Proof-Outline Logic (Concurrent Programming Edition)

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

Draft typeset May 12, 2014

Abstract

An introduction to proof outlines, compiled as background reading for Engi 8893 Concurrent Programming and Engi 9869 Advanced Concurrent Programming.

Note on editions: This is the Concurrent Programming edition, designed to support MUN courses Engi-7893 and Engi-9869. The first part of this note —the part up to the section entitled Concurrent Programming— also appears in an edition for MUN course Engi-6892 Algorithms: Correctness and Complexity. The content is the same, but the notation is a bit different. The notation used in this edition better matches the notation used in Andrews' text book [Andrews, 2000]. [For students who took Engi-6892 in 2013 or earlier: since earlier editions of this note and the edition for Engi-6892 I've made the following changes in terminology. I formerly said that a condition was "valid" where I now say it is "universally true". And I formally said that a Hoare triple or a proof outline was "valid" where I now say it is "partially correct".]

1 Preface

This note provides background on assertions and the use of assertions in designing correct programs.

The ideas presented are mainly due, for the sequential programming part, to Floyd [Floyd, 1967] and Hoare [Hoare, 1969], and, for the concurrent programming part, to Lamport [Lamport, 1977] and Gries and Owicki

[Owicki and Gries, 1976b, Owicki and Gries, 1976a]. Blikle [Blikle, 1979] presented an early version of proof-outline logic for sequential programs. An excellent and detailed study of Owicki/Gries theory is to be found in [Feijen and van Gasteren, 1999], where the theory is expanded from a set of rules for checking proofs of parallel programs to a method for developing proofs of parallel programs.

Sections 2 and 3 deal with sequential programming. As such they are an elaboration of Hoare's excellent 'Axiomatic basis' paper [Hoare, 1969]. I suggest reading Hoare's paper first. For sequential programs, proof-outline logic is just like Hoare logic except that commands contain internal assertions. Section 4 then extends the rules to cover concurrent programs.

Section 5 describes the syntax and semantics a bit more formally than the preceding sections. It is entirely optional reading.

Section 6 gives some example and shows some handy tricks for showing noninterference. Section 7 discusses global invariants, which are assertions that are true throughout the execution of a concurrent program. It gives a handy abbreviation that saves you from having to write global invariants over and over, and, more importantly, concludes with an example of proving a communication protocol. The final two sections show applications of global invariants. Section 8 shows how introducing extra variables can simplify proofs and Section 9 shows how to use a sort of coordinate transformation to change the set of variables used in a program.

The programming notation that I use is based on that in Gregory Andrews's text [Andrews, 2000], which is based on C. I'll make a few "improvements" to Andrews's notation. I will mention them as I go along but will summarize them here:

- Andrews uses the Fortran/C/Java notation for assignment: $v = E$;. I use the Algol/Pascal/Ada notation: $v := E$;.

- Andrews uses the C/Java notation for equality: $E == F$. I use the mathematical notation: $E = F$.

- Andrews brackets assertions with "##" and the end of the line (eol). I do this too, but as an alternative, I'll sometimes bracket them with "{" and "}", which is the tradition in Hoare logic. The latter notation is particularly useful when you don't want to be forced to put line breaks in the middle of a formula.

- Andrews groups commands with "{" and "}", as in C and Java. I do that too, but to avoid confusion with assertions, I'll sometimes use "("

and ")". For larger examples I'll use ##/eol for assertions with {/} for grouping commands; for smaller examples and in the theory, I'll use {/} for assertions with (/) for grouping commands.

2 Conditions and assertions

2.1 Assertions

A **condition** is a boolean expression with free variables chosen from the state variables of a program. For example if we have variables

```
int x ;
int y ;
```

Then the following are all examples of conditions

$$\begin{aligned} x &< y \\ x &= y \\ x + y &= 0 \\ x &\geq 0 \\ x &\geq 0 \land x + y = 0 \qquad . \end{aligned}$$

A condition that is expected to be true every time execution passes a particular point in a program is called an **assertion**. In this course, assertions are preceded by ## and are followed by an end-of-line like this:[1]

```
int x ;
int y ;
x := 5 ;
y := -5 ;
## $x \geq 0 \land x + y = 0$
z := x+z ;
```

Sometimes I'll write assertions insides curly brackets like this:

```
int x ; int y ; x := 5 ; y := -5 ; {$x \geq 0 \land x + y = 0$} z := x+y ;
```

[1] This fragment uses both the symbol := and =.This is one of those "notational improvements" I mentioned. I will use := for assignment and either = or == for equality when writing pseudo-code. In C, C++, and Java, = is used for assignment and == for equality. Andrews follows the C/C++/Java convention. If you want to be on the safe side of any possible misunderstanding, you can use := for assignment and == for equality.

2.2 Assertions in C, C++, and Java

If one is programming in C or C++, then assertions may be written either as comments or using the assert macro from the standard C library. E.g.

```
#include <assert.h>
...
int x ;
int y ;
x = 5 ;
y = -5 ;
assert( x >=0 && x+y == 0 ) ;
```

Assertions written using the assert macro will be evaluated at run time and the program will come to a grinding halt, should the assertion ever evaluate to false.[2]

In Java one easily create one's own Assert class with a check method in it.[3]

```
public class Assert {
    public static void check( boolean b ) {
        if( !b ) { throw new java.lang.AssertionError() ; }
    }
}
```

This can be used in your code as follows:

```
int x ;
int y ;
x = 5 ;
y = -5 ;
Assert.check( x >=0 && x+y == 0 ) ;
```

[2] By using a different include file, one can, of course, make the action followed on a false assertion be whatever you like. For example, in a desktop application, one might cause all files to be saved and an error report to be assembled and e-mailed back to the developers; in an embedded system, one might cause the system to go into safe mode. If you program in C or C++, I strongly suggest redefining the assert macro in a way that suits your application. And use it!

[3] As of Java 1.4, there is actually an assert keyword in Java. However I don't recommend its use. Assertion checking is turned off by default in most (if not all) JVMs. This can be compared to removing the seat belts from a car's design once it goes into production. In my own work I use my own assertion checking class. I recommend you do the same.

2.2.1 Be an assertive programmer

Using assertions has several benefits.

- In the design process, they help you articulate what conditions you expect to be true at various points in program execution.
- In testing, executable assertions can help you identify errors in your code or in your design.
- In execution, executable assertions —combined with a recovery mechanism— can help make your program more fault tolerant.
- Assertions provide valuable documentation. Executable assertions are more valuable than comments, as they are more likely to be accurate.

Whether to code assertions as comments or as executable checks is a question that depends on the local conditions of the project you are working on. Sometimes has to be answered on a case-by-case basis. In this course we will concentrate on the use of assertions in the design process, rather than on their (nevertheless important) uses in testing, documentation, and in making systems fault-tolerant. My general advice is to make assertions executable as much as is practical.[4]

2.3 Substitutions

Sometimes it is useful to create a new condition by replacing all free occurrences of a variable x in a condition P by an expression (E). We write $P_{x \leftarrow E}$

[4] In concurrent programming there is an additional complication in making assertions executable, namely that they should be evaluated atomically. Consider the assertion

$$x = 0 \vee y = 0$$

if we evaluate this in parallel with the following sequence of assignments

$$x := 0; y := 1;$$

it is possible that the assertion will evaluate to false even though there is no time at which it is in fact false.

This problem can be solved by evaluating assertions only when the thread has exclusive access to the data they refer to.

for the new condition.[5] For example

$$\begin{array}{lll}(x \geq 0 \wedge x+y=0)_{x \leftarrow z} & \text{is} & (z) \geq 0 \wedge (z)+y=0 \\ (x \geq 0 \wedge x+y=0)_{x \leftarrow x+y} & \text{is} & (x+y) \geq 0 \wedge (x+y)+y=0 \\ (2y=5)_{y \leftarrow y+z} & \text{is} & 2(y+z)=5 \quad .\end{array}$$

It is useful to extend this notation to allow the simultaneous substitution for more than one variable. For example

$$(x \geq 0 \wedge x+y=0)_{x,y \leftarrow z,x} \text{ is } (z) \geq 0 \wedge (z)+(x)=0$$

Usually we omit the parentheses in contexts where they are not required.

2.4 Propositional and predicate logic

In this section, I will review a little bit of propositional and predicate logic. We use notations $\neg$ (not), $=$ (equality), $\wedge$ (and), $\vee$, (or), and $\Rightarrow$ (implication). Precedence between the operators is in the same order. Some of the laws of propositional logic that will be useful in this course are

$(P \Rightarrow Q) = (\neg P \vee Q)$	Material implication
$(\mathit{true} \Rightarrow P) = P$	Identity
$\mathit{false} \Rightarrow P$	Antidomination
$P \Rightarrow P$	Reflexivity
$P \Rightarrow \mathit{true}$	Domination
$(P \Rightarrow (Q \Rightarrow R)) = (P \wedge Q \Rightarrow R)$	Shunting
$(P0 \Rightarrow Q) \Rightarrow (P0 \wedge P1 \Rightarrow Q)$	Subsetting the antecedent
$(P \Rightarrow Q \wedge R) = (P \Rightarrow Q) \wedge (P \Rightarrow R)$	Distributivity

Also frequently useful are the **one-point laws**, which let you make use information from equalities. E and F range over expressions of any type, v is a variable.

$$(E = F \Rightarrow P_{v \leftarrow E}) = (E = F \Rightarrow P_{v \leftarrow F})$$
$$(E = F \wedge P_{v \leftarrow E}) = (E = F \wedge P_{v \leftarrow F})$$

To see that these are true, consider the case where $E = F$ and then the case where $E \neq F$.

