Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 6: April 24, 2012

Lecturer: Mooly Sagiv *Scribe: Michal Balas and Yair Asa*

Axiomatic Semantics

6.1 The basic idea

The problem we would like to solve is how to prove that a program does what we require of it. Given a program, we specify its required behavior based on our intuitive understanding of it. We can run it according to the operational semantics or denotational semantics and compare to its behavior there. For some programs we need to be more abstract (for example programs that receive input), and then it is necessary to use some logic to reason about the program (and how it behaves on a set of inputs and not in one specific execution path). In this case we may eventually develop a formal proof system for properties of the program or showing that it satisfies a requirement. We can then use the proof system to show correctness. These rules of the proof system are called Hoare or Floyd-Hoare rules. Floyd-rules are for flow-charts and Hoare-rules are for structured languages. Originally their approach was advocated not just for proving properties of programs but also giving a method for explaining the meaning of program. The meaning of a program was specified in terms of "axioms" saying how to prove properties of it, in other words it is given by a set of verification rules. Therefore, this approach was named *axiomatic semantics*.
Axiomatic semantics has many applications, such as:

- Program verifiers
- Symbolic execution tools for bug hunting
- Software validation tools
- Malware detection
- Automatic test generation

It is also used for proving the correctness of algorithms or hardware descriptions, "extended static checking (e.g., checking array bounds), and documenting programs and interfaces.

6.1.1 Example

An example program that computes the sum of the first hundred numbers: $\sum_{1 \le m \le 100} m$.

$S := 0;$
$N := 1;$
while $\neg(N = 101)$ **do**
    $S := S + N;$
    $N := N + 1;$

We can see that the commands $S := 0; N := 1;$ initialize the values in the locations. so we add comments as follows;

$S := 0;$
$\{S = 0\}$
$N := 1;$
$\{N = 1\}$
while $\neg(N = 101)$ **do**
    $S := S + N;$
    $N := N + 1;$

We can also add the comment after the execution of the while loop meaning that S will have the required value.

$S := 0;$
$\{S = 0\}$
$N := 1;$
$\{N = 1\}$
while $\neg(N = 101)$ **do**
    $S := S + N;$
    $N := N + 1;$
$\{N = 101 \wedge S = \sum_{1 \le m \le 100} m\}$

Inside the while loop we add comment on both N and S: N ranges from 1 to 101 and S represents the partial sum. This comment express the key relationship between the value at location S and the value at location N.

$S := 0;$
$\{S = 0\}$
$N := 1;$
$\{N = 1\}$
while $\neg(N = 101)$ **do**
    $\{1 \le N < 101 \wedge S = \sum_{1 \le m < N} m\}$
    $S := S + N;$
    $\{1 \le N < 101 \wedge S = \sum_{1 \le m \le N} m\}$
    $N := N + 1;$
$\{N = 101 \wedge S = \sum_{1 \le m \le 100} m\}$

The assertion $S = \sum_{1 \leq m \leq N} m$ is called an *invariant* of the while-loop because it remains true under each iteration of the loop.

6.1.2 Partial Correctness and Total Correctness

We can base a proof system on assertions of the form

$$\{P\}S\{Q\}$$

where P, Q are assertions, i.e. extensions of boolean expressions, and S is a statement (also called: command). The interpretation of an assertion of this form is: for all states σ which satisfy P, if the execution of S from state σ terminates in state σ', then σ' satisfies Q. In other words, any terminating execution of S from a state satisfying P ends up in a state satisfying Q. P is the precondition and Q is the postcondition of the assertion $\{P\}S\{Q\}$.
For example: $\{y \leq x\}z := x; z := z + 1\{y < z\}$ is a valid assertion.

Assertions of this form are known as *Hoare triples* or *Hoare assertions*, named after the logician *C.A.R. Hoare*.

These assertions are called *partial correctness assertions* because they do not say anything about the command S if it fails to terminate.
An example for a statement that does not terminate when executed from any state is:

$$S \equiv \text{while true do skip}$$

Since S does not terminate, $\{true\}S\{false\}$ is a valid partial correctness assertion. In fact any partial correctness assertion for this specific S is valid.

Total correctness assertions, on the other hand, are assertions of the form

$$[P]S[Q]$$

and the interpretation of such an assertion is: for all states σ which satisfy P, the execution of S from state σ must terminate in a state σ' that satisfies Q.

