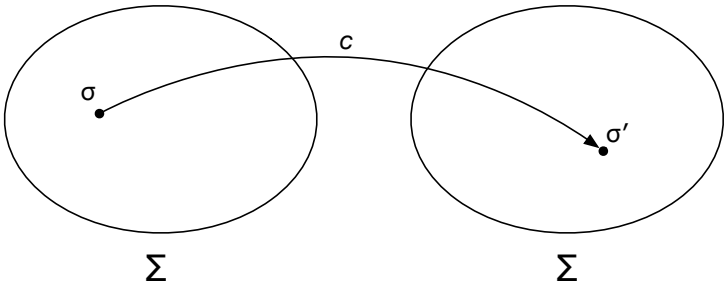
1 Axiomatic semantics

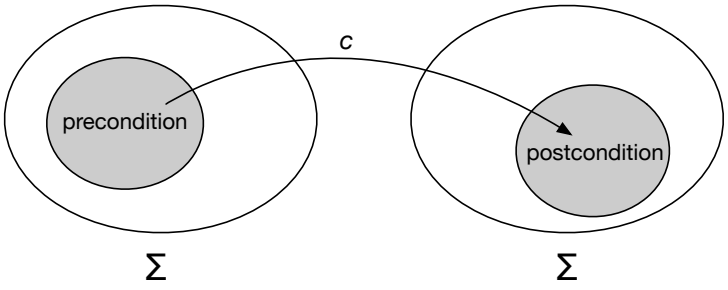
Thus far we have seen two styles of programming language semantics: operational semantics, in which we describe how to evaluate programs, and translational semantics, in which we convert programs to a simpler code (or mathematical) representation. A third important style of semantics is *axiomatic semantics*, in which the possible executions of the program are characterized by logical assertions that describe to *sets* of possible program configurations rather than individual configurations.

Axiomatic semantics provides a coarser way to describe behavior of programs, which is useful because it supports more modular reasoning about programs. As programmers, we often we do not care about exactly what happens inside some code; all we want to know is that when the code is used in an appropriate context, the code achieves its desired goals. For example, we might want to know that when a function is called in a way that satisfies its precondition, its postcondition is satisfied. Axiomatic semantics allows us to separately verify the implementation of the function and the implementation of its caller.

Axiomatic semantics was originally developed for imperative programming, so we will use the IMP programming language as the basis for studying it. Recall that the big-step semantics for IMP defined a relation describing the full evaluation of an IMP command: $\langle c, \sigma\rangle \Downarrow \sigma'$, where σ is the initial store (state) and σ' is the final store. If the set of possible stores is Σ, we can view the big-step semantics as defining a function relating initial and final states, with the following pictorial representation.



The axiomatic semantics lifts this semantics to talk about sets of states. The axiomatic semantics relates a set of states characterized by a *precondition* specified as a logical assertion to a set of states characterized by a *postcondition*.



For example, consider the command $c =$ **while** $x > 0$ **do** $(y := x * y; x := x - 1)$. Starting from a state in which $x = 5$ and $y = 1$, it results in a state in which $y = 5! = 120$. More generally, given an arbitrary value n, starting from a state in which $x = n$ and $y = 1$, it results in a state in which $y = n!$. We can express this idea through a *partial correctness assertion* (PCA):

$$\{x = n \wedge y = 1 \wedge n > 0\}c\{y = n!\}$$

Here, the precondition and postcondition of the command are written as assertions on the left and right side of the command. Each of them may use *logical variables* such as n.

We write $\sigma \vDash A$ to mean that the state σ satisfies the logical assertion A.

Using this notation, the meaning of this PCA can be expressed using the big-step semantics:

$$\begin{array}{l} \forall n, \sigma. \\ \quad \sigma \models (x = n \wedge n > 0 \wedge y = 1) \wedge \langle c, \sigma \rangle \Downarrow \sigma' \\ \quad\quad \Rightarrow \quad \sigma \models y = n! \end{array}$$

Note that because successful evaluation appears on the left side of the implication, partial correctness assertions do not require that the command terminates. It is also possible to write *total correctness assertions*, whose meaning is that the command terminates *and* satisfies the corresponding PCA:

$$[A]\ c\ [B] \triangleq \sigma \models A \Rightarrow\ \{A\}\ c\ \{B\} \wedge\ \langle c, \sigma \rangle \Downarrow$$

However, we will focus on partial correctness assertions.

