Proof Outline Logic

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

Draft typeset April 20, 2009

Abstract

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

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 statements 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 statements 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 statements; for smaller examples and in the theory, I'll use {/} for assertions with (/) for grouping statements.

2 Assertions and logic

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

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

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 preceeded by ## and are followed by an end-of-line like this:[1]

```
int x ;
int y ;
x := 5 ;
y := -5 ;
## x ≥ 0 ∧ 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 ≥ 0 ∧ x + y = 0} z := x+y ;
```

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 ;
```

[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.

```
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 ) ;
```

2.2.1 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.

[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.

- In execution, executable assertions 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 an assertion P by an expression (E). We write $P_{x \leftarrow E}$ for the new condition.[5] For example

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

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.

[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 is taken care of by evaluating assertions only when they have exclusive access to the data they refer to.

[5] The number of different notations that different people use for substitution is stupendous. In other courses, I usually use $P[x := E]$. Here I am following Andrews's book.

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
$\mathit{false} \Rightarrow P$	Anti-Domination
$P \Rightarrow P$	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

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

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.

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

$$V = 1000I \qquad ,$$

provided

$$-10 \le V \le +10 \qquad .$$

In programming, we can use a pair of assertions as a specification or "contract" for a statement or subroutine. The first assertion is the so-called precondition, it specifies the operating conditions, that is, the state of the program when the statement begins operation. The second assertion specifies the state of the program when (and if) the statement ends operation. For example, the following pair of conditions specifies a statement 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

---

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

Such a triple, consisting of a precondition, a statement, 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:

$\#\#\ y = 4$
$x := 5\ ;$
$\#\#\ x = 5 \qquad .$

Here is one more:

$\#\#\ y = 4$
$y := y + 1\ ;$

[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.

$x := y$;
$x = 5$.

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 traditionally written with capital letters. The following contract specifies that y must not change and that the final value of x must be larger than that of y:

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

where it is understood that x and y are state variables of type `int` while Y is a constant[7] of type `int`. This particular contract may or may not have a solution, depending on how we interpret the type `int`. If we consider `int` variables to range over all mathematical integers, then $x := y + 1$ is a solution. On the other hand, if, as in most programming languages, `int` is a finite subset of the mathematical integers, then there is no solution to this particular contract.

3.2 The meaning of a triple

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

P
S
Q

We define that the triple **is valid** if and only if, for all possible values of all constants, whenever the execution of S is started in a state satisfying P then the execution of S can only end in a state satisfying Q.

Note that the definition of validity does not require that S should terminate. Thus the following triple is valid even if we interpret x to have the type `int` and `int` to range over all mathematical integers

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

[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 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.

If we consider states where x is initially 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!

Hoare triples indicate correctness up-to-but-not-including termination. This form of correctness is sometimes called **partial correctness**. 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.

We write $\vdash \{P\}\ S\ \{Q\}$ to mean "$\{P\}\ S\ \{Q\}$ is valid"[8]

3.3 Some examples of assignments

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 valid? (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 valid.

[8] The use of $\vdash$ and the equivalent verb phrase "is valid" may seem a bit fussy. In mathematical writing, a mathematical formula is either a noun (e.g. $x + y$) or an independent clause (e.g. $x < y$) —i.e. a statement of fact. Relations are usually treated as verbs. We write

"If $x < y$, then $f(x) < f(y)$."

rather than

"If $x < y$ is valid, then $f(x) < f(y)$ is also valid."

We could think of a statement S as being a relation between conditions and thus think of "$\{P\}\ S\ \{Q\}$" as an independent clause stating that the relationship S holds between conditions P and Q. However, it is useful to think of Hoare triples (and proof outlines, which we will see later) as structures and hence as nouns. For example we will want to say things such as that one proof outline does not interfere with another. Thus to form an independant clause out of the noun phrase "$\{P\}\ S\ \{Q\}$" , we need to supply a verb, either in English or via the unary relation $\vdash$. Luckily the term "Hoare triple" —as opposed to, say, "Hoare sentence"— reenforces that we are dealing with an object.

This is a common quandary in writing about logic: What is clearly a sentence to a practitioner is an object of study to the logician and hence a noun. When I put on my practitioner's hat, I'll often drop the phrase "is valid" or the equivalent notation $\vdash$.

How about

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

Is this valid? Well, if initially $2x$ is $3y$ then after changing x to $2x$ then 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

$$\vdash \{P\}\ x := E\ \{Q\}\ \text{ exactly if } P \text{ is as strong as } Q_{x\leftarrow E}$$

What I mean by 'P is as strong as $Q_{x\leftarrow E}$' is that, for all possible values of any free variables, $P \Rightarrow Q_{x\leftarrow E}$. Here is an example where the precondition is stronger than it needs to be:

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

The necessary and sufficient precondition for $x := 2x$ to establish $x \leq 3y$ is $(x \leq 3y)_{x\leftarrow 2x}$ which is $2x \leq 3y$; this is implied (for all values of x and y) by the given precondition of $2x < 3y$.

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

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

is valid, where

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

To show that this triple is valid, we'll need to deal with constructs other than assignments.

3.5 Proof outlines

A **proof outline** is a statement 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.[9]

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

3.6 Validity of proof outlines

We can formally define valid proof outlines for sequential programs as follows:

Assignment Rule: $\{P\}\ x := E\ \{Q\}$ is a valid proof outline if

$$P \Rightarrow Q_{x \leftarrow E}, \text{ for all values of all variables} \qquad .$$

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

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

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

[9] In keeping with Andrews's text, I use braces to group statements. Generally I'll use braces to group statements and ## to indicate assertions in multiline examples. In single line examples, I'll use parentheses to group statements and braces to indicate predicates. I hope this inconsistency is not too confusing.

