loop and guard is <i>false</i>
1b	$\{R\}$	postcondition

which we will call the *worksheet*. The column labeled “Steps” indicates the order in which the worksheet will be filled, as we will discuss next.

In the remainder of this section we will use a few examples to illustrate the approach.

6.2 Scanning an array

Example 26 Let $b[0 \dots (n - 1)]$ be an array of integers. Develop a program that computes i , the index of the first element of b that equals zero.

Step 1: Specify the input and output The example indicates what is to be computed. What we need to do first is translate this into a mathematical specification of the precondition Q and postcondition R :

- $Q : 1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)$.
- $R : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0$.

These are entered for Step 1a and 1b in the worksheet.

Step	
1a	$\{1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
4	S_I
2	$\{P\}$
	do
2	$\{P\}$
3	$B \rightarrow$
2, 3	$\{P \wedge B\}$
5	S_L
2	$\{P\}$
	od
2,3	$\{P \wedge \neg B\}$
1b	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0\}$

Step 2: Determine an invariant The next step is to determine a loop invariant.

No computation happens between where $P \wedge \neg B$ holds and where R must hold. Thus, it must be the case that

$$P \wedge \neg B \rightarrow 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0.$$

Frequently it is the case that $P \wedge \neg B$ is exactly R . (Notice that then certainly $P \wedge \neg B \rightarrow R$, since in this case $P \wedge \neg B \leftrightarrow R$). In other words $R = (P \wedge \neg B)$. (Recall that $p \wedge q \rightarrow p$ and hence p is weaker than $p \wedge q$.)

Now, in our example the post condition is

$$R : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0.$$

While the loop is executing, that i is such that $b[i] = 0$ has not necessarily been achieved. This suggests weakening R to

$$P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)$$

which can be further manipulated to

$$\begin{aligned}
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid 0 \leq j < n : b[j] = 0) \\
\leftrightarrow & \text{ < split range >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \\
& \quad \wedge \left((\exists j \mid 0 \leq j < i : b[j] = 0) \vee (\exists j \mid i \leq j < n : b[j] = 0) \right) \\
\leftrightarrow & \text{ < distributivity >} \\
& 0 \leq i < n \wedge \left[\begin{aligned} & \left((\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid 0 \leq j < i : b[j] = 0) \right) \\ & \vee \left((\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \right) \end{aligned} \right] \\
\leftrightarrow & \text{ < DeMorgan's Law >} \\
& 0 \leq i < n \wedge \left[\begin{aligned} & \left(\neg(\exists j \mid 0 \leq j < i : b[j] = 0) \wedge (\exists j \mid 0 \leq j < i : b[j] = 0) \right) \\ & \vee \left((\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \right) \end{aligned} \right]
\end{aligned}$$

↔ < excluded middle >

$$0 \leq i < n \wedge (F \vee ((\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)))$$

↔ < \vee -simplification >

$$0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)$$

This last line represents the following state:

No zero has yet been found in the part of the array that has already been scanned ($0 \leq j < i$) and there exists a zero in the part of the array that remains ($i \leq j < n$).

It is entered for Step 2 in the worksheet:

Step	
1a	$\{1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
4	S_I
2	$\{P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
	do
2	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
3	$b[i] \neq 0 \rightarrow$
2, 3	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge B\}$
5,6	S_L
2	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
	od
2,3	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge \neg B\}$
1b	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0\}$

Note: it pays to examine the precondition Q and see if all the information in that predicate is somehow relevant to the invariant. This, in part, motivates the given weakening of R to yield P .

Step 3: Determine the loop guard The next question is how to pick B . We note that it must be the case that $P \wedge \neg B \rightarrow R$ and thus B must be chosen to make this *true*.

In the case of our example, P was derived from R by weakening $b[i] = 0$. Thus, it seems logical to try $\neg B = (b[i] = 0)$, or, equivalently, $B = (b[i] \neq 0)$. Then, indeed, $P \wedge \neg B \rightarrow R$:

$$P \wedge \neg B \rightarrow R$$

↔ < instantiation >

$$0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge \neg b[i] \neq 0 \rightarrow \\ 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0$$

↔ < arithmetic >

$$0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge b[i] = 0 \rightarrow \\ 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0$$

↔ < weakening/strengthening >

T

Inserting $b[i] \neq 0$ for B in the worksheet yields

Step	
1a	$\{1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
4	S_I
2	$\{P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
	do
2	$\{P\}$
3	$b[i] \neq 0 \rightarrow$
2, 3	$\{P \wedge b[i] \neq 0\}$
5,6	S_L
2	$\{P\}$
	od
2,3	$\{P \wedge \neg(b[i] \neq 0)\}$
1b	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0\}$

Note: Filling in $b[i] \neq 0$ in the worksheet is the first component of the actual final algorithm. Until Step 3, we had *only* been deriving the annotations (assertions) that were guiding the derivation of the algorithm itself.