The semantics of assertions for partial correctness:
$\sigma \models A$ means that the state σ satisfies the assertion A (or: A is true at state σ).
In $\{P\}S\{Q\}$, the statement S denotes a partial function from initial states to final states. Thus, the partial correctness assertion means:

$$\forall \sigma, \sigma' \in \Sigma.(\sigma \models P \wedge \langle S, \sigma \rangle \rightarrow \sigma') \Rightarrow \sigma' \models Q.$$

By the denotational semantics:

$$\forall \sigma \in \Sigma.(\sigma \models P \wedge \mathbf{S}[\![S]\!]\sigma \neq \perp) \Rightarrow \mathbf{S}[\![S]\!]\sigma \models Q.$$

If we adopt the convention that $\perp$ represents an undefined state that satisfies all assertions, that is, for all A, $\perp \models A$, the partial correctness assertion $\{P\}S\{Q\}$ can be defined as follows:

$$\forall \sigma \in \Sigma.\sigma \models P \Rightarrow \mathbf{S}[\![S]\!]\sigma \models Q.$$

6.2 Assn - an Assertion Language

What kind of assertions should we include? Since we would like to reason about boolean expressions, we include all the assertions in **Bexp**. We want to make assertions using the quantifiers "$\forall i \cdots$" and "$\exists i \cdots$", therefore we work with extensions of **Bexp** and **Aexp** which include integer variables i over which we can quantify. Then , for example, we use $\exists i.k = i \times l.$ to denote that an integer k is a multiple of another l. We also import well known mathematical concepts, for example, we use: $n! = n \times (n-1) \times \cdots \times 2 \times 1$ for the factorial function.

We first define **Aexpv** which is, basically, **Aexp** extended to include integer variables i, j, k, etc.. So **Aexpv** is given by:

$$a ::= n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1$$

where n ranges over numbers, X ranges over locations and i ranges over integer variables.

We extend boolean expressions to include these more general arithmetic expressions and quantifiers, as well as implication:

$$A ::= \text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid A_0 \wedge A_1 \mid A_0 \vee A_1 \mid \neg A \mid A_0 \Rightarrow a_1 \mid \forall i.A \mid \exists i.A$$

We call the set of extended boolean assertions, **Assn**.

6.2.1 Example

A program that computes the gcd of two numbers:

```
while ¬(M = N) do
    if M ≤ N
        then N := N − M
        else M := M − N
```

In this example we would like to say that there are no preconditions (true) and that the postcondition is that M, N both hold the gcd of the original M, N. In order to formulate this we need to add two variables m, n. Now we can set:

precondition: $(M = m) \wedge (N = n) \wedge (m \geq 0) \wedge (n \geq 0)$

postcondition: $M = N = gcd(m, n)$

6.2.2 Free and Bound Variables

An occurrence of an integer variable i in an assertion is *bound* if it occurs in the scope of an enclosing quantifier $\forall i$ or $\exists i$. Otherwise, it is *free*.

Examples:

- $\exists i.k = i \times l$
  the occurrence of the integer variable i is bound, while those of k, l are free.
- $(i + 100 \leq 77) \wedge (\forall i.j + 1 = i + 3)$
  here the same integer variable i has two different occurrences in the same assertion: the first is free and the second is bound. j is free.

A formal definition using definition by structural induction:
Let $FV(a)$ be the set of free variables of arithmetic expressions, extended by integer variables ($a \in$ **Aexpv**). By structural induction:

$$FV(n) = FV(X) = \emptyset$$
$$FV(i) = \{i\}$$
$$FV(a_0 + a_1) = FV(a_0 - a_1) = FV(a_0 \times a_1) = FV(a_0) \bigcup FV(a_1)$$

for all numbers n, locations X, and integer variables i.
Let $FV(A)$ be the free variables of an assertion A. By structural induction:

$$FV(true) = FV(false) = \emptyset$$
$$FV(a_0 = a_1) = FV(a_0 \leq a_1) = FV(a_0) \bigcup FV(a_1)$$
$$FV(A_0 \wedge A_1) = FV(A_0 \vee A_1) = FV(A_0 \Rightarrow A_1) = FV(A_0) \bigcup FV(A_1)$$
$$FV(\neg A) = FV(A)$$
$$FV(\forall i.A) = FV(\exists i.A) = FV(A) \setminus \{i\}$$

for all $a_0, a_1 \in$ **Aexpv**, integer variables i and assertions A_0, A_1, A.
Any variable which occurs in an assertion A and is not free is said to be bound. An assertion with no free variables is *closed*.

6.2.3 Substitution

Visualization of an assertion A, with free occurrences of integer variable i:

$$\text{- - - } i \text{ - - - } i \text{ - - - -}$$

Let a be a "pure" arithmetic expression (contains no integer variables). Then the result of substituting a for i is:

$$A[a/i] \equiv \text{- - - } a \text{ - - - } a \text{ - - - -}$$

In general substitutions, if a contains integer variables then it may be necessary to rename some bound variables of A in order to avoid some variables of a becoming bound by quantifiers in A.