```
## y ≥ 0 ∧ x = X ∧ y = Y
z := 1 ;
## I : y ≥ 0 ∧ X^Y = z × x^y
while( y > 0 )
## X^Y = z × x^y ∧ y > 0
{
    if( odd(y) )
    ## X^Y = z × x^y ∧ y > 0 ∧ odd(y)
    {
        z := z × y;
        ## y − 1 ≥ 0 ∧ X^Y = z × x^(y−1)
        y := y − 1;
    }
    else
    ## X^Y = z × x^y ∧ y > 0 ∧ even(y)
    {
        x := x × x;
        ## even(y) ∧ y/2 ≥ 0 ∧ X^Y = z × x^(y/2)
        y := y/2;
    }
}
## z = X^Y
```

Figure 1: An example proof outline.

2-Tailed If Rule: $\{P\}$ **if**(E) $\{Q_0\}$ S **else** $\{Q_1\}$ T $\{R\}$ is a valid proof outline, provided $\{Q_0\}$ S $\{R\}$ and $\{Q_1\}$ T $\{R\}$ are both valid proof outlines, and that $P \wedge E \Rightarrow Q_0$ and $P \wedge \neg E \Rightarrow Q_1$ are true for all values of all variables.

1-Tailed If Rule: $\{P\}$ **if**(E) $\{Q\}$ S $\{R\}$ is a valid proof outline, provided $\{Q\}$ S $\{R\}$ is a valid proof outline and that $P \wedge E \Rightarrow Q$ and $P \wedge \neg E \Rightarrow R$ are true for all values of all variables.

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

- that $P \wedge E \Rightarrow Q$ is true for all values of all variables,
- that $\{Q\}$ S $\{P\}$ is a valid proof outline, and

- that $P \wedge \neg E \Rightarrow R$ is true for all values of all variables.

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 statement 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.[10]

Parentheses Rule: $\{P\}\ (S)\ \{Q\}$ is a valid proof outline, provided $\{P\}\ S\ \{Q\}$ is a valid proof outline.[11]

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

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

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

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

Suppose $\{P\}\ S\ \{Q\}$ is a valid 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 valid Hoare triple.

---

[10] 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.

[11] 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-statement. 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

$$\{P\}\ (S)\ \{Q\}$$

Equivalently in Andrew's notation, I'll write

$$\begin{array}{l} \{\ \#\#\ P \\ \quad S\ \} \\ \#\#\ Q \end{array}$$

rather than

$$\begin{array}{l} \#\#\ P \\ \{\ S\ \} \\ \#\#\ Q \end{array}$$

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 Validity of the example

There is a little, but not much, work left to show that the example proof outline is valid. You should verify that each of these implications is a tautology, i.e. true for all values of their variables (including constants).

Let

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

- Because of the first assignment, we must show

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

  is a tautology. After substitution we have

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

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

- From the rule for while-loops we must show

  $$I \land y > 0 \Rightarrow X^Y = z \times x^y \land y > 0$$
  $$I \land \neg(y > 0) \Rightarrow z = X^Y$$

  are tautologies. Both are fairly straight-forward

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

  $$X^Y = z \times x^y \land y > 0 \land odd(y) \Rightarrow X^Y = z \times x^y \land y > 0 \land odd(y)$$
  $$X^Y = z \times x^y \land y > 0 \land \neg odd(y) \Rightarrow X^Y = z \times x^y \land y > 0 \land even(y)$$

  are tautologies. Both are trivial.

- The four assignments in the loop body give rise to four expressions to show to be tautologies:

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

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[12]

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

---

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[13] 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

---

$\#\#\ true$

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

[13] 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

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

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

---

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, then 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 statements.

4.2 Rule for concurrent execution

We can extend the logic to include a statement for concurrent execution, by which we mean an arbitrary interleaving of the atomic actions of two processes.[14] We'll use the notation

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

for the concurrent composition of two statements 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

[14] 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.

P is an assertion from $\{P_T\}T\{Q_T\}$ (possibly P_T or Q_T, but also possibly an assertion embedded with statement 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.[15]

Concurrent execution Rule:

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

is a valid proof outline iff

- the precondition of the co statement implies the precondition of each component:

  $$P \Rightarrow P_S, \text{ for all values of all variables and}$$
  $$P \Rightarrow P_T, \text{ for all values of all variables,}$$

- the conjunction of the postconditions of its components implies the postcondition of the co statement:

  $$Q_S \wedge Q_T \Rightarrow Q, \text{ for all values of all variables,}$$

- $\{P_S\}S\{Q_S\}$ and $\{P_T\}T\{Q_T\}$ are both valid proof outlines (local correctness) and

- they do not interfere with each other (freedom from interference).

4.3 Awaiting

Processes can synchronize by means of an await statement

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

where E is a boolean expression and S is any statement[16]. The idea is that the process delays until E becomes true and then the statement S is executed atomically with the evaluation of E.

