AXIOMS OF AN IMPERATIVE LANGUAGE

We will use the same language, with the same abstract syntax that we used for operational semantics. However, we will only be concerned with the commands, since the language has expressions that have no side effects. In other words, only commands (and essentially only assignment) can alter program state. There will be an axiom for alternative command in the abstract syntax. The main form in each axiom is a conclusion with the ‘assertion-command-assertion’ (or ACA) form:

\[
\{ p \} C \{ q \}
\]

In this form C is any command, \( p \) is a true assertion before C is executed, and \( q \) is true after it. \( p \) is the pre-condition and \( q \) is the post-condition, and are expressed in the language of assertions, which contains expressions about arithmetic variables; each variable corresponds to a program language variable in the whole program. Some axioms are expressed as rules, in a similar fashion to the operational semantic rules. Below the line is a conclusion, and above the line will be one or more ACA triples or possibly other assertions or logical expressions. The meaning of these will be similar to the operational semantics rules, i.e. the conclusion below the line can only be made if the expressions above the line are true. We will build correctness proofs of programs by incorporating axioms relating to individual assignments, and building these up with the help of the compound command forms (sequence, conditional, loop) to build a proof for the whole program.

PARTIAL CORRECTNESS

To start with we will make one big assumption concerning each ACA triple. They will only hold if the command C terminates. We can make nor real judgment about the state after the command unless it does. This is called partial correctness. It can expressed as:

if \( p \) is true before C, and if C terminates, then \( q \) is true after it has finished

Each axiom makes the same assumption. Total correctness will be looked at later, when we will be able to prove that C terminates.

If C is a whole program, then \([p,q]\) is the specification of the program. Correctness proofs start with a specification, and break the program down into its simple commands, when the axioms apply.

We will take the axioms in the same order that the abstract syntax presents them:

\[
\text{nop} \mid I := E \mid C_1; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ end} \mid \text{while } B \text{ do } C \text{ end}
\]

WEAK AND STRONG CONDITIONS

Since we are dealing with assertions that can only be true or false, and there can be an infinite number of equivalent expressions, there is one extra goal that we must look out for. When we build our pre- and post-conditions, we should always try to be as accurate as possible. This means that the post-conditions should be as strong as possible, so that they express the most specific version of what the command actually does. Conversely, the pre-condition should be as weak (or general) as possible so as not to overly constrain our expression of what the command does. Later we shall express this goal in an axiom, but let us turn now to the basic command types.

THE AXIOM FOR \text{nop}

Since \text{nop} is meant to do nothing, not surprisingly, it is very simple:

\[
\{ p \} \text{nop} \{ p \}
\]

What this says is that \text{nop} does not alter the program state. Whatever was true before the command is also true after it.
THE AXIOM FOR ASSIGNMENT

The axiom for assignment is counter-intuitive, so we will present it after a couple of examples. First, expression evaluation cannot alter program state, so we need to concentrate on how to express the change in state caused by the assignment of the value of an expression to the identifier in \( I := E \). We might think that we should add an additional assertion to the post-condition. Something like this:

\[
\{ p \} I := E \{ p \land \text{something about } I \text{ and } E \}
\]

e.g. for \( x := 2 \), we could use:

\[
\{ p \} x := 2 \{ p \land x=2 \}
\]

Unfortunately, this will not work, as a simple example shows:

\[
\{ x > 4 \} x := 2 \{ x > 4 \land x = 2 \}
\]

The post-condition is clearly false, for any \( x \), so we have gone from true in the pre-condition, to false in the post-condition. This is clearly no good. All axioms and the proofs that contain them must proceed from true to true. Logic cannot tolerate a rule such as this attempt. Instead, we must instead think like this: what must be the pre-condition be for the post-condition to the true? Expressions not involving the assigned variable can remain the same for pre- to post-condition. Expressions that do involve the assigned variable must be true \textit{for the value of the expression to be assigned} for the post-condition to be true. In this way, the assignment has already been pre-judged in a way. Only certain expressions can be true after an assignment since assignments are destructive. We could never have:

\[
\{ \ldots \} x := 2 \{ x > 4 \}
\]

because 2 (the value of the assigned expression) is not greater than 4. However:

\[
\{ \ldots \} x := 2 \{ x > 0 \}
\]

could work because \( 2 > 0 \). Following this line of thought, we can see that the relationship between pre and post-condition must be such that, for all possible post-conditions that could be true, they must always work as pre-conditions with the value of the expression substituted. This is where our substitution form comes in very handily. We can write the assignment axiom as:

\[
\{ p[E \setminus I] \} I := E \{ p \}
\]