Let i be an integer variable and a an arithmetic expression without integer variables. We use structural induction to define substitution into arithmetic expressions:

$n[a/i] \equiv n$
$X[a/i] \equiv X$
$j[a/i] \equiv j$
$i[a/i] \equiv a$
$(a_0 + a_1)[a/i] \equiv (a_0[a/i] + a_1[a/i])$
$(a_0 - a_1)[a/i] \equiv (a_0[a/i] - a_1[a/i])$
$(a_0 \times a_1)[a/i] \equiv (a_0[a/i] \times a_1[a/i])$

when n is a number, X a location, j an integer variable ($j \neq i$) and $a_0, a_1 \in$ **Aexpv**. Now we define substitutions of a (no free variables in a) for i in assertions:

$true[a/i] \equiv true$
$false[a/i] \equiv false$
$(A_0 = A_1)[a/i] \equiv (A_0[a/i] = A_1[a/i])$
$(A_0 \leq A_1)[a/i] \equiv (A_0[a/i] \leq A_1[a/i])$
$(A_0 \wedge A_1)[a/i] \equiv (A_0[a/i] \wedge A_1[a/i])$
$(A_0 \vee A_1)[a/i] \equiv (A_0[a/i] \vee A_1[a/i])$
$(\neg A)[a/i] \equiv \neg(A[a/i])$
$(A_0 \Rightarrow A_1)[a/i] \equiv (A_0[a/i] \Rightarrow A_1[a/i])$
$(\forall j.A)[a/i] \equiv (\forall j.A[a/i])$
$(\forall i.A)[a/i] \equiv (\forall i.A)$
$(\exists j.A)[a/i] \equiv (\exists j.A[a/i])$
$(\exists i.A)[a/i] \equiv (\exists i.A)$

where $a_0, a_1 \in$ **Aexpv**, A_0, A_1, A are assertions and j is an integer variable with $j \neq i$.

For substitution in place of a location X we use the same notation. The formal definition is similar to the one above.

$A \equiv$ - - - X - - - X - - - -
$A[a/X] \equiv$ - - - a - - - a - - - -

Example Assertions:

- i is a prime number:
  $PRIME \equiv \forall j, k \;.\; j \times k = i \Rightarrow (j = 1 \vee k = 1)$

- i is the least common multiple of j, k:
  $LCM \equiv i = (\mid j \times k \mid / gcd(j, k))$

6.3 Semantics of Assertions

The extended arithmetic expressions include integer variables. In order to describe the value of such an expression we must first interpret integer variables as particular integers.

An *interpretation* is a function assigning integer to each integer variable: I :**Intvar** $\to$ **N**.

The meaning of Aexpv:
We define a semantic function Av giving the value of an arithmetic expression a with integer variables in a particular state σ in a particular interpretation I. It is written as $Av[\![a]\!]I\sigma$ or $(Av[\![a]\!](I))(\sigma)$. It is defined by structural induction:

$Av[\![n]\!]I\sigma = n$
$Av[\![X]\!]I\sigma = \sigma(X)$
$Av[\![i]\!]I\sigma = I(i)$
$Av[\![a_0 + a_1]\!]I\sigma = Av[\![a_0]\!]I\sigma + Av[\![a_1]\!]I\sigma$
$Av[\![a_0 - a_1]\!]I\sigma = Av[\![a_0]\!]I\sigma - Av[\![a_1]\!]I\sigma$
$Av[\![a_0 \times a_1]\!]I\sigma = Av[\![a_0]\!]I\sigma \times Av[\![a_1]\!]I\sigma$

This extends the semantics for arithmetics expressions without integer variables.

For all $a \in$ **Aexp**, for all states σ and for all interpretations I
$A[\![a]\!]\sigma = Av[\![a]\!]I\sigma$

The meaning of Assn:
The semantic function requires an interpretation function as a further argument, since integer variables are included. The interpretation function provides a value in **N**, which is the interpretation of integer variables.
We use $I[n/i]$ to define the interpretation got from I by changing the value of i to n.

We can specify the meaning of assertions in **Assn** in the same way we did for expressions in **Aexpv** using a semantics function. Or given an interpretation I define directly the states which satisfy an assertion.

We extend the set of states Σ to the set $\Sigma_\perp$ which includes the value $\perp$ for non-terminating computation. By structural induction we define when $\sigma \models^I A$ ($A \in$ **Assn**), i.e., when state σ satisfies A in interpretation I. Then we extend it so $\perp \models^I A$. We define by structural induction for all $\sigma \in \Sigma$:

$\sigma \models^I true$
$\sigma \models^I (a_0 = a_1)$ if $Av[\![a_0]\!]I\sigma = Av[\![a_1]\!]I\sigma$
$\sigma \models^I (a_0 \leq a_1)$ if $Av[\![a_0]\!]I\sigma \leq Av[\![a_1]\!]I\sigma$
$\sigma \models^I A \wedge B$ if $\sigma \models^I A$ and $\sigma \models^I B$
$\sigma \models^I A \vee B$ if $\sigma \models^I A$ or $\sigma \models^I B$
$\sigma \models^I \neg A$ if not $\sigma \models^I A$
$\sigma \models^I A \Rightarrow B$ if (not $\sigma \models^I A$) or $\sigma \models^I B$
$\sigma \models^I \forall i.A$ if $\sigma \models^{I[n/i]} A$ for all $n \in$ **N**
$\sigma \models^I \exists i.A$ if $\sigma \models^{I[n/i]} A$ for some $n \in$ **N**
$\perp \models^I A$

The semantics of boolean expressions (which are certain kinds of assertions): For all $b \in$

Bexp, for all states σ and for all interpretations I
$B[\![b]\!]\sigma = true$ iff $\sigma \models^I b$, and
$B[\![b]\!]\sigma = false$ iff not $\sigma \models^I b$

6.3.1 Partial Correctness Assertions

The partial correctness, as defined in subsection 6.1.2, does not interest us as programmers, since finding out whether it is true at a particular state is not that important to us. What will usually be more interesting is whether it is true at all states. This will be discussed next.

6.3.2 Validity

Let I be an interpretation and consider $\{P\}c\{Q\}$. We want this partial correctness assertion to be true at all states with respect to the interpretation I, i.e.

$$\forall\sigma \in \Sigma_\perp.\sigma \models^I \{P\}c\{Q\}$$

or:

$$\models^I \{P\}c\{Q\}$$

In fact, we are interested to know whether or not it is true at all states for all interpretations I, i.e.

$$\models \{P\}c\{Q\}$$

This is called the *validaity*. When $\models \{P\}c\{Q\}$ we say the partial correctness assertion $\{P\}c\{Q\}$ is *valid*.

For example, consider

$$\{i < X\}X := X + 1\{i < x\}$$

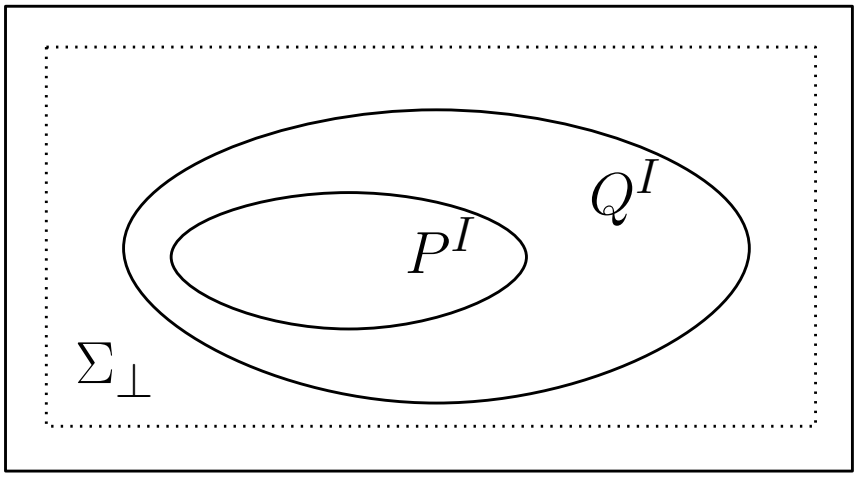
we are interested in whether or not it is true at all states for all interpretations I rather then in a particular value associated with i by the interpretation I.

Similarly, for any assertion A we say $\models A$ iff for all interpretations I and states σ, $\sigma \models^I A$. Then A is valid.

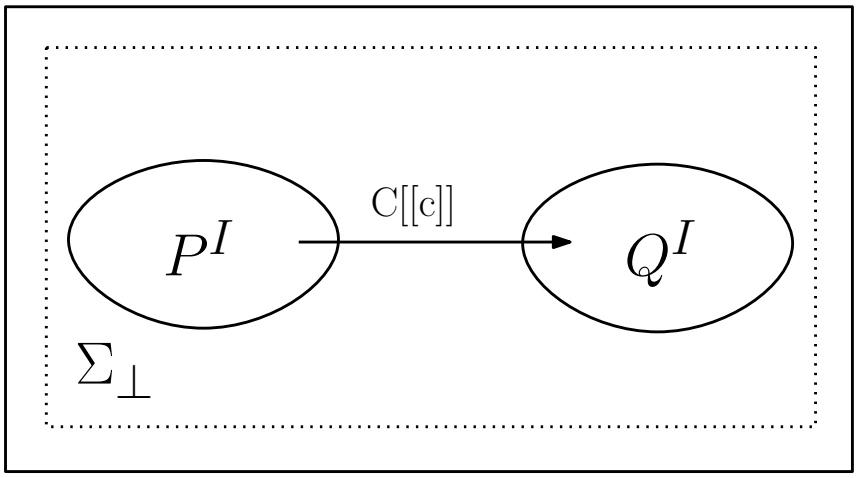
6.3.3 Examples

- Suppose $\models (P \Rightarrow Q)$.
  Then for any interpretation I
  $$\forall\sigma \in \Sigma.((\sigma \models^I P) \Rightarrow (\sigma \models^I Q))$$
  i.e. $P^I \subseteq Q^I$.
  So $\models (P \Rightarrow Q)$ iff for all interpretations I, all states which satisfy P also satisfy Q.