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

[16] Except that await statements are not allowed inside of await statements, nor are co statements.

Await rule: A proof outline

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

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

Thus, in a sense, the await statement causes E to become true, almost as if by magic. For example the following program 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 statement 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 valid proof outline may still embody a program that will never terminate.

As abbreviations we have:

- $\langle S\rangle$ abbreviates $\langle \textbf{await}(\ \mathit{true}\ )\ S\rangle$, and
- $\langle \textbf{await}(\ E\ )\rangle$ abbreviates $\langle \textbf{await}(\ E\ )\ \textbf{skip}\rangle$

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

$$P \wedge E \Rightarrow R, \text{ for all values of all variables.}$$

To account for the atomicity of the await statements, we modify the definition of 'proof outlines do not interfere with each other' to exclude assertions that are contained within await statements. 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 statement..

5 A formalization of proof outline logic

This section is optional reading. I wrote it mainly to clarify my own understanding. I include it as it may also help clarify yours.

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, statements S, blocks B and proof outlines O is given by:

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

There are a number of restrictions not indicated by the syntax

- In an assignment statement, 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 statements must be boolean.
- The scope of a variable is the block that follows its declaration.
- Await statements may not occur within await statements, directly or indirectly.
- The choice between 1- and 2-tailed if statements 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

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

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

A few other comments are in order.

- I only include the binary case for the concurrent statement. The generalization to more processes is straight-forward.
- I omit the abbreviations for await statements. These are $\langle B\rangle$ abbreviates $\langle \mathbf{await}(\ \mathit{true}\ )\ B\rangle$ and $\langle \mathbf{await}(\ P\ )\rangle$ abbreviates $\langle \mathbf{await}(\ P\ )\ \mathbf{skip}\rangle$.
- I use round parentheses to turn blocks into statements 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.
- I use $\{P\}$ to indicate an assertion whereas Andrews uses $\#\#P$.
- In practice a fragment "$\{P\}\,($" (or "$\begin{array}{l}\#\#P\\ \{\end{array}$") 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.

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.

5.2.2 Rules

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

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

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

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.[17]

$$\frac{\vdash P \Rightarrow Q_{v_0,v_1,...,v_{n-1} \leftarrow e_0,e_1,...,e_{n-1}}}{\vdash \{P\}\ v_0, v_1, ..., v_{n-1} := e_0, e_1, ..., e_{n-1};\ \{Q\}}\text{(Assign)}$$

$$\frac{\vdash P \Rightarrow Q}{\vdash \{P\}\ \textbf{skip}\ \{Q\}}\text{(Skip)} \qquad \frac{\vdash \{P \wedge E\}\ B\ \{Q\}}{\vdash \{P\}\ \langle \textbf{await}(\ E\ )\ B\rangle\ \{Q\}}\text{(Await)}$$

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

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

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

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

[17] 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.

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 statements are executed atomically. Similarly the guard expressions in if and while statements are assumed to be evaluated atomically. If you are not comfortable with this —and you probably shouldn't be— then restrict all expressions and assignments that appear outside await statements to those that obey the the 'At Most Once property' defined next.

A *critical reference* is a reference to a variable that is changed in another process.

An expression E satisfies the *At Most Once property* iff

- E contains at most one critical reference.

An assignment $x := E$ satisfies the *At Most Once* property if either:

1. E contains at most one critical reference and x is not read by another process, or

2. E contains no critical references.

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 valid.

Generally we need to show

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

is valid. Some observations:

- The Hoare-triple is valid if and only if

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

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

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

- If the precondition is a conjunction, for example $P = (P0 \wedge P1 \wedge P2)$, 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 valid.

- We can show the postcondition in parts. For example if $Q = Q0 \wedge Q1$ 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 valid. 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 valid. This is because $Q_{x \leftarrow E}$ is then simply Q and we have to show

$$\vdash Q \wedge P \Rightarrow Q \qquad ,$$

  which is trivially valid. We call this situation **disjoint variables**.

6.2 An example with no interference

Consider the following program that increments x and y in parallel

---