For example

$$x = X \wedge y = Y \Rightarrow x^y = X^Y$$

[5] A variety different notations are used by authors for substitution. In other courses, I usually use $P[x : E]$. Here I am following Andrews's book.

simplifies, using one-point (and shunting), to

$$x = X \wedge y = Y \Rightarrow X^Y = X^Y$$

which then simplifies to

$$x = X \wedge y = Y \Rightarrow \mathit{true}$$

which is then true.

Any condition that is true for all assignments of values to its free variables, is called a **universally true** formula. For example, (assuming x and y and z are integer variables) the following are all universally true

$$\begin{aligned}
&2 + 2 = 4 \\
&x < y \wedge y < z \Rightarrow x < z \\
&x + 1 > x
\end{aligned}$$

However, $x^2 + y^2 = z^2$ is not universally true, because there is an assignment for which it is not true; for example $3^2 + 4^2 = 6^2$ is not true.

Whether a condition is universally true may depend on the types ascribed to its variables. For example, if we are using Java and x has type **int**, then, by the rules of the Java language, the value of $x + 1$, when x is $2^{31} - 1$, is -2^{31}; so $x + 1 > x$ is not universally true in that case.

Sometimes expressions are undefined, for example, supposing x and y are rational variables, x/y is undefined when y is 0, so this raises the question of whether $1 = x/x$ is universally true. We'll say that such an expression is not universally true. However, the expression $x \neq 0 \Rightarrow 1 = x/x$ is universally true; if we consider the case of $x = 0$, we have false $\Rightarrow$?, and applying the principle of antidomination, this is true. Using ? to represent an unknown or undefined truth value, we can fill in truth tables for the propositional logic as follows

$\neg$	
false	true
?	?
true	false

$\wedge$	false	?	true
false	false	false	false
?	false	?	?
true	false	?	true

$\vee$	false	?	true
false	false	?	true
?	?	?	true
true	true	true	true

$\Rightarrow$	false	?	true
false	true	true	true
?	?	?	true
true	false	?	true

3 Sequential programming

3.1 Contracts

A specification for a component indicates the operating conditions (i.e. the conditions under which the component is expected to operate) and the function of the component (i.e. the relationship between the components inputs and outputs). For example we might specify a resistor by saying that the relationship between the voltage and current across the resistor is given by

$$953I \leq V \leq 1050I \qquad ,$$

provided

$$0 \leq V \leq +10 \qquad .$$

The latter formula gives the operating conditions, the former the relation between inputs and outputs.

In programming, we can use a pair of assertions as a specification or "contract" for a command or subroutine. The first assertion is the so-called precondition, it specifies the operating conditions, that is, the state of the program when the command begins operation. The second assertion specifies the state of the program when (and if) the command ends operation. For example, the following pair of conditions specifies a command that results in x being assigned the value 5, provided that y is initially 4:

$$[y = 4, x = 5]$$

where x and y are understood to be state variables of type int. One solution to this particular contract is

```
## $y = 4$
$x := y + 1$ ;
## $x = 5$      .
```

Such a triple, consisting of a precondition, a command, and a postcondition (ignoring the variable declarations), is called a **Hoare triple** after C.A.R. Hoare, who introduced the idea to programming.[6] Here is another solution:

```
## $y = 4$
```

[6] Traditionally Hoare triples are written with the assertions in braces. So we would, traditionally, write

$$\{y = 4\} \quad x := y + 1 \quad \{x = 5\}$$

I'm going to use braces sometimes and the ## convention at others.

```
$x := 5$ ;
$\#\# \; x = 5$      .
```

Here is one more:

```
$\#\# \; y = 4$
$y := y + 1$ ;
$x := y$ ;
$\#\# \; x = 5$      .
```

Nothing in the contract says that y must not change!

If you do want to specify that a variable does not change, then 'constants' can be used. Constants are conventionally written with capital letters. The following contract specifies that, provided y is initially less than 100, y must not change and the final value of x must be larger than that of y:

$$[y < 100 \wedge y = Y, y = Y \wedge x > y]$$

where it is understood that x and y are state variables of type int, while Y is a constant[7] of type int.

3.2 Partial correctness

Consider a Hoare triple $\{P\}\ S\ \{Q\}$ or equivalently

```
$\#\# \; P$
$S$
$\#\# \; Q$
```

We define that the triple is **partially correct** if and only if, for all possible values of all constants, whenever the execution of S is started in a state

[7] The word 'constant' is the traditional term to use. In the mathematical sense, Y is a variable. We use the term 'constant' to distinguish such mathematical variables from "program variables" which refer to components of the program state. The point is that Y can't be changed by the execution of the program so Y represents the same value in both the precondition and in the postcondition. Since the precondition implies $y = Y$ and the postcondition implies $y = Y$, it is clear that y has the same value in the final state as it has in the initial state.

It is so common to use constants equated to the initial values of variables that the following convention has evolved. The notation e_o is used to refer to the value of expression e in the original state. Thus this contract could be written as

$$[y < 100, y = y_o \wedge x > y]$$

satisfying P, the execution of S does not crash and can only end in a state satisfying Q.

Note that the definition of partial correctness does not require that S should terminate. Thus the following triple is partially correct even if we interpret x to have the type $\mathbb{Z}$, that is, to range over all mathematical integers

$$\{\text{true}\} \quad \textbf{while}(x \neq 0)\ x := x - 1; \quad \{x = 0\} \qquad .$$

If we consider initial states where x is positive or zero, then eventually the while-loop will terminate and the program will halt in a state where $x = 0$, satisfying the postcondition. If we start the while-loop in an initial state where x is negative, then the while-loop will never terminate and so execution "can only end in a state satisfying" the postcondition by virtue of the fact it never ends at all!

We write $\vdash \{P\}\ S\ \{Q\}$ to mean "$\{P\}\ S\ \{Q\}$ is partially correct"

In concurrent programming, we are often interested in processes that do not terminate (e.g., in embedded systems) so dealing with partial correctness is an appropriate and desirable thing to do. If termination is important, we can deal with it as a separate concern. From here on, we won't be worried about any other kind of correctness, so we will just say "correct".

3.3 Some examples of assignments and a rule

Here are some small examples of Hoare triples. In each case the variables should be understood to be integers

$$\{x + 1 = y\}\ x := x + 1;\ \{x = y\}$$

Is this triple correct? (Answer for yourself before reading on...) If initially y is $x + 1$ and we change x to $x + 1$, then finally both x and y will equal the original value of $x + 1$, and so they will equal each other. Yes, it is correct.

How about

$$\{2x = 3y\}\ x := 2x;\ \{x = 3y\}$$

Is this correct? Well, if initially $2x$ is $3y$, then, after changing x to $2x$, finally x will be $3y$.

These two examples suggest a general rule, which is

$$\vdash \{Q_{x \leftarrow E}\}\ x := E;\ \{Q\}$$

When Q is the postcondition of an assignment $x := E$, we call $Q_{x \leftarrow E}$ the **substituted postcondition**. More generally, it is sufficient for the precondition to imply $Q_{x \leftarrow E}$, so a better rule is

$$\vdash \{P\}\ v := E;\ \{Q\} \text{ exactly if } P \Rightarrow Q_{v \leftarrow E} \text{ is universally true}$$

where v is any variable, E is any expression, and P and Q are any conditions. Here is an example where the precondition is stronger than it needs to be:

$$\{2x < 3y\}\ x := 2x;\ \{x \leq 3y\}$$

The substituted postcondition is $(x \leq 3y)_{x \leftarrow 2x}$, which is $2x \leq 3y$. By the assignment rule, this triple is correct exactly if $2x < 3y \Rightarrow 2x \leq 3x$ is universally true.

There is one more aspect to assignment that should be mentioned. This is that the expression might not always be well defined. For example, if we divide by 0, this is an error and we should consider that if this happens the program has crashed. Since a command that crashes is not partially correct we should really ensure that our rule for assignments includes checking that the expression is well defined. Let's suppose that for each expression E there is a condition $\mathrm{df}[E]$ that says that E is well-defineed, i.e. does not crash when evaluated. For example $\mathrm{df}[x/y]$ might be $y \neq 0$. Now the improved assignment rule is

$$\begin{array}{ll} \vdash \{P\}\ v := E;\ \{Q\} & \text{exactly if } P \Rightarrow Q_{v \leftarrow E} \text{ is universally true} \\ & \text{and } P \Rightarrow \mathrm{df}[E] \text{ is universally true} \end{array} \qquad \text{(assignment rule)}$$

In many cases $\mathrm{df}[E]$ is simply true, and so it is trivial that $P \Rightarrow \mathrm{df}[E]$ is universally true.