- Suppose $\models \{P\}c\{Q\}$.
  Then for any interpretation I



$\forall \sigma \in \Sigma.((\sigma \models^I P) \Rightarrow (C[\![c]\!]\sigma \models^I Q))$
i.e. the image of P under $C[\![c]\!]$ is included in Q, i.e. $C[\![c]\!]P^I \subseteq Q^I$.



So $\models \{P\}c\{Q\}$ iff for all interpretations I, if c is executed from a state which satisfies A then if its execution terminates in a state, that state will satisfy B.
Remark: P^I and Q^I are not necessarily disjoint.

6.4 Proof Rules for Partial Correctness

Hoare rules are a set of proof rules that are syntax-directed. This rules reduce proving a partial correctness assertion of a compound command to proving partial correctness assertions of its immediate subcommands.

6.4.1 Hoare Proof Rules for Partial Correctness

1. Rule for skip: $\{A\}skip\{A\}$
2. Rule for assignments: $\{B\,[a/X]\}X := a\{B\}$
3. Rule for sequencing: $\dfrac{\{A\}c_0\{C\}\{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$
4. Rule for conditionals: $\dfrac{\{A \wedge b\}c_0\{B\}\{A \wedge \neg b\}c_1\{B\}}{\{A\}\text{if}\quad b\quad \text{then}\quad c_0\quad \text{else}\quad c_1\{B\}}$
5. Rule for while loops: $\dfrac{\{A \wedge b\}c\{A\}}{\{A\}\text{while}\quad b\quad \text{do}\quad c\{A \wedge \neg B\}}$
6. Rule of consequence: $\dfrac{\models (A \to A')\{A'\}c\{B\} \models (B \to B')}{\{A\}c\{B\}}$

6.5 Example - Using Hoare rules

$Y := 1;$
while $X > 0$ do
    $Y := X * Y;$
    $X := X - 1;$

This code computes $n!$ where n is the initial value of X.
We will use Hoare's rules to prove this.
We begin by setting our precondition :

$$P := \{X = n \wedge n \geq 0\}$$

Now we go to the first statement :

$$Y := 1;$$

This is assignment so we will use rule no. 2 :

$$\{X = n \wedge n \geq 0\}Y := 1; \{X = n \wedge Y = 1 \wedge n \geq 0\}$$

We need the postcondition to match the precondition of the next statement (While) so can latter use rule no. 3. Therefore we will now use rule no. 6 :

$$\{X = n \wedge n \geq 0\}Y := 1; \{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}$$

We move on to next statement which is the while. We begin with it's most inner statements. Therefore we begin with $Y := X * Y;$ using rule no. 2:

$$\{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}Y := X * Y; \{X > n \wedge n \geq 0 \wedge Y = n!/(X-1)!\}$$

In the same way we do statement $X := X - 1;$:

$$\{X > 0 \wedge n \geq 0 \wedge Y = n!/(X-1)!\}X := X - 1; \{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}$$

We combine the two using rule no. 3 :

$$\{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}Y := X * Y; X := X - 1; \{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}$$

In order to use rule no. 5 on the loop we need the precondition of the inner statement to match the invariant and the boolean condition. Therefore we will now use rule no. 6 on the last statement :

$$\{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X! \wedge X > 0\}Y := X * Y; X := X - 1; \{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\}$$

No we can use rule no. 5 on the while statement:

$$\{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\} \text{ while } X > 0 \text{ do } Y := X * Y; X := X - 1; \{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X! \wedge \not{X} >$$

We will use rule no. 6 on the while statement in order for the postcondition the match the final wanted result:

$$\{X \geq 0 \wedge n \geq 0 \wedge Y = n!/X!\} \text{ while } X > 0 \text{ do } Y := X * Y; X := X - 1; \{Y = n!\}$$

Lastly we will combine the while with first statement using rule no. 3 to get the complete proof:

$$\{X = n \wedge n \geq 0\}Y := 1; \text{ while } X > 0 \text{ do } Y := X * Y; X := X - 1; \{Y = n!\}$$