```
## x = X ∧ y = Y
co
    ## x = X
    ⟨x := x + 1;⟩
    ## x = X + 1
//
    ## y = Y
    ⟨y := y + 1;⟩
    ## y = Y + 1
oc
## x = X + 1 ∧ y = Y + 1
```

To show this proof outline valid we must check the following

- *The precondition of the co statement implies the preconditions of each component of the co statement.*
  - $x = X \wedge y = Y \Rightarrow x = X$
    This is 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 statement.*

  The postconditions of the components are respectively $x = X + 1$ and $y = Y + 1$. The conjunction of them is the postcondition of the co statement itself.
- *Local correctness.* We need to show that each component is itself a valid proof outline. By application of the assignment rule. We have that
  $$\{x = X\}\ x := x + 1;\ \{x = X + 1\}$$
  is a valid Hoare triple and hence, it is a valid proof outline. 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

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

  is valid. From the assignment rule, we must show

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

  which simplifies to

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

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

  The other three action/assertion pairs are 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 statement 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 statement.* As they are the same, this implication is also trivial.
- *Local correctness.* Each component is a valid 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 valid. 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 a bit. Initially $x = X$ is true, but while the first thread is waiting to execute its statement, the statement 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 statement 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 leading to

---

$x = X$
co

$$
\begin{array}{l}
\quad\quad \#\#\ x = X \vee x = X+3 \\
\quad\quad \langle x := x + 2; \rangle \\
\quad\quad \#\#\ ? \\
// \\
\quad\quad \#\#\ x = X \vee x = X+2 \\
\quad\quad \langle x := x + 3; \rangle \\
\quad\quad \#\#\ ? \\
\textbf{oc} \\
\#\#\ x = X + 5
\end{array}
$$

From the preconditions and the assignments, we can see that after the first statement 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:

$$
\begin{array}{l}
\#\#\ x = X \\
\textbf{co} \\
\quad\quad \#\#\ x = X \vee x = X+3 \\
\quad\quad \langle x := x + 2; \rangle \\
\quad\quad \#\#\ x = X+2 \vee x = X+5 \\
// \\
\quad\quad \#\#\ x = X \vee x = X+2 \\
\quad\quad \langle x := x + 3; \rangle \\
\quad\quad \#\#\ x = X+3 \vee x = X+5 \\
\textbf{oc} \\
\#\#\ x = X + 5
\end{array}
$$

Now is this proof outline valid? We must check:

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

$$P \Rightarrow P \vee Q$$

- *The conjunction of the postconditions of its components implies the postcondition of the co statement.* We must check

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

which is true.

- *Local correctness.* We need to check each assignment statement. I.e. the validity of

  $$\{x = X \lor x = X + 3\}\ \langle x := x + 2; \rangle\ \{x = X + 2 \lor x = X + 5\} \qquad \text{, and}$$
  $$\{x = X \lor x = X + 2\}\ \langle x := x + 3; \rangle\ \{x = X + 3 \lor x = X + 5\} \qquad .$$

  For the first we substitute in the postcondition to get

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

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

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

  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 validity of

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

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

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

    which is true.

  - For the second, we need to check

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

The precondition simplifies to $x = X + 2$. The postcondition, under the substitution, is $x + 3 = X + 2 \vee x + 3 = X + 5$ which simplifies to $x = X - 1 \vee x = X + 2$. So we need to check

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

– 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 statement if

- it is implied by the precondition of the co statement,
- each atomic action in the co statement preserves the invariant in the sense that
$$\{G \wedge P_a\}\ a\ \{G\}$$
is valid for each atomic action a in the co statement, 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 triples of the form

$$\{G \wedge P_L\}\ a\ \{G \wedge Q_L\}$$

valid, 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")

$$\{G \wedge P_L\}\ a\ \{Q_L\}$$

valid. For freedom from interference, we need to show triples of the form

$$\{Q_L \wedge G \wedge P_L\}\ a\ \{G \wedge Q_L\}$$

to be valid, 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")

$$\{Q_L \wedge G \wedge P_L\}\ a\ \{Q_L\}$$

valid.

We can save a lot of writing by noting the relevant global invariants between the co statement 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 statements 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

---

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 valid proof outline with the stated pre- and postconditions..*

Figure 2 on page 32 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