2 Assertions

We now formalize assertions and their meaning. The grammar of assertions is as follows, built on top of the existing IMP grammar for arithmetic expressions **AExp** and boolean expressions **BExp**:

$$A ::= T \mid F \mid a_1 = a_2 \mid a_1 \ \leq\ a_2 \mid A_1 \vee A_2 \mid A_1 \wedge\ A_2 \mid \neg A \mid A_1 \Rightarrow A_2 \mid \forall i.A \mid \exists i.A$$

The metavariable i represents a logical variable like n, taken from a set **LVar**. Note that the set of assertions **Assn** is a superset of IMP boolean expressions, extended with logical variables and quantification over them. In addition, arithmetic expressions appearing in assertions may mention logical variables.

To give assertions a semantics, we assume an *interpretation* $I \in \mathbf{LVar} \to \mathbb{Z}$ of logical variables is provided. We write $\sigma \vDash^I A$ if σ satisfies the assertion A under logical interpretation I.

We can give a meaning (as an integer) to any arithmetic expression a under an interpretation I. The definition is by structural induction on a. We write $\mathcal{A}[\![a]\!]I\sigma$ to mean the value of a in state σ and interpretation I.

$$\begin{array}{rll} \mathcal{A}[\![n]\!]I\sigma = n & & \\ \mathcal{A}[\![x]\!]I\sigma = \sigma(x) & & \text{(look up ordinary variables in } \sigma) \\ \mathcal{A}[\![i]\!]I\sigma = I(i) & & \text{(look up logical variables in } I) \\ \mathcal{A}[\![a_1 + a_2]\!]I\sigma = \mathcal{A}[\![a_1]\!]I\sigma + \mathcal{A}[\![a_2]\!]I\sigma & & \text{(and so on for other operators)} \end{array}$$

3 Satisfying assertions

Using $\mathcal{A}[\![a]\!]I\sigma$, we next define $\sigma \vDash^I A$ by structural induction on A:

$$\begin{array}{l} \sigma \vDash^I T \\ \sigma \vDash^I a_1 = a_2 \iff \mathcal{A}[\![a_1]\!]I\sigma = \mathcal{A}[\![a_2]\!]I\sigma \\ \sigma \vDash^I A_1 \wedge A_2 \iff (\sigma \vDash^I A_1) \wedge (\sigma \vDash^I A_2) \\ \sigma \vDash^I A_1 \vee A_2 \iff (\sigma \vDash^I A_1) \vee (\sigma \vDash^I A_2) \\ \sigma \vDash^I \neg A \iff \neg(\sigma \vDash^I A) \\ \sigma \vDash^I A \Rightarrow B \iff (\sigma \vDash^I A) \Rightarrow\ (\sigma \vDash^I B) \end{array}$$

These rules can also be expressed in terms of sets. For example, we could write the rule for implication as a statement about sets:

$$\sigma \vDash^I A \Rightarrow B \iff \{\sigma \mid \sigma \vDash^I A\} \subseteq \{\sigma \mid \sigma \vDash^I B\}$$

A state σ satisfies an assertion A if it satisfies it in *any* interpretation; that is, assertions are implicitly universally quantified over free logical variables:

$$\sigma \vDash A \iff \forall I.\, \sigma \vDash^I A$$

PCAs are also implicitly quantified over interpretations:

$$\{A\}\ c\ \{B\} \iff \forall I, \sigma \vDash^I A. \quad \langle c, \sigma\rangle \Downarrow \sigma' \Rightarrow \sigma' \vDash^I B$$

We write $\vDash \{A\}c\{B\}$ to mean that this PCA is semantically true. We can now express the semantics of a command c in terms of all PCAs that it satisfies; that is, as the set of pairs of assertions (A, B) such that $\vDash \{A\}c\{B\}$ holds.

4 Hoare Logic

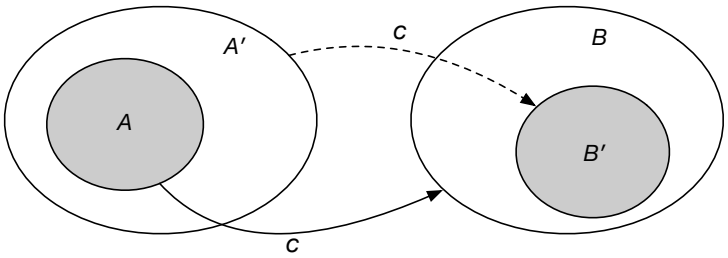
How can we show that a given PCA is true? Hoare's idea was to define a logic for deriving true PCAs: that is, a set of proof rules that can derive all true PCAs. Recall the grammar of commands:

$$c ::= \textbf{skip} \mid x := a \mid c_0\ ;\ c_1 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ b\ \textbf{do}\ c$$