Step 4: Determine the initialization S_I The command S_I typically initializes variables. The question one should ask oneself is “What command, S_I , when started in a state where Q holds, will establish P ?” Command S_I clearly must have the property that $Q \rightarrow wp(S_I, P)$.

For our example, focus on

$\{Q : 1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
S_I
$\{P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$

Notice that $S_I : i := 0$ has the desired property:

$$\begin{aligned}
& Q \rightarrow wp(\text{“}S_I\text{“}, P) \\
& \leftrightarrow \text{ < instantiation >} \\
& Q \rightarrow wp(\text{“}i := 0\text{“}, 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)) \\
& \leftrightarrow \text{ < definition := >} \\
& Q \rightarrow 0 \leq 0 < n \wedge (\forall j \mid 0 \leq j < 0 : b[j] \neq 0) \\
& \leftrightarrow \text{ < arithmetic, empty range >} \\
& Q \rightarrow 0 \leq n \wedge T \\
& \leftrightarrow \text{ < instantiation, \wedge-simplification >} \\
& 1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0) \rightarrow 0 \leq n \\
& \leftrightarrow \text{ < weakening/strengthening >} \\
& T
\end{aligned}$$

Inserting $i := 0$ for S_I in the worksheet yields

Step	
1a	$\{1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
4	$i := 0$
2	$\{P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
	do
2	$\{P\}$
3	$b[i] \neq 0 \rightarrow$
2, 3	$\{P \wedge b[i] \neq 0\}$
5,6	S_L
2	$\{P\}$
	od
2,3	$\{P \wedge \neg(b[i] \neq 0)\}$
1b	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0\}$

Step 5: Determine S_L Next, we need to determine how to march through the loop. Notice that initially $i = 0$ by virtue of $S_I : i := 0$. This means that we will be scanning through array b incrementally from element with smallest index, $i = 0$. This suggests that the loop index, i , should increase each time through the loop

$$S_{L2} : i := i + 1.$$

This leaves us to determine S_{L1} so that

$\{P \wedge B\}$
$S_{L1}; S_{L2}$
$\{P\}$

which can also be written and annotated as

$\{P \wedge B\}$
S_{L1}
$\{wp(S_{L2}, P)\}$
S_{L2}
$\{P\}$

Now

$$\begin{aligned}
& wp(S_{L2}, P) \\
& \leftrightarrow \text{ < instantiate >} \\
& wp("i := i + 1", 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)) \\
& \leftrightarrow \text{ < definition := >} \\
& 0 \leq i + 1 < n \wedge (\forall j \mid 0 \leq j < i + 1 : b[j] \neq 0) \wedge (\exists j \mid i + 1 \leq j < n : b[j] = 0) \\
& \leftrightarrow \text{ < arithmetic >} \\
& -1 \leq i < n - 1 \wedge (\forall j \mid 0 \leq j < i + 1 : b[j] \neq 0) \wedge (\exists j \mid i + 1 \leq j < n : b[j] = 0)
\end{aligned}$$

so that S_{L1} must be chosen to make

$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge b[i] \neq 0\}$
S_{L1}
$\{-1 \leq i < n - 1 \wedge (\forall j \mid 0 \leq j < i + 1 : b[j] \neq 0) \wedge (\exists j \mid i + 1 \leq j < n : b[j] = 0)\}$

hold.