```
## x = X
int q := 0 , i := 0
## x = X ∧ q = 0 ∧ i = 0
## Global Inv: 0 ≤ q ≤ 2
## Global Inv: i ≥ 0
## Global Inv: q ≠ 1 ⇒ x = h^i(X)
## Global Inv: q = 1 ⇒ x = g(h^{i-1}(X)) ∧ i > 0
co
    while( q ≠ 2 ) {
        ⟨await (q = 1)⟩
        ## q = 1
        ⟨x, q := f(x), 0;⟩ }
//
    ## i ≤ N ∧ q = 0
    while( i < N ) {
        ## i < N ∧ q = 0
        ⟨x, q, i := g(x), 1, i + 1;⟩
        ## i ≤ N
        ⟨await (q = 0)⟩
    ## q = 0 ∧ i = N
    ⟨q := 2;⟩
    ## q = 2 ∧ i = N
oc
## x = h^N(X)
```

Figure 2: A client and server system.

statement. These invariants are to be thought of as being conjoined with each assertion within the co statement. Thus the full precondition of the statement $\langle x, q := f(x), 0; \rangle$ 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 $\langle$**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 valid, 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 valid.
- *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.
  - $0 \leq q \leq 2$. 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 $0 \leq 0 \leq 2$, $0 \leq 1 \leq 2$, and $0 \leq 2 \leq 2$ which are all trivially true.
  - $i \geq 0$. By disjoint variables we only need to show

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

    which is implied by

    $$i \geq 0 \Rightarrow i + 1 \geq 0$$

    which is clearly valid.[18]

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

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

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

- $q \neq 1 \Rightarrow x = h^i(X)$.
  - The assignment

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

    sets q to 1. Doing the substitution we get

    $$1 \neq 1 \Rightarrow ...$$

    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

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

    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

  $$\langle x, q := f(x), 0; \rangle \qquad \text{and}$$
  $$\langle q := 2; \rangle$$

  both give a substituted postcondition of the form

  $$false \Rightarrow ...$$

  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

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

  valid. 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

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

  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

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

  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 completing a valid 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 ∨ x = 1
    ⟨ x := x + 1; ⟩
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.