One important proof rule is the *rule of consequence*, also called the *weakening* rule:

$$\frac{A \Rightarrow A' \quad \{A'\}\ c\ \{B'\} \quad B' \Rightarrow B}{\{A\}\ c\ \{B\}}$$

This rule allows us to adjust the preconditions and postconditions of a PCA, by strengthening the precondition and weakening the postcondition. The rule has two side conditions, which require proving $A \Rightarrow A'$ and $B' \Rightarrow B$. To see intuitively why this rule is correct, we can viewing the various assertions in this rule as sets:



The rule says that if we want to show that command c transforms each state in A to some state in B, it suffices to show that it transforms each state in the larger set A' into some state in the smaller set B'.

In addition to the rule of consequence, we have rules for each of possible commands. For example, since **skip** does not change the current state, the most precise rule simply relates all assertions to themselves:

$$\frac{}{\{B\}\ \textbf{skip}\ \{B\}}$$

The rule for assignment is more interesting. A postcondition B mentioning the assigned variable x will hold if B would have held before the assignment took place, but with x replaced with the assigned value a:

$$\frac{}{\{B\{a/x\}\}\ x := a\ \{B\}}$$

For example, suppose the assignment takes the form $x := 2 * 3$ and the postcondition B is $(x = 2 + 4)$. Then the precondition is $2 * 3 = 2 + 4$.

The rule for **if** requires that the command achieves the postcondition regardless of which branch is taken:

$$\frac{\{A \wedge b\}\ c_1\ \{B\} \quad \{A \wedge \neg b\}\ c_2\ \{B\}}{\{A\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{B\}}$$

When sequencing two commands ($c = c_1; c_2$), we need an intermediate assertion describing the state that exists in the middle:

$$\frac{\{A\}\ c_1\ \{D\} \quad \{D\}\ c_2\ \{B\}}{\{A\}\ c_1; c_2\ \{B\}}$$

The rule does not say how to choose D. In practice we would like to use the weakest possible precondition of c_2 that allows proving the second premise.

Finally, we need a rule for **while**. This rule requires us to choose the precondition of the command, which must be a *loop invariant* that is preserved by loop execution.

$$\frac{\{A \wedge b\}\ c\ \{A\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{A \wedge \neg b\}}$$

On termination of the command, we know that the loop invariant still holds, but also that the loop guard b does not. The rule requires proving c preserves the loop invariant holds, and we may make the additional assumption that the loop guard b does hold.

5 Example

Consider the example command from earlier, which we will call w:

$$w = \textbf{while}\ x > 0\ \textbf{do}\ (y := x * y; x := x - 1)$$

We would like to prove the PCA $\{pre\}\ w\ \{post\}$, where pre and $post$ are defined as follows:

$$\begin{aligned} pre &= (x = n \wedge y = 1 \wedge n > 0) \\ post &= (y = n!) \end{aligned}$$

This PCA doesn't look quite like the rule for **while**, so we use the rule of consequence to align the PCA with that rule. We choose the following loop invariant, which is clearly implied by pre:

$$INV = x \geq 0 \wedge\ n > 0 \wedge y = n!/x!$$

On the postcondition side of the rule of consequence, we see that $INV \wedge\ \neg(x > 0) \Rightarrow\ post$, as required, since the left-hand side ensures $x = 0$. We now apply the rules driven by syntax until we reach the body of the loop:

$$\frac{\dfrac{\{INV \wedge\ x > 0\}\ y := y * x; x := x - 1\ \{INV\}}{\{INV\}\ w\ \{INV \wedge x \leq 0\}}}{\{pre\}\ w\ \{post\}}$$

To prove the right-hand premise of the sequence rule, we choose an intermediate assertion D according to the assignment rule for $x := x - 1$:

$$\begin{aligned} D &= (INV \wedge x \leq\ 0)\{x - 1/x\} \\ &= (x - 1) \geq 0 \wedge\ n > 0 \wedge y = n!/(x - 1)! \end{aligned}$$

Now we just have to prove the assertion for the left-hand assignment $y := x * y$. We use the rule of consequence again, choosing its precondition A' according to the assignment rule: $A' = D\{x * y/y\}$. We only need to show $INV \wedge\ x > 0 \Rightarrow D\{x * y/y\}$. Expanding, we have:

$$x \geq 0 \wedge\ n > 0 \wedge y = n!/x! \wedge\ x > 0 \Rightarrow (x - 1) \geq 0 \wedge\ n > 0 \wedge y * x = n!/(x - 1)!$$

This implication clearly holds, so the original PCA is proven.