This rule generalizes to simultaneous assignments to multiple variables. For example

$$\{x < y\}\ x, y := y, x;\ \{y \leq x\}$$

The substituted postcondition is $(y \leq x)_{x,y \leftarrow y,x}$, which is $x \leq y$; this is implied (for all values of x and y) by $x < y$.

Here is one last example of an assignment; it will be of use later.

$$\{y \geq 0 \wedge x = X \wedge y = Y\}\ z := 1;\ \left\{y \geq 0 \wedge X^Y = z \times x^y\right\}$$

First we find the substituted precondition

$$y \geq 0 \wedge X^Y = 1 \times x^y$$

which simplifies to

$$y \geq 0 \wedge X^Y = x^y$$

This (using one-point laws) is implied by the precondition $y \geq 0 \wedge x = X \wedge y = Y$.

3.4 A bigger example

Here is another example. I claim that

$$\{y \geq 0 \land x = X \land y = Y\}\ S\ \left\{z = X^Y\right\} \qquad , \tag{1}$$

is correct, where

$$\begin{aligned} S &\triangleq (z := 1; \textbf{while}(\ y > 0\ )\ T) \\ T &\triangleq \textbf{if}(\ \text{odd}(y)\ )\ U\ \textbf{else}\ V \\ U &\triangleq (z := z \times x; y := y - 1;) \\ V &\triangleq (x := x \times x; y := y/2;) \qquad . \end{aligned}$$

To show that this triple is correct, we'll need to deal with constructs other than assignments. For that we introduce a new idea: proof outlines.

3.5 Proof outlines

A **proof outline** is a command that is annotated with assertions. It represents the outline of a proof of the program. Figure 1 is a proof outline for the example of the last section.

A proof outline is not a proof: it is (if correct) a summary of a proof. This is why it is called a 'proof outline'.

3.6 Correctness of proof outlines

We can formally define **partially correct** proof outlines for sequential programs as follows:

Assignment Rule: $\{P\}\ v := E;\ \{Q\}$ is a partially correct proof outline if

$$P \Rightarrow Q_{v \leftarrow E} \text{ is universally true and } P \Rightarrow \text{df}[E] \text{ is universally true} \qquad .$$

Skip Rule: $\{P\}$ **skip** $\{Q\}$ is a partially correct proof outline if

$$P \Rightarrow Q, \text{ for all values of all variables} \qquad .$$

(Sequential) Composition Rule: $\{P\}\ S\ \{Q\}\ T\ \{R\}$ is a partially correct proof outline, provided $\{P\}\ S\ \{Q\}$ and $\{Q\}\ T\ \{R\}$ are both partially correct proof outlines.

2-Tailed If Rule: $\{P\}$ $\textbf{if}(\ E\ )\ \{Q_0\}\ S\ \textbf{else}\ \{Q_1\}\ T\ \{R\}$ is a partially correct proof outline, provided $\{Q_0\}\ S\ \{R\}$ and $\{Q_1\}\ T\ \{R\}$ are both

```
## $y \geq 0 \land x = X \land y = Y$
$z := 1$ ;
## $I : y \geq 0 \land X^Y = z \times x^y$
while( $y > 0$ )
## $X^Y = z \times x^y \land y > 0$
{
    if( odd($y$) )
    ## $X^Y = z \times x^y \land y > 0 \land \text{odd}(y)$
    {
        $z := z \times y$;
        ## $y - 1 \geq 0 \land X^Y = z \times x^{y-1}$
        $y := y - 1$;
    }
    else
    ## $X^Y = z \times x^y \land y > 0 \land \text{even}(y)$
    {
        $x := x \times x$;
        ## $\text{even}(y) \land y/2 \geq 0 \land X^Y = z \times x^{y/2}$
        $y := y/2$;
    }
}
## $z = X^Y$
```

Figure 1: An example proof outline.

partially correct proof outlines and that $P \Rightarrow \text{df}[E]$, $P \land E \Rightarrow Q_0$, and $P \land \neg E \Rightarrow Q_1$ are all universally true.

1-Tailed If Rule: $\{P\}$ **if**(E) $\{Q\}$ S $\{R\}$ is a partially correct proof outline, provided $\{Q\}$ S $\{R\}$ is a partially correct proof outline and that $P \Rightarrow \text{df}[E]$, $P \land E \Rightarrow Q$, and $P \land \neg E \Rightarrow R$ are all universally true.

Iteration Rule: $\{P\}$ **while**(E) $\{Q\}$ S $\{R\}$ is a partially correct proof outline, provided

- that $P \Rightarrow \text{df}[E]$ is universally true,
- that $P \land E \Rightarrow Q$ is universally true,
- that $P \land \neg E \Rightarrow R$ is universally true, and

- that $\{Q\} \quad S \quad \{P\}$ is a partially correct proof outline.

By the way, the loop's precondition, P, is called an **invariant** of the loop. Loop invariants are crucial in designing and documenting loops. Note that, provided it is true when the while command starts, the invariant will be true at the start of each iteration and when the loop terminates. Loop invariants allow us to analyze the effect of a loop by considering only the effect of a single iteration.[8]

Parentheses Rule: $\{P\}\ (S)\ \{Q\}$ is a partially correct proof outline, provided $\{P\}\ S\ \{Q\}$ is a partially correct proof outline.[9]

From here on we will write "**correct**" in place of "partially correct", as we won't be concerned with any other sort of correctness.

In a proof outline, all commands will be preceded by an assertion. This is its **precondition**. In the example, the precondition of $y := y/2;$ is

$$\text{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{y/2}$$

and the precondition of $x := x \times x;$ is

$$X^Y = z \times x^y \wedge y > 0 \wedge \text{even}(y) \qquad .$$

Suppose $\{P\}\ S\ \{Q\}$ is a correct proof outline. Let $\widehat{S}$ be formed by deleting all assertions from S or by treating them as comments. Now $\{P\}\ \widehat{S}\ \{Q\}$ is a correct Hoare triple.

[8] Loop invariants are closely related to global invariants, class or module invariants, and monitor invariants that we will see later in this course. All can be considered a kind of loop invariant.

[9] Remember that in larger examples, we use braces instead of parentheses.

In Figure 1 I was careful to place the assertions before the braces at the start of the loop body and at the start of each tail of the if command. Throughout the course, I'll place such assertions after the brace at times, just to save some vertical space. I.e., I'll write

```
{ ## P
  S }
## Q
```

rather than

```
## P
{ S }
## Q
```

With the former notation, the "{ ## P" can often be conveniently placed on the same line as an "**if**(E)" or a "**while**(E)"

3.7 Correctness of the example

There is a little, but not much, work left to show that the example proof outline in Figure 1 is correct. First a recall that a boolean formula is said to be universally true if it evaluates to true regardless of the values chosen for its free variables (including our so-called constants).

Let's call the loop invariant I.

$$I \triangleq y \geq 0 \wedge X^Y = z \times x^y$$

- Because of the first assignment, we must show

  $$y \geq 0 \wedge x = X \wedge y = Y \Rightarrow I_{z \leftarrow 1}$$

  is universally true. After substitution we have

  $$y \geq 0 \wedge x = X \wedge y = Y \Rightarrow y \geq 0 \wedge X^Y = 1 \times x^y$$

  which (using a one-point law) we can easily see is universally true.

- From the rule for while-loops, we must show

  $$\begin{aligned} &I \wedge y > 0 \Rightarrow X^Y = z \times x^y \wedge y > 0 \\ &I \wedge \neg\,(y > 0) \Rightarrow z = X^Y \end{aligned}$$

  are each universally true. Both are fairly straight-forward

- From the rule for 2-tailed if commands, we must show

  $$\begin{aligned} &X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{odd}(y) \Rightarrow X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{odd}(y) \text{ and} \\ &X^Y = z \times x^y \wedge y > 0 \wedge \neg\mathrm{odd}(y) \Rightarrow X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{even}(y) \end{aligned}$$

  are each universally true. Both are trivial.

- The four assignments in the loop body give rise to four expressions that should be shown to be universally true:

  $$\begin{aligned} &X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{odd}(y) \Rightarrow \left(y - 1 \geq 0 \wedge X^Y = z \times x^{y-1}\right)_{z \leftarrow z \times y} \\ &y - 1 \geq 0 \wedge X^Y = z \times x^{y-1} \Rightarrow I_{y \leftarrow y-1} \\ &X^Y = z \times x^y \wedge y > 0 \wedge \mathrm{even}(y) \Rightarrow \left(\mathrm{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{y/2}\right)_{x \leftarrow x \times x} \\ &\mathrm{even}(y) \wedge y/2 \geq 0 \wedge X^Y = z \times x^{y/2} \Rightarrow I_{y \leftarrow y/2} \end{aligned}$$

4 Concurrent programming

4.1 Interference

It was recognized early on that the style of reasoning shown above can be invalid in the presence of concurrent programs sharing the same variables. For example, if we have the program[10]

## *true*
$\langle x := 1; \rangle$
$x = 1$
$\langle y := x; \rangle$
$y > 0$

and we run it concurrently with the program $\langle x := x + 1; \rangle$, then the proof outline above is problematic. Consider what happens if the assignment $x := x + 1;$ is scheduled after the assignment $x := 1;$ and before the assignment $y := x;$. Then the precondition of $y := x;$, i.e. $x = 1$ is not true after the increment happens. In a sense the program $x := x+1;$ has "interfered with" the state of the other program. Worse still, the assignment has interfered with the proof of the other program, so we can no longer trust our reasoning!