```
## x = 0
# int a := 0
# int b := 0
## x = 0 ∧ a = 0 ∧ b = 0
# Global invariant: x = a + b
co
    ## a = 0
    ⟨ x := x + 1; a := 1; ⟩
    ## a = 1
//
    ## b = 0
    ⟨ x := x + 1; b := 1; ⟩
    ## 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 statements 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 $a := 0, b := 0, c := 0$;
$a = 0 \land b = 0 \land c = 0$
Global invariant: $0 \leq a, b, c \leq 1$
Global invariant: $a + b + c < 2$
co
    while(*true*) {
        $\langle$**await**$(a + b + c = 0)\ a := 1; \rangle$
        A
        $\langle a := 0 \rangle$ }
//
    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 statements 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 ∧ b = 0 ∧ c = 0 ∧ s = 0
# Global invariant: 0 ≤ a, b, c ≤ 1
# Global invariant: a + b + c < 2
# Global invariant: s = a + b + c
co
    while( true ) {
        ⟨await(a + b + c = 0) a := 1; s := 1; ⟩
        A
        ⟨a := 0; s := 0; ⟩ }
//
    while( true ) {
        ⟨await(a + b + c = 0) b := 1; s := 1; ⟩
        B
        ⟨b := 0; s := 0; ⟩ }
//
    while( true ) {
        ⟨await(a + b + c = 0) c := 1; s := 1; ⟩
        C
        ⟨c := 0; s := 0; ⟩ }
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 ∧ b = 0 ∧ c = 0 ∧ s = 0
# Global invariant: 0 ≤ a, b, c ≤ 1
# Global invariant: s < 2
# Global invariant: s = a + b + c
co
    while( true ) {
        ⟨await(s = 0) a := 1; s := 1;⟩
        A
        ⟨a := 0; s := 0;⟩ }
//
    while( true ) {
        ⟨await(s = 0) b := 1; s := 1;⟩
        B
        ⟨b := 0; s := 0;⟩ }
//
    while( true ) {
        ⟨await(s = 0) c := 1; s := 1;⟩
        C
        ⟨c := 0; s := 0;⟩ }
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 ) {
        ⟨await(s = 0) s := 1;⟩
        A
```

$\langle s := 0; \rangle$ }

//

while(*true*) {

$\langle \mathbf{await}(s = 0)\ s := 1; \rangle$

B

$\langle s := 0; \rangle$ }

//

while(*true*) {

$\langle \mathbf{await}(s = 0)\ s := 1; \rangle$

C

$\langle s := 0; \rangle$ }

oc

The statement $\langle \mathbf{await}(s = 0)\ s := 1; \rangle$ 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.