Now, the precondition can be manipulated via the steps

$$\begin{aligned}
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge b[i] \neq 0 \\
\leftrightarrow & \text{ < rearrange, split range >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] \neq 0 \\
& \quad \wedge \left(b[i] = 0 \vee (\exists j \mid i+1 \leq j < n : b[j] = 0) \right) \\
\leftrightarrow & \text{ < } p \wedge (\neg p \vee q) \leftrightarrow p \wedge q \text{ >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] \neq 0 \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \\
\leftrightarrow & \text{ < combine range >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \\
\leftrightarrow & \text{ < arithmetic >} \\
& (0 \leq i < n-1 \vee i = n-1) \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge \\
& \quad (\exists j \mid i+1 \leq j < n : b[j] = 0) \\
\leftrightarrow & \text{ < distributivity >} \\
& \left(0 \leq i < n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \right) \\
& \quad \vee \left(i = n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \right) \\
\leftrightarrow & \text{ < instantiation >} \\
& \left(0 \leq i < n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \right) \\
& \quad \vee \left((\forall j \mid 0 \leq j < n : b[j] \neq 0) \wedge (\exists j \mid n \leq j < n : b[j] = 0) \right) \\
\leftrightarrow & \text{ < empty range >} \\
& \left(0 \leq i < n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \right) \\
& \quad \vee \left((\forall j \mid 0 \leq j < n : b[j] \neq 0) \wedge F \right) \\
\leftrightarrow & \text{ < } \wedge\text{-simplification, } \vee\text{-simplification >} \\
& 0 \leq i < n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0)
\end{aligned}$$

This yields

$\{0 \leq i < n-1 \wedge (\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0)\}$
S_{L1}
$\left\{ \begin{array}{l} (i = -1 \vee 0 \leq i < n-1) \wedge \left((\forall j \mid 0 \leq j < i+1 : b[j] \neq 0) \wedge b[i] \neq 0 \right) \\ \wedge (\exists j \mid i+1 \leq j < n : b[j] = 0) \end{array} \right\}$

We conclude that the postcondition to S_{L1} is weaker than its precondition. This means that $S_{L1} : \mathbf{skip}$ will yield the desired result!

The resulting $S_L : \mathbf{skip}; i := i + 1$ is entered in the worksheet as $i := 1i + 1$:

Step	
1a	$\{1 \leq n \wedge (\exists j \mid 0 \leq j < n : b[j] = 0)\}$
4	$i := 0$
2	$\{P : 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0)\}$
	do
2	$\{P\}$
3	$b[i] \neq 0 \rightarrow$
2, 3	$\{P \wedge b[i] \neq 0\}$
5	$i := i + 1$
2	$\{P\}$
	od
2,3	$\{P \wedge \neg(b[i] \neq 0)\}$
1b	$\{0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge b[i] = 0\}$

Proving that the loop terminates We now need to check that the loop terminates. Since the index i increases to n , it makes sense to take the bound function to equal $t = (n - i)$. We must check that $P \wedge B \rightarrow t > 0$:

$$\begin{aligned}
& P \wedge B \rightarrow t > 0 \\
& \leftrightarrow \text{ < instantiation >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge B \rightarrow (n - i) > 0 \\
& \leftrightarrow \text{ < arithmetic >} \\
& 0 \leq i < n \wedge (\forall j \mid 0 \leq j < i : b[j] \neq 0) \wedge (\exists j \mid i \leq j < n : b[j] = 0) \wedge B \rightarrow i < n \\
& \leftrightarrow \text{ < weakening/strengthening >} \\
& T
\end{aligned}$$

and that $P \wedge B \rightarrow wp(\text{“}t^{\text{old}} := t; S_L\text{“}, t < t^{\text{old}})$:

$$\begin{aligned}
& P \wedge B \rightarrow wp(\text{“}t^{\text{old}} := t; S_L\text{“}, t < t^{\text{old}}) \\
& \leftrightarrow \text{ < definition := >} \\
& P \wedge B \rightarrow wp(\text{“}t^{\text{old}} := t\text{“}, wp(S_L\text{“}, t < t^{\text{old}})) \\
& \leftrightarrow \text{ < instantiation >} \\
& P \wedge B \rightarrow wp(\text{“}t^{\text{old}} := n - i\text{“}, wp(i := i + 1\text{“}, (n - i) < t^{\text{old}})) \\
& \leftrightarrow \text{ < definition := >} \\
& P \wedge B \rightarrow wp(\text{“}t^{\text{old}} := n - i\text{“}, (n - (i + 1)) < t^{\text{old}}) \\
& \leftrightarrow \text{ < definition := >} \\
& P \wedge B \rightarrow (n - (i + 1)) < n - i \\
& \leftrightarrow \text{ < arithmetic >} \\
& P \wedge B \rightarrow T \\
& \leftrightarrow \text{ < } \rightarrow\text{-simplification >} \\
& T
\end{aligned}$$

We conclude that the loop terminates.