This says that the only post-conditions which are possibly true are those for which it is also true with \( E \) substituted for \( I \) in \( p \). In fact, this usually means that we work backwards with assignments. We have a post-condition, and we ‘push it through’ the assignment to produce the correct pre-condition. Simple examples are:

\[
\{ 2 > 0 \} x := 2 \{ x > 0 \}
\]
\[
\{ x-1 < x \} y := x - 1 \{ y < x \}
\]
\[
\{ x+1 = 3 \} x := x + 1 \{ x = 3 \}
\]

In each case the left hand is either true because of a fact \( \{ 2 > 0 \} \), true because of the axioms of arithmetic, like \( \{ x-1 < x \} \), or true by solving some constraint, like \( \{ x+1 = 3 \} \). What is not possible is to use post-conditions which cannot possibly be true:

\[
\{ 2 < 0 \} x := 2 \{ x < 0 \}
\]
\[
\{ x-1 = x \} y := x - 1 \{ y = x \}
\]
\[
\{ x+1 = 0 \land x+1 = 1 \} x := x + 1 \{ x = 0 \land x = 1 \}
\]
In each of these cases the post-condition produces a pre-condition that is false, or it is false to start with, like the last example. Strictly, these might be considered correct since false produces false, but they are not useful in this form. We only want to apply these axioms when both assertions are true.

**THE AXIOM FOR THE COMMAND SEQUENCE**

Having got the base cases right, i.e. nop and assignment, the rest are recursive in nature, as expressed in the abstract syntax. In the case of the sequence, two commands are executed one after the other. There are possibly two changes of state. We can express this by asking that the post-condition of the first command is the pre-condition of the second. These ACA triples will be conditions to be proved before the sequence can be proved correct:

\[
\{p\} C_1 \{r\} \quad \{r\} C_1 \{q\} \\
\{p\} C_1 ; C_2 \{q\}
\]

As an example, let us put two assignments together, and apply this rule by asserting a post-condition and pushing it back twice through the commands:

The two commands are: \(x := 2; y := x + 1\).

The post-condition will be \(\{y > x\}\).

Pushing this back through the second command gives: \(\{x + 1 > x\} y := x + 1 \{y > x\}\).

Pushing this pre-condition back through the first gives: \(2 + 1 > 2 x := 2 \{x + 1 > x\}\)

This pre-condition is true, so we can assert: \(\{true\} x := 2; y := x + 1 \{y > x\}\) for the sequence.

This says that it does not matter what values \(x\) and \(y\) have to start with, after these two assignments, \(x\) will be greater than \(y\).

It is pointless trying to generate separate ACA triples for each command since our correctness proofs will be goal-directed, i.e. we will have some specification that we are trying to prove. The method does not lend itself well to forward reasoning, such as trying to find all possible assertions which might be true after the program terminates. Operational semantics is much better at that since it can propagate values through different stores as the program executes.

**THE AXIOM FOR THE CONDITIONAL**

The conditional axiom has two difficulties. First we must ensure that both pre- and post-conditions apply regardless of which branch is taken. Second we have somehow to get the condition itself into the act. To do this, we must ensure that all Boolean expressions in the language have a corresponding form in the arithmetic language. This is the case with ours since we only have equal, greater and less, which correspond to =, >, < in arithmetic. We can put both branches in the same axiom by asking that they both be prerequisites for the conditional to be asserted:

\[
\{p \land b\} C_1 \{q\} \quad \{p \land \neg b\} C_2 \{q\} \\
\{p\} \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ end } \{q\}
\]