6.6 Soundness

Every rule should preserve validity, in the sense that if the assumptions in the rule's premise is valid then so is its conclusion. When this holds of a rule it is called sound. When every rule of a proof system is sound, the proof system itself is said to be sound. It follows then by rule-induction that every theorem obtained from the proof system of Hoare rules is a valid partial correctness assertion.
The proof of soundness of the rules depends on some facts about substitution.
Lemma 6.8: Let I be an interpretation. *Let* $a, a_0 \in Aexpv$. Let $X \in XLoc$. Then for all interpretions I and states σ

$$Av[[a_0[a/X]]]I\sigma = Av[[a_0]]I\sigma[Av[[a]]I\sigma/X]$$

Lemma 6.9: Let I be an interpretation. Let $B \in Assn, X \in Loc$ and $a \in Aexp$. For all states $\sigma \in \Sigma$

$$\sigma \models^I B[a/X] iff \sigma[A[[a]]\sigma/X] \models^I B.$$

Theorem: Let $\{A\}c\{B\}$ be a partial correctness assertion. If $\vdash \{A\}c\{B\}$ then $\{A\}c\{B\}$.

Proof: Clearly if we can show each rule is sound (i.e preserves validity in the sense that if its premise consists of valid assertions and partial correctness assertions then so is its conclusion) then by rule-induction we can see that every theorem is valid.

The rule for skip : Clearly $\models \{A\}skip\{A\}$ so the rule for skip is sound.

The rule for assignment: Assume $c \equiv (X := a)$.
Let I be an interpretation. We have $\sigma \models^I B[a/X] \quad iff \quad \sigma[A[[a]]\sigma/X] \models^I B$, by Lemma 6.9 Thus.$\sigma \models^I B[a/X] \rightarrow C[[X := a]]\sigma \models^I B$, and hence $\models \{B[a/X]\}X;= a\{B\}$, showing the soundness of the assignment rule.

The rule for sequencing: Assume $\models \{A\}c_0\{C\}$ and $\models \{C\}c_1\{B\}$.
Let I be an interpretation. Suppose $\sigma \models^I A$. Either $\sigma \models^I b$ or $\sigma \models^I \neg b$. In the former case $\sigma \models^I A \wedge b$ so $C[[c_0]]\sigma \models^I B$, as $\models^I \{A \wedge b\}c_0\{B\}$. In the latter case $\sigma \models^I A \wedge \neg b$ so $C[[c_1]]\sigma \models^I B$, as $\models^I \{A \wedge \neg b\}c_1\{B\}$. This ensures $\models \{A\}if \quad b \quad then \quad c_0 \quad else \quad c_1\{B\}$

The rule for while-loops: *Assume* $\models \{A \wedge b\}c\{A\}$, i.e. A is an invariant. $w \equiv while \quad b \quad do \quad c$.

Let I be an interpretation. Recall that $C[[w]] = \cup_{n\in w}\Theta_n$ where $\Theta_0 = \Theta$, $\Theta_{n+1} = \{(\sigma, \sigma')|B[[b]]\sigma = true$ and $(\sigma, \sigma') \in \Theta_n \circ C[[c]]\} \cup \{(\sigma, \sigma'|B[[b]]\sigma = false)\}$

We shall show by mathematical induction that $P(n)$ holds where $P(n) \leftrightarrow def^{\forall_{\sigma,\sigma'} \in \Sigma}(\sigma, \sigma') \in \Theta_n$ and $\sigma \models^I A \rightarrow \sigma' \models^I A \wedge \neg b$

For all $\models^I A \models C[[\omega]]\sigma \models^I A \wedge \neg b$

For all states σ, and hence that $\models \{A\}\omega\{A \wedge \neg b\}$, as required.

Base case $n = 0$;When $n = 0, \Theta_0 = \emptyset$ so that induction hypothesis $P(0)$ is vacuously true.

Induction Step: We assume the induction hypothesis $P(n)$ holds for $n \geq 0$ and attempt to prove $P(n+1)$. Suppose $(\omega, \omega') \in \Theta_{n+1}$ and $\omega \models^I A$. Either (i)$B[[b]]\omega = true$ and $(\omega, \omega') \in \Theta_n \circ C[[c]]$, or (ii)$B[[b]]\omega = false$ and $\omega' = \omega$

We show in either case that $\omega' \models^I A \wedge \neg b$. Assume (i). As $B[[b]]\omega = true$ we have $\omega \models^I b$ and hence $\omega \models^I A \wedge b$. Also $(\omega, \omega'') \in C[[c]]$ and $(\omega'', \omega') \in \Theta_n$ for some state ω''. We obtain $\omega' \models^I A$, as $\models \{A \wedge b\}c\{A\}$. From the assumption $P(n)$, we obtain $\omega' \models^I A \wedge \neg b$.
Assume (ii). As $B[[b]]\omega = false$ we have $\omega \models^I \neg b$ and hence $\omega \models^I A \wedge \neg b$. But $\omega' = \omega$. This establishes the induction hypothesis $P(n+1)$. By mathematical induction we conclude $P(n)$ holds for all n. Hence the rule for while loops is sound.