Let's consider the concurrent program above again. Our intuition is that the concurrent assignment $x := x + 1;$ shouldn't invalidate the conclusion that in the end $y > 0$. One way to validate this intuition is to consider all possible interleavings[11] of the assignments — in this example there are only three possible interleavings, so this is not much work — however, we will find that, in general, it is impractical to consider all possible interleavings and to use sequential reasoning.

So what can we do? If we use a different proof of the sequential program, namely

## *true*
$\langle x := 1; \rangle$

[10] The angle brackets indicate 'atomic actions', that is commands that are executed without interruption. We will formalize this notion later.

[11] An interleaving of two sequences of actions is a sequence that consists of all the actions of the two sequences in the same relative order. For example if we have two sequences $[a_0, a_1]$ and $[b_0, b_1]$, the possible interleavings are

$$[a_0, a_1, b_0, b_1]\,,\ [a_0, b_0, a_1, b_1]\,,\ [a_0, b_0, b_1, a_1]\,,$$
$$[b_0, a_0, a_1, b_1]\,,\ [b_0, a_0, b_1, a_1]\,, \text{ and } [b_0, b_1, a_0, a_1]\,.$$

$\#\# \; x > 0$
$\langle y := x; \rangle$
$\#\# \; y > 0$

then the assignment $\langle x := x + 1; \rangle$ can not change the condition $x > 0$ from true to false. Thus no interleaving of the two programs will invalidate the proof above. Since we don't know when the assignment $x := x + 1$ will happen, we should check that the assignment will not interfere with any of the assertions in the longer program. That is, it will not cause them, once made true by the first program, to become false. Formally we should check that

$$\begin{array}{llll} \vdash \{true\} & \langle x := x + 1; \rangle & \{true\} & , \\ \vdash \{x > 0\} & \langle x := x + 1; \rangle & \{x > 0\} & \text{, and} \\ \vdash \{y > 0\} & \langle x := x + 1; \rangle & \{y > 0\} & . \end{array}$$

The insight that Owicki and Gries provided is that, when the proof of a program is not interfered with by another program, it doesn't matter if the state is interfered with. This allows us to still use Hoare logic and proof outlines, provided we are careful, even when dealing with concurrent programs — and this is very important because, while we can often trust ourselves to reason intuitively rather than formally about sequential programs, the same can not be said for concurrent programs. Proof outlines provide a useful record of all the assertions that must not be interfered with by actions of concurrently running commands.

4.2 Rule for concurrent execution

We can extend the logic of partial correctness to include a command for concurrent execution, by which we mean an arbitrary interleaving of the atomic actions of two processes.[12] We'll use the notation

$$\mathbf{co}\; S \;//\; T \;\mathbf{oc}$$

for the concurrent composition of two commands S and T.

Suppose $\{P_S\}S\{Q_S\}$ and $\{P_T\}T\{Q_T\}$ are two proof outlines. Suppose that a is an atomic action from $\{P_S\}S\{Q_S\}$ with a precondition of R, and P is an assertion from $\{P_T\}T\{Q_T\}$ (possibly P_T or Q_T, but also possibly an

[12] This may seem an odd definiton of concurrent, since it doesn't suggest that two actions might happen at the same time. However, if action a_i has started already then action b_j can be started if it is independant of a_i.

assertion embedded within command T), then a **does not interfere with** P if

$$\vdash \{P \wedge R\} \quad a \quad \{P\}$$

Furthermore the two proof outlines **do not interfere with each other** if no action of one interferes with any assertion in the other.[13]

Concurrent execution Rule:

$$\{P\} \ \textbf{co} \ \{P_S\}S\{Q_S\} \ // \ \{P_T\}T\{Q_T\} \ \textbf{oc} \ \{Q\}$$

is a (partially) correct proof outline iff

- the precondition of the co command implies the precondition of each component, i.e.,

  $$P \Rightarrow P_S \text{ is universally true and}$$
  $$P \Rightarrow P_T \text{ is universally true,}$$

- the conjunction of the postconditions of its components implies the postcondition of the co command, i.e.,

  $$Q_S \wedge Q_T \Rightarrow Q \text{ is universally true,}$$

- $\{P_S\}S\{Q_S\}$ and $\{P_T\}T\{Q_T\}$ are both correct proof outlines, and

- $\{P_S\}S\{Q_S\}$ and $\{P_T\}T\{Q_T\}$ do not interfere with each other (freedom from interference).

4.3 Awaiting

Processes can synchronize by means of an await command

$$\langle \textbf{await}(\ E\ )\ S \rangle$$

where E is a boolean expression and S is any command[14]. The idea is that the process delays until E becomes true and then the command S is executed atomically with the evaluation of E.

Await rule: A proof outline

$$\{P\} \ \langle \textbf{await}(\ E\ )\ S \rangle \ \{R\}$$

[13] Later, when we get to await commands, we'll have to modify this definition a little.

[14] Except that await commands are not allowed inside of await commands, nor are co commands.

is correct iff $\{P \wedge E\}\ S\ \{R\}$ is a correct proof outline.

Thus, in a sense, the await command causes E to become true, almost as if by magic. For example the following correct proof outline appears to set x to 99 without doing any real work

```
## true
co
    ## true
    ⟨await( $x = 99$ ) skip; ⟩
    ## $x = 99$
//
    ## true
    skip
    ## true
oc
## $x = 99$
```

Of course there is no magic about it; if E is not true when the await command starts to delay, then it is up to the other process to eventually make E true; if that doesn't happen, the await will delay the process forever. Remember that we are dealing only with partial correctness here, so a correct proof outline may still embody a program that will never terminate.

As abbreviations we have:

- $\langle S \rangle$ abbreviates $\langle$**await**(*true*) $S\rangle$, and
- $\langle$**await**(E)$\rangle$ abbreviates $\langle$**await**(E) **skip**$\rangle$

The derived rules are that $\{P\}\ \langle S \rangle\ \{R\}$ is a correct proof outline if $\{P\}\ S\ \{R\}$ is a correct proof outline, and that $\{P\}\ \langle$**await**(E)$\rangle\ \{R\}$ is a correct proof outline if

$$P \wedge E \Rightarrow R \text{ is universally true.}$$

To account for the atomicity of the await commands, we modify the definition of 'proof outlines do not interfere with each other' to exclude assertions that are contained within await commands. I.e. two proof outlines **do not interfere with each other** exactly if no action of one interferes with any assertion in the other not contained in an await command.

5 A formalization of proof-outline logic

The next two subsections are optional reading. I wrote them mainly to clarify my own understanding. I include them as they may also help clarify yours. If you skip them, proceed to the important caveat in subsection 5.3.

5.1 Syntax

Let's take V to be a set of variables, T to be set of types, E to be a set of expressions, and P to be a set of conditions. The syntax for assignments A, commands S, blocks B and proof outlines O is given by:

A	$\rightarrow$	$V := E;$	Assignment
A	$\rightarrow$	V, A, E	Multiple assignment
S	$\rightarrow$	$A;$	Assignment command
S	$\rightarrow$	$\textbf{skip}$	Skip command
S	$\rightarrow$	(B)	Block command
S	$\rightarrow$	$\textbf{if}(\ E\ )\ \{P\}\ S\ \textbf{else}\ \{P\}\ S$	2-tailed if command
S	$\rightarrow$	$\textbf{if}(\ E\ )\ \{P\}\ S$	1-tailed if command
S	$\rightarrow$	$\textbf{while}(\ E\ )\ \{P\}\ S$	While command
S	$\rightarrow$	$\textbf{co}\ \{P\}\ B\ \{P\}\ //\ \{P\}\ B\ \{P\}\ \textbf{oc}$	Concurrent command
S	$\rightarrow$	$\langle \textbf{await}(\ E\ )\ B \rangle$	Await command
B	$\rightarrow$	$T\ V\ ;\ B$	Variable declaration
B	$\rightarrow$	$S\ \{P\}\ B$	Sequential composition
B	$\rightarrow$	S	Simple block
O	$\rightarrow$	$\{P\}\ B\ \{P\}$	Proof outline

There are a number of restrictions not indicated by the syntax

- In an assignment command, the type of the i^{th} variable must match the type of the i^{th} expression, for each i.
- The expressions in await, if, and while commands must be boolean.
- The scope of a variable is the block that follows its declaration.
- Await commands may not occur within await commands, directly or indirectly.
- The choice between 1- and 2-tailed if commands leads to a syntactic ambiguity. As we read from left to right, each 'else' is considered

attached to the nearest as-yet-umatched ‘if’ to its left to which it could be attached. For example

$$\textbf{if}(E_0)\ \{P_0\}\ \textbf{if}(E_1)\ \{P_1\}\ S_0\ \textbf{else}\ \{P_2\}\ S_1$$

is treated as a 2-tailed if command within a 1-tailed if command.