Here \(b\) is a meta-variable for the expression obtained from the Boolean expression \(B\). Two things are notable. One is that \(p\) and \(q\) are the conditions for both branches; the other is that \(q\) must be one assertion, that essentially handles both of these branches together. Sometimes this assertion is quite simple:

\[
\{true\} \text{if } x \text{ less } 0 \text{ then } x := 1 \text{ else } x := 2 \text{ end } \{x > 0\}
\]

is provable because the test is really irrelevant. If we push the post-condition through both branches we get \(true\) in both cases, since \(2 > 0\) and \(1 > 0\). Since \(true\) is already the weakest possible pre-condition, we don’t care whether the test is true or false. However, \(\{true\} \text{if } x \text{ less } 0 \text{ then } y := x + 1 \text{ else } y := \neg x \text{ end } \{y \leq 0\}\) is more interesting (neg is the
unary negative operator). Using the assignment axiom on both branches gives: \( \{ x + 1 \leq 0 \} \) on the then branch and \( \{-x \leq 0\} \) on the else branch. In order to use the conditional axiom, we must show, since \( p \) is just \( true \), that \( \{ x + 1 \leq 0 \} \) is the same as the test \( \{ x < 0 \} \), and that \( \{-x \leq 0\} \) is the same as its negation \( \{ -x < 0 \} \). In this case both are true, but not without some work: \( x + 1 \leq 0 \rightarrow x \leq -1 \rightarrow x < 0 \) and \( -x \leq 0 \rightarrow x \geq 0 \rightarrow -x < 0 \) using algebraic rewriting rules.

**THE AXIOM FOR THE LOOP**

The loop is the most difficult axiom since it is clearly impossible in our assertion language to express the iterative nature of the execution. What we must do is to capture the state changes in such a way that we can express the semantics of the loop by looking at what happens each time we execute the body of the loop. We would like to write something like:

\[
\{ p \} \quad \{ p \} \quad \{ C \} \quad \{ p \} \quad \{ \text{while } B \text{ do } C \text{ end} \} \quad \{ q \}
\]

But what are the proper pre- and post-conditions for \( C \), and how do we handle the loop exit? The solution is to find a single assertion that is true before the loop starts, after each iteration (so it is the next pre-condition as well), and after the loop ends. This is the loop invariant. It expresses, logically, the fact that the loop continues until the test becomes false, and then the loop exits. The axiom is:

\[
\{ p \} \quad \{ p \} \quad \{ p \wedge b \} \quad \{ C \} \quad \{ p \} \quad \{ \text{while } B \text{ do } C \text{ end } \{ p \wedge \neg b \} \}
\]

This is very subtle. We are unable to show the loop explicitly as we were in operational semantics. Instead we characterize the loop by what it keeps the same, rather than what it changes. While the invariant \( p \) does the loop continuation, the test \( b \) handles the non-continuation. Essentially we have the annotated sequence:

\[
\{ p \wedge b \} \quad \{ C \} \quad \{ p \wedge b \} \quad \{ C \} \quad \{ p \wedge \neg b \}
\]

Obviously, finding the loop invariant is crucial to handling loops. Let’s start very simply with the loop “while \( x \) greater 0 do \( x := x \text{ minus } 1 \) end”, assuming that \( x \) starts out positive. What is the invariant in this case? What is true before, during and after the loop? Since this is really a count-down loop, the only thing we can say is \( \{ x \geq 0 \} \). Can we prove the condition above the line? i.e. is:

\[
\{ x \geq 0 \wedge x > 0 \} \quad x := x - 1 \quad \{ x \geq 0 \} \quad true?
\]

If we push the post-condition back through the assignment we get \( \{ x - 1 \geq 0 \} \). Is this the same as \( \{ x \geq 0 \wedge x > 0 \} \)? First, \( x \geq 0 \wedge x > 0 \) is just the same as \( x > 0 \), since they are both false or both true together. Then we have \( x - 1 \geq 0 \rightarrow x \geq 1 \rightarrow x > 0 \) by algebraic rewrite, so they are indeed the same. So, by applying the axiom we can write:

\[
\{ x \geq 0 \} \quad \text{while } x \text{ greater } 0 \text{ do } x := x \text{ minus } 1 \text{ end } \{ x \geq 0 \wedge \neg (x > 0) \}
\]

The post-condition is interesting, since \( \neg (x > 0) \rightarrow x \leq 0 \), and \( x \geq 0 \wedge x \leq 0 \) is the same as \( x = 0 \). We have proved, by choosing the right invariant, that the loop terminates with \( x = 0 \), quite a result. Other loops will be harder, as we shall see.