The consequence rule: Assume $\models (A \rightarrow A')$ and $\models \{A'\}c\{B'\}$ and $\models (B' \rightarrow B)$.
Let I be an interpretation. Suppose $\omega \models^I A$. Then $\omega^I A'$, hence $C[[c]]\omega \models^I B'$ and hence $C[[c]]\omega \models^I B$. Thus $\{A\}c\{B\}$. The consequence rule is sound.

By rule-induction, every theorem is valid.

6.7 Ideal Completeness

Gödel's Incompleteness Theorem implies there is no complete proof system for establishing precisely the valid assertions. The Hoare rules inherit this incompleteness. However by separating incompleteness of the assertion language from incompleteness due to inadequacies in the axioms and rules for the programming language constructs, we can obtain relative completeness. The proof that Hoare rules are relatively complete relies on the idea of weakest liberal precondition, and leads into a discussion of verification-condition generators.

6.7.1 Weakest Precondition

Definition: wp(c,Q) - the weakest condition such that every terminating computation of s results in a state satisfying Q.

$$[[wp^I(c,Q)]] = \{\sigma \in \Sigma^{\perp} | S[[c]]\sigma \in^I Q\}$$

Theorem: Assn is expressive
Proof: We show by structural induction on commands c that for all assertions B there is an assertion w[[c,B]] such that for all interpretations I

$$wp^I[[c,B]] = w[[c,B]]^I$$

for all commands c.
For example we will show the if command : $c \equiv if \quad b \quad then \quad c_0 \quad else \quad c_1$: Define

$$w[[if \quad b \quad then \quad c_0 \quad else \quad c_1, B]] \equiv [(b \wedge w[[c_0,B]])] \vee [(\neg b \wedge w[[c_1,B]])]$$

Then, for $\sigma \in \Sigma$ and interpretation I,

$$\sigma \in wp^I[[c,B]] \quad iff \quad \mathcal{C}[[c]]\sigma \models^I B$$

iff

$$([\mathcal{B}[[b]]\sigma = \mathbf{true} \quad and \quad \mathcal{C}[[c_0]]\sigma \models^I B]$$

or

$$([\mathcal{B}[[b]]\sigma = \mathbf{false} \quad and \quad \mathcal{C}[[c_1]]\sigma \models^I B]$$

iff

$$(\sigma \models^I b \quad and \quad \sigma \models^I w[[c_0,B]$$

or

$$(\sigma \models^I \neg b \quad and \quad \sigma \models^I w[[c_1,B]$$

, by induction iff

$$\sigma \models I[(b \wedge w[[c_0,B]])] \vee [(\neg b \wedge w[[c_1,B]])]$$

iff

$$\sigma \models Iw[[c,B]]$$

Lemma: For command c and $B \in$ **Assn**, let w[[c,B]] be an assertion expressing the weakest precondition i.e. $w[[c,B]]^I = wpI[[c,B]]$. Then

$$\{w[[c,B]]\}c\{B\}$$

Proof: Let w[[c,B]] be an assertion which expresses the weakest precondition of a command c and postcondition B. We show by structural induction on c that

$$\vdash \{w[[c, B]]\}c\{B\} \quad for \quad all \quad B \in \mathbf{Assn}$$

for all commands c.
For example we will show the if command : $c \equiv if \quad b \quad then \quad c_0 \quad else \quad c_1$: In this case, for $\sigma \in \Sigma$ and interpretation I,

$$\sigma \models Iw[[c, B]] \quad iff \quad \mathcal{C}[[c]]\sigma \models^I B$$

iff

$$([\mathcal{B}[[b]]\sigma = \mathbf{true} \quad and \quad \mathcal{C}[[c_0]]\sigma \models^I B]$$

or

$$([\mathcal{B}[[b]]\sigma = \mathbf{false} \quad and \quad \mathcal{C}[[c_1]]\sigma \models^I B]$$

iff

$$(\sigma \models^I b \quad and \quad \sigma \models^I w[[c_0, B]$$

or

$$(\sigma \models^I \neg b \quad and \quad \sigma \models^I w[[c_1, B]$$

iff

$$\sigma \models I[(b \wedge w[[c_0, B]])] \vee [(\neg b \wedge w[[c_1, B]])]$$

Hence

$$\models w[[c, B]] \Leftrightarrow [(b \wedge w[[c_0, B]]) \vee [(\neg b \wedge w[[c_1, B]]$$

Theorem: The proof system for partial correctness is relatively complete, i.e. for any partial correctness assertion $\{A\}c\{B\}$,

$$\vdash \{A\}c\{B\} \quad if \quad \models \{A\}c\{B\}$$

Proof: Suppose $\models \{A\}c\{B\}$. Then by the above lemma $\vdash \{w[[c, B]]\}c\{B\}$ where $w[[c, B]]^I = wp^I[[c, B]]$ for any interpretation I. Thus as $\models (A \Rightarrow w[[c, B]])$, by the consequence rule, we obtain $\models \{A\}c\{B\}$.