A few other comments are in order.

- I only include the binary case for the concurrent command. The generalization to more processes is straight-forward.
- I omit the abbreviations for await commands. These are $\langle B\rangle$ abbreviates $\langle$**await**(*true*) $B\rangle$ and $\langle$**await**(P)$\rangle$ abbreviates $\langle$**await**(P) **skip**$\rangle$.
- I use round parentheses to turn blocks into commands rather than the curly braces used by Andrews and in my examples. This is so that the curly braces can be saved to delimit assertions. See next point.
- In this section I use $\{P\}$ to indicate an assertion, rather than the ##P notation used in other sections and in Andrew’s text.
- In practice a fragment “$\{P\}$ (” (or “ $\begin{array}{c}\#\#P\\ \{\end{array}$ ”) may be written as “($\{P\}$” (or “{ ##P”) . In Andrews’s notation, the latter often formats a bit better. In dealing with the theory, the former is easier to cope with.
- I haven’t gone in to detail about the set of expressions. I will assume though that, for each expression E, there is a condition df$[E]$ that expresses the weakest condition for E to be well-defined, i.e. not to crash. For example if dividing by 0 is considered to be a cause for crashing then df$[x/y]$ would be $y \neq 0$. If a is an array of size n then df$[a[i]]$ would be $0 \leq i < n$.

5.2 Semantics

5.2.1 Underlying logic

We’ll assume there is an underlying logic with judgements of the form $\vdash P$ where P is a predicate logic formula. So, for example,

$$\vdash x + y = y + x$$

means that “$x + y = y + x$” is a theorem in the underlying logic, i.e. that it is universally true and can be proved so in the underlying logic.

5.2.2 Rules

We put a turnstile ($\vdash$) in front of a proof outline to indicate that it is (partially) correct.

The following inference rules allow us to conclude that certain proof outlines are correct. Each inference rule

$$\frac{\begin{array}{c} A_0 \\ A_1 \\ \ldots \\ A_n \end{array}}{C}$$

is interpreted as follows: If the list of antecedents A on the top of the line is true, then the consequent C, below the line is true.[15]

$$\frac{\begin{array}{c} \vdash P \Rightarrow \mathrm{df}[e_0] \wedge \mathrm{df}[e_1] \wedge \cdots \wedge \mathrm{df}[e_{n-1}] \\ \vdash P \Rightarrow Q_{v_0,v_1,...,v_{n-1} \leftarrow e_0,e_1,...,e_{n-1}} \end{array}}{\vdash \{P\}\ v_0, v_1, ..., v_{n-1} := e_0, e_1, ..., e_{n-1};\ \{Q\}}\text{(Assign)}$$

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \mathbf{skip}\ \{Q\}}\text{(Skip)} \qquad \frac{\begin{array}{c} \vdash P \Rightarrow \mathrm{df}[E] \\ \vdash \{P \wedge E\}\ B\ \{Q\} \end{array}}{\vdash \{P\}\ \langle \mathbf{await}(\ E\ )\ B \rangle\ \{Q\}}\text{(Await)}$$

$$\frac{\begin{array}{l} \vdash \{Q_0\}\ S_0\ \{R\} \\ \vdash \{Q_1\}\ S_1\ \{R\} \\ \vdash P \Rightarrow \mathrm{df}[E] \\ \vdash P \wedge E \Rightarrow Q_0 \\ \vdash P \wedge \neg E \Rightarrow Q_1 \end{array}}{\vdash \{P\}\ \mathbf{if}(\ E\ )\ \{Q_0\}\ S_0\ \mathbf{else}\ \{Q_0\}\ S_1\ \{R\}}\text{(If 2)} \qquad \frac{\begin{array}{l} \vdash \{Q\}\ S\ \{R\} \\ \vdash P \Rightarrow \mathrm{df}[E] \\ \vdash P \wedge E \Rightarrow Q \\ \vdash P \wedge \neg E \Rightarrow R \end{array}}{\vdash \{P\}\ \mathbf{if}(\ E\ )\ \{Q\}\ S\ \{R\}}\text{(If 1)}$$

$$\frac{\begin{array}{l} \vdash \{Q\}\ S\ \{P\} \\ \vdash P \Rightarrow \mathrm{df}[E] \\ \vdash P \wedge E \Rightarrow Q \\ \vdash P \wedge \neg E \Rightarrow R \end{array}}{\vdash \{P\}\ \mathbf{while}(\ E\ )\ \{Q\}\ S\ \{R\}}\text{(While)} \qquad \frac{\begin{array}{l} \vdash \{P\}\ S\ \{Q\} \\ \vdash \{Q\}\ B\ \{R\} \end{array}}{\vdash \{P\}\ S\ \{Q\}\ B\ \{R\}}\text{(Seq)}$$

[15] Those who have studied Hoare's logic will note the absence of the rule(s) of consequence. Consequence is incorporated as needed in the various rules.

$$\frac{\begin{array}{l} \vdash \{P_0\}\ B_0\ \{Q_0\} \\ \vdash \{P_1\}\ B_1\ \{Q_1\} \\ \vdash P \Rightarrow P_0 \\ \vdash P \Rightarrow P_1 \\ \vdash Q_0 \wedge Q_1 \Rightarrow Q \\ \{P_0\}\ B_0\ \{Q_0\} \text{ does not interfere with } \{P_1\}\ B_1\ \{Q_1\} \end{array}}{\vdash \{P\}\ \mathbf{co}\ \{P_0\}\ B_0\ \{Q_0\}\ //\ \{P_1\}\ B_1\ \{Q_1\}\ \mathbf{oc}\ \{Q\}}\text{(Co)}$$

$$\frac{\begin{array}{l} \vdash \{P\}\ B\ \{Q\} \\ v \text{ is not free in } P \\ v \text{ is not free in } Q \end{array}}{\vdash \{P\}\ T\ v\ ;\ B\ \{Q\}}\text{(Decl)} \qquad \frac{\vdash \{P\}\ B\ \{Q\}}{\vdash \{P\}\ (B)\ \{Q\}}\text{(Paren)}$$

5.3 A caveat

These rules assume a certain granularity of execution that may not be realistic. You can see that $x := E;$ and $\langle x := E; \rangle$ mean exactly the same thing, so we are assuming that assignment commands are executed atomically. Similarly the guard expressions in if and while commands are assumed to be evaluated atomically. These assumptions are not actually realistic. Languages such as C, C++, and Java have carefully defined rules defining the meaning of reads from and writes to shared variables; these are the so-called 'memory models' for the languages.

6 Simple examples

6.1 A note on showing triples involving assignments

To show noninterference, one has to show that atomic actions do not interfere with assertions. Typically the atomic actions are assignments, so let's look a bit at showing Hoare triples that involve assignments to be correct.

Generally we need to show

$$\{P\}\ \langle x := E; \rangle\ \{Q\}$$

is correct. Some observations:

- The Hoare-triple is correct if and only if

  $$P \Rightarrow Q_{x \leftarrow E}$$

  is universally true. We call $Q_{x \leftarrow E}$ **the substituted postcondition**. So we must show that the precondition implies the substituted precondition.

- If P is universally false,[16] then the Hoare triple is correct. This follows immediately from the previous point. Usually when a precondition is universally false, it is because you have come across some form of **mutual exclusion**.

- If the precondition is a conjunction, for example P is $(P0 \wedge P1 \wedge P2)$, it is safe to use only some of the conjuncts of a precondition. For example, it would be sufficient to show

  $$\vdash P0 \Rightarrow Q_{x \leftarrow E} \qquad .$$

  We call this **subsetting the precondition**.

- Extending this a bit further, it is safe to replace the precondition P by any condition R that is implied by the P. At the extreme (taking R as **true**), it suffices to ignore the precondition altogether and simply show $Q_{x \leftarrow E}$ is universally true.

- We can show the postcondition in parts. For example if Q is $Q0 \wedge Q1$ then we can separately show

  $$\begin{array}{ll} \vdash P \Rightarrow Q0_{x \leftarrow E} & \text{, and} \\ \vdash P \Rightarrow Q1_{x \leftarrow E} & \end{array}$$

  are correct. We call this **proof by parts**.

When showing non-interference, the postcondition is also part of the precondition. We need to show

$$\vdash \{Q \wedge P\} \ \langle x := E; \rangle \ \{Q\}$$

where Q is some assertion made in one component, $\langle x := E; \rangle$ is an atomic action from another component, and P is the precondition of $\langle x := E; \rangle$.

- If x does not occur in Q, then the Hoare triple is correct. This is because $Q_{x \leftarrow E}$ is then simply Q and we have to show

  $$Q \wedge P \Rightarrow Q$$

  universally true, which it trivially is. We call this situation **disjoint variables**.

[16] I.e. $\neg P$ is universally true.

The idea of **proof by parts** applies to outlines in general, not just to assignment commands. If we want to show

$$\{P\}\ S\ \{Q0 \wedge Q1\}$$

to be correct, it suffices to show that both

$$\{P\}\ S\ \{Q0\} \text{ and}$$
$$\{P\}\ S\ \{Q1\}$$

are correct.