For another simple example, let’s look at the loop “while \( y \) greater \( x \) do \( y := y \text{ minus } 1; x := x \text{ plus } 1 \) end”. In this loop, the value of \( y \) and \( x \) approach each other until either \( x = y \) or \( y \) ‘crosses over’ so that \( y = x - 1 \). Potentially we can prove that, so let’s set the final assertion to \( \{ y = x \vee y = x - 1 \} \) which makes it explicit. The only realistic invariant is \( y > x - 2 \) which takes both possible end results into account. In order to prove the loop correct with respect to the final assertion, we need to prove:

\[
\{ y \geq x - 2 \wedge y > x \} \quad y := y \text{ minus } 1; x := x \text{ plus } 1 \quad \{ y > x - 2 \}
\]

If we take the invariant back through the two assignments, using the assignment axiom, we get a pre-condition of \( y - 1 > (x + 1) - 2 \), or \( y > x \) using algebra and arithmetic. As before, we need to show that this is the same as \( y > x - 2 \wedge y > x \). First, we know that \( A \wedge B \Rightarrow B \) for all \( A \) and \( B \). Then, just like before, \( y > x - 2 \wedge y > x \) is the same as \( y > x \), so we do indeed have the right invariant. We can this write the loop conclusion:

\[
\{ y > x - 2 \} \quad \text{while } y \text{ greater } x \text{ do } y := y \text{ minus } 1; x := x \text{ plus } 1 \quad \{ y > x - 2 \wedge \neg (y > x) \}
\]
The proof that this post-condition is the same as \( \{ y = x \lor y = x - 1 \} \) is hard. Showing that they have the same value in all cases is easier. We leave that as an exercise.

**STRENGTHENING AND WEAKENING**

The final axiom (really two axioms) is called the consequence axiom. Very often we will end up with a pre-condition that is not immediately convertible into some desired assertion. Are we doomed to failure in this case? The answer is no, because we may be able to show that one assertion implies the other. What this means on the pre-condition side is that we may not have the weakest pre-condition, but a stronger one. If we can show that the strong one implies the weaker, then we can still apply the axiom in question. As an axiom of its own this is:

\[
\begin{align*}
& p \Rightarrow r \quad \{ r \} C \{ q \} \\
\Rightarrow & \quad \{ p \} C \{ q \}
\end{align*}
\]

Sometimes this is called the ‘strengthening’ axiom. There is a corresponding one on the post-condition side, the weakening axiom:

\[
\begin{align*}
& \{ p \} C \{ r \} \quad r \Rightarrow q \\
\Rightarrow & \quad \{ p \} C \{ q \}
\end{align*}
\]

Here we make the post-condition weaker by finding something that is implied by it. These can be combined in the consequence axiom:

\[
\begin{align*}
& p \Rightarrow r \quad \{ r \} C \{ s \} \quad s \Rightarrow q \\
\Rightarrow & \quad \{ p \} C \{ q \}
\end{align*}
\]

Now we can see that in fact we were a little too strong in our previous loop examples in insisting that, for instance, \( x \geq 0 \land x \leq 0 \) is the same as \( x = 0 \). Logically this could be written as \( (x \geq 0 \land x \leq 0) \Leftrightarrow (x = 0) \), whereas we only really need the forward implication \( (x \geq 0 \land x \leq 0) \Rightarrow (x = 0) \). We can then use the weakening axiom to prove the final result. Whether this is easier than proving the equivalence is not obvious, but if we cannot prove equivalence, we may still be able to prove the implication.