Step	
1a	$\{0 \leq n \wedge \alpha = \hat{\alpha}\}$
4	$i := 0$
2	$\{P : 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j])\}$
	do
2	$\{P\}$
3	$i < n \rightarrow$
2, 3	$\{P \wedge i < n\}$
5	$\alpha := \alpha + x[i] \cdot y[i];$ $i := i + 1$
2	$\{P\}$
	od
2,3	$\{P \wedge \neg(i < n)\}$
1b	$\{0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])\}$

Figure 6.1: Worksheet for deriving an algorithm for updating a variable α by adding the result of an innerproduct.

Final algorithm The algorithm is now presented without the annotations that prove correct correctness by deleting the grey boxes:

```

i := 0
do
  b[i] ≠ 0 → i := i + 1
od

```

6.3 Inner product

Example 27 Let $x[0 \dots (n-1)]$ and $y[0 \dots (n-1)]$ be arrays of reals and let α be a real. Develop a program that adds the inner product of x and y to α , overwriting α :

$$\alpha := \alpha + \sum_{j=0}^{n-1} x[j] \cdot y[j]$$

or, in our notation,

$$\alpha := \alpha + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j]).$$

(In linear algebra notation: $\alpha := \alpha + x^T y$.)

It helps to review some properties of summation:

- If $n \leq i$ then $(\Sigma j \mid i \leq j < n : x[j] \cdot y[j]) = 0$.
- One can split the range: if $0 \leq i \leq n$, then

$$(\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j]) = (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + (\Sigma j \mid i \leq j < n : x[j] \cdot y[j]).$$

The worksheet for deriving an algorithm is given in Figure ???. We next describe the steps that filled that worksheet.

Step 1: Specify the input and output The example indicates what is to be computed. What we need to do first is translate this into a mathematical specification of the precondition Q and postcondition R :

- $Q : 0 \leq n \wedge \alpha = \hat{\alpha}$.
- $R : 0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])$.

Here $\hat{\alpha}$ indicates the original contents of α so that the postcondition expresses that the inner product has been added to the original contents.

These are entered for Step 1a and 1b in Figure ??.

Step 2: Determine an invariant The next step is to determine a loop invariant. In our example the post condition is

$$0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])$$

↔ [< split range >](#)

$$0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + (\Sigma j \mid i \leq j < n : x[j] \cdot y[j]).$$

While the loop is executing, we would expect only part of the summation to have completed, which suggests the invariant

$$P : 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]).$$

This is entered for Step 2 in Figure ??.

Step 3: Determine the loop guard It must be the case that $P \wedge \neg B \rightarrow R$. We notice that $i < n$ has this property:

$$P \wedge \neg B \rightarrow R$$

↔ [< instantiation >](#)

$$0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \wedge \neg(i < n) \rightarrow$$

$$0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])$$

↔ [< arithmetic >](#)

$$0 \leq i \leq n \wedge i = n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \rightarrow$$

$$0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])$$

↔ [< substitution >](#)

$$0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j]) \rightarrow$$

$$0 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < n : x[j] \cdot y[j])$$

↔ [< identity >](#)

T

The guard $i < n$ is entered in Step 3 of Figure ??.

Step 4: Determine the initialization S_I Command S_I must have the property that $Q \rightarrow wp(S_I, P)$.

For our example, focus on

$\{Q : 0 \leq n \wedge \alpha = \hat{\alpha}\}$
S_I
$\{P : 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j])\}$

Notice that $S_I : i := 0$ has the desired property:

$$\begin{aligned}
& Q \rightarrow wp("S_I", P) \\
& \leftrightarrow \text{ < instantiation >} \\
& Q \rightarrow wp("i := 0", 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j])) \\
& \leftrightarrow \text{ < definition := >} \\
& Q \rightarrow 0 \leq 0 < n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < 0 : x[j] \cdot y[j]) \\
& \leftrightarrow \text{ < instantiation, arithmetic, empty range >} \\
& 0 \leq n \wedge \alpha = \hat{\alpha} \rightarrow 0 \leq n \wedge \alpha = \hat{\alpha} \\
& \leftrightarrow \text{ < identity >} \\
& T
\end{aligned}$$

The initialization S_I is entered in Step 4 of Figure ??.