6.2 An example with no interference

Consider the following program that increments x and y in parallel

```
$\#\#\ x = X \wedge y = Y$
**co**
    $\#\#\ x = X$
    $\langle x := x + 1; \rangle$
    $\#\#\ x = X + 1$
//
    $\#\#\ y = Y$
    $\langle y := y + 1; \rangle$
    $\#\#\ y = Y + 1$
**oc**
$\#\#\ x = X + 1 \wedge y = Y + 1$
```

To show this proof outline correct we must check the following

- *The precondition of the co command implies the preconditions of each component of the co command.*
  - $x = X \wedge y = Y \Rightarrow x = X$
    This is universally true from propositional calculus.
  - $x = X \wedge y = Y \Rightarrow y = Y$
    Similarly.
- *The conjunction of the postconditions of its components implies the postcondition of the co command.*

The postconditions of the components are respectively $x = X + 1$ and $y = Y + 1$. The conjunction of them is the postcondition of the co command itself.

- *Local correctness.* We need to show that each component is itself a correct proof outline. By application of the assignment rule. We have that

  $$\{x = X\}\ x := x + 1;\ \{x = X + 1\}$$

  is correct. Similarly for

  $$\{y = Y\}\ y := y + 1;\ \{y = Y + 1\}$$

- *Freedom from interference.* We can consider each pair consisting of an assertion in one component and an atomic action in the other. There are four such pairs

$\{x = X\}$	$\langle y := y + 1; \rangle$
$\{x = X + 1\}$	$\langle y := y + 1; \rangle$
$\{y = Y\}$	$\langle x := x + 1; \rangle$
$\{y = Y + 1\}$	$\langle x := x + 1; \rangle$

  To check the first we must check that

  $$\{x = X \wedge y = Y\}\ \langle y := y + 1; \rangle\ \{x = X\}$$

  is correct. From the assignment rule, we must show

  $$x = X \wedge y = Y \Rightarrow (x = X)_{y \leftarrow y+1}$$

  to be universally true; it simplifies to

  $$x = X \wedge y = Y \Rightarrow x = X$$

  which is universally true by propositional calculus. In the terminology above, we have "disjoint variables".

  The other three action/assertion pairs are each free of interference by reason of disjoint variables.

6.3 An example with interference

Consider the following proof outline

```
## $x = X$
**co**
    ## $x = X$
    $\langle x := x + 2; \rangle$
    ## $x = x + 2$
//
    ## $x = X$
    $\langle x := x + 3; \rangle$
    ## $x = X + 3$
**oc**
## $x = X + 2 \wedge x = X + 3$
```

This has the rather startling postcondition that x is both $X+2$ and that it is $X + 3$!

What is wrong? Let's check everything

- *The precondition of the co command implies the precondition of each component.* As they are the same, the implication is trivial.
- *The conjunction of the postconditions of its components implies the postcondition of the co command.* As they are the same, this implication is also trivial.
- *Local correctness.* Each component is a correct proof outline.
- *Freedom from interference.* There are four assertion/action pairs to check

$\{x = X\}$	$\langle x := x + 3; \rangle$
$\{x = x + 2\}$	$\langle x := x + 3; \rangle$
$\{x = X\}$	$\langle x := x + 2; \rangle$
$\{x = X + 3\}$	$\langle x := x + 2; \rangle$

For the first we must show that

$$\{x = X\}\ \langle x := x + 3; \rangle\ \{x = X\}$$

is correct. But it is not. Thus there is interference. In fact every pair exhibits interference.

6.4 Fixing the last example.

Looking at the code for the last example, we would expect the postcondition to be $x = X+5$. Let's see how we can prove that. We'll reason operationally: Initially $x = X$ is true, but while the first thread is waiting to execute its command, the command from the second thread may execute, changing x to $X + 3$. Thus either $x = X$ or $x = X + 3$ could be true. Similarly, before the command in the second component is waiting to execute, the state could be either $x = X$ or $x = X + 2$ (at least). Let's try these as preconditions; we get

```
## $x = X$
co
    ## $x = X \vee x = X + 3$
    $\langle x := x + 2; \rangle$
    ## ?
//
    ## $x = X \vee x = X + 2$
    $\langle x := x + 3; \rangle$
    ## ?
oc
## $x = X + 5$
```

From the preconditions and the assignments, we can see that after the first command, the state could be either $x = X + 2$ or $x = X + 5$ and after the second the state could be either $x = X + 3$ or $x = X + 5$. So we can complete the proof outline:

```
## $x = X$
co
    ## $x = X \vee x = X + 3$
    $\langle x := x + 2; \rangle$
    ## $x = X + 2 \vee x = X + 5$
//
    ## $x = X \vee x = X + 2$
    $\langle x := x + 3; \rangle$
    ## $x = X + 3 \vee x = X + 5$
oc
## $x = X + 5$
```

Now is this proof outline correct? We must check:

- *The precondition of the co command implies the precondition of each component.* These are true by propositional reasoning as

$$P \Rightarrow P \vee Q$$

- *The conjunction of the postconditions of the components implies the postcondition of the co command.* We must check whether

$$(x = X + 2 \vee x = X + 5) \wedge (x = X + 3 \vee x = X + 5) \Rightarrow x = X + 5$$

  is universally true. We have

$$\begin{array}{l} \quad (x = X + 2 \vee x = X + 5) \wedge (x = X + 3 \vee x = X + 5) \\ =\text{Distributivity} \\ \quad (x = X + 2 \wedge x = X + 3) \vee (x = X + 2 \wedge x = X + 5) \\ \quad \vee (x = X + 5 \wedge x = X + 3) \vee (x = X + 5 \wedge x = X + 5) \\ =\text{Simplification} \\ \quad \text{false} \vee \text{false} \vee \text{false} \vee\ x = X + 5 \\ =\text{Identity} \\ \quad x = X + 5 \end{array}$$

- *Local correctness.* We need to check the correctness of each component. I.e. the correctness of

$$\begin{array}{ll} \{x = X \vee x = X + 3\}\ \langle x := x + 2; \rangle\ \{x = X + 2 \vee x = X + 5\} & \text{, and} \\ \{x = X \vee x = X + 2\}\ \langle x := x + 3; \rangle\ \{x = X + 3 \vee x = X + 5\} & . \end{array}$$

  For the first we substitute in the postcondition to get

$$x + 2 = X + 2 \vee x + 2 = X + 5$$

  which after a bit of algebraic simplification, is equivalent to the precondition $x = X \vee x = X + 3$. Similarly for the second after substitution we get

$$x + 3 = X + 3 \vee x + 3 = X + 5 \qquad ,$$

  which simplifies to the precondition.

- *Freedom from interference.* There are four assertion/action pairs to check

$\{x = X \lor x = X + 3\}$	$\langle x := x + 3; \rangle$
$\{x = X + 2 \lor x = X + 5\}$	$\langle x := x + 3; \rangle$
$\{x = X \lor x = X + 2\}$	$\langle x := x + 2; \rangle$
$\{x = X + 3 \lor x = X + 5\}$	$\langle x := x + 2; \rangle$

  - For the first we need to check the correctness of

$$\begin{array}{l} \{(x = X \lor x = X + 3) \land (x = X \lor x = X + 2)\} \\ \langle x := x + 3; \rangle \\ \{x = X \lor x = X + 3\} \end{array}$$

    The precondition simplifies to $x = X$. The substituted postcondition is $x + 3 = X \lor x + 3 = X + 3$. So we need to check that

$$x = X \Rightarrow x + 3 = X \lor x + 3 = X + 3$$

    is universally true, which, by a one-point law, it is.

  - For the second, we need to check

$$\begin{array}{l} \{(x = X + 2 \lor x = X + 5) \land (x = X \lor x = X + 2)\} \\ \langle x := x + 3; \rangle \\ \{x = X + 2 \lor x = X + 5\} \end{array}$$

    The precondition simplifies to $x = X + 2$. The substituted postcondition is $x + 3 = X + 2 \lor x + 3 = X + 5$ which simplifies to $x = X - 1 \lor x = X + 2$. So we need to check

$$x = X + 2 \Rightarrow x = X - 1 \lor x = X + 2$$

  - The last two are essentially the same as the first two.

7 Global invariants

7.1 Global invariants

The purpose of the next example is to illustrate the important technique of using “global invariants”. A condition G is a **global invariant** with respect to a co command if

- it is implied by the precondition of the co command,
- each atomic action in the co command preserves the invariant in the sense that
$$\{G \wedge P_a\}\ a\ \{G\}$$
is correct for each atomic action a in the co command, where P_a is the precondition of a.