6.7.2 Verification Conditions

Since **Assn** is expressive, the validity of a partial correctness assertion of the form $\{P\}c\{Q\}$ is equivalent to the validity of the assertion $A \Rightarrow w[\![c, B]\!]$, from which the command has been eliminated. In this way, given a theorem prover for predicate calculus we may hope to derive a theorem prover for our programs. Unfortunately, obtaining $w[\![c, B]\!]$ is inefficient and not practical.

However, we can use automatic theorem provers to show partial correctness and to check the validity of such assertions. We start by generating assertions that describe the partial correctness of the program. This is also referred to as annotating the program by assertions. We define the syntactic set of annotated commands by:

$$c ::= \text{skip} \mid X := a \mid c_0; (X := a) \mid c_0; \{D\}c_1 \mid$$
$$\text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } \{D\}c$$

where X is a location, a an arithmetic expression, b a boolean expression, c, c_0, c_1 are annotated commands and D is an assertion such that in $c_0; \{D\}c_1$, the annotated command c_1 is not an assignment.
The general idea: an assertion at a point in the annotated command is true whenever flow of control reaches that point. Thus we only annotate commands of the form $c_0; c_1$ at the point where the control shifts from c_0 to c_1. When c_1 is an assignment the annotation can be derived simply from a postcondition.

An *annotated partial correctness assertion* is of the form $\{A\}c\{B\}$ where c is an annotated command. We can say that an annotated partial correctness assertion is valid when its associated (unannotated) partial correctness assertion is.
For example, the following annotated while loop:

$$\{A\} \text{ while } b \text{ do } \{D\}c\{B\}$$

contains an assertion D which we hope is an invariant, meaning that:

$$\{D \wedge b\}c\{D\}$$

is valid.
In order to ensure that

$$\{A\} \text{ while } b \text{ do } \{D\}c\{B\}$$

is valid, if D is an invariant, it suffices to show that both:

$$A \Rightarrow D,\ D \wedge \neg b \Rightarrow B$$

are valid.
We can derive $\{A\}$ while b do $c\{B\}$ from $\{D \wedge b\}c\{D\}$ using the Hoare rules.

Not all annotated partial correctness assertions are valid. We define the *verification conditions* (vc) by structural induction of annotated commands:

$$
\begin{array}{rcl}
vc(\{A\}\text{skip}\{B\}) & = & \{A \Rightarrow B\} \\
vc(\{A\}X := a\{B\}) & = & \{A \Rightarrow B[a/X]\} \\
vc(\{A\}c_0; X := a\{B\}) & = & vc(\{A\}c_0\{B[a/X]\}) \\
vc(\{A\}c_0; \{D\}c_1\{B\}) & = & vc(\{A\}c_0\{D\}) \cup vc(\{D\}c_1\{B\}) \\
 & & \text{where } c_1 \text{ is not an assignment} \\
vc(\{A\}\text{if } b \text{ then } c_0 \text{ else } c_1\{B\}) & = & vc(\{A \wedge b\}c_0\{B\} \cup vc(\{A \wedge \neg b\}c_1\{B\}) \\
vc(\{A\}\text{while } b \text{ do } \{D\}c\{B\}) & = & vc(\{D \wedge b\}c\{D\}) \cup \{A \Rightarrow D\} \cup \{D \wedge \neg b \Rightarrow B\}
\end{array}
$$

To show that an annotated partial correctness assertion is valid it is sufficient to show its verification conditions are valid. Thus, the program verification can be done by the theorem prover for predicate calculus. An example for such a program is Gypsy(CITE).

While validity of verification conditions is sufficient to guarantee the validity of an annotated partial correctness assertion - it is not necessary (can happen when the invariant chosen is in appropriate for the pre and post conditions).
Example:

{true}while false do {false}skip{true}

The above annotated while-loop is valid with **false** as an invariant. However, its verification conditions contain:

true $\Rightarrow$ false

which is not a valid assertion.

Various theorem-provers:

- Z3 - `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
- Isabelle - `http://isabelle.in.tum.de/nominal/activities/cas09/`
- ESC/Java - `http://en.wikipedia.org/wiki/ESC/Java`
- Spec# - `http://research.microsoft.com/en-us/projects/specsharp/`

6.8 Summary

Axiomatic semantics provides an abstract semantics. It is appropriate for arguing program correctness, and therefore can be used to explain programming.

It has many extensions, such as:

- Procedures
- Concurrency
- Events

- Rely/Guarantee
- Heaps

There are many automatic tools based on axiomatic semantics for various applications, e.g. theorem-provers and program verifiers. However, more effort is required to make it more efficient and practical.

6.9 Further Reading

- Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993. (Chapters 6 and 7)