Step 5: Determine S_L Next, we need to determine how to march through the loop. Notice that initially $i = 0$ by virtue of $S_I : i := 0$. This suggests that we should increment i until it no longer less than n :

$$S_{L2} : i := i + 1.$$

This leaves us to determine S_{L1} so that

$\{P \wedge B\}$
$S_{L1}; S_{L2}$
$\{P\}$

which can also be written and annotated as

$\{P \wedge B\}$
S_{L1}
$\{wp(S_{L2}, P)\}$
S_{L2}
$\{P\}$

Now

$$\begin{aligned}
& wp(S_{L2}, P) \\
& \leftrightarrow \text{ < instantiate >} \\
& wp("i := i + 1", 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j])) \\
& \leftrightarrow \text{ < definition := >} \\
& 0 \leq i + 1 \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i + 1 : x[j] \cdot y[j]) \\
& \leftrightarrow \text{ < arithmetic, split range >} \\
& -1 \leq i < n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + x[i] \cdot y[i]
\end{aligned}$$

so that S_{L1} must be chosen to make

$\{0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \wedge i < n\}$
S_{L1}
$\{-1 \leq i < n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + x[i] \cdot y[i]\}$

hold. It is not hard to see that $S_{L1} : \alpha := \alpha + x[i] \cdot y[i]$ is the desired update:

$$\begin{aligned}
& 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \wedge i < n \rightarrow \\
& \quad wp(\alpha := \alpha + x[i] \cdot y[i], -1 \leq i < n \wedge \\
& \quad \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + x[i] \cdot y[i]) \\
& \leftrightarrow \text{ < definition := >} \\
& 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \wedge i < n \rightarrow \\
& \quad -1 \leq i < n \wedge \alpha + x[i] \cdot y[i] = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) + x[i] \cdot y[i] \\
& \leftrightarrow \text{ < arithmetic >} \\
& 0 \leq i \leq n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \wedge i < n \rightarrow \\
& \quad -1 \leq i < n \wedge \alpha = \hat{\alpha} + (\Sigma j \mid 0 \leq j < i : x[j] \cdot y[j]) \\
& \leftrightarrow \text{ < weakening/strengthening >} \\
& T
\end{aligned}$$

The update S_L is entered in Step 5 of Figure ??.

Proving the loop terminates We now need to check that the loop terminates. Since the index i increases to n , it makes sense to take the bound function to equal $t = (n - i)$. We must check that $P \wedge B \rightarrow t > 0$:

$$\begin{aligned}
& P \wedge B \rightarrow t > 0 \\
& \leftrightarrow \text{ < instantiation >} \\
& P \wedge i < n \rightarrow (n - i) > 0 \\
& \leftrightarrow \text{ < arithmetic >} \\
& P \wedge i < n \rightarrow i > n \\
& \leftrightarrow \text{ < weakening/strengthening >} \\
& T
\end{aligned}$$

and that $P \wedge B \rightarrow wp(\alpha := \alpha + x[i] \cdot y[i], t < t^{\text{old}})$:

$$\begin{aligned}
& P \wedge B \rightarrow wp(\alpha := \alpha + x[i] \cdot y[i], t < t^{\text{old}}) \\
& \leftrightarrow \text{ < definition ; >} \\
& P \wedge B \rightarrow wp(\alpha := \alpha + x[i] \cdot y[i], wp(\alpha := \alpha + x[i] \cdot y[i], t < t^{\text{old}})) \\
& \leftrightarrow \text{ < instantiation >} \\
& P \wedge B \rightarrow wp(\alpha := \alpha + x[i] \cdot y[i], wp(\alpha := \alpha + x[i] \cdot y[i], t < t^{\text{old}})) \\
& \leftrightarrow \text{ < definition ; definition := \times 2 >} \\
& P \wedge B \rightarrow wp(\alpha := \alpha + x[i] \cdot y[i], (n - (i + 1)) < t^{\text{old}}) \\
& \leftrightarrow \text{ < definition := >} \\
& P \wedge B \rightarrow (n - (i + 1)) < n - i \\
& \leftrightarrow \text{ < arithmetic >} \\
& P \wedge B \rightarrow T \\
& \leftrightarrow \text{ < \rightarrow-simplification >} \\
& T
\end{aligned}$$

We conclude that the loop terminates.

Final algorithm The algorithm is now presented without the annotations that prove correct correctness by deleting the grey boxes:

```
i := 0
do
  i < n → α := α + x[i] · y[i];
  i := i + 1
od
```