Once we've shown a condition to be a global invariant, we can assume it as a precondition in showing the local correctness of any assertion and when showing interference freedom. Each assertion P can be split into the global part and a local part:
$$P = G \wedge P_L \qquad ,$$
where G is the conjunction of all global invariants. For local correctness we need to show outlines of the form
$$\{G \wedge P_L\}\ a\ \{G \wedge Q_L\}$$
correct, where P_L is the local part of the precondition and Q_L is the local part of the postcondition. Since we've already shown G is a global invariant, all that remains is to show (using "proof by parts")
$$\{G \wedge P_L\}\ a\ \{Q_L\}$$
correct. For freedom from interference, we need to show outlines of the form
$$\{Q_L \wedge G \wedge P_L\}\ a\ \{G \wedge Q_L\}$$
to be correct, where P_L is the local part of the precondition of an action a and Q_L is the local part of some assertion from another component. If we've already shown that G is a global invariant, it only remains to show (again, using "proof by parts") that
$$\{Q_L \wedge G \wedge P_L\}\ a\ \{Q_L\}$$
is correct.

We can save a lot of writing by noting the relevant global invariants between the co command and its precondition. For example,

```
## $P$
## Global Inv: $G$
co
```

```
    ## $P_0$
    $\langle S \rangle$
    ## $Q_0$
//
    ## $P_1$
    $\langle T \rangle$
    ## $Q_1$
oc
## $Q$
```

abbreviates

```
## $P$
co
    ## $G \wedge P_0$
    $\langle S \rangle$
    ## $G \wedge Q_0$
//
    ## $G \wedge P_1$
    $\langle T \rangle$
    ## $G \wedge Q_1$
oc
## $Q$
```

The next section illustrates the use of global invariants, of await commands for coordinating cooperating processes, and of mutual exclusion.

7.2 A client and a server

In this example, we analyze a shared-variable client-server system. The client sends a message in shared variable x. The server computes a function, f, of x and returns it via the same variable. The server looks like this

```
while( $q \neq 2$ ) {
    $\langle$await $(q = 1)\rangle$
    $\langle x, q := f(x), 0; \rangle$ }
```

The client computes a function $g(x)$ and sends the result to the server, getting back the result $f(g(x))$. The client ends after N iterations. The client looks like this

```
## $x = X$
int $q := 0$ , $i := 0$
## $x = X \wedge q = 0 \wedge i = 0$
## Global Inv: $0 \leq q \leq 2$
## Global Inv: $i \geq 0$
## Global Inv: $q \neq 1 \Rightarrow x = h^i(X)$
## Global Inv: $q = 1 \Rightarrow x = g(h^{i-1}(X)) \wedge i > 0$
co
    while( $q \neq 2$ ) {
        $\langle$await $(q = 1)\rangle$
        ## $q = 1$
        $\langle x, q := f(x), 0; \rangle$ }
//
    ## $i \leq N \wedge q = 0$
    while( $i < N$ ) {
        ## $i < N \wedge q = 0$
        $\langle x, q, i := g(x), 1, i + 1; \rangle$
        ## $i \leq N$
        $\langle$await $(q = 0)\rangle$
    ## $q = 0 \wedge i = N$
    $\langle q := 2; \rangle$
    ## $q = 2 \wedge i = N$
oc
## $x = h^N(X)$
```

Figure 2: A client and server system.

```
int $i := 0$ ;
while( $i < N$ ) {
    $\langle x, q, i := g(x), 1, i + 1; \rangle$
    $\langle$await $(q = 0)\rangle$ }
$\langle q := 2; \rangle$
```

Initially we will have $x = X$ and $q = 0$. Let $h = (f \circ g)$ be the composition of g and f. Then the parallel composition of the client and server should compute $x = h^N(X)$.

We need to come up with a proof outline for the composition. *It is a* *good idea to keep the annotation as minimal as possible, that is to resist the temptation to write down everything we might be able to prove about the state at each point. Rather we only make note of those facts required to achieve our goal: namely, to have a correct proof outline with the stated pre- and postconditions..*

Figure 2 on page 33 shows the composition.

In the proof outline in Figure 2, I've used the abbreviation mentioned at the end of the previous section. Rather than repeating the global invariants in nine different places, I've listed them just once at the start of the co command. These invariants are to be thought of as being conjoined with each assertion within the co command. Thus the full precondition of the command $\langle x, q := f(x), 0; \rangle$ is

$$q = 1 \wedge 0 \leq q \leq 2 \wedge i \geq 0 \wedge \big(q \neq 1 \Rightarrow x = h^i(X)\big) \wedge \big(q = 1 \Rightarrow x = g(h^{i-1}(X))\big)$$

which simplifies to

$$q = 1 \wedge i \geq 0 \wedge x = g(h^{i-1}(X))$$

Furthermore, I haven't written assertions that are simply *true.* So the precondition of $\langle \textbf{await}\ (q = 1)\rangle$ is simply the conjunction of the four global invariants.

I made a slight change to the client algorithm in that I expanded the scope of the variable i so that it can appear in the assertions both sides of the //.

To show that the proof outline is correct, we need to show that each assertion is justified.

- $x = X \wedge q = 0 \wedge i = 0$. This follows from the initializations.

- *The preconditions of the components.* We need to check that each global invariant follows from

  $$x = X \wedge q = 0 \wedge i = 0$$

  and also that $i \leq N \wedge q = 0$ does. These five implications are all trivially universally true.

- *The postconditions of the components imply the overall postcondition.* Combining the postcondition $q = 2 \wedge i = N$ with the global invariant $q \neq 1 \Rightarrow x = h^i(X)$, we get $x = h^N(X)$.

- *Global invariants.* We show that the claimed global invariants are preserved by each atomic action.
  - $0 \le q \le 2$. The assignment actions we have to worry about are

    $$\begin{array}{ll}\langle x, q := f(x), 0; \rangle & , \\ \langle x, q, i := g(x), 1, i+1; \rangle & \text{, and} \\ \langle q := 2; \rangle & \end{array}$$

    The substituted postconditions are $0 \le 0 \le 2$, $0 \le 1 \le 2$, and $0 \le 2 \le 2$ which are all trivially true.
  - $i \ge 0$. By disjoint variables we only need to show

    $$\{i \ge 0\}\ \langle x, q, i := g(x), 1, i+1; \rangle\ \{i \ge 0\}$$

    which is implied by

    $$i \ge 0 \Rightarrow i + 1 \ge 0$$

    which is universally true.[17]
  - $q \ne 1 \Rightarrow x = h^i(X)$.
    - The assignment

      $$\langle x, q, i := g(x), 1, i+1; \rangle$$

      sets q to 1. Doing the substitution we get

      $$1 \ne 1 \Rightarrow ...$$

      which we can see is true without considering what is in the ... , nor considering the preconditions.
    - For the assignment

      $$\langle x, q := f(x), 0; \rangle$$

      the substituted postcondition simplifies to $f(x) = h^i(X)$. The precondition includes the conjuncts

      $$q = 1 \wedge \big(q = 1 \Rightarrow x = g(h^{i-1}(X)) \wedge i > 0\big) \qquad ,$$

[17] Well maybe and maybe not. If our ints are true integers, there is no issue. If ints are bounded in range, it's not universally true. We could use a larger subset of the precondition and show

$$i \ge 0 \wedge i < N \Rightarrow i + 1 \ge 0$$

Given that N is an int, this will be universally trie for bounded ints..

which imply $x = g(h^{i-1}(X)) \wedge i > 0$. Now apply f to both sides and we get $f(x) = f(g(h^{i-1}(x))$, which is the same as $f(x) = h^i(X)$.

- $*$ For the assignment $\langle q := 2; \rangle$, the substituted postcondition simplifies to $x = h^i(X)$. The preconditions include

$$q = 0 \wedge \big(q \neq 1 \Rightarrow x = h^i(X)\big) \qquad ,$$

which clearly imply $x = h^i(x)$.

- $-$ $q = 1 \Rightarrow x = g(h^{i-1}(X)) \wedge i > 0$. The assignments

$$\begin{array}{ll} \langle x, q := f(x), 0; \rangle & \text{and} \\ \langle q := 2; \rangle & \end{array}$$

both give a substituted postcondition of the form

$$\text{false} \Rightarrow ...$$

which is trivially true. The interesting action is $\langle x, q, i := g(x), 1, i + 1; \rangle$. The substituted postcondition simplifies to

$$g(x) = g(h^i(X)) \wedge i + 1 > 0 \qquad ,$$

which is implied by the following subset of the precondition:

$$q = 0 \wedge \big(q \neq 1 \Rightarrow x = h^i(X)\big) \wedge i \geq 0 \qquad .$$

- *Local correctness.* Showing the that the global invariants really are invariant is part of local correctness. What remains is to show the local parts of each assertion follow from the preceding atomic actions. This is straight forward in each case. It suffices to show

$$\begin{array}{ll}
\vdash \{\text{true}\}\ \langle \textbf{await}\ (q = 1)\rangle\ \{q = 1\} & , \\
\vdash q = 0 \wedge i = 0 \Rightarrow 0 \leq i \leq N \wedge q = 0 & , \\
\vdash 0 \leq i \leq N \wedge q = 0 \wedge i < N \Rightarrow 0 \leq i < N \wedge q = 0 & , \\
\vdash \{0 \leq i < N\}\ \langle x, q, i := g(x), 1, i + 1; \rangle\ \{0 \leq i \leq N\} & , \\
\vdash \{0 \leq i \leq N\}\ \langle \textbf{await}\ (q = 0)\rangle\ \{0 \leq i \leq N \wedge q = 0\} & , \\
\vdash 0 \leq i \leq N \wedge q = 0 \wedge \neg\,(i < N) \Rightarrow q = 0 \wedge i = N & , \text{and} \\
\vdash \{i = N\}\ \langle q := 2; \rangle\ \{q = 2 \wedge i = N\} &
\end{array}$$

In each case the reasoning is straight-forward.

- *Freedom from interference.* Showing the global invariants really are invariant is part of showing freedom from interference. What remains is to show that the following pairs don't interfere

$q = 1$	$\langle x, q, i := g(x), 1, i + 1; \rangle$
$q = 1$	$\langle q := 2; \rangle$
$q = 0$	$\langle x, q := f(x), 0; \rangle$
$q = 0$	$\langle x, q := f(x), 0; \rangle$

  I've omitted all conjuncts that have to do with i because they are free from interference by reason of "disjoint variables". Now let's deal with the pairs. In each case we actually have mutual exclusion. For example, in the first, the precondition of the action includes the conjunct $q = 0$. Now, subsetting the precondition, it suffices to show

  $$\vdash \{q = 1 \wedge q = 0\} \ \langle x, q, i := g(x), 1, i + 1; \rangle \ \{q = 1\}$$

  We need do no more work than to observe that the precondition is false. This technique works in all four cases.

  Three of the pairs (all but the second) can also be shown in another easy way. Consider the first again. The substituted postcondition is $1 = 1$ which is *true*. We need not even look at the precondition. We have

  $$\vdash \{\text{true}\} \ \langle x, q, i := g(x), 1, i + 1; \rangle \ \{q = 1\}$$

  This is an extreme case of subsetting the precondition.

This concludes the example.

8 Ghost Variables

A technique that is often useful and occasionally indispensable is that of ghost variables. The idea is to introduce extra variables that are useful for creating a correct proof outline, but that are not needed in the actual implementation. These variables are called '**ghost variables.'** (Some writers call them 'auxiliary variables,' 'thought variables,' or 'dummy variables.')

Here is a simple example. Consider the algorithm

```
## $x = 0$
co
    $\langle\ x := x + 1;\ \rangle$
```

```
//
    $\langle\ x := x + 1;\ \rangle$
oc
## $x = 2$
```

How can we complete the outline? If we put in a precondition for each component of $x = 0$, like this

```
## $x = 0$
co
    ## $x = 0$
    $\langle\ x := x + 1;\ \rangle$
//
    ## $x = 0$
    $\langle\ x := x + 1;\ \rangle$
oc
## $x = 2$
```

there is interference. Weakening the preconditions to try to avoid interference, like this

```
## $x = 0$
co
    ## $x = 0 \vee x = 1$
    $\langle\ x := x + 1;\ \rangle$
//
    ## $x = 0 \vee x = 1$
    $\langle\ x := x + 1;\ \rangle$
oc
## $x = 2$
```

doesn't work; there is still interference. *We* know that once the other process has incremented x, it won't do it again, but the definition of interference only considers the actions of the other process, not the sequence or frequency that they might happen in.

We can introduce ghost variables a and b to track the local changes to x; a represents how much the first component has incremented x, while b represents how much the second component has incremented x. Thus $x = a + b$, at all times. To emphasize that a and b are ghost variables, we put their declarations in comments marked by "#".

```
## $x = 0$
# int $a := 0$
# int $b := 0$
## $x = 0 \wedge a = 0 \wedge b = 0$
# Global invariant: $x = a + b$
co
    ## $a = 0$
    $\langle\ x := x + 1;\ a := 1;\rangle$
    ## $a = 1$
//
    ## $b = 0$
    $\langle\ x := x + 1;\ b := 1;\rangle$
    ## $b = 1$
oc
## $x = 2$
```

By disjoint variables, there is no interference. We need only show that each action is correct locally and that each maintains the global invariant. The final assertion that $x = 2$ follows from the global invariant together with the assertions $a = 1$ and $b = 1$.

9 Data refinement

In engineering, we often replace an abstract part of a design by something more concrete that refines it. For example, an early version of the design of a building might specify a column capable of transmitting a given force. Later in the design process, that column might be replaced by a set of smaller columns that together do the same job.

In programming we can replace one set of variables with another set of variables. This process is known as data refinement. Data refinement allows us to go from abstract solutions that are easily seen to be correct, but that may be difficult to implement, to concrete solutions that are easy to implement, but whose correctness may be less obvious. It is a useful technique in concurrent programming and also in sequential programming.

Data refinement is best done in three stages:

Augment First, we introduce the new set of variables, linked to the other variables by an invariant.

Transform Second, we transform the program until (some of) the original variables are no longer needed.

Diminish Finally, we eliminate any variables that are no longer needed, or demote them to the status of ghost variables.

As an example, consider a solution to the mutual exclusion problem. We start with a program that ensures that only one of commands A, B, and C are being executed at any one time. We assume that A, B, and C don't change variables a, b, or c.

```
int $a := 0, b := 0, c := 0$ ;
## $a = 0 \wedge b = 0 \wedge c = 0$
# Global invariant: $0 \le a, b, c \le 1$
# Global invariant: $a + b + c < 2$
co
    while( true ) {
        ⟨await$(a + b + c = 0)$ $a := 1;$⟩
        $A$
        ⟨$a := 0;$⟩ }
//
    while( true ) {
        ⟨await$(a + b + c = 0)$ $b := 1;$⟩
        $B$
        ⟨$b := 0;$⟩ }
//
    while( true ) {
        ⟨await$(a + b + c = 0)$ $c := 1;$⟩
        $C$
        ⟨$c := 0;$⟩ }
oc
```

The problem with this program is that the await commands must read three variables at once, which might be difficult to engineer.

Augmenting: Let's introduce a new variable s, tied to a, b, and c by a global invariant $s = a + b + c$.

```
int $a := 0, b := 0, c := 0$ ;
int $s := 0$ ;
```

```
## $a = 0 \wedge b = 0 \wedge c = 0 \wedge s = 0$
# Global invariant: $0 \leq a, b, c \leq 1$
# Global invariant: $a + b + c < 2$
# Global invariant: $s = a + b + c$
co
    while( true ) {
        $\langle$await$(a + b + c = 0)$ $a := 1; s := 1;\rangle$
        $A$
        $\langle a := 0; s := 0;\rangle$ }
//
    while( true ) {
        $\langle$await$(a + b + c = 0)$ $b := 1; s := 1;\rangle$
        $B$
        $\langle b := 0; s := 0;\rangle$ }
//
    while( true ) {
        $\langle$await$(a + b + c = 0)$ $c := 1; s := 1;\rangle$
        $C$
        $\langle c := 0; s := 0;\rangle$ }
oc
```

Transforming: Now, by the global invariant $s = a+b+c$, we can replace the expression $a + b + c$ with s anywhere it appears:

```
int $a := 0, b := 0, c := 0$ ;
int $s := 0$ ;
## $a = 0 \wedge b = 0 \wedge c = 0 \wedge s = 0$
# Global invariant: $0 \leq a, b, c \leq 1$
# Global invariant: $s < 2$
# Global invariant: $s = a + b + c$
co
    while( true ) {
        $\langle$await$(s = 0)$ $a := 1; s := 1;\rangle$
        $A$
        $\langle a := 0; s := 0;\rangle$ }
//
    while( true ) {
        $\langle$await$(s = 0)$ $b := 1; s := 1;\rangle$
        $B$
        $\langle b := 0; s := 0; \rangle$ }
//
    while( $true$ ) {
        $\langle$await$(s = 0)\ c := 1; s := 1; \rangle$
        $C$
        $\langle c := 0; s := 0; \rangle$ }
oc
```

Diminishing: At this point, variables a, b, and c are not used in the algorithm; we can demote a, b, and c to the status of ghosts, or eliminate them altogether.

```
int $s := 0$ ;
## $s = 0$
# Global invariant: $s < 2$
co
    while( $true$ ) {
        $\langle$await$(s = 0)\ s := 1; \rangle$
        $A$
        $\langle s := 0; \rangle$ }
//
    while( $true$ ) {
        $\langle$await$(s = 0)\ s := 1; \rangle$
        $B$
        $\langle s := 0; \rangle$ }
//
    while( $true$ ) {
        $\langle$await$(s = 0)\ s := 1; \rangle$
        $C$
        $\langle s := 0; \rangle$ }
oc
```

The command $\langle$**await**$(s = 0)\ s := 1; \rangle$ can easily be implemented with the test-and-set instruction found on many computers.

References

[Andrews, 2000] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and distributed programming.* Addison Wesley Longman, 2000.

[Blikle, 1979] Andrzej J. Blikle. Assertion programming. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, number 74 in Lecture Notes in Computer Science, pages 26–42, 1979.

[Feijen and van Gasteren, 1999] W. H. J. Feijen and A. J. M. van Gasteren. *On a Method of Multiprogramming.* Springer-Verlag, 1999.

[Floyd, 1967] Robert Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics, Volume XIX*, 1967.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Lamport, 1977] L. Lamport. Proving the correctness of multiprocess programs. *Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[Owicki and Gries, 1976a] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6:319–340, 1976.

[Owicki and Gries, 1